Appendix

A. Project Structure

Codes for this research is available on Github repository of Computational Medicine Laboratory, Kumoh National Institute of Technology (https://github.com/kit-cml). There are some versions of the code in our Github profile, depending on the cell models and research scenario. This section will discuss deeply the code's version for ORd 2011 cell model (https://github.com/kit-cml/MultiConcGPU). Codes explained in this appendix is in same condition as the time this thesis being written, including some files that might be deleted due to redundancy in the future. More recent updates are available in the Github repository. Structure of the repository can be written as such:

```
MultiConcGPU
├── .gitignore
├── Makefile
├── Makefile_commercial
├── bin
│   ├── CVAR
│   │   ├── HF_male.csv
│   │   └── sens_healthy_male.csv
│   ├── autorun.sh
│   ├── control
│   │   ├── IC50_samples.csv
│   │   └── init_state.csv
│   ├── drug
│   │   ├── IC50_terfenadine.csv
│   │   ├── IC50_verapamil.csv
│   │   └── testing
│   │       └── IC50_Mexiletine.csv
```

```
|   ├── drug_sim
|   ├── ic50_sens
|   |   └── mitoxantrone_sens.csv
|   ├── input_deck.txt
|   ├── mitoxantrone
|   |   └── mitoxantrone_100_samples_50_conc.csv
|   └── result
|       ├── do_not_delete_this_folder
|       └── state_only.zip
├── cellmodels
|   ├── Ohara_Rudy_2011.cpp
|   ├── Ohara_Rudy_2011.hpp
|   ├── cellmodel.hpp
|   └── enums
|       ├── enum_Ohara_Rudy_2011.hpp
|       └── enum_ord2011.hpp
├── main.cu
├── modules
|   ├── cipa_t.cu
|   ├── cipa_t.cuh
|   ├── drug_conc.cpp
|   ├── drug_conc.hpp
|   ├── glob_funct.cpp
|   ├── glob_funct.hpp
|   ├── glob_type.cpp
|   ├── glob_type.hpp
|   ├── gpu.cu
|   ├── gpu.cuh
|   ├── gpu_cu_.backup
|   ├── gpu_cuh.backup
|   ├── gpu_glob_type.cu
|   ├── gpu_glob_type.cuh
|   ├── param.cpp
|   └── param.hpp
└── test_compile.bat
```

The main structure, such as folder names, will less likely to be revised. Scripts inside the folder, especially in the 'modules' folder, more likely to be changed by removing some redundant functions. Next section will deeply discuss each file and their possibility of redundancy.

B. Root folder

Root folder is the main folder that contains makefile, gitignore file, main code, and test_compile.bat. The next sub section will discuss each of them deeply.

a. Makefile

This script is used for compiling the whole project in correct order, and enables easy clean-up and re-compilation. Binary files will be cleaned, and the simulator will be re-compiled with 'make clean all' command. Compilation configuration and steps described as follow:

```
# Some notes:
# - Using ':=' instead of '=' assign the value at Makefile parsing time,
#   others are evaluated at usage time. This discards
# - Use ':set list' in Vi/Vim to show tabs (Ctrl-v-i force tab insertion)
#

# List to '.PHONY' all fake targets, those that are neither files nor folders.
# "all" and "clean" are good candidates.
.PHONY: all, clean

# Define the final program name
PROGNAME := drug_sim

# Pre-processor flags to be used for includes (-I) and defines (-D)
CPPFLAGS := -I/usr/local/cuda/include
# CPPFLAGS :=
# CXX to set the compiler
# CXX := g++
CXX := nvcc
CXXLINK := nvlink

# CXXFLAGS is used for C++ compilation options.
```

```
        #CXXFLAGS += -Wall -O0 -fpermissive -std=c++11
        #CXXFLAGS += -Wall -O2 -fno-alias -fpermissive
        # CXXFLAGS += -Wall
        # Use this if you want to use ToR-ORd 2019 cell model.
        # Otherwise, comment it
        #CXXFLAGS += -DTOR-ORD_2019

        # LDFLAGS is used for linker (-g enables debug symbols)
        # LDFLAGS  += -g -L/usr/local/cuda/lib64
        LDFLAGS  += -g -L/usr/local/cuda/lib64 -arch=sm_86 -rdc=true

        # List the project' sources to compile or let the Makefile recognize
        # them for you using 'wildcard' function.
        #
        SOURCES  = $(wildcard *.cpp) $(wildcard **/*.cpp) $(wildcard *.c) $(wildcard **/*.c)
$(wildcard **/*.cu) $(wildcard *.cu)

        # List the project' headers or let the Makefile recognize
        # them for you using 'wildcard' function.
        #
        HEADERS  = $(wildcard *.hpp) $(wildcard **/*.hpp) $(wildcard *.h) $(wildcard **/*.h)
$(wildcard **/*.cuh) $(wildcard *.cuh)

        # Construct the list of object files based on source files using
        # simple extension substitution.
        OBJECTS := $(SOURCES:%.cpp=%.o)
        LIBS= -lopenblas -lpthread -lcudart -lcublas

        #
        # Now declare the dependencies rules and targets
        #
        # Starting with 'all' make it  becomes the default target when none
        # is specified on 'make' command line.
        all : $(PROGNAME)

        # Declare that the final program depends on all objects and the Makfile
        $(PROGNAME) : $(OBJECTS) Makefile
                $(CXX) -o bin/$@ $(OBJECTS) $(LDFLAGS)

        # Now the choice of using implicit rules or not (my choice)...
        #
        # Choice 1: use implicit rules and then we only need to add some dependencies
        #           to each object.
        #
        ## Tells make that each object file depends on all headers and this Makefile.
        #$(OBJECTS) : $(HEADERS) Makefile
        #
        # Choice 2: don't use implicit rules and specify our will
        %.o: %.cpp $(HEADERS) Makefile
```

```
	$(CXX) -x cu $(CXXFLAGS) $(CPPFLAGS) -dc -arch=sm_86 $(OUTPUT_OPTION) $<
# -dc -rdc=true

# Simple clean-up target
# notes:
# - the '@' before 'echo' informs make to hide command invocation.
# - the '-' before 'rm' command to informs make to ignore errors.
clean :
	@echo "Clean."
	rm -rf *.o bin/$(PROGNAME)
	rm -rf **/*.o
```

b. .gitignore

.gitignore file specifies files and folders to be ignored by git, the version control used in this research. The git will ignore simulation results, binary files, CUDA related libraries, log files, and a jupyter notebook used in plotting. Below is the list of files and folder I ignore in the research:

```
*.i
*.ii
*.gpu
*.ptx
*.cubin
*.fatbin
.DS_Store
bin/drug_sim
*.o
*.plt
*.out
bin/result/*/*.csv
bin/result/*/*/*.csv
bin/result/parse.ipynb
output.*
*.old
bin/result/*
```

c. main.cu

The main.cu file serves as the primary entry point for executing the drug simulation program. This file orchestrates the interaction between the core modules, manages the GPU-based computations, and handles input and output processes. Every CPU related orchestration for the simulation happen in main.cu. Its primary responsibilities are initialisation, loading input data, initialise GPU environment, core number calculation, executing simulation, output handling, memory clean-up, logging, and making sure all input and output are correct. I found that CUDA debugger does not really help the debugging process due to unique parallelisation in this research. Self-created debugging points were also introduced in the main.cu. The main.cu coded as below:

```
#include <cuda.h>
#include <cuda_runtime.h>

// #include "modules/drug_sim.hpp"
#include "modules/glob_funct.hpp"
#include "modules/glob_type.hpp"
#include "modules/gpu.cuh"
#include "modules/cipa_t.cuh"

#include <cstdio>
#include <cstdlib>
#include <cstring>
#include <ctime>
#include <iostream>
#include <math.h>
#include <vector>
#include <sys/stat.h>

#define ENOUGH ((CHAR_BIT * sizeof(int) - 1) / 3 + 2)
char buffer[255];


// unsigned int datapoint_size = 7000;
const unsigned int sample_limit = 10000;
```

```
clock_t START_TIMER;

clock_t tic();
void toc(clock_t start = START_TIMER);

clock_t tic()
{
    return START_TIMER = clock();
}

void toc(clock_t start)
{
    std::cout
        << "Elapsed time: "
        << (clock() - start) / (double)CLOCKS_PER_SEC << "s"
        << std::endl;
}

int gpu_check(unsigned int datasize){
    int num_gpus;
    float percent;
    int id;
    size_t free, total;
    cudaGetDeviceCount( &num_gpus );
    for ( int gpu_id = 0; gpu_id < num_gpus; gpu_id++ ) {
        cudaSetDevice( gpu_id );
        cudaGetDevice( &id );
        cudaMemGetInfo( &free, &total );
        percent = (free/(float)total);
        printf("GPU No %d\nFree Memory: %ld, Total Memory: %ld (%f percent free)\n",
id,free,total,percent*100.0);
    }
    percent = 1.0-(datasize/(float)total);
    //// this code strangely gave out too small value, so i disable the safety switch for now

    // printf("The program uses GPU No %d and %f percent of its memory\n", id,percent*100.0);
    // printf("\n");
    // if (datasize<=free) {
    //   return 0;
    // }
    // else {
    //   return 1;
    // }


    return 0;

}
```

```
// get the IC50 data from file
drug_t get_IC50_data_from_file(const char* file_name);
// return error and message based on the IC50 data
int check_IC50_content(const drug_t* ic50, const param_t* p_param);


int get_IC50_data_from_file(const char* file_name, double *ic50)
{
    /*
    a host function to take all samples from the file, assuming each sample has 14 features.

    it takes the file name, and an ic50 (already declared in 1D, everything become 1D)
    as a note, the data will be stored in 1D array, means this functions applies flatten.

    it returns 'how many samples were detected?' in integer.
    */
  FILE *fp_drugs;
//   drug_t ic50;
  char *token;
  char buffer_ic50[255];
  unsigned int idx;

  if( (fp_drugs = fopen(file_name, "r")) == NULL){
    printf("Cannot open file %s\n",
      file_name);
    return 0;
  }
  idx = 0;
  int sample_size = 0;
  fgets(buffer_ic50, sizeof(buffer_ic50), fp_drugs); // skip header
  while( fgets(buffer_ic50, sizeof(buffer_ic50), fp_drugs) != NULL )
  { // begin line reading
    token = strtok( buffer_ic50, "," );
    while( token != NULL )
    { // begin data tokenizing
      ic50[idx++] = strtod(token, NULL);
      token = strtok(NULL, ",");
    } // end data tokenizing
    sample_size++;
  } // end line reading

  fclose(fp_drugs);
  return sample_size;
}

int get_cvar_data_from_file(const char* file_name, unsigned int limit, double *cvar)
{
  // buffer for writing in snprintf() function
```

```
  char buffer_cvar[255];
  FILE *fp_cvar;
  // cvar_t cvar;
  char *token;
  // std::array<double,18> temp_array;
  unsigned int idx;

  if( (fp_cvar = fopen(file_name, "r")) == NULL){
    printf("Cannot open file %s\n",
      file_name);
  }
  idx = 0;
  int sample_size = 0;
  fgets(buffer_cvar, sizeof(buffer_cvar), fp_cvar); // skip header
  while( (fgets(buffer_cvar, sizeof(buffer_cvar), fp_cvar) != NULL) && (sample_size<limit))
  { // begin line reading
    token = strtok( buffer_cvar, "," );
    while( token != NULL )
    { // begin data tokenizing
      cvar[idx++] = strtod(token, NULL);
      token = strtok(NULL, ",");
    } // end data tokenizing
    // printf("\n");
    sample_size++;
    // cvar.push_back(temp_array);
  } // end line reading

  fclose(fp_cvar);
  return sample_size;
}


int get_init_data_from_file(const char* file_name, double *init_states)
{
  // buffer for writing in snprintf() function
  char buffer_cache[1023];
  FILE *fp_cache;
  // cvar_t cvar;
  char *token;
  // std::array<double,18> temp_array;
  unsigned long idx;

  if( (fp_cache = fopen(file_name, "r")) == NULL){
    printf("Cannot open file %s\n",
      file_name);
  }
  idx = 0;
  unsigned int sample_size = 0;
  // fgets(buffer_cvar, sizeof(buffer_cvar), fp_cvar); // skip header
```

```
  while( (fgets(buffer_cache, sizeof(buffer_cache), fp_cache) != NULL) )
  { // begin line reading
    token = strtok( buffer_cache, "," );
    while( token != NULL )
    { // begin data tokenizing
      init_states[idx++] = strtod(token, NULL);
      // if(idx < 82){
      //     printf("%d: %lf\n",idx-1,init_states[idx-1]);
      // }
      token = strtok(NULL, ",");
    } // end data tokenizing
    // printf("\n");
    sample_size++;
    // cvar.push_back(temp_array);
  } // end line reading

  fclose(fp_cache);
  return sample_size;
}
int exists(const char *fname)
{
    FILE *file;
    if ((file = fopen(fname, "r")))
    {
        fclose(file);
        return 1;
    }
    // fclose(file);
    return 0;
}

int check_IC50_content(const drug_t* ic50, const param_t* p_param)
{
        if(ic50->size() == 0){
                printf("Something problem with the IC50 file!\n");
                return 1;
        }
        else if(ic50->size() > 2000){
                printf( "Too much input! Maximum sample data is 2000!\n");
                return 2;
        }
        else if(p_param->pace_max < 750 && p_param->pace_max > 1000){
                printf("Make sure the maximum pace is around 750 to 1000!\n");
                return 3;
        }
        // else if(mympi::size > ic50->size()){
        //       printf("%s\n%s\n",
  //              "Overflow of MPI Process!",
  //              "Make sure MPI Size is less than or equal the number of sample");
```

```
        //      return 4;
        // }
        else{
                return 0;
        }
}

int main(int argc, char **argv)
{
        // enable real-time output in stdout
        //setvbuf( stdout, NULL, _IONBF, 0 );

// NEW CODE STARTS HERE //
    // mycuda *thread_id;
    // cudaMalloc(&thread_id, sizeof(mycuda));
```

This is how the main.cu manages multi-concentration in the simulation. Since 'kernel_DoDrugSim' uses less memory, I used them to put more samples, then I can put more concentration in the modified IC50 file.

```
  // TODO: Automation 3. check file inside folder
    for (const auto &entry : fs::directory_iterator(drug_dir)) {
        param_t *p_param, *d_p_param;
        p_param = new param_t();
        p_param->init();
        edison_assign_params(argc, argv, p_param);

        std::filesystem::directory_entry dir_entry = entry;
        std::string entry_str = dir_entry.path().string();
        std::cout << entry_str << std::endl;
        std::regex pattern("/([a-zA-Z0-9_\.]+)\.csv");
        std::smatch match;
        std::regex_search(entry_str, match, pattern);

        // TODO: Automation 2. create drug_name and conc

        // TODO: NewFile 2. disable drug name for now since the file name is inside it
        // strcpy(p_param->drug_name, match[1].str().c_str());
        strcpy(p_param->hill_file, entry_str.c_str());
        // strcat(p_param->hill_file, ".csv");
        // strcat(p_param->hill_file, "/IC50_samples.csv");

        // TODO: NewFile 3. getvalue from source is unnecessary
        // p_param->conc = getValue(drugConcentration, match[1].str()) * cmax;
```

```
        // p_param->show_val();

    // for qinwards calculation
    double inal_auc_control = -90.547322;    // AUC of INaL under control model
    double ical_auc_control = -105.935067;   // AUC of ICaL under control model

    // input variables for cell simulation
    param_t *p_param = new param_t();  // input data for CPU
    param_t *d_p_param;  // input data for GPU parsing

         p_param->init();
    edison_assign_params(argc,argv,p_param);
    p_param->show_val();

    double* ic50 = (double *)malloc(14 * sample_limit * sizeof(double));
    // if (p_param->is_cvar == true) cvar = (double *)malloc(18 * sample_limit *
sizeof(double));
    double* cvar = (double *)malloc(18 * sample_limit * sizeof(double));  // conductance
variability

    const int num_of_constants = 146;
    const int num_of_states = 41;
    const int num_of_algebraic = 199;
    const int num_of_rates = 41;
    const double CONC = p_param->conc;
```

Below is how the main.cu manages memory and data output for ‘kernel_DoDrugSim_single’. This kernel function requires more memory due to its more detailed output, hence requires a slightly adapted way to ensure all temporary memory wiped after simulation. In practice, this function also takes up to 60% more GPU memory compared to ‘kernel_DoDrugSim’.

```
    ////////// if we are in write time series mode (post processing) //////////
    if(p_param->is_time_series == 1 /*&& exists(p_param->cache_file) == 1 <- still
unstable*/){

    printf("Using cached initial state from previous result!!!! \n\n");

    const unsigned int datapoint_size = p_param->sampling_limit; // sampling_limit: limit of
num of data points in one sample
```

```
    double* cache = (double *)malloc((num_of_states+2) * sample_limit * sizeof(double)); //
array for in silico results


    static const int CALCIUM_SCALING = 1000000;
    static const int CURRENT_SCALING = 1000;

    // snprintf(buffer, sizeof(buffer),
    //   "./drugs/bepridil/IC50_samples.csv"
    //   // "./drugs/bepridil/IC50_optimal.csv"
    //   // "./IC50_samples.csv"
    //   );

    int sample_size = get_IC50_data_from_file(p_param->hill_file, ic50);
    if(sample_size == 0)
        printf("Something problem with the IC50 file!\n");
    // else if(sample_size > 2000)
    //     printf("Too much input! Maximum sample data is 2000!\n");
    printf("Sample size: %d\n",sample_size);
    printf("Set GPU Number: %d\n",p_param->gpu_index);

    cudaSetDevice(p_param->gpu_index);  // select a specific GPU

    if(p_param->is_cvar == true){
      int cvar_sample = get_cvar_data_from_file(p_param->cvar_file,sample_size,cvar);
      printf("Reading: %d Conductance Variability samples\n",cvar_sample);
    }

    printf("preparing GPU memory space \n");

      // char buffer_cvar[255];
      // snprintf(buffer_cvar, sizeof(buffer_cvar),
      // "./result/66_00.csv"
      // // "./drugs/optimized_pop_10k.csv"
      // );
      int cache_num = get_init_data_from_file(p_param->cache_file,cache);  //

      printf("Found cache for %d samples\n",cache_num);
      // note to self:
      // num of states+2 gave you at the very end of the file (pace number)
      // the very beginning -> the core number
    //   for (int z = 0; z <  num_of_states; z++) {printf("%lf\n", cache[z+1]);}
    //   printf("\n");
    //   for (int z = 0; z <  num_of_states; z++) {printf("%lf\n", cache[ 1*(num_of_states+2)
+ (z+2)]);}
    //   printf("\n");
    //   for (int z = 0; z <  num_of_states; z++) {printf("%lf\n", cache[ 2*(num_of_states+2)
+ (z+3)]);}
    // return 0 ;
```

```
    double *d_ic50;
    double *d_cvar;
    double *d_ALGEBRAIC;
    double *d_CONSTANTS;
    double *d_RATES;
    double *d_STATES;
    double *d_STATES_cache;
    // actually not used but for now, this is only for satisfiying the GPU regulator
parameters
    double *d_STATES_RESULT;
    double *d_all_states;
    cudaMalloc(&d_ALGEBRAIC, num_of_algebraic * sample_size * sizeof(double));
    cudaMalloc(&d_CONSTANTS, num_of_constants * sample_size * sizeof(double));
    cudaMalloc(&d_RATES, num_of_rates * sample_size * sizeof(double));
    cudaMalloc(&d_STATES, num_of_states * sample_size * sizeof(double));
    cudaMalloc(&d_STATES_cache, (num_of_states+2) * sample_size * sizeof(double));
    cudaMalloc(&d_p_param,  sizeof(param_t));

    double *time;
    double *dt;
    double *states;
    double *ical;
    double *inal;
    double *cai_result;
    double *ina;
    double *ito;
    double *ikr;
    double *iks;
    double *ik1;
    cipa_t *temp_result, *cipa_result;
    // prep for 1 cycle plus a bit (7000 * sample_size)
    cudaMalloc(&temp_result, sample_size * sizeof(cipa_t));  // for temporal ??
    cudaMalloc(&cipa_result, sample_size * sizeof(cipa_t));  // output of postprocessing

    cudaMalloc(&time, sample_size * datapoint_size * sizeof(double));
    cudaMalloc(&dt, sample_size * datapoint_size * sizeof(double));

    cudaMalloc(&states, sample_size * datapoint_size * sizeof(double));
    cudaMalloc(&ical, sample_size * datapoint_size * sizeof(double));
    cudaMalloc(&inal, sample_size * datapoint_size * sizeof(double));
    cudaMalloc(&cai_result, sample_size * datapoint_size * sizeof(double));
    cudaMalloc(&ina, sample_size * datapoint_size * sizeof(double));
    cudaMalloc(&ito, sample_size * datapoint_size * sizeof(double));
    cudaMalloc(&ikr, sample_size * datapoint_size * sizeof(double));
    cudaMalloc(&iks, sample_size * datapoint_size * sizeof(double));
    cudaMalloc(&ik1, sample_size * datapoint_size * sizeof(double));
    // cudaMalloc(&d_STATES_RESULT, (num_of_states+1) * sample_size * sizeof(double));
    // cudaMalloc(&d_all_states, num_of_states * sample_size * p_param->find_steepest_start *
sizeof(double));
```

```
    cudaMalloc(&d_ic50, sample_size * 14 * sizeof(double));  // ic50s of 7 channels
    cudaMalloc(&d_cvar, sample_size * 18 * sizeof(double));  // conductances of 18

    printf("Copying sample files to GPU memory space \n");
    cudaMemcpy(d_STATES_cache, cache, (num_of_states+2) * sample_size * sizeof(double),
cudaMemcpyHostToDevice);
    cudaMemcpy(d_ic50, ic50, sample_size * 14 * sizeof(double), cudaMemcpyHostToDevice);
    cudaMemcpy(d_cvar, cvar, sample_size * 18 * sizeof(double), cudaMemcpyHostToDevice);
    cudaMemcpy(d_p_param, p_param, sizeof(param_t), cudaMemcpyHostToDevice);

    // // Get the maximum number of active blocks per multiprocessor
    // cudaOccupancyMaxActiveBlocksPerMultiprocessor(&numBlocks, do_drug_sim_analytical,
threadsPerBlock);

    // // Calculate the total number of blocks
    // int numTotalBlocks = numBlocks * cudaDeviceGetMultiprocessorCount();
    tic();
    printf("Timer started, doing simulation.... \n\n\nGPU Usage at this moment: \n");
    int thread;
    if (sample_size>=16) thread = 16; // change this according to hardware
// optimal number of thread by experience -> might be different for each GPU, can be 16, can
be 32
    else thread = sample_size;
    // int block = int(ceil(sample_size*1.0/thread)+1);
    int block = (sample_size + thread - 1) / thread;
    // int block = (sample_size + thread - 1) / thread;
    if(gpu_check(15 * sample_size * sizeof(double) + sizeof(param_t)) == 1){
      printf("GPU memory insufficient!\n");
      return 0;
    }
    printf("Sample size: %d\n",sample_size);
    cudaSetDevice(p_param->gpu_index);
    printf("\n   Configuration: \n\n\tblock\t||\tthread\n-----------------------------------
--\n  \t%d\t||\t%d\n\n\n", block,thread);
    // initscr();
    //
printf("[________________________________________________________________________________________
_____________]  0.00 %% \n");


    kernel_DrugSimulation<<<block,thread>>>(d_ic50, d_cvar, d_CONSTANTS, d_STATES,
d_STATES_cache, d_RATES, d_ALGEBRAIC,
                                            d_STATES_RESULT, d_all_states,
                                            time, states, dt, cai_result,
                                            ina, inal,
                                            ical, ito,
                                            ikr, iks,
                                            ik1,
                                            sample_size,
```

```
                                        temp_result, cipa_result,
                                        d_p_param
                                        );
                                    //block per grid, threads per block
    // endwin();

    cudaDeviceSynchronize();


    printf("allocating memory for computation result in the CPU, malloc style \n");
    double
*h_states,*h_time,*h_dt,*h_ical,*h_inal,*h_cai_result,*h_ina,*h_ito,*h_ikr,*h_iks,*h_ik1;
    cipa_t *h_cipa_result;

    h_states = (double *)malloc(datapoint_size * sample_size * sizeof(double));
    printf("...allocated for STATES, \n");
    h_time = (double *)malloc(datapoint_size * sample_size * sizeof(double));
    printf("...allocated for time, \n");
    h_dt = (double *)malloc(datapoint_size * sample_size * sizeof(double));
    printf("...allocated for dt, \n");
    h_cai_result= (double *)malloc(datapoint_size * sample_size * sizeof(double));
    printf("...allocated for Cai, \n");
     h_ina= (double *)malloc(datapoint_size * sample_size * sizeof(double));
    printf("...allocated for iNa, \n");
     h_ito= (double *)malloc(datapoint_size * sample_size * sizeof(double));
    printf("...allocated for ito, \n");
     h_ikr= (double *)malloc(datapoint_size * sample_size * sizeof(double));
    printf("...allocated for ikr, \n");
     h_iks= (double *)malloc(datapoint_size * sample_size * sizeof(double));
    printf("...allocated for iks, \n");
     h_ik1= (double *)malloc(datapoint_size * sample_size * sizeof(double));
    printf("...allocated for ik1, \n");
     h_ical= (double *)malloc(datapoint_size * sample_size * sizeof(double));
    printf("...allocated for ICaL, \n");
    h_inal = (double *)malloc(datapoint_size * sample_size * sizeof(double));

    h_cipa_result = (cipa_t *)malloc( sample_size * sizeof(cipa_t));
    printf("...allocating for INaL and postprocessing, all set!\n");

    ////// copy the data back to CPU, and write them into file ////////
    printf("copying the data back to the CPU \n");

    cudaMemcpy(h_states, states, sample_size * datapoint_size * sizeof(double),
cudaMemcpyDeviceToHost);
    cudaMemcpy(h_time, time, sample_size * datapoint_size * sizeof(double),
cudaMemcpyDeviceToHost);
    cudaMemcpy(h_dt, dt, sample_size * datapoint_size * sizeof(double),
cudaMemcpyDeviceToHost);
```

```
    cudaMemcpy(h_ical, ical, sample_size * datapoint_size * sizeof(double),
cudaMemcpyDeviceToHost);
    cudaMemcpy(h_inal, inal, sample_size * datapoint_size * sizeof(double),
cudaMemcpyDeviceToHost);
    cudaMemcpy(h_cai_result, cai_result, sample_size * datapoint_size * sizeof(double),
cudaMemcpyDeviceToHost);
    cudaMemcpy(h_ina, ina, sample_size * datapoint_size * sizeof(double),
cudaMemcpyDeviceToHost);
    cudaMemcpy(h_ito, ito, sample_size * datapoint_size * sizeof(double),
cudaMemcpyDeviceToHost);
    cudaMemcpy(h_ikr, ikr, sample_size * datapoint_size * sizeof(double),
cudaMemcpyDeviceToHost);
    cudaMemcpy(h_iks, iks, sample_size * datapoint_size * sizeof(double),
cudaMemcpyDeviceToHost);
    cudaMemcpy(h_ik1, ik1, sample_size * datapoint_size * sizeof(double),
cudaMemcpyDeviceToHost);

    cudaMemcpy(h_cipa_result, cipa_result, sample_size * sizeof(cipa_t),
cudaMemcpyDeviceToHost);

    FILE *writer;
    int check;
    bool folder_created = false;

    printf("writing to file... \n");
    // sample loop
    for (int sample_id = 0; sample_id<sample_size; sample_id++){
      // printf("writing sample %d... \n",sample_id);
      char sample_str[ENOUGH];
      char conc_str[ENOUGH];
      char filename[500] = "./result/";
      sprintf(sample_str, "%d", sample_id);
      sprintf(conc_str, "%.2f", CONC);
      strcat(filename,conc_str);
      strcat(filename,"/");
      if (folder_created == false){
        check = mkdir(filename,0777);
        // check if directory is created or not
        if (!check){
          printf("Directory created\n");
          }
        else {
          printf("Unable to create directory, or the folder is already created, relax
mate...\n");
      }
      folder_created = true;
      }

      strcat(filename,sample_str);
```

```
      strcat(filename,"_timeseries.csv");

      writer = fopen(filename,"w");
      fprintf(writer, "Time,Vm,dVm/dt,Cai,INa,INaL,ICaL,IKs,IKr,IK1,Ito\n");
      for (int datapoint = 1; datapoint<datapoint_size; datapoint++){
       if (h_time[ sample_id + (datapoint * sample_size)] == 0.0) {break;}
        fprintf(writer,"%lf,%lf,%lf,%lf,%lf,%lf,%lf,%lf,%lf,%lf,%lf\n", // change this into
string, or limit the decimal accuracy, so we can decrease filesize
        h_time[ sample_id + (datapoint * sample_size)],
        h_states[ sample_id + (datapoint * sample_size)],
        h_dt[ sample_id + (datapoint * sample_size)],
        h_cai_result[ sample_id + (datapoint * sample_size)],

        h_ina[ sample_id + (datapoint * sample_size)],
        h_inal[ sample_id + (datapoint * sample_size)],

        h_ical[ sample_id + (datapoint * sample_size)],
        h_iks[ sample_id + (datapoint * sample_size)],

        h_ikr[ sample_id + (datapoint * sample_size)],
        h_ik1[ sample_id + (datapoint * sample_size)],

        h_ito[ sample_id + (datapoint * sample_size)]
        );
      }
      fclose(writer);
    }

    printf("writing each biomarkers value... \n");
    // sample loop
      char conc_str[ENOUGH];
      char filename[500] = "./result/";
      // sprintf(sample_str, "%d", sample_id);
      sprintf(conc_str, "%.2f", CONC);
      strcat(filename,conc_str);
      strcat(filename,"/");
      // printf("creating %s... \n", filename);
      if (folder_created == false){
        check = mkdir(filename,0777);
        // check if directory is created or not
        if (!check){
          printf("Directory created\n");
          }
        else {
          printf("Unable to create directory, or the folder is already created, relax
mate...\n");
      }
      folder_created = true;
      }
```

```
      // strcat(filename,sample_str);
    strcat(filename,"_biomarkers.csv");

    writer = fopen(filename,"a");

    fprintf(writer,
"sample,qnet,qInward,inal_auc,ical_auc,apd90,apd50,apd_tri,cad90,cad50,cad_tri,dvmdt_repol,vm_
peak,vm_valley,vm_dia,ca_peak,ca_valley,ca_dia\n");
    for (int sample_id = 0; sample_id<sample_size; sample_id++){
      // printf("writing sample %d... \n",sample_id);


fprintf(writer,"%d,%lf,%lf,%lf,%lf,%lf,%lf,%lf,%lf,%lf,%lf,%lf,%lf,%lf,%lf,%lf,%lf,%lf\n", //
change this into string, or limit the decimal accuracy, so we can decrease filesize
        sample_id,
        h_cipa_result[sample_id].qnet,
        0.5*((h_cipa_result[sample_id].ical_auc /
ical_auc_control)+(h_cipa_result[sample_id].inal_auc / inal_auc_control)),
        h_cipa_result[sample_id].inal_auc,
        h_cipa_result[sample_id].ical_auc,

        h_cipa_result[sample_id].apd90,
        h_cipa_result[sample_id].apd50,
        h_cipa_result[sample_id].apd90 - h_cipa_result[sample_id].apd50,

        h_cipa_result[sample_id].cad90,
        h_cipa_result[sample_id].cad50,
        h_cipa_result[sample_id].cad90 - h_cipa_result[sample_id].cad50,

        h_cipa_result[sample_id].dvmdt_repol,
        h_cipa_result[sample_id].vm_peak,
        h_cipa_result[sample_id].vm_valley,
        h_cipa_result[sample_id].vm_dia,

        h_cipa_result[sample_id].ca_peak,
        h_cipa_result[sample_id].ca_valley,
        h_cipa_result[sample_id].ca_dia

    //      temp_result[sample_id].qnet = 0.;
    // temp_result[sample_id].inal_auc = 0.;
    // temp_result[sample_id].ical_auc = 0.;

    // temp_result[sample_id].dvmdt_repol = -999;
    // temp_result[sample_id].dvmdt_max = -999;
    // temp_result[sample_id].vm_peak = -999;
    // temp_result[sample_id].vm_valley = d_STATES[(sample_id * num_of_states) +V];
    // temp_result[sample_id].vm_dia = -999;
```

```
    // temp_result[sample_id].apd90 = 0.;
    // temp_result[sample_id].apd50 = 0.;
    // temp_result[sample_id].ca_peak = -999;
    // temp_result[sample_id].ca_valley = d_STATES[(sample_id * num_of_states) +cai];
    // temp_result[sample_id].ca_dia = -999;
    // temp_result[sample_id].cad90 = 0.;
    // temp_result[sample_id].cad50 = 0.;
        );

    }
     fclose(writer);

    toc();

    return 0;
  }
```

The following is the *in silico* mode. This part of the main.cu explains how differ the handling for 'kernel_DoDrugSim' compared to 'kernel_DoDrugSim_single'. 'kernel_DoDrugSim' requires less memory, so this following section of the code removes some memory assignments compared to 'kernel_DoDrugSim_single:

```
    ////////// find cache mode (in silico code) //////////
    else{
    printf("in silico mode, creating cache file because we don't have that yet, or
is_time_series is intentionally false \n\n");
    double *d_ic50;
    double *d_cvar;
    double *d_ALGEBRAIC;
    double *d_CONSTANTS;
    double *d_RATES;
    double *d_STATES;

    // not used, only to satisfy the parameters of the GPU regulator's function
    double *d_STATES_cache;
    double *time;
    double *dt;
    double *states;
    double *cai_result;
    double *ical;
    double *inal;
    double *ina;
    double *ito;
```

```
    double *ikr;
    double *iks;
    double *ik1;

    double *d_STATES_RESULT;
    double *d_all_states;

    cipa_t *temp_result, *cipa_result;

    int sample_size = get_IC50_data_from_file(p_param->hill_file, ic50);
    if(sample_size == 0)
        printf("Something problem with the IC50 file!\n");
    // else if(sample_size > 2000)
    //     printf("Too much input! Maximum sample data is 2000!\n");
    printf("Sample size: %d\n",sample_size);
    cudaSetDevice(p_param->gpu_index);
    printf("preparing GPU memory space \n");

    if(p_param->is_cvar == true){
      int cvar_sample = get_cvar_data_from_file(p_param->cvar_file,sample_size,cvar);
      printf("Reading: %d Conductance Variability samples\n",cvar_sample);
    }

    cudaMalloc(&d_ALGEBRAIC, num_of_algebraic * sample_size * sizeof(double));
    cudaMalloc(&d_CONSTANTS, num_of_constants * sample_size * sizeof(double));
    cudaMalloc(&d_RATES, num_of_rates * sample_size * sizeof(double));
    cudaMalloc(&d_STATES, num_of_states * sample_size * sizeof(double));

    cudaMalloc(&d_p_param,  sizeof(param_t));

    // prep for 1 cycle plus a bit (7000 * sample_size)
    cudaMalloc(&temp_result, sample_size * sizeof(cipa_t));
    cudaMalloc(&cipa_result, sample_size * sizeof(cipa_t));

    cudaMalloc(&d_STATES_RESULT, (num_of_states+1) * sample_size * sizeof(double)); // for
cache file
    cudaMalloc(&d_all_states, num_of_states * sample_size * p_param->find_steepest_start *
sizeof(double)); // for each sample

    printf("Copying sample files to GPU memory space \n");
    cudaMalloc(&d_ic50, sample_size * 14 * sizeof(double));
    // if(p_param->is_cvar == true) cudaMalloc(&d_cvar, sample_size * 18 * sizeof(double));
    cudaMalloc(&d_cvar, sample_size * 18 * sizeof(double));

    cudaMemcpy(d_ic50, ic50, sample_size * 14 * sizeof(double), cudaMemcpyHostToDevice);
    // if(p_param->is_cvar == true) cudaMemcpy(d_cvar, cvar, sample_size * 18 *
sizeof(double), cudaMemcpyHostToDevice);
    cudaMemcpy(d_cvar, cvar, sample_size * 18 * sizeof(double), cudaMemcpyHostToDevice);
    cudaMemcpy(d_p_param, p_param, sizeof(param_t), cudaMemcpyHostToDevice);
```

```
    // // Get the maximum number of active blocks per multiprocessor
    // cudaOccupancyMaxActiveBlocksPerMultiprocessor(&numBlocks, do_drug_sim_analytical,
threadsPerBlock);

    // // Calculate the total number of blocks
    // int numTotalBlocks = numBlocks * cudaDeviceGetMultiprocessorCount();

    tic();
    printf("Timer started, doing simulation.... \n GPU Usage at this moment: \n");
    int thread;
    if (sample_size>=16){
      thread = 16; // change this according to hardware
    }
    else thread = sample_size;
    // int block = int(ceil(sample_size*1.0/thread)+1);
    int block = (sample_size + thread - 1) / thread;
    // int block = (sample_size + thread - 1) / thread;
    if(gpu_check(15 * sample_size * sizeof(double) + sizeof(param_t)) == 1){
      printf("GPU memory insufficient!\n");
      return 0;
    }
    printf("Sample size: %d\n",sample_size);
    cudaSetDevice(p_param->gpu_index);
    printf("\n   Configuration: \n\n\tblock\t||\tthread\n-------------------------------------
--\n  \t%d\t||\t%d\n\n\n", block,thread);
    // initscr();
    //
printf("[________________________________________________________________________________________
_____________]  0.00 %% \n");

    kernel_DrugSimulation<<<block,thread>>>(d_ic50, d_cvar, d_CONSTANTS, d_STATES,
d_STATES_cache, d_RATES, d_ALGEBRAIC,
                                            d_STATES_RESULT, d_all_states,
                                            time, states, dt, cai_result,
                                            ina, inal,
                                            ical, ito,
                                            ikr, iks,
                                            ik1,
                                            sample_size,
                                            temp_result, cipa_result,
                                            d_p_param
                                            );
                                      //block per grid, threads per block
    // endwin();

    cudaDeviceSynchronize();
```

```
    printf("allocating memory for computation result in the CPU, malloc style \n");
    double *h_states, *h_all_states;
    cipa_t *h_cipa_result;

    h_states = (double *)malloc((num_of_states+1) * sample_size * sizeof(double)); //cache
file
    h_all_states = (double *)malloc( (num_of_states) * sample_size * p_param-
>find_steepest_start * sizeof(double)); //all core
    h_cipa_result = (cipa_t *)malloc(sample_size * sizeof(cipa_t));
    printf("...allocating for all states, all set!\n");

    ////// copy the data back to CPU, and write them into file ////////
    printf("copying the data back to the CPU \n");

    cudaMemcpy(h_cipa_result, cipa_result, sample_size * sizeof(cipa_t),
cudaMemcpyDeviceToHost);
    cudaMemcpy(h_states, d_STATES_RESULT, sample_size * (num_of_states+1) *  sizeof(double),
cudaMemcpyDeviceToHost);
    cudaMemcpy(h_all_states, d_all_states, (num_of_states) * sample_size  * p_param-
>find_steepest_start  *  sizeof(double), cudaMemcpyDeviceToHost);

    FILE *writer;
    int check;
    bool folder_created = false;


    // char sample_str[ENOUGH];
    char conc_str[ENOUGH];
    char filename[500] = "./result/";
    sprintf(conc_str, "%.2f", CONC);
    strcat(filename,conc_str);
    // strcat(filename,"_steepest");
      if (folder_created == false){
        check = mkdir(filename,0777);
        // check if directory is created or not
        if (!check){
          printf("Directory created\n");
          }
        else {
          printf("Unable to create directory, or the folder is already created, relax
mate...\n");
      }
      folder_created = true;
      }

    // strcat(filename,sample_str);
    strcat(filename,".csv");
    printf("writing to %s ... \n", filename);
    writer = fopen(filename,"w");
```

```
    // sample loop
    for (int sample_id = 0; sample_id<sample_size; sample_id++){
      // writer = fopen(filename,"a"); // because we have multiple fwrites
      fprintf(writer,"%d,",sample_id); // write core number at the front
      for (int datapoint = 0; datapoint<num_of_states; datapoint++){
       // if (h_time[ sample_id + (datapoint * sample_size)] == 0.0) {continue;}
        fprintf(writer,"%.5f,", // change this into string, or limit the decimal accuracy, so
we can decrease filesize
        h_states[(sample_id * (num_of_states+1)) + datapoint]
        );
      }
        // fprintf(writer,"%lf,%lf\n", // write last data
        // h_states[(sample_id * num_of_states+1) + num_of_states],
        // h_states[(sample_id * num_of_states+1) + num_of_states+1]
        // );
        fprintf(writer,"%.5f\n", h_states[(sample_id * (num_of_states+1))+num_of_states] );
        // fprintf(writer, "\n");

      // fclose(writer);
    }
     fclose(writer);

    // // FILE *writer;
    // // int check;
    // // bool folder_created = false;

    // printf("writing each core value... \n");
    // // sample loop
    // for (int sample_id = 0; sample_id<sample_size; sample_id++){
    //   // printf("writing sample %d... \n",sample_id);
    //   char sample_str[ENOUGH];
    //   char conc_str[ENOUGH];
    //   char filename[500] = "./result/";
    //   sprintf(sample_str, "%d", sample_id);
    //   sprintf(conc_str, "%.2f", CONC);
    //   strcat(filename,conc_str);
    //   strcat(filename,"/");
    //   // printf("creating %s... \n", filename);
    //   if (folder_created == false){
    //     check = mkdir(filename,0777);
    //     // check if directory is created or not
    //     if (!check){
    //       printf("Directory created\n");
    //       }
    //     else {
    //       printf("Unable to create directory, or the folder is already created, relax
mate...\n");
    //   }
    //   folder_created = true;
```

```
    //   }

    //   strcat(filename,sample_str);
    //   strcat(filename,".csv");

    //   writer = fopen(filename,"w");
    //   for (int pacing = 0; pacing < p_param->find_steepest_start; pacing++){ //pace loop
    //    // if (h_time[ sample_id + (datapoint * sample_size)] == 0.0) {continue;}
    //     for(int datapoint = 0; datapoint < num_of_states; datapoint++){ // each data loop
    //     fprintf(writer,"%lf,",h_all_states[((sample_id * num_of_states)) + ((sample_size) *
pacing) + datapoint]);
    //     // fprintf(writer,"%lf,",h_all_states[((sample_id * num_of_states))+ datapoint]);
    //     }
    //     // fprintf(writer,"%d",p_param->find_steepest_start + pacing);
    //     fprintf(writer,"%d\n",pacing + (p_param->pace_max - p_param-
>find_steepest_start)+1 );

    //   }
    //   fclose(writer);
    // }
    toc();
    return 0;

    }

        }
```

d. test_compile.bat

This file is a result from previous iteration. There was a moment that I wanted to develop the simulator for Windows Operating System outside of Linux. It is finally decided this repository will be containerised using Docker instead of separately develop Windows version of this simulator. This script will more likely to be deleted in the next update.

C. ‘bin’ Folder

The bin folder serves as the central directory for storing various input files, intermediate data, and simulation outputs. This folder is organised into subdirectories that categorise data for ease of access and maintainability.

a. CVAR

This folder will be used to store inter-individual conductance variability file for future development of this research.

b. Control

The control subfolder contains files that are fundamental to running simulations under control conditions (without drug effects). It typically includes:

- IC50_samples.csv: This file provides a baseline reference for the IC50 values of various ionic currents, which are used for comparison in simulations involving drug-induced conditions. IC50 file formatted as:

```
drug_name,conc,ICaL_IC50,ICaL_h,IK1_IC50,IK1_h,IKs_IC50,IKs_h,INa_IC50,INa_h,IN
aL_IC50,INaL_h,Ito_IC50,Ito_h,hERG_IC50,hERG_h
bepridil,0,2704,0.6954,NA,NA,NA,NA,2371,1.984,1947,1.473,NA,NA,139.1,3.199
bepridil,0,2818,0.6409,NA,NA,NA,NA,2734,1.225,1802,1.212,NA,NA,181.4,2.77
bepridil,0,3939,0.718,NA,NA,NA,NA,3064,1.108,1921,1.421,NA,NA,194.8,0.8339
```

- init_state.csv: This file contains the initial states of all variables in the cell model, such as membrane voltage and ion concentrations. These states serve as the starting point for the simulations, ensuring consistent and reproducible results.

By isolating control data in its dedicated subfolder, the simulation framework ensures clarity when comparing control and drug-altered conditions.

c. drug

The drug subfolder stores input files related to simulations involving specific drugs. These files typically include the IC50 values and Hill coefficients for the drugs under study, which define their effects on various

ionic currents. For example, IC50_terfenadine.csv and IC50_verapamil.csv: These files describe the pharmacological properties of terfenadine and verapamil, respectively.

d. result

The result subfolder is used to store simulation outputs, ensuring that data generated during the computational runs is systematically archived. Simulation output classified as two types, the init file, and post-processing folder. Init file is the output from the first phase of the simulation. It contains the simulation's state when gradient of action potential is at its steepest. Named _state_only.csv in a folder named from the IC50 file, this initial state usually shaped like:

0,-89.14848,0.01218,0.00007,12.14017,12.14051,144.11277,144.11273,1.56242,1.55949,0.00008,0.00074,0.83611,0.83590,0.68309,0.83534,0.00015,0.53126,0.28205,0.00093,0.99964,0.56097,0.00047,0.99964,0.61777,0.00000,1.00000,0.92678,1.00000,0.99980,0.99996,1.00000,1.00000,0.00051,0.00087,0.00070,0.00083,0.99790,0.00002,0.00060,0.27721,0.00017,0.00000,0.00000,999.00000

1,-89.14444,0.01211,0.00007,12.12804,12.12838,144.10139,144.10135,1.55818,1.55521,0.00008,0.00074,0.83604,0.83582,0.68296,0.83526,0.00015,0.53107,0.28182,0.00093,0.99963,0.56062,0.00047,0.99963,0.61738,0.00000,1.00000,0.92684,1.00000,0.99980,0.99996,1.00000,1.00000,0.00051,0.00086,0.00070,0.00083,0.99789,0.00002,0.00060,0.27735,0.00017,0.00000,0.00000,999.00000

With the first column as sample ID number, and the last column acts as number of pace recorded. This file will become the input of the second phase. Post-processing folder contains a biomarker file and time-series data per sample. The biomarker file will be used for next researches that requires data analysis. In the research, I validate the result, and manually analyse action potential and ion channels with the time-series files.

This subfolder's structure allows for easy retrieval of outputs for validation and analysis, ensuring that results from different runs are preserved without overwriting.

D. 'cellmodels' Folder

This folder contains mathematical models of the cell. The folder is created to make cell models easier to change and modify. I put the converted C file in here, with an enum file as an addition.

a. Ohara_Rudy_2011.hpp

This file declares all the function in ORd 2011 cell model. This header is required because the main parallelisation will be handled by gpu.cu, hence requiring an object-oriented programming approach. Notice that every function here is a kernel function, since each cell model will be simulated in parallel using each computing thread of the GPU. Below is the format of header file I implement, and this should be changed with any function changes in Ohara_Rudy_2011.cpp file:

```
#ifndef OHARA_RUDY_2011_HPP
#define OHARA_RUDY_2011_HPP

#include "enums/enum_Ohara_Rudy_2011.hpp"
#include <cuda_runtime.h>

        // void initConsts();
        // void initConsts(double type);
        // void initConsts(bool is_dutta);
        __device__ void initConsts(double *CONSTANTS, double *STATES, double type,
double conc, double *ic50, double *cvar, bool is_dutta, bool is_cvar,  int offset);
        __device__ void computeRates(double TIME, double* CONSTANTS, double* RATES,
double* STATES, double* ALGEBRAIC, int offset);
        __device__ void solveAnalytical(double *CONSTANTS, double *STATES, double
*ALGEBRAIC, double *RATES, double dt, int offset);
```

```
                __device__ void solveEuler( double *STATES, double *RATES, double dt, int
offset);
                __device__   double   set_time_step(double   TIME,double   time_point,double
max_time_step,
          double* CONSTANTS,
          double* RATES,
          double* STATES,
          double* ALGEBRAIC,
          int offset);
            __device__ void applyDrugEffect(double *CONSTANTS, double conc, double *ic50, double
epsilon, int offset);
                __device__ void ___applyDutta(double *CONSTANTS, int offset);
                __device__ void ___applyCvar(double *CONSTANTS, double *cvar, int offset);
                // void ___initConsts(double *CONSTANTS, double *STATES, double type, int
offset);

        #endif
```

b. Ohara_Rudy_2011.cpp

This file contains the mathematical cell model and its solver. I added the solver and drug effect simulation function in this script. The rest of the script follows the conversion from CellML. Below is how I implement the modification and the whole script:

```
        /*
           There are a total of 198 entries in the algebraic variable array.
           There are a total of 41 entries in each of the rate and state variable arrays.
           There are a total of 139+2 entries in the constant variable array.
         */

        #include "Ohara_Rudy_2011.hpp"
        #include <cmath>
        #include <cstdlib>
        #include <cstdio>
        #include "../modules/glob_funct.hpp"
        #include <cuda_runtime.h>
        #include <cuda.h>

        __device__ void ___initConsts(double *CONSTANTS, double *STATES, double type, int offset)
        {

        int num_of_constants = 145;
        int num_of_states = 41;
        // printf("%d\n", offset);
        CONSTANTS[(offset * num_of_constants) + celltype] = type;
```

```
CONSTANTS[(offset * num_of_constants) + nao] = 140;
CONSTANTS[(offset * num_of_constants) + cao] = 1.8;
CONSTANTS[(offset * num_of_constants) + ko] = 5.4;
CONSTANTS[(offset * num_of_constants) + R] = 8314;
CONSTANTS[(offset * num_of_constants) + T] = 310;
CONSTANTS[(offset * num_of_constants) + F] = 96485;
CONSTANTS[(offset * num_of_constants) + zna] = 1;
CONSTANTS[(offset * num_of_constants) + zca] = 2;
CONSTANTS[(offset * num_of_constants) + zk] = 1;
CONSTANTS[(offset * num_of_constants) + L] = 0.01;
CONSTANTS[(offset * num_of_constants) + rad] = 0.0011;
CONSTANTS[(offset * num_of_constants) + stim_start] = 10.0;
// bcl edited in the gpu.cu
CONSTANTS[(offset * num_of_constants) + BCL] = 1000.0;
// cvar starts here
CONSTANTS[(offset * num_of_constants) + Jrel_scale] = 1.0;
CONSTANTS[(offset * num_of_constants) + Jup_scale] = 1.0;
CONSTANTS[(offset * num_of_constants) + Jtr_scale] = 1.0;
CONSTANTS[(offset * num_of_constants) + Jleak_scale] = 1.0;
//CONSTANTS[(offset * num_of_constants) + KCaMK_scale] = 1.0;
// cvar ends here
STATES[(offset * num_of_states) + V] = -87;
CONSTANTS[(offset * num_of_constants) + amp] = -80;
CONSTANTS[(offset * num_of_constants) + duration] = 0.5;
CONSTANTS[(offset * num_of_constants) + KmCaMK] = 0.15;
CONSTANTS[(offset * num_of_constants) + aCaMK] = 0.05;
CONSTANTS[(offset * num_of_constants) + bCaMK] = 0.00068;
CONSTANTS[(offset * num_of_constants) + CaMKo] = 0.05;
CONSTANTS[(offset * num_of_constants) + KmCaM] = 0.0015;
STATES[(offset * num_of_states) + CaMKt] = 0;
STATES[(offset * num_of_states) + cass] = 1e-4;
CONSTANTS[(offset * num_of_constants) + cmdnmax_b] = 0.05;
CONSTANTS[(offset * num_of_constants) + kmcmdn] = 0.00238;
CONSTANTS[(offset * num_of_constants) + trpnmax] = 0.07;
CONSTANTS[(offset * num_of_constants) + kmtrpn] = 0.0005;
CONSTANTS[(offset * num_of_constants) + BSRmax] = 0.047;
CONSTANTS[(offset * num_of_constants) + KmBSR] = 0.00087;
CONSTANTS[(offset * num_of_constants) + BSLmax] = 1.124;
CONSTANTS[(offset * num_of_constants) + KmBSL] = 0.0087;
CONSTANTS[(offset * num_of_constants) + csqnmax] = 10;
CONSTANTS[(offset * num_of_constants) + kmcsqn] = 0.8;
STATES[(offset * num_of_states) + nai] = 7;
STATES[(offset * num_of_states) + nass] = 7;
STATES[(offset * num_of_states) + ki] = 145;
STATES[(offset * num_of_states) + kss] = 145;
STATES[(offset * num_of_states) + cansr] = 1.2;
STATES[(offset * num_of_states) + cajsr] = 1.2;
STATES[(offset * num_of_states) + cai] = 1e-4;
CONSTANTS[(offset * num_of_constants) + cm] = 1;
```

```
CONSTANTS[(offset * num_of_constants) + PKNa] = 0.01833;
CONSTANTS[(offset * num_of_constants) + mssV1] = 39.57;
CONSTANTS[(offset * num_of_constants) + mssV2] = 9.871;
CONSTANTS[(offset * num_of_constants) + mtV1] = 11.64;
CONSTANTS[(offset * num_of_constants) + mtV2] = 34.77;
CONSTANTS[(offset * num_of_constants) + mtD1] = 6.765;
CONSTANTS[(offset * num_of_constants) + mtD2] = 8.552;
CONSTANTS[(offset * num_of_constants) + mtV3] = 77.42;
CONSTANTS[(offset * num_of_constants) + mtV4] = 5.955;
STATES[(offset * num_of_states) + m] = 0;
CONSTANTS[(offset * num_of_constants) + hssV1] = 82.9;
CONSTANTS[(offset * num_of_constants) + hssV2] = 6.086;
CONSTANTS[(offset * num_of_constants) + Ahf] = 0.99;
STATES[(offset * num_of_states) + hf] = 1;
STATES[(offset * num_of_states) + hs] = 1;
CONSTANTS[(offset * num_of_constants) + GNa] = 75;
STATES[(offset * num_of_states) + j] = 1;
STATES[(offset * num_of_states) + hsp] = 1;
STATES[(offset * num_of_states) + jp] = 1;
STATES[(offset * num_of_states) + mL] = 0;
CONSTANTS[(offset * num_of_constants) + thL] = 200;
STATES[(offset * num_of_states) + hL] = 1;
STATES[(offset * num_of_states) + hLp] = 1;
CONSTANTS[(offset * num_of_constants) + GNaL_b] = 0.0075;
CONSTANTS[(offset * num_of_constants) + Gto_b] = 0.02;
STATES[(offset * num_of_states) + a] = 0;
STATES[(offset * num_of_states) + iF] = 1;
STATES[(offset * num_of_states) + iS] = 1;
STATES[(offset * num_of_states) + ap] = 0;
STATES[(offset * num_of_states) + iFp] = 1;
STATES[(offset * num_of_states) + iSp] = 1;
CONSTANTS[(offset * num_of_constants) + Kmn] = 0.002;
CONSTANTS[(offset * num_of_constants) + k2n] = 1000;
CONSTANTS[(offset * num_of_constants) + PCa_b] = 0.0001;
STATES[(offset * num_of_states) + d] = 0;
STATES[(offset * num_of_states) + ff] = 1;
STATES[(offset * num_of_states) + fs] = 1;
STATES[(offset * num_of_states) + fcaf] = 1;
STATES[(offset * num_of_states) + fcas] = 1;
STATES[(offset * num_of_states) + jca] = 1;
STATES[(offset * num_of_states) + ffp] = 1;
STATES[(offset * num_of_states) + fcafp] = 1;
STATES[(offset * num_of_states) + nca] = 0;
CONSTANTS[(offset * num_of_constants) + GKr_b] = 0.046;
STATES[(offset * num_of_states) + xrf] = 0;
STATES[(offset * num_of_states) + xrs] = 0;
CONSTANTS[(offset * num_of_constants) + GKs_b] = 0.0034;
STATES[(offset * num_of_states) + xs1] = 0;
STATES[(offset * num_of_states) + xs2] = 0;
```

```
        CONSTANTS[(offset * num_of_constants) + GK1_b] = 0.1908;
        STATES[(offset * num_of_states) + xk1] = 1;
        CONSTANTS[(offset * num_of_constants) + kna1] = 15;
        CONSTANTS[(offset * num_of_constants) + kna2] = 5;
        CONSTANTS[(offset * num_of_constants) + kna3] = 88.12;
        CONSTANTS[(offset * num_of_constants) + kasymm] = 12.5;
        CONSTANTS[(offset * num_of_constants) + wna] = 6e4;
        CONSTANTS[(offset * num_of_constants) + wca] = 6e4;
        CONSTANTS[(offset * num_of_constants) + wnaca] = 5e3;
        CONSTANTS[(offset * num_of_constants) + kcaon] = 1.5e6;
        CONSTANTS[(offset * num_of_constants) + kcaoff] = 5e3;
        CONSTANTS[(offset * num_of_constants) + qna] = 0.5224;
        CONSTANTS[(offset * num_of_constants) + qca] = 0.167;
        CONSTANTS[(offset * num_of_constants) + KmCaAct] = 150e-6;
        CONSTANTS[(offset * num_of_constants) + Gncx_b] = 0.0008;
        CONSTANTS[(offset * num_of_constants) + k1p] = 949.5;
        CONSTANTS[(offset * num_of_constants) + k1m] = 182.4;
        CONSTANTS[(offset * num_of_constants) + k2p] = 687.2;
        CONSTANTS[(offset * num_of_constants) + k2m] = 39.4;
        CONSTANTS[(offset * num_of_constants) + k3p] = 1899;
        CONSTANTS[(offset * num_of_constants) + k3m] = 79300;
        CONSTANTS[(offset * num_of_constants) + k4p] = 639;
        CONSTANTS[(offset * num_of_constants) + k4m] = 40;
        CONSTANTS[(offset * num_of_constants) + Knai0] = 9.073;
        CONSTANTS[(offset * num_of_constants) + Knao0] = 27.78;
        CONSTANTS[(offset * num_of_constants) + delta] = -0.155;
        CONSTANTS[(offset * num_of_constants) + Kki] = 0.5;
        CONSTANTS[(offset * num_of_constants) + Kko] = 0.3582;
        CONSTANTS[(offset * num_of_constants) + MgADP] = 0.05;
        CONSTANTS[(offset * num_of_constants) + MgATP] = 9.8;
        CONSTANTS[(offset * num_of_constants) + Kmgatp] = 1.698e-7;
        CONSTANTS[(offset * num_of_constants) + H] = 1e-7;
        CONSTANTS[(offset * num_of_constants) + eP] = 4.2;
        CONSTANTS[(offset * num_of_constants) + Khp] = 1.698e-7;
        CONSTANTS[(offset * num_of_constants) + Knap] = 224;
        CONSTANTS[(offset * num_of_constants) + Kxkur] = 292;
        CONSTANTS[(offset * num_of_constants) + Pnak_b] = 30;
        CONSTANTS[(offset * num_of_constants) + GKb_b] = 0.003;
        CONSTANTS[(offset * num_of_constants) + PNab] = 3.75e-10;
        CONSTANTS[(offset * num_of_constants) + PCab] = 2.5e-8;
        CONSTANTS[(offset * num_of_constants) + GpCa] = 0.0005;
        CONSTANTS[(offset * num_of_constants) + KmCap] = 0.0005;
        CONSTANTS[(offset * num_of_constants) + bt] = 4.75;
        STATES[(offset * num_of_states) + Jrelnp] = 0;
        STATES[(offset * num_of_states) + Jrelp] = 0;
        CONSTANTS[(offset * num_of_constants) + cmdnmax] = (CONSTANTS[(offset * num_of_constants)
+  celltype]==1.00000  ?   CONSTANTS[(offset  *  num_of_constants)  +  cmdnmax_b]*1.30000  :
CONSTANTS[(offset * num_of_constants) + cmdnmax_b]);
```

```
        CONSTANTS[(offset * num_of_constants) + Ahs] = 1.00000 - CONSTANTS[(offset *
num_of_constants) + Ahf];
        CONSTANTS[(offset * num_of_constants) + thLp] = 3.00000 * CONSTANTS[(offset *
num_of_constants) + thL];
        CONSTANTS[(offset * num_of_constants) + GNaL] = (CONSTANTS[(offset * num_of_constants)
+ celltype]==1.00000 ? CONSTANTS[(offset * num_of_constants) + GNaL_b]*0.600000 :
CONSTANTS[(offset * num_of_constants) + GNaL_b]);
        CONSTANTS[(offset * num_of_constants) + Gto] = (CONSTANTS[(offset * num_of_constants)
+ celltype]==1.00000 ? CONSTANTS[(offset * num_of_constants) + Gto_b]*4.00000 :
CONSTANTS[(offset * num_of_constants) + celltype]==2.00000 ? CONSTANTS[(offset *
num_of_constants) + Gto_b]*4.00000 : CONSTANTS[(offset * num_of_constants) + Gto_b]);
        CONSTANTS[(offset * num_of_constants) + Aff] = 0.600000;
        CONSTANTS[(offset * num_of_constants) + PCa] = (CONSTANTS[(offset * num_of_constants)
+ celltype]==1.00000 ? CONSTANTS[(offset * num_of_constants) + PCa_b]*1.20000 :
CONSTANTS[(offset * num_of_constants) + celltype]==2.00000 ? CONSTANTS[(offset *
num_of_constants) + PCa_b]*2.50000 : CONSTANTS[(offset * num_of_constants) + PCa_b]);
        CONSTANTS[(offset * num_of_constants) + tjca] = 75.0000;
        CONSTANTS[(offset * num_of_constants) + GKr] = (CONSTANTS[(offset * num_of_constants)
+ celltype]==1.00000 ? CONSTANTS[(offset * num_of_constants) + GKr_b]*1.30000 :
CONSTANTS[(offset * num_of_constants) + celltype]==2.00000 ? CONSTANTS[(offset *
num_of_constants) + GKr_b]*0.800000 : CONSTANTS[(offset * num_of_constants) + GKr_b]);
        CONSTANTS[(offset * num_of_constants) + GKs] = (CONSTANTS[(offset * num_of_constants)
+ celltype]==1.00000 ? CONSTANTS[(offset * num_of_constants) + GKs_b]*1.40000 :
CONSTANTS[(offset * num_of_constants) + GKs_b]);
        CONSTANTS[(offset * num_of_constants) + GK1] = (CONSTANTS[(offset * num_of_constants)
+ celltype]==1.00000 ? CONSTANTS[(offset * num_of_constants) + GK1_b]*1.20000 :
CONSTANTS[(offset * num_of_constants) + celltype]==2.00000 ? CONSTANTS[(offset *
num_of_constants) + GK1_b]*1.30000 : CONSTANTS[(offset * num_of_constants) + GK1_b]);
        CONSTANTS[(offset * num_of_constants) + vcell] = 1000.00*3.14000*CONSTANTS[(offset *
num_of_constants) + rad]*CONSTANTS[(offset * num_of_constants) + rad]*CONSTANTS[(offset *
num_of_constants) + L];
        CONSTANTS[(offset * num_of_constants) + GKb] = (CONSTANTS[(offset * num_of_constants)
+ celltype]==1.00000 ? CONSTANTS[(offset * num_of_constants) + GKb_b]*0.600000 :
CONSTANTS[(offset * num_of_constants) + GKb_b]);
        CONSTANTS[(offset * num_of_constants) + a_rel] = 0.500000*CONSTANTS[(offset *
num_of_constants) + bt];
        CONSTANTS[(offset * num_of_constants) + btp] = 1.25000*CONSTANTS[(offset *
num_of_constants) + bt];
        CONSTANTS[(offset * num_of_constants) + upScale] = (CONSTANTS[(offset * num_of_constants)
+ celltype]==1.0000 ? 1.30000 : 1.00000);
        CONSTANTS[(offset * num_of_constants) + Afs] = 1.00000 - CONSTANTS[(offset *
num_of_constants) + Aff];
        CONSTANTS[(offset * num_of_constants) + PCap] = 1.10000*CONSTANTS[(offset *
num_of_constants) + PCa];
        CONSTANTS[(offset * num_of_constants) + PCaNa] = 0.00125000*CONSTANTS[(offset *
num_of_constants) + PCa];
        CONSTANTS[(offset * num_of_constants) + PCaK] = 0.000357400*CONSTANTS[(offset *
num_of_constants) + PCa];
```

```
        CONSTANTS[(offset * num_of_constants) + Ageo] =  2.00000*3.14000*CONSTANTS[(offset *
num_of_constants)     +     rad]*CONSTANTS[(offset     *     num_of_constants)     +     rad]+
2.00000*3.14000*CONSTANTS[(offset    *    num_of_constants)    +    rad]*CONSTANTS[(offset    *
num_of_constants) + L];
        CONSTANTS[(offset  *  num_of_constants)  +  a_relp]  =   0.500000*CONSTANTS[(offset  *
num_of_constants) + btp];
        CONSTANTS[(offset  *  num_of_constants)  +  PCaNap]  =   0.00125000*CONSTANTS[(offset  *
num_of_constants) + PCap];
        CONSTANTS[(offset  *  num_of_constants)  +  PCaKp]  =   0.000357400*CONSTANTS[(offset  *
num_of_constants) + PCap];
        CONSTANTS[(offset  *  num_of_constants)  +  Acap]  =    2.00000*CONSTANTS[(offset  *
num_of_constants) + Ageo];
        CONSTANTS[(offset  *  num_of_constants)  +  vmyo]  =    0.680000*CONSTANTS[(offset  *
num_of_constants) + vcell];
        CONSTANTS[(offset  *  num_of_constants)  +  vnsr]  =    0.0552000*CONSTANTS[(offset  *
num_of_constants) + vcell];
        CONSTANTS[(offset  *  num_of_constants)  +  vjsr]  =    0.00480000*CONSTANTS[(offset  *
num_of_constants) + vcell];
        CONSTANTS[(offset  *  num_of_constants)  +  vss]  =    0.0200000*CONSTANTS[(offset  *
num_of_constants) + vcell];
        CONSTANTS[(offset * num_of_constants) + h10_i] = CONSTANTS[(offset * num_of_constants)
+  kasymm]+1.00000+  (CONSTANTS[(offset  *  num_of_constants)  +  nao]/CONSTANTS[(offset  *
num_of_constants)     +     kna1])*(1.00000+CONSTANTS[(offset     *     num_of_constants)     +
nao]/CONSTANTS[(offset * num_of_constants) + kna2]);
        CONSTANTS[(offset * num_of_constants) + h11_i] = ( CONSTANTS[(offset * num_of_constants)
+ nao]*CONSTANTS[(offset * num_of_constants) + nao])/( CONSTANTS[(offset * num_of_constants) +
h10_i]*CONSTANTS[(offset  *  num_of_constants)  +  kna1]*CONSTANTS[(offset  *  num_of_constants)  +
kna2]);
        CONSTANTS[(offset  *  num_of_constants)  +  h12_i]  =  1.00000/CONSTANTS[(offset  *
num_of_constants) + h10_i];
        CONSTANTS[(offset * num_of_constants) + k1_i] =  CONSTANTS[(offset * num_of_constants)
+ h12_i]*CONSTANTS[(offset * num_of_constants) + cao]*CONSTANTS[(offset * num_of_constants) +
kcaon];
        CONSTANTS[(offset * num_of_constants) + k2_i] = CONSTANTS[(offset * num_of_constants)
+ kcaoff];
        CONSTANTS[(offset * num_of_constants) + k5_i] = CONSTANTS[(offset * num_of_constants)
+ kcaoff];
        CONSTANTS[(offset * num_of_constants) + Gncx] = (CONSTANTS[(offset * num_of_constants)
+  celltype]==1.00000  ?    CONSTANTS[(offset  *  num_of_constants)  +  Gncx_b]*1.10000  :
CONSTANTS[(offset  *  num_of_constants)  +  celltype]==2.00000  ?    CONSTANTS[(offset  *
num_of_constants) + Gncx_b]*1.40000 : CONSTANTS[(offset * num_of_constants) + Gncx_b]);
        CONSTANTS[(offset * num_of_constants) + h10_ss] = CONSTANTS[(offset * num_of_constants)
+  kasymm]+1.00000+  (CONSTANTS[(offset  *  num_of_constants)  +  nao]/CONSTANTS[(offset  *
num_of_constants)     +     kna1])*(1.00000+CONSTANTS[(offset     *     num_of_constants)     +
nao]/CONSTANTS[(offset * num_of_constants) + kna2]);
        CONSTANTS[(offset * num_of_constants) + h11_ss] = ( CONSTANTS[(offset * num_of_constants)
+ nao]*CONSTANTS[(offset * num_of_constants) + nao])/( CONSTANTS[(offset * num_of_constants) +
h10_ss]*CONSTANTS[(offset * num_of_constants) + kna1]*CONSTANTS[(offset * num_of_constants) +
kna2]);
```

```
        CONSTANTS[(offset * num_of_constants) + h12_ss] = 1.00000/CONSTANTS[(offset *
num_of_constants) + h10_ss];
        CONSTANTS[(offset * num_of_constants) + k1_ss] = CONSTANTS[(offset * num_of_constants)
+ h12_ss]*CONSTANTS[(offset * num_of_constants) + cao]*CONSTANTS[(offset * num_of_constants) +
kcaon];
        CONSTANTS[(offset * num_of_constants) + k2_ss] = CONSTANTS[(offset * num_of_constants)
+ kcaoff];
        CONSTANTS[(offset * num_of_constants) + k5_ss] = CONSTANTS[(offset * num_of_constants)
+ kcaoff];
        CONSTANTS[(offset * num_of_constants) + b1] = CONSTANTS[(offset * num_of_constants) +
k1m]*CONSTANTS[(offset * num_of_constants) + MgADP];
        CONSTANTS[(offset * num_of_constants) + a2] = CONSTANTS[(offset * num_of_constants) +
k2p];
        CONSTANTS[(offset * num_of_constants) + a4] = (( CONSTANTS[(offset * num_of_constants)
+ k4p]*CONSTANTS[(offset * num_of_constants) + MgATP])/CONSTANTS[(offset * num_of_constants) +
Kmgatp])/(1.00000+CONSTANTS[(offset * num_of_constants) + MgATP]/CONSTANTS[(offset *
num_of_constants) + Kmgatp]);
        CONSTANTS[(offset * num_of_constants) + Pnak] = (CONSTANTS[(offset * num_of_constants)
+ celltype]==1.00000 ? CONSTANTS[(offset * num_of_constants) + Pnak_b]*0.900000 :
CONSTANTS[(offset * num_of_constants) + celltype]==2.00000 ? CONSTANTS[(offset *
num_of_constants) + Pnak_b]*0.700000 : CONSTANTS[(offset * num_of_constants) + Pnak_b]);
        }

        __device__ void ___applyDutta(double *CONSTANTS, int offset)
        {
        int num_of_constants = 145;
        //sisanya ganti jadi G (GKs for example)
        CONSTANTS[GKs + (offset * num_of_constants)] *= 1.870;
        CONSTANTS[GKr + (offset * num_of_constants)] *= 1.013;
        CONSTANTS[GK1 + (offset * num_of_constants)] *= 1.698;
        CONSTANTS[PCa + (offset * num_of_constants)] *= 1.007; //pca
        CONSTANTS[GNaL + (offset * num_of_constants)] *= 2.661;
        }

        /*===============*/
        /* Added by ALI */
        /*===============*/
        __device__ void ___applyCvar(double *CONSTANTS, double *cvar, int offset)
        {
          int num_of_constants = 145;

          CONSTANTS[(offset * num_of_constants) +GNa] *= cvar[0 + (offset*18)];
        // GNa
          CONSTANTS[(offset * num_of_constants) +GNaL] *= cvar[1 + (offset*18)];
        // GNaL
          CONSTANTS[(offset * num_of_constants) +Gto] *= cvar[2 + (offset*18)];
        // Gto
          CONSTANTS[(offset * num_of_constants) +GKr] *= cvar[3 + (offset*18)];
        // GKr
```

```
  CONSTANTS[(offset * num_of_constants) +GKs] *= cvar[4 + (offset*18)];
// GKs
  CONSTANTS[(offset * num_of_constants) +GK1] *= cvar[5 + (offset*18)];
// GK1
  CONSTANTS[(offset * num_of_constants) +Gncx] *= cvar[6 + (offset*18)];
// GNaCa
  CONSTANTS[(offset * num_of_constants) +GKb] *= cvar[7 + (offset*18)];
// GKb
  CONSTANTS[(offset * num_of_constants) +PCa] *= cvar[8 + (offset*18)];
// PCa
  CONSTANTS[(offset * num_of_constants) +Pnak] *= cvar[9 + (offset*18)];
// INaK
  CONSTANTS[(offset * num_of_constants) +PNab] *= cvar[10 + (offset*18)];
// PNab
  CONSTANTS[(offset * num_of_constants) +PCab] *= cvar[11 + (offset*18)];
// PCab
  CONSTANTS[(offset * num_of_constants) +GpCa] *= cvar[12 + (offset*18)];
// GpCa
  CONSTANTS[(offset * num_of_constants) +KmCaMK] *= cvar[17 + (offset*18)];  // KCaMK

  // Additional constants
  CONSTANTS[(offset  *  num_of_constants)  +Jrel_scale]  *=  cvar[13  +  (offset*18)];
// SERCA_Total (release)
  CONSTANTS[(offset  *  num_of_constants)  +Jup_scale]  *=  cvar[14  +  (offset*18)];
// RyR_Total (uptake)
  CONSTANTS[(offset  *  num_of_constants)  +Jtr_scale]  *=  cvar[15  +  (offset*18)];
// Trans_Total (NSR to JSR translocation)
  CONSTANTS[(offset  *  num_of_constants)  +Jleak_scale]  *=  cvar[16  +  (offset*18)];
// Leak_Total (Ca leak from NSR)
  // CONSTANTS[(offset * num_of_constants) +KCaMK_scale] *= cvar[17 + (offset*18)];
// KCaMK
}

__device__ void applyDrugEffect(double *CONSTANTS, double conc, double *ic50, double
epsilon, int offset)
{
int num_of_constants = 145;

CONSTANTS[PCa_b+(offset    *    num_of_constants)]    =    CONSTANTS[PCa_b+(offset    *
num_of_constants)] * ( (ic50[0 + (offset*14)] > epsilon && ic50[1+ (offset*14)] > epsilon) ?
1./(1.+pow(conc/ic50[0+ (offset*14)],ic50[1+ (offset*14)])) : 1.);
CONSTANTS[GK1_b+(offset    *    num_of_constants)]    =    CONSTANTS[GK1_b+(offset    *
num_of_constants)] * ((ic50[2 + (offset*14)] > epsilon && ic50[3+ (offset*14)] > epsilon) ?
1./(1.+pow(conc/ic50[2+ (offset*14)],ic50[3+ (offset*14)])) : 1.);
CONSTANTS[GKs_b+(offset    *    num_of_constants)]    =    CONSTANTS[GKs_b+(offset    *
num_of_constants)] * ((ic50[4 + (offset*14)] > epsilon && ic50[5+ (offset*14)] > epsilon) ?
1./(1.+pow(conc/ic50[4+ (offset*14)],ic50[5+ (offset*14)])) : 1.);
```

```
        CONSTANTS[GNa+(offset * num_of_constants)] = CONSTANTS[GNa+(offset * num_of_constants)]
* ((ic50[6 + (offset*14)] > epsilon && ic50[7+ (offset*14)] > epsilon) ? 1./(1.+pow(conc/ic50[6+
(offset*14)],ic50[7+ (offset*14)])) : 1.);
        CONSTANTS[GNaL_b+(offset    *    num_of_constants)]    =    CONSTANTS[GNaL_b+(offset    *
num_of_constants)] * ((ic50[8+ (offset*14)] > epsilon && ic50[9+ (offset*14)] > epsilon) ?
1./(1.+pow(conc/ic50[8+ (offset*14)],ic50[9+ (offset*14)])) : 1.);
        CONSTANTS[Gto_b+(offset    *    num_of_constants)]    =    CONSTANTS[Gto_b+(offset    *
num_of_constants)] * ((ic50[10 + (offset*14)] > epsilon && ic50[11+ (offset*14)] > epsilon) ?
1./(1.+pow(conc/ic50[10+ (offset*14)],ic50[11+ (offset*14)])) : 1.);
        CONSTANTS[GKr_b+(offset    *    num_of_constants)]    =    CONSTANTS[GKr_b+(offset    *
num_of_constants)] * ((ic50[12+ (offset*14)] > epsilon && ic50[13+ (offset*14)] > epsilon) ?
1./(1.+pow(conc/ic50[12+ (offset*14)],ic50[13+ (offset*14)])) : 1.);

        }

        // void initConsts(int offset)
        // {
        //      ___initConsts(0.,offset);
        // }

        // void initConsts(double type)
        // {
        //      ___initConsts(type, offset);
        // }

        __device__ void initConsts(double *CONSTANTS, double *STATES, double type, double conc,
double *ic50, double *cvar, bool is_dutta, bool is_cvar,  int offset)
        {
          // int num_of_constants = 145;
          //  printf("ic50:%d  %lf,  %lf,  %lf\n",offset,ic50[0  +  (offset*14)],ic50[1  +
(offset*14)],ic50[2 + (offset*14)]);

                ___initConsts(CONSTANTS, STATES, type, offset); // initconst kan minta
                // // mpi_printf(0,"Celltype: %lf\n", CONSTANTS[celltype]);
                // #ifndef COMPONENT_PATCH // for patch clamp component based research
                //  //  mpi_printf(0,"Control  %lf  %lf  %lf  %lf  %lf\n",  CONSTANTS[PCa],
CONSTANTS[GK1], CONSTANTS[GKs], CONSTANTS[GNaL], CONSTANTS[GKr]);
                // #endif
                if(is_dutta == true){
                        ___applyDutta(CONSTANTS, offset);
                }

          if(is_cvar == true){
                        ___applyCvar(CONSTANTS, cvar, offset);
                }
                // #ifndef COMPONENT_PATCH
                //  //  mpi_printf(0,"After  Dutta  %lf  %lf  %lf  %lf  %lf\n",  CONSTANTS[PCa],
CONSTANTS[GK1], CONSTANTS[GKs], CONSTANTS[GNaL], CONSTANTS[GKr]);
                // #endif
```

```
                // ___applyDrugEffect(CONSTANTS, conc, ic50, 10E-14, offset);
                // #ifndef COMPONENT_PATCH
                //  //  mpi_printf(0,"After  drug  %lf  %lf  %lf  %lf  %lf\n",  CONSTANTS[PCa],
CONSTANTS[GK1], CONSTANTS[GKs], CONSTANTS[GNaL], CONSTANTS[GKr]);
                // #endif


        }

        __device__ void computeRates( double TIME, double *CONSTANTS, double *RATES, double
*STATES, double *ALGEBRAIC, int offset )
        {
        int num_of_constants = 145; //done
        int num_of_states = 41; //done
        int num_of_algebraic = 199; //done
        int num_of_rates = 41; //done

        //new part
        CONSTANTS[(offset * num_of_constants) + cmdnmax] = (CONSTANTS[(offset * num_of_constants)
+  celltype]==1.00000  ?    CONSTANTS[(offset  *  num_of_constants)  +  cmdnmax_b]*1.30000  :
CONSTANTS[(offset * num_of_constants) + cmdnmax_b]);
        CONSTANTS[(offset * num_of_constants) + GNaL] = (CONSTANTS[(offset * num_of_constants)
+  celltype]==1.00000  ?    CONSTANTS[(offset  *  num_of_constants)  +  GNaL_b]*0.600000  :
CONSTANTS[(offset * num_of_constants) + GNaL_b]);
        CONSTANTS[(offset * num_of_constants) + Gto] = (CONSTANTS[(offset * num_of_constants)
+  celltype]==1.00000  ?    CONSTANTS[(offset  *  num_of_constants)  +  Gto_b]*4.00000  :
CONSTANTS[(offset  *  num_of_constants)  +  celltype]==2.00000  ?    CONSTANTS[(offset  *
num_of_constants) + Gto_b]*4.00000 : CONSTANTS[(offset * num_of_constants) + Gto_b]);
        CONSTANTS[(offset * num_of_constants) + PCa] = (CONSTANTS[(offset * num_of_constants)
+  celltype]==1.00000  ?    CONSTANTS[(offset  *  num_of_constants)  +  PCa_b]*1.20000  :
CONSTANTS[(offset  *  num_of_constants)  +  celltype]==2.00000  ?    CONSTANTS[(offset  *
num_of_constants) + PCa_b]*2.50000 : CONSTANTS[(offset * num_of_constants) + PCa_b]);
        CONSTANTS[(offset * num_of_constants) + GKr] = (CONSTANTS[(offset * num_of_constants)
+  celltype]==1.00000  ?    CONSTANTS[(offset  *  num_of_constants)  +  GKr_b]*1.30000  :
CONSTANTS[(offset  *  num_of_constants)  +  celltype]==2.00000  ?    CONSTANTS[(offset  *
num_of_constants) + GKr_b]*0.800000 : CONSTANTS[(offset * num_of_constants) + GKr_b]);
        CONSTANTS[(offset * num_of_constants) + GKs] = (CONSTANTS[(offset * num_of_constants)
+  celltype]==1.00000  ?    CONSTANTS[(offset  *  num_of_constants)  +  GKs_b]*1.40000  :
CONSTANTS[(offset * num_of_constants) + GKs_b]);
        CONSTANTS[(offset * num_of_constants) + GK1] = (CONSTANTS[(offset * num_of_constants)
+  celltype]==1.00000  ?    CONSTANTS[(offset  *  num_of_constants)  +  GK1_b]*1.20000  :
CONSTANTS[(offset  *  num_of_constants)  +  celltype]==2.00000  ?    CONSTANTS[(offset  *
num_of_constants) + GK1_b]*1.30000 : CONSTANTS[(offset * num_of_constants) + GK1_b]);
        CONSTANTS[(offset * num_of_constants) + GKb] = (CONSTANTS[(offset * num_of_constants)
+  celltype]==1.00000  ?    CONSTANTS[(offset  *  num_of_constants)  +  GKb_b]*0.600000  :
CONSTANTS[(offset * num_of_constants) + GKb_b]);
        CONSTANTS[(offset * num_of_constants) + upScale] = (CONSTANTS[(offset * num_of_constants)
+ celltype]==1.0000 ? 1.30000 : 1.00000);
```

```
        CONSTANTS[(offset * num_of_constants) + Gncx] = (CONSTANTS[(offset * num_of_constants)
+ celltype]==1.00000 ? CONSTANTS[(offset * num_of_constants) + Gncx_b]*1.10000 :
CONSTANTS[(offset * num_of_constants) + celltype]==2.00000 ? CONSTANTS[(offset *
num_of_constants) + Gncx_b]*1.40000 : CONSTANTS[(offset * num_of_constants) + Gncx_b]);
        CONSTANTS[(offset * num_of_constants) + Pnak] = (CONSTANTS[(offset * num_of_constants)
+ celltype]==1.00000 ? CONSTANTS[(offset * num_of_constants) + Pnak_b]*0.900000 :
CONSTANTS[(offset * num_of_constants) + celltype]==2.00000 ? CONSTANTS[(offset *
num_of_constants) + Pnak_b]*0.700000 : CONSTANTS[(offset * num_of_constants) + Pnak_b]);
        // new part ends
        ALGEBRAIC[(offset * num_of_algebraic) +Istim] = (TIME>=CONSTANTS[(offset *
num_of_constants) + stim_start] && (TIME - CONSTANTS[(offset * num_of_constants) + stim_start])
- floor((TIME - CONSTANTS[(offset * num_of_constants) + stim_start])/CONSTANTS[(offset *
num_of_constants) + BCL])*CONSTANTS[(offset * num_of_constants) + BCL]<=CONSTANTS[(offset *
num_of_constants) + duration] ? CONSTANTS[(offset * num_of_constants) + amp] : 0.000000);
        // in libcml there is ifdef TISSUE, ask further

        ALGEBRAIC[(offset * num_of_algebraic) +hLss] = 1.00000/(1.00000+exp((STATES[(offset *
num_of_states) + V]+87.6100)/7.48800));
        ALGEBRAIC[(offset * num_of_algebraic) +hLssp] = 1.00000/(1.00000+exp((STATES[(offset *
num_of_states) + V]+93.8100)/7.48800));
        ALGEBRAIC[(offset * num_of_algebraic) +mss] = 1.00000/(1.00000+exp(- (STATES[(offset *
num_of_states) + V]+CONSTANTS[(offset * num_of_constants) + mssV1])/CONSTANTS[(offset *
num_of_constants) + mssV2]));
        ALGEBRAIC[(offset * num_of_algebraic) +tm] = 1.00000/( CONSTANTS[(offset *
num_of_constants) + mtD1]*exp((STATES[(offset * num_of_states) + V]+CONSTANTS[(offset *
num_of_constants) + mtV1])/CONSTANTS[(offset * num_of_constants) + mtV2])+ CONSTANTS[(offset *
num_of_constants) + mtD2]*exp(- (STATES[(offset * num_of_states) + V]+CONSTANTS[(offset *
num_of_constants) + mtV3])/CONSTANTS[(offset * num_of_constants) + mtV4]));
        ALGEBRAIC[(offset * num_of_algebraic) +hss] = 1.00000/(1.00000+exp((STATES[(offset *
num_of_states) + V]+CONSTANTS[(offset * num_of_constants) + hssV1])/CONSTANTS[(offset *
num_of_constants) + hssV2]));
        ALGEBRAIC[(offset * num_of_algebraic) +thf] = 1.00000/( 1.43200e-05*exp(-
(STATES[(offset * num_of_states) + V]+1.19600)/6.28500)+ 6.14900*exp((STATES[(offset *
num_of_states) + V]+0.509600)/20.2700));
        ALGEBRAIC[(offset * num_of_algebraic) +ths] = 1.00000/( 0.00979400*exp(-
(STATES[(offset * num_of_states) + V]+17.9500)/28.0500)+ 0.334300*exp((STATES[(offset *
num_of_states) + V]+5.73000)/56.6600));
        ALGEBRAIC[(offset * num_of_algebraic) +ass] = 1.00000/(1.00000+exp(- (STATES[(offset *
num_of_states) + V] - 14.3400)/14.8200));
        ALGEBRAIC[(offset * num_of_algebraic) +ta] = 1.05150/(1.00000/( 1.20890*(1.00000+exp(-
(STATES[(offset * num_of_states) + V] - 18.4099)/29.3814)))+3.50000/(1.00000+exp((STATES[(offset
* num_of_states) + V]+100.000)/29.3814)));
        ALGEBRAIC[(offset * num_of_algebraic) +dss] = 1.00000/(1.00000+exp(- (STATES[(offset *
num_of_states) + V]+3.94000)/4.23000));
        ALGEBRAIC[(offset * num_of_algebraic) +td] = 0.600000+1.00000/(exp( -
0.0500000*(STATES[(offset * num_of_states) + V]+6.00000))+exp( 0.0900000*(STATES[(offset *
num_of_states) + V]+14.0000)));
        ALGEBRAIC[(offset * num_of_algebraic) + fss] = 1.00000/(1.00000+exp((STATES[(offset *
num_of_states) + V]+19.5800)/3.69600));
```

```
        ALGEBRAIC[(offset * num_of_algebraic) + tff] = 7.00000+1.00000/( 0.00450000*exp(-
(STATES[(offset * num_of_states) + V]+20.0000)/10.0000)+ 0.00450000*exp((STATES[(offset *
num_of_states) + V]+20.0000)/10.0000));
        ALGEBRAIC[(offset * num_of_algebraic) + tfs] = 1000.00+1.00000/( 3.50000e-05*exp(-
(STATES[(offset * num_of_states) + V]+5.00000)/4.00000)+ 3.50000e-05*exp((STATES[(offset *
num_of_states) + V]+5.00000)/6.00000));
        ALGEBRAIC[(offset * num_of_algebraic) + fcass] = ALGEBRAIC[(offset * num_of_algebraic)
+ fss];
        ALGEBRAIC[(offset * num_of_algebraic) + km2n] =  STATES[(offset * num_of_states) +
jca]*1.00000;
        ALGEBRAIC[(offset * num_of_algebraic) + anca] = 1.00000/(CONSTANTS[(offset *
num_of_constants) + k2n]/ALGEBRAIC[(offset * num_of_algebraic) +
km2n]+pow(1.00000+CONSTANTS[(offset * num_of_constants) + Kmn]/STATES[(offset * num_of_states)
+ cass], 4.00000));
        ALGEBRAIC[(offset * num_of_algebraic) + xrss] = 1.00000/(1.00000+exp(- (STATES[(offset
* num_of_states) + V]+8.33700)/6.78900));
        ALGEBRAIC[(offset * num_of_algebraic) + txrf] =
12.9800+1.00000/( 0.365200*exp((STATES[(offset * num_of_states) + V] - 31.6600)/3.86900)+
4.12300e-05*exp(- (STATES[(offset * num_of_states) + V] - 47.7800)/20.3800));
        ALGEBRAIC[(offset * num_of_algebraic) + txrs] =
1.86500+1.00000/( 0.0662900*exp((STATES[(offset * num_of_states) + V] - 34.7000)/7.35500)+
1.12800e-05*exp(- (STATES[(offset * num_of_states) + V] - 29.7400)/25.9400));
        ALGEBRAIC[(offset * num_of_algebraic) + xs1ss] = 1.00000/(1.00000+exp(- (STATES[(offset
* num_of_states) + V]+11.6000)/8.93200));
        ALGEBRAIC[(offset * num_of_algebraic) + txs1] =
817.300+1.00000/( 0.000232600*exp((STATES[(offset * num_of_states) + V]+48.2800)/17.8000)+
0.00129200*exp(- (STATES[(offset * num_of_states) + V]+210.000)/230.000));
        ALGEBRAIC[(offset * num_of_algebraic) + xk1ss] = 1.00000/(1.00000+exp(- (STATES[(offset
* num_of_states) + V]+ 2.55380*CONSTANTS[(offset * num_of_constants) +
ko]+144.590)/( 1.56920*CONSTANTS[(offset * num_of_constants) + ko]+3.81150)));
        ALGEBRAIC[(offset * num_of_algebraic) + txk1] = 122.200/(exp(- (STATES[(offset *
num_of_states) + V]+127.200)/20.3600)+exp((STATES[(offset * num_of_states) +
V]+236.800)/69.3300));
        ALGEBRAIC[(offset * num_of_algebraic) + jss] = ALGEBRAIC[(offset * num_of_algebraic) +
hss];
        ALGEBRAIC[(offset * num_of_algebraic) + tj] = 2.03800+1.00000/( 0.0213600*exp(-
(STATES[(offset * num_of_states) + V]+100.600)/8.28100)+ 0.305200*exp((STATES[(offset *
num_of_states) + V]+0.994100)/38.4500));
        ALGEBRAIC[(offset * num_of_algebraic) + assp] = 1.00000/(1.00000+exp(- (STATES[(offset
* num_of_states) + V] - 24.3400)/14.8200));
        ALGEBRAIC[(offset * num_of_algebraic) + tfcaf] = 7.00000+1.00000/( 0.0400000*exp(-
(STATES[(offset * num_of_states) + V] - 4.00000)/7.00000)+ 0.0400000*exp((STATES[(offset *
num_of_states) + V] - 4.00000)/7.00000));
        ALGEBRAIC[(offset * num_of_algebraic) + tfcas] = 100.000+1.00000/( 0.000120000*exp(-
STATES[(offset * num_of_states) + V]/3.00000)+ 0.000120000*exp(STATES[(offset * num_of_states)
+ V]/7.00000));
        ALGEBRAIC[(offset * num_of_algebraic) + tffp] =  2.50000*ALGEBRAIC[(offset *
num_of_algebraic) + tff];
```

```
        ALGEBRAIC[(offset * num_of_algebraic) + xs2ss] = ALGEBRAIC[(offset * num_of_algebraic)
+ xs1ss];
        ALGEBRAIC[(offset * num_of_algebraic) + txs2] = 1.00000/( 0.0100000*exp((STATES[(offset
* num_of_states) + V] - 50.0000)/20.0000)+ 0.0193000*exp(- (STATES[(offset * num_of_states) +
V]+66.5400)/31.0000));
        ALGEBRAIC[(offset * num_of_algebraic) + CaMKb] = ( CONSTANTS[(offset * num_of_constants)
+ CaMKo]*(1.00000 - STATES[(offset * num_of_states) + CaMKt]))/(1.00000+CONSTANTS[(offset *
num_of_constants) + KmCaM]/STATES[(offset * num_of_states) + cass]);
        ALGEBRAIC[(offset * num_of_algebraic) + hssp] = 1.00000/(1.00000+exp((STATES[(offset *
num_of_states) + V]+89.1000)/6.08600));
        ALGEBRAIC[(offset * num_of_algebraic) + thsp] = 3.00000*ALGEBRAIC[(offset *
num_of_algebraic) + ths];
        ALGEBRAIC[(offset * num_of_algebraic) + tjp] = 1.46000*ALGEBRAIC[(offset *
num_of_algebraic) + tj];
        ALGEBRAIC[(offset * num_of_algebraic) + mLss] = 1.00000/(1.00000+exp(- (STATES[(offset
* num_of_states) + V]+42.8500)/5.26400));
        ALGEBRAIC[(offset * num_of_algebraic) + tmL] = ALGEBRAIC[(offset * num_of_algebraic) +
tm];
        ALGEBRAIC[(offset * num_of_algebraic) + tfcafp] = 2.50000*ALGEBRAIC[(offset *
num_of_algebraic) + tfcaf];
        ALGEBRAIC[(offset * num_of_algebraic) + iss] = 1.00000/(1.00000+exp((STATES[(offset *
num_of_states) + V]+43.9400)/5.71100));
        ALGEBRAIC[(offset * num_of_algebraic) + delta_epi] = (CONSTANTS[(offset *
num_of_constants) + celltype]==1.00000 ? 1.00000 - 0.950000/(1.00000+exp((STATES[(offset *
num_of_states) + V]+70.0000)/5.00000)) : 1.00000);
        ALGEBRAIC[(offset * num_of_algebraic) + tiF_b] = 4.56200+1.00000/( 0.393300*exp(-
(STATES[(offset * num_of_states) + V]+100.000)/100.000)+ 0.0800400*exp((STATES[(offset *
num_of_states) + V]+50.0000)/16.5900));
        ALGEBRAIC[(offset * num_of_algebraic) + tiF] = ALGEBRAIC[(offset * num_of_algebraic)
+ tiF_b]*ALGEBRAIC[(offset * num_of_algebraic) + delta_epi];
        ALGEBRAIC[(offset * num_of_algebraic) + tiS_b] = 23.6200+1.00000/( 0.00141600*exp(-
(STATES[(offset * num_of_states) + V]+96.5200)/59.0500)+ 1.78000e-08*exp((STATES[(offset *
num_of_states) + V]+114.100)/8.07900));
        ALGEBRAIC[(offset * num_of_algebraic) + tiS] = ALGEBRAIC[(offset * num_of_algebraic)
+ tiS_b]*ALGEBRAIC[(offset * num_of_algebraic) + delta_epi];
        ALGEBRAIC[(offset * num_of_algebraic) + dti_develop] =
1.35400+0.000100000/(exp((STATES[(offset * num_of_states) + V] - 167.400)/15.8900)+exp(-
(STATES[(offset * num_of_states) + V] - 12.2300)/0.215400));
        ALGEBRAIC[(offset * num_of_algebraic) + dti_recover] = 1.00000 -
0.500000/(1.00000+exp((STATES[(offset * num_of_states) + V]+70.0000)/20.0000));
        ALGEBRAIC[(offset * num_of_algebraic) + tiFp] = ALGEBRAIC[(offset * num_of_algebraic)
+ dti_develop]*ALGEBRAIC[(offset * num_of_algebraic) + dti_recover]*ALGEBRAIC[(offset *
num_of_algebraic) + tiF];
        ALGEBRAIC[(offset * num_of_algebraic) + tiSp] = ALGEBRAIC[(offset * num_of_algebraic)
+ dti_develop]*ALGEBRAIC[(offset * num_of_algebraic) + dti_recover]*ALGEBRAIC[(offset *
num_of_algebraic) + tiS];
        ALGEBRAIC[(offset * num_of_algebraic) + f] = CONSTANTS[(offset * num_of_constants) +
Aff]*STATES[(offset * num_of_states) + ff]+ CONSTANTS[(offset * num_of_constants) +
Afs]*STATES[(offset * num_of_states) + fs];
```

```
        ALGEBRAIC[(offset * num_of_algebraic) + Afcaf] = 0.300000+0.600000/(1.00000+exp((STATES[(offset * num_of_states) + V] - 10.0000)/10.0000));
        ALGEBRAIC[(offset * num_of_algebraic) + Afcas] = 1.00000 - ALGEBRAIC[(offset * num_of_algebraic) + Afcaf];
        ALGEBRAIC[(offset * num_of_algebraic) + fca] = ALGEBRAIC[(offset * num_of_algebraic) + Afcaf]*STATES[(offset * num_of_states) + fcaf]+ ALGEBRAIC[(offset * num_of_algebraic) + Afcas]*STATES[(offset * num_of_states) + fcas];
        ALGEBRAIC[(offset * num_of_algebraic) + fp] = CONSTANTS[(offset * num_of_constants) + Aff]*STATES[(offset * num_of_states) + ffp]+ CONSTANTS[(offset * num_of_constants) + Afs]*STATES[(offset * num_of_states) + fs];
        ALGEBRAIC[(offset * num_of_algebraic) + fcap] = ALGEBRAIC[(offset * num_of_algebraic) + Afcaf]*STATES[(offset * num_of_states) + fcafp]+ ALGEBRAIC[(offset * num_of_algebraic) + Afcas]*STATES[(offset * num_of_states) + fcas];
        ALGEBRAIC[(offset * num_of_algebraic) + vffrt] = ( STATES[(offset * num_of_states) + V]*CONSTANTS[(offset * num_of_constants) + F]*CONSTANTS[(offset * num_of_constants) + F])/( CONSTANTS[(offset * num_of_constants) + R]*CONSTANTS[(offset * num_of_constants) + T]);
        ALGEBRAIC[(offset * num_of_algebraic) + vfrt] = ( STATES[(offset * num_of_states) + V]*CONSTANTS[(offset * num_of_constants) + F])/( CONSTANTS[(offset * num_of_constants) + R]*CONSTANTS[(offset * num_of_constants) + T]);
        ALGEBRAIC[(offset * num_of_algebraic) + PhiCaL] = ( 4.00000*ALGEBRAIC[(offset * num_of_algebraic) + vffrt]*( STATES[(offset * num_of_states) + cass]*exp( 2.00000*ALGEBRAIC[(offset * num_of_algebraic) + vfrt]) - 0.341000*CONSTANTS[(offset * num_of_constants) + cao]))/(exp( 2.00000*ALGEBRAIC[(offset * num_of_algebraic) + vfrt]) - 1.00000);
        ALGEBRAIC[(offset * num_of_algebraic) + CaMKa] = ALGEBRAIC[(offset * num_of_algebraic) + CaMKb]+STATES[(offset * num_of_states) + CaMKt];
        ALGEBRAIC[(offset * num_of_algebraic) + fICaLp] = 1.00000/(1.00000+CONSTANTS[(offset * num_of_constants) + KmCaMK]/ALGEBRAIC[(offset * num_of_algebraic) + CaMKa]);
        ALGEBRAIC[(offset * num_of_algebraic) + ICaL] = (1.00000 - ALGEBRAIC[(offset * num_of_algebraic) + fICaLp])*CONSTANTS[(offset * num_of_constants) + PCa]*ALGEBRAIC[(offset * num_of_algebraic) + PhiCaL]*STATES[(offset * num_of_states) + d]*( ALGEBRAIC[(offset * num_of_algebraic) + f]*(1.00000 - STATES[(offset * num_of_states) + nca])+ STATES[(offset * num_of_states) + jca]*ALGEBRAIC[(offset * num_of_algebraic) + fca]*STATES[(offset * num_of_states) + nca])+ ALGEBRAIC[(offset * num_of_algebraic) + fICaLp]*CONSTANTS[(offset * num_of_constants) + PCap]*ALGEBRAIC[(offset * num_of_algebraic) + PhiCaL]*STATES[(offset * num_of_states) + d]*( ALGEBRAIC[(offset * num_of_algebraic) + fp]*(1.00000 - STATES[(offset * num_of_states) + nca])+ STATES[(offset * num_of_states) + jca]*ALGEBRAIC[(offset * num_of_algebraic) + fcap]*STATES[(offset * num_of_states) + nca]);
        ALGEBRAIC[(offset * num_of_algebraic) + Jrel_inf_temp] = ( CONSTANTS[(offset * num_of_constants) + a_rel]*- ALGEBRAIC[(offset * num_of_algebraic) + ICaL])/(1.00000+1.00000*pow(1.50000/STATES[(offset * num_of_states) + cajsr], 8.00000));
        ALGEBRAIC[(offset * num_of_algebraic) + Jrel_inf] = (CONSTANTS[(offset * num_of_constants) + celltype]==2.00000 ? ALGEBRAIC[(offset * num_of_algebraic) + Jrel_inf_temp]*1.70000 : ALGEBRAIC[(offset * num_of_algebraic) + Jrel_inf_temp]);
        ALGEBRAIC[(offset * num_of_algebraic) + tau_rel_temp] = CONSTANTS[(offset * num_of_constants) + bt]/(1.00000+0.0123000/STATES[(offset * num_of_states) + cajsr]);
        ALGEBRAIC[(offset * num_of_algebraic) + tau_rel] = (ALGEBRAIC[(offset * num_of_algebraic) + tau_rel_temp]<0.00100000 ? 0.00100000 : ALGEBRAIC[(offset * num_of_algebraic) + tau_rel_temp]);
```

```
        ALGEBRAIC[(offset * num_of_algebraic) + Jrel_temp] = ( CONSTANTS[(offset *
num_of_constants) + a_relp]*- ALGEBRAIC[(offset * num_of_algebraic) +
ICaL])/(1.00000+pow(1.50000/STATES[(offset * num_of_states) + cajsr], 8.00000));
        ALGEBRAIC[(offset * num_of_algebraic) + Jrel_infp] = (CONSTANTS[(offset *
num_of_constants) + celltype]==2.00000 ? ALGEBRAIC[(offset * num_of_algebraic) +
Jrel_temp]*1.70000 : ALGEBRAIC[(offset * num_of_algebraic) + Jrel_temp]);
        ALGEBRAIC[(offset * num_of_algebraic) + tau_relp_temp] = CONSTANTS[(offset *
num_of_constants) + btp]/(1.00000+0.0123000/STATES[(offset * num_of_states) + cajsr]);
        ALGEBRAIC[(offset * num_of_algebraic) + tau_relp] = (ALGEBRAIC[(offset *
num_of_algebraic) + tau_relp_temp]<0.00100000 ? 0.00100000 : ALGEBRAIC[(offset *
num_of_algebraic) + tau_relp_temp]);
        ALGEBRAIC[(offset * num_of_algebraic) + EK] = (( CONSTANTS[(offset * num_of_constants)
+ R]*CONSTANTS[(offset * num_of_constants) + T])/CONSTANTS[(offset * num_of_constants) +
F])*log(CONSTANTS[(offset * num_of_constants) + ko]/STATES[(offset * num_of_states) + ki]);
        ALGEBRAIC[(offset * num_of_algebraic) + AiF] = 1.00000/(1.00000+exp((STATES[(offset *
num_of_states) + V] - 213.600)/151.200));
        ALGEBRAIC[(offset * num_of_algebraic) + AiS] = 1.00000 - ALGEBRAIC[(offset *
num_of_algebraic) + AiF];
        ALGEBRAIC[(offset * num_of_algebraic) + i] = ALGEBRAIC[(offset * num_of_algebraic) +
AiF]*STATES[(offset * num_of_states) + iF]+ ALGEBRAIC[(offset * num_of_algebraic) +
AiS]*STATES[(offset * num_of_states) + iS];
        ALGEBRAIC[(offset * num_of_algebraic) + ip] = ALGEBRAIC[(offset * num_of_algebraic) +
AiF]*STATES[(offset * num_of_states) + iFp]+ ALGEBRAIC[(offset * num_of_algebraic) +
AiS]*STATES[(offset * num_of_states) + iSp];
        ALGEBRAIC[(offset * num_of_algebraic) + fItop] = 1.00000/(1.00000+CONSTANTS[(offset *
num_of_constants) + KmCaMK]/ALGEBRAIC[(offset * num_of_algebraic) + CaMKa]);
        ALGEBRAIC[(offset * num_of_algebraic) + Ito] = CONSTANTS[(offset * num_of_constants)
+ Gto]*(STATES[(offset * num_of_states) + V] - ALGEBRAIC[(offset * num_of_algebraic) +
EK])*( (1.00000 - ALGEBRAIC[(offset * num_of_algebraic) + fItop])*STATES[(offset * num_of_states)
+ a]*ALGEBRAIC[(offset * num_of_algebraic) + i]+ ALGEBRAIC[(offset * num_of_algebraic) +
fItop]*STATES[(offset * num_of_states) + ap]*ALGEBRAIC[(offset * num_of_algebraic) + ip]);
        ALGEBRAIC[(offset * num_of_algebraic) + Axrf] = 1.00000/(1.00000+exp((STATES[(offset *
num_of_states) + V]+54.8100)/38.2100));
        ALGEBRAIC[(offset * num_of_algebraic) + Axrs] = 1.00000 - ALGEBRAIC[(offset *
num_of_algebraic) + Axrf];
        ALGEBRAIC[(offset * num_of_algebraic) + xr] = ALGEBRAIC[(offset * num_of_algebraic) +
Axrf]*STATES[(offset * num_of_states) + xrf]+ ALGEBRAIC[(offset * num_of_algebraic) +
Axrs]*STATES[(offset * num_of_states) + xrs];
        ALGEBRAIC[(offset * num_of_algebraic) + rkr] = ( (1.00000/(1.00000+exp((STATES[(offset
* num_of_states) + V]+55.0000)/75.0000)))*1.00000)/(1.00000+exp((STATES[(offset * num_of_states)
+ V] - 10.0000)/30.0000));
        ALGEBRAIC[(offset * num_of_algebraic) + IKr] = CONSTANTS[(offset * num_of_constants)
+ GKr]* pow((CONSTANTS[(offset * num_of_constants) + ko]/5.40000), 1.0 / 2)*ALGEBRAIC[(offset *
num_of_algebraic) + xr]*ALGEBRAIC[(offset * num_of_algebraic) + rkr]*(STATES[(offset *
num_of_states) + V] - ALGEBRAIC[(offset * num_of_algebraic) + EK]);
        ALGEBRAIC[(offset * num_of_algebraic) + EKs] = (( CONSTANTS[(offset * num_of_constants)
+ R]*CONSTANTS[(offset * num_of_constants) + T])/CONSTANTS[(offset * num_of_constants) +
F])*log((CONSTANTS[(offset * num_of_constants) + ko]+ CONSTANTS[(offset * num_of_constants) +
```

```
PKNa]*CONSTANTS[(offset * num_of_constants) + nao])/(STATES[(offset * num_of_states) + ki]+
CONSTANTS[(offset * num_of_constants) + PKNa]*STATES[(offset * num_of_states) + nai]));
        ALGEBRAIC[(offset * num_of_algebraic) + KsCa] = 1.00000+0.600000/(1.00000+pow(3.80000e-
05/STATES[(offset * num_of_states) + cai], 1.40000));
        ALGEBRAIC[(offset * num_of_algebraic) + IKs] =  CONSTANTS[(offset * num_of_constants)
+ GKs]*ALGEBRAIC[(offset * num_of_algebraic) + KsCa]*STATES[(offset * num_of_states) +
xs1]*STATES[(offset * num_of_states) + xs2]*(STATES[(offset * num_of_states) + V] -
ALGEBRAIC[(offset * num_of_algebraic) + EKs]);
        ALGEBRAIC[(offset * num_of_algebraic) + rk1] = 1.00000/(1.00000+exp(((STATES[(offset *
num_of_states) + V]+105.800) -  2.60000*CONSTANTS[(offset * num_of_constants) + ko])/9.49300));
        ALGEBRAIC[(offset * num_of_algebraic) + IK1] =  CONSTANTS[(offset * num_of_constants)
+ GK1]*  pow(CONSTANTS[(offset * num_of_constants) + ko], 1.0 / 2)*ALGEBRAIC[(offset *
num_of_algebraic) + rk1]*STATES[(offset * num_of_states) + xk1]*(STATES[(offset * num_of_states)
+ V] - ALGEBRAIC[(offset * num_of_algebraic) + EK]);
        ALGEBRAIC[(offset * num_of_algebraic) + Knao] =  CONSTANTS[(offset * num_of_constants)
+ Knao0]*exp(( (1.00000 - CONSTANTS[(offset * num_of_constants) + delta])*STATES[(offset *
num_of_states) + V]*CONSTANTS[(offset * num_of_constants) + F])/( 3.00000*CONSTANTS[(offset *
num_of_constants) + R]*CONSTANTS[(offset * num_of_constants) + T]));
        ALGEBRAIC[(offset * num_of_algebraic) + a3] = ( CONSTANTS[(offset * num_of_constants)
+ k3p]*pow(CONSTANTS[(offset * num_of_constants) + ko]/CONSTANTS[(offset * num_of_constants) +
Kko], 2.00000))/((pow(1.00000+CONSTANTS[(offset * num_of_constants) + nao]/ALGEBRAIC[(offset *
num_of_algebraic) + Knao], 3.00000)+pow(1.00000+CONSTANTS[(offset * num_of_constants) +
ko]/CONSTANTS[(offset * num_of_constants) + Kko], 2.00000)) - 1.00000);
        ALGEBRAIC[(offset * num_of_algebraic) + P] = CONSTANTS[(offset * num_of_constants) +
eP]/(1.00000+CONSTANTS[(offset * num_of_constants) + H]/CONSTANTS[(offset * num_of_constants) +
Khp]+STATES[(offset * num_of_states) + nai]/CONSTANTS[(offset * num_of_constants) +
Knap]+STATES[(offset * num_of_states) + ki]/CONSTANTS[(offset * num_of_constants) + Kxkur]);
        ALGEBRAIC[(offset * num_of_algebraic) + b3] = ( CONSTANTS[(offset * num_of_constants)
+ k3m]*ALGEBRAIC[(offset * num_of_algebraic) + P]*CONSTANTS[(offset * num_of_constants) +
H])/(1.00000+CONSTANTS[(offset * num_of_constants) + MgATP]/CONSTANTS[(offset * num_of_constants)
+ Kmgatp]);
        ALGEBRAIC[(offset * num_of_algebraic) + Knai] =  CONSTANTS[(offset * num_of_constants)
+ Knai0]*exp(( CONSTANTS[(offset * num_of_constants) + delta]*STATES[(offset * num_of_states) +
V]*CONSTANTS[(offset * num_of_constants) + F])/( 3.00000*CONSTANTS[(offset * num_of_constants)
+ R]*CONSTANTS[(offset * num_of_constants) + T]));
        ALGEBRAIC[(offset * num_of_algebraic) + a1] = ( CONSTANTS[(offset * num_of_constants)
+ k1p]*pow(STATES[(offset * num_of_states) + nai]/ALGEBRAIC[(offset * num_of_algebraic) + Knai],
3.00000))/((pow(1.00000+STATES[(offset * num_of_states) + nai]/ALGEBRAIC[(offset *
num_of_algebraic) + Knai], 3.00000)+pow(1.00000+STATES[(offset * num_of_states) +
ki]/CONSTANTS[(offset * num_of_constants) + Kki], 2.00000)) - 1.00000);
        ALGEBRAIC[(offset * num_of_algebraic) + b2] = ( CONSTANTS[(offset * num_of_constants)
+ k2m]*pow(CONSTANTS[(offset * num_of_constants) + nao]/ALGEBRAIC[(offset * num_of_algebraic) +
Knao], 3.00000))/((pow(1.00000+CONSTANTS[(offset * num_of_constants) + nao]/ALGEBRAIC[(offset *
num_of_algebraic) + Knao], 3.00000)+pow(1.00000+CONSTANTS[(offset * num_of_constants) +
ko]/CONSTANTS[(offset * num_of_constants) + Kko], 2.00000)) - 1.00000);
        ALGEBRAIC[(offset * num_of_algebraic) + b4] = ( CONSTANTS[(offset * num_of_constants)
+ k4m]*pow(STATES[(offset * num_of_states) + ki]/CONSTANTS[(offset * num_of_constants) + Kki],
2.00000))/((pow(1.00000+STATES[(offset * num_of_states) + nai]/ALGEBRAIC[(offset *
```

```
num_of_algebraic)  +  Knai],  3.00000)+pow(1.00000+STATES[(offset  *  num_of_states)  +
ki]/CONSTANTS[(offset * num_of_constants) + Kki], 2.00000)) - 1.00000);
        ALGEBRAIC[(offset * num_of_algebraic) + x1] =  CONSTANTS[(offset * num_of_constants) +
a4]*ALGEBRAIC[(offset * num_of_algebraic) + a1]*CONSTANTS[(offset * num_of_constants) + a2]+
ALGEBRAIC[(offset  *  num_of_algebraic)  +  b2]*ALGEBRAIC[(offset  *  num_of_algebraic)  +
b4]*ALGEBRAIC[(offset  *  num_of_algebraic)  +  b3]+  CONSTANTS[(offset  *  num_of_constants)  +
a2]*ALGEBRAIC[(offset * num_of_algebraic) + b4]*ALGEBRAIC[(offset * num_of_algebraic) + b3]+
ALGEBRAIC[(offset  *  num_of_algebraic)  +  b3]*ALGEBRAIC[(offset  *  num_of_algebraic)  +
a1]*CONSTANTS[(offset * num_of_constants) + a2];
        ALGEBRAIC[(offset * num_of_algebraic) + x2] =  ALGEBRAIC[(offset * num_of_algebraic) +
b2]*CONSTANTS[(offset * num_of_constants) + b1]*ALGEBRAIC[(offset * num_of_algebraic) + b4]+
ALGEBRAIC[(offset  *  num_of_algebraic)  +  a1]*CONSTANTS[(offset  *  num_of_constants)  +
a2]*ALGEBRAIC[(offset  *  num_of_algebraic)  +  a3]+  ALGEBRAIC[(offset  *  num_of_algebraic)  +
a3]*CONSTANTS[(offset * num_of_constants) + b1]*ALGEBRAIC[(offset * num_of_algebraic) + b4]+
CONSTANTS[(offset  *  num_of_constants)  +  a2]*ALGEBRAIC[(offset  *  num_of_algebraic)  +
a3]*ALGEBRAIC[(offset * num_of_algebraic) + b4];
        ALGEBRAIC[(offset * num_of_algebraic) + x3] =  CONSTANTS[(offset * num_of_constants) +
a2]*ALGEBRAIC[(offset * num_of_algebraic) + a3]*CONSTANTS[(offset * num_of_constants) + a4]+
ALGEBRAIC[(offset  *  num_of_algebraic)  +  b3]*ALGEBRAIC[(offset  *  num_of_algebraic)  +
b2]*CONSTANTS[(offset  *  num_of_constants)  +  b1]+  ALGEBRAIC[(offset  *  num_of_algebraic)  +
b2]*CONSTANTS[(offset * num_of_constants) + b1]*CONSTANTS[(offset * num_of_constants) + a4]+
ALGEBRAIC[(offset  *  num_of_algebraic)  +  a3]*CONSTANTS[(offset  *  num_of_constants)  +
a4]*CONSTANTS[(offset * num_of_constants) + b1];
        ALGEBRAIC[(offset * num_of_algebraic) + x4] =  ALGEBRAIC[(offset * num_of_algebraic) +
b4]*ALGEBRAIC[(offset * num_of_algebraic) + b3]*ALGEBRAIC[(offset * num_of_algebraic) + b2]+
ALGEBRAIC[(offset  *  num_of_algebraic)  +  a3]*CONSTANTS[(offset  *  num_of_constants)  +
a4]*ALGEBRAIC[(offset  *  num_of_algebraic)  +  a1]+  ALGEBRAIC[(offset  *  num_of_algebraic)  +
b2]*CONSTANTS[(offset * num_of_constants) + a4]*ALGEBRAIC[(offset * num_of_algebraic) + a1]+
ALGEBRAIC[(offset  *  num_of_algebraic)  +  b3]*ALGEBRAIC[(offset  *  num_of_algebraic)  +
b2]*ALGEBRAIC[(offset * num_of_algebraic) + a1];
        ALGEBRAIC[(offset * num_of_algebraic) + E1] = ALGEBRAIC[(offset * num_of_algebraic) +
x1]/(ALGEBRAIC[(offset  *  num_of_algebraic)  +  x1]+ALGEBRAIC[(offset  *  num_of_algebraic)  +
x2]+ALGEBRAIC[(offset * num_of_algebraic) + x3]+ALGEBRAIC[(offset * num_of_algebraic) + x4]);
        ALGEBRAIC[(offset * num_of_algebraic) + E2] = ALGEBRAIC[(offset * num_of_algebraic) +
x2]/(ALGEBRAIC[(offset  *  num_of_algebraic)  +  x1]+ALGEBRAIC[(offset  *  num_of_algebraic)  +
x2]+ALGEBRAIC[(offset * num_of_algebraic) + x3]+ALGEBRAIC[(offset * num_of_algebraic) + x4]);
        ALGEBRAIC[(offset  *  num_of_algebraic)  +  JnakNa]  =  3.00000*(  ALGEBRAIC[(offset  *
num_of_algebraic) + E1]*ALGEBRAIC[(offset * num_of_algebraic) + a3] -  ALGEBRAIC[(offset *
num_of_algebraic) + E2]*ALGEBRAIC[(offset * num_of_algebraic) + b3]);
        ALGEBRAIC[(offset * num_of_algebraic) + E3] = ALGEBRAIC[(offset * num_of_algebraic) +
x3]/(ALGEBRAIC[(offset  *  num_of_algebraic)  +  x1]+ALGEBRAIC[(offset  *  num_of_algebraic)  +
x2]+ALGEBRAIC[(offset * num_of_algebraic) + x3]+ALGEBRAIC[(offset * num_of_algebraic) + x4]);
        ALGEBRAIC[(offset * num_of_algebraic) + E4] = ALGEBRAIC[(offset * num_of_algebraic) +
x4]/(ALGEBRAIC[(offset  *  num_of_algebraic)  +  x1]+ALGEBRAIC[(offset  *  num_of_algebraic)  +
x2]+ALGEBRAIC[(offset * num_of_algebraic) + x3]+ALGEBRAIC[(offset * num_of_algebraic) + x4]);
        ALGEBRAIC[(offset  *  num_of_algebraic)  +  JnakK]  =  2.00000*(  ALGEBRAIC[(offset  *
num_of_algebraic) + E4]*CONSTANTS[(offset * num_of_constants) + b1] -  ALGEBRAIC[(offset *
num_of_algebraic) + E3]*ALGEBRAIC[(offset * num_of_algebraic) + a1]);
```

```
        ALGEBRAIC[(offset * num_of_algebraic) + INaK] =  CONSTANTS[(offset * num_of_constants)
+ Pnak]*( CONSTANTS[(offset * num_of_constants) + zna]*ALGEBRAIC[(offset * num_of_algebraic) +
JnakNa]+ CONSTANTS[(offset * num_of_constants) + zk]*ALGEBRAIC[(offset * num_of_algebraic) +
JnakK]);
        ALGEBRAIC[(offset * num_of_algebraic) + xkb] = 1.00000/(1.00000+exp(- (STATES[(offset
* num_of_states) + V] - 14.4800)/18.3400));
        ALGEBRAIC[(offset * num_of_algebraic) + IKb] =  CONSTANTS[(offset * num_of_constants)
+ GKb]*ALGEBRAIC[(offset * num_of_algebraic) + xkb]*(STATES[(offset * num_of_states) + V] -
ALGEBRAIC[(offset * num_of_algebraic) + EK]);
        ALGEBRAIC[(offset * num_of_algebraic) + JdiffK] = (STATES[(offset * num_of_states) +
kss] - STATES[(offset * num_of_states) + ki])/2.00000;
        ALGEBRAIC[(offset * num_of_algebraic) + PhiCaK] = ( 1.00000*ALGEBRAIC[(offset *
num_of_algebraic)     +     vffrt]*(     0.750000*STATES[(offset     *     num_of_states)     +
kss]*exp( 1.00000*ALGEBRAIC[(offset * num_of_algebraic) + vfrt]) -  0.750000*CONSTANTS[(offset
* num_of_constants) + ko]))/(exp( 1.00000*ALGEBRAIC[(offset * num_of_algebraic) + vfrt]) -
1.00000);
        ALGEBRAIC[(offset * num_of_algebraic) + ICaK] =   (1.00000 - ALGEBRAIC[(offset *
num_of_algebraic) + fICaLp])*CONSTANTS[(offset * num_of_constants) + PCaK]*ALGEBRAIC[(offset *
num_of_algebraic) + PhiCaK]*STATES[(offset * num_of_states) + d]*( ALGEBRAIC[(offset *
num_of_algebraic) + f]*(1.00000 - STATES[(offset * num_of_states) + nca])+ STATES[(offset *
num_of_states) + jca]*ALGEBRAIC[(offset * num_of_algebraic) + fca]*STATES[(offset *
num_of_states) + nca])+ ALGEBRAIC[(offset * num_of_algebraic) + fICaLp]*CONSTANTS[(offset *
num_of_constants) + PCaKp]*ALGEBRAIC[(offset * num_of_algebraic) + PhiCaK]*STATES[(offset *
num_of_states) + d]*( ALGEBRAIC[(offset * num_of_algebraic) + fp]*(1.00000 - STATES[(offset *
num_of_states) + nca])+ STATES[(offset * num_of_states) + jca]*ALGEBRAIC[(offset *
num_of_algebraic) + fcap]*STATES[(offset * num_of_states) + nca]);
        ALGEBRAIC[(offset * num_of_algebraic) + ENa] =  (( CONSTANTS[(offset * num_of_constants)
+ R]*CONSTANTS[(offset * num_of_constants) + T])/CONSTANTS[(offset * num_of_constants) +
F])*log(CONSTANTS[(offset * num_of_constants) + nao]/STATES[(offset * num_of_states) + nai]);
        ALGEBRAIC[(offset * num_of_algebraic) + h] =  CONSTANTS[(offset * num_of_constants) +
Ahf]*STATES[(offset * num_of_states) + hf]+ CONSTANTS[(offset * num_of_constants) +
Ahs]*STATES[(offset * num_of_states) + hs];
        ALGEBRAIC[(offset * num_of_algebraic) + hp] =  CONSTANTS[(offset * num_of_constants) +
Ahf]*STATES[(offset * num_of_states) + hf]+ CONSTANTS[(offset * num_of_constants) +
Ahs]*STATES[(offset * num_of_states) + hsp];
        ALGEBRAIC[(offset * num_of_algebraic) + fINap] = 1.00000/(1.00000+CONSTANTS[(offset *
num_of_constants) + KmCaMK]/ALGEBRAIC[(offset * num_of_algebraic) + CaMKa]);
        ALGEBRAIC[(offset * num_of_algebraic) + INa] =  CONSTANTS[(offset * num_of_constants)
+ GNa]*(STATES[(offset * num_of_states) + V] - ALGEBRAIC[(offset * num_of_algebraic) +
ENa])*pow(STATES[(offset * num_of_states) + m], 3.00000)*( (1.00000 - ALGEBRAIC[(offset *
num_of_algebraic) + fINap])*ALGEBRAIC[(offset * num_of_algebraic) + h]*STATES[(offset *
num_of_states) + j]+ ALGEBRAIC[(offset * num_of_algebraic) + fINap]*ALGEBRAIC[(offset *
num_of_algebraic) + hp]*STATES[(offset * num_of_states) + jp]);
        ALGEBRAIC[(offset * num_of_algebraic) + fINaLp] = 1.00000/(1.00000+CONSTANTS[(offset *
num_of_constants) + KmCaMK]/ALGEBRAIC[(offset * num_of_algebraic) + CaMKa]);
        ALGEBRAIC[(offset * num_of_algebraic) + INaL] =  CONSTANTS[(offset * num_of_constants)
+ GNaL]*(STATES[(offset * num_of_states) + V] - ALGEBRAIC[(offset * num_of_algebraic) +
ENa])*STATES[(offset * num_of_states) + mL]*( (1.00000 - ALGEBRAIC[(offset * num_of_algebraic)
```

```
+ fINaLp])*STATES[(offset * num_of_states) + hL]+ ALGEBRAIC[(offset * num_of_algebraic) +
fINaLp]*STATES[(offset * num_of_states) + hLp]);
        ALGEBRAIC[(offset        *        num_of_algebraic)        +        allo_i]        =
1.00000/(1.00000+pow(CONSTANTS[(offset   *   num_of_constants)   +   KmCaAct]/STATES[(offset   *
num_of_states) + cai], 2.00000));
        ALGEBRAIC[(offset   *   num_of_algebraic)   +   hna]   =   exp((   CONSTANTS[(offset   *
num_of_constants)   +   qna]*STATES[(offset   *   num_of_states)   +   V]*CONSTANTS[(offset   *
num_of_constants)  +  F])/(  CONSTANTS[(offset  *  num_of_constants)  +  R]*CONSTANTS[(offset  *
num_of_constants) + T]));
        ALGEBRAIC[(offset  *  num_of_algebraic)  +  h7_i]  =  1.00000+  (CONSTANTS[(offset  *
num_of_constants)       +       nao]/CONSTANTS[(offset       *       num_of_constants)       +
kna3])*(1.00000+1.00000/ALGEBRAIC[(offset * num_of_algebraic) + hna]);
        ALGEBRAIC[(offset * num_of_algebraic) + h8_i] = CONSTANTS[(offset * num_of_constants)
+ nao]/( CONSTANTS[(offset * num_of_constants) + kna3]*ALGEBRAIC[(offset * num_of_algebraic) +
hna]*ALGEBRAIC[(offset * num_of_algebraic) + h7_i]);
        ALGEBRAIC[(offset * num_of_algebraic) + k3pp_i] =  ALGEBRAIC[(offset * num_of_algebraic)
+ h8_i]*CONSTANTS[(offset * num_of_constants) + wnaca];
        ALGEBRAIC[(offset * num_of_algebraic) + h1_i] = 1.00000+ (STATES[(offset * num_of_states)
+   nai]/CONSTANTS[(offset   *   num_of_constants)   +   kna3])*(1.00000+ALGEBRAIC[(offset   *
num_of_algebraic) + hna]);
        ALGEBRAIC[(offset * num_of_algebraic) + h2_i] = ( STATES[(offset * num_of_states) +
nai]*ALGEBRAIC[(offset * num_of_algebraic) + hna])/( CONSTANTS[(offset * num_of_constants) +
kna3]*ALGEBRAIC[(offset * num_of_algebraic) + h1_i]);
        ALGEBRAIC[(offset * num_of_algebraic) + k4pp_i] =  ALGEBRAIC[(offset * num_of_algebraic)
+ h2_i]*CONSTANTS[(offset * num_of_constants) + wnaca];
        ALGEBRAIC[(offset * num_of_algebraic) + h4_i] = 1.00000+ (STATES[(offset * num_of_states)
+ nai]/CONSTANTS[(offset * num_of_constants) + kna1])*(1.00000+STATES[(offset * num_of_states)
+ nai]/CONSTANTS[(offset * num_of_constants) + kna2]);
        ALGEBRAIC[(offset * num_of_algebraic) + h5_i] = ( STATES[(offset * num_of_states) +
nai]*STATES[(offset  *  num_of_states)  +  nai])/(  ALGEBRAIC[(offset  *  num_of_algebraic)  +
h4_i]*CONSTANTS[(offset  *  num_of_constants)  +  kna1]*CONSTANTS[(offset  *  num_of_constants)  +
kna2]);
        ALGEBRAIC[(offset * num_of_algebraic) + k7_i] =  ALGEBRAIC[(offset * num_of_algebraic)
+ h5_i]*ALGEBRAIC[(offset * num_of_algebraic) + h2_i]*CONSTANTS[(offset * num_of_constants) +
wna];
        ALGEBRAIC[(offset * num_of_algebraic) + k8_i] =  ALGEBRAIC[(offset * num_of_algebraic)
+ h8_i]*CONSTANTS[(offset * num_of_constants) + h11_i]*CONSTANTS[(offset * num_of_constants) +
wna];
        ALGEBRAIC[(offset   *   num_of_algebraic)   +   h9_i]   =   1.00000/ALGEBRAIC[(offset   *
num_of_algebraic) + h7_i];
        ALGEBRAIC[(offset * num_of_algebraic) + k3p_i] =  ALGEBRAIC[(offset * num_of_algebraic)
+ h9_i]*CONSTANTS[(offset * num_of_constants) + wca];
        ALGEBRAIC[(offset * num_of_algebraic) + k3_i] = ALGEBRAIC[(offset * num_of_algebraic)
+ k3p_i]+ALGEBRAIC[(offset * num_of_algebraic) + k3pp_i];
        ALGEBRAIC[(offset   *   num_of_algebraic)   +   hca]   =   exp((   CONSTANTS[(offset   *
num_of_constants)   +   qca]*STATES[(offset   *   num_of_states)   +   V]*CONSTANTS[(offset   *
num_of_constants)  +  F])/(  CONSTANTS[(offset  *  num_of_constants)  +  R]*CONSTANTS[(offset  *
num_of_constants) + T]));
```

```
        ALGEBRAIC[(offset * num_of_algebraic) + h3_i] = 1.00000/ALGEBRAIC[(offset *
num_of_algebraic) + h1_i];
        ALGEBRAIC[(offset * num_of_algebraic) + k4p_i] = ( ALGEBRAIC[(offset * num_of_algebraic)
+ h3_i]*CONSTANTS[(offset * num_of_constants) + wca])/ALGEBRAIC[(offset * num_of_algebraic) +
hca];
        ALGEBRAIC[(offset * num_of_algebraic) + k4_i] = ALGEBRAIC[(offset * num_of_algebraic)
+ k4p_i]+ALGEBRAIC[(offset * num_of_algebraic) + k4pp_i];
        ALGEBRAIC[(offset * num_of_algebraic) + h6_i] = 1.00000/ALGEBRAIC[(offset *
num_of_algebraic) + h4_i];
        ALGEBRAIC[(offset * num_of_algebraic) + k6_i] = ALGEBRAIC[(offset * num_of_algebraic)
+ h6_i]*STATES[(offset * num_of_states) + cai]*CONSTANTS[(offset * num_of_constants) + kcaon];
        ALGEBRAIC[(offset * num_of_algebraic) + x1_i] = CONSTANTS[(offset * num_of_constants)
+ k2_i]*ALGEBRAIC[(offset * num_of_algebraic) + k4_i]*(ALGEBRAIC[(offset * num_of_algebraic) +
k7_i]+ALGEBRAIC[(offset * num_of_algebraic) + k6_i])+ CONSTANTS[(offset * num_of_constants) +
k5_i]*ALGEBRAIC[(offset * num_of_algebraic) + k7_i]*(CONSTANTS[(offset * num_of_constants) +
k2_i]+ALGEBRAIC[(offset * num_of_algebraic) + k3_i]);
        ALGEBRAIC[(offset * num_of_algebraic) + x2_i] = CONSTANTS[(offset * num_of_constants)
+ k1_i]*ALGEBRAIC[(offset * num_of_algebraic) + k7_i]*(ALGEBRAIC[(offset * num_of_algebraic) +
k4_i]+CONSTANTS[(offset * num_of_constants) + k5_i])+ ALGEBRAIC[(offset * num_of_algebraic) +
k4_i]*ALGEBRAIC[(offset * num_of_algebraic) + k6_i]*(CONSTANTS[(offset * num_of_constants) +
k1_i]+ALGEBRAIC[(offset * num_of_algebraic) + k8_i]);
        ALGEBRAIC[(offset * num_of_algebraic) + x3_i] = CONSTANTS[(offset * num_of_constants)
+ k1_i]*ALGEBRAIC[(offset * num_of_algebraic) + k3_i]*(ALGEBRAIC[(offset * num_of_algebraic) +
k7_i]+ALGEBRAIC[(offset * num_of_algebraic) + k6_i])+ ALGEBRAIC[(offset * num_of_algebraic) +
k8_i]*ALGEBRAIC[(offset * num_of_algebraic) + k6_i]*(CONSTANTS[(offset * num_of_constants) +
k2_i]+ALGEBRAIC[(offset * num_of_algebraic) + k3_i]);
        ALGEBRAIC[(offset * num_of_algebraic) + x4_i] = CONSTANTS[(offset * num_of_constants)
+ k2_i]*ALGEBRAIC[(offset * num_of_algebraic) + k8_i]*(ALGEBRAIC[(offset * num_of_algebraic) +
k4_i]+CONSTANTS[(offset * num_of_constants) + k5_i])+ ALGEBRAIC[(offset * num_of_algebraic) +
k3_i]*CONSTANTS[(offset * num_of_constants) + k5_i]*(CONSTANTS[(offset * num_of_constants) +
k1_i]+ALGEBRAIC[(offset * num_of_algebraic) + k8_i]);
        ALGEBRAIC[(offset * num_of_algebraic) + E1_i] = ALGEBRAIC[(offset * num_of_algebraic)
+ x1_i]/(ALGEBRAIC[(offset * num_of_algebraic) + x1_i]+ALGEBRAIC[(offset * num_of_algebraic)
+ x2_i]+ALGEBRAIC[(offset * num_of_algebraic) + x3_i]+ALGEBRAIC[(offset * num_of_algebraic) +
x4_i]);
        ALGEBRAIC[(offset * num_of_algebraic) + E2_i] = ALGEBRAIC[(offset * num_of_algebraic)
+ x2_i]/(ALGEBRAIC[(offset * num_of_algebraic) + x1_i]+ALGEBRAIC[(offset * num_of_algebraic) +
x2_i]+ALGEBRAIC[(offset * num_of_algebraic) + x3_i]+ALGEBRAIC[(offset * num_of_algebraic) +
x4_i]);
        ALGEBRAIC[(offset * num_of_algebraic) + E3_i] = ALGEBRAIC[(offset * num_of_algebraic)
+ x3_i]/(ALGEBRAIC[(offset * num_of_algebraic) + x1_i]+ALGEBRAIC[(offset * num_of_algebraic) +
x2_i]+ALGEBRAIC[(offset * num_of_algebraic) + x3_i]+ALGEBRAIC[(offset * num_of_algebraic) +
x4_i]);
        ALGEBRAIC[(offset * num_of_algebraic) + E4_i] = ALGEBRAIC[(offset * num_of_algebraic)
+ x4_i]/(ALGEBRAIC[(offset * num_of_algebraic) + x1_i]+ALGEBRAIC[(offset * num_of_algebraic) +
x2_i]+ALGEBRAIC[(offset * num_of_algebraic) + x3_i]+ALGEBRAIC[(offset * num_of_algebraic) +
x4_i]);
        ALGEBRAIC[(offset * num_of_algebraic) + JncxNa_i] = ( 3.00000*( ALGEBRAIC[(offset *
num_of_algebraic) + E4_i]*ALGEBRAIC[(offset * num_of_algebraic) + k7_i] - ALGEBRAIC[(offset *
```

```
num_of_algebraic) + E1_i]*ALGEBRAIC[(offset * num_of_algebraic) + k8_i])+ ALGEBRAIC[(offset *
num_of_algebraic) + E3_i]*ALGEBRAIC[(offset * num_of_algebraic) + k4pp_i]) - ALGEBRAIC[(offset
* num_of_algebraic) + E2_i]*ALGEBRAIC[(offset * num_of_algebraic) + k3pp_i];
        ALGEBRAIC[(offset * num_of_algebraic) + JncxCa_i] = ALGEBRAIC[(offset *
num_of_algebraic) + E2_i]*CONSTANTS[(offset * num_of_constants) + k2_i] - ALGEBRAIC[(offset *
num_of_algebraic) + E1_i]*CONSTANTS[(offset * num_of_constants) + k1_i];
        ALGEBRAIC[(offset * num_of_algebraic) + INaCa_i] = 0.800000*CONSTANTS[(offset *
num_of_constants) + Gncx]*ALGEBRAIC[(offset * num_of_algebraic) + allo_i]*( CONSTANTS[(offset *
num_of_constants) + zna]*ALGEBRAIC[(offset * num_of_algebraic) + JncxNa_i]+ CONSTANTS[(offset *
num_of_constants) + zca]*ALGEBRAIC[(offset * num_of_algebraic) + JncxCa_i]);
        ALGEBRAIC[(offset * num_of_algebraic) + INab] = ( CONSTANTS[(offset * num_of_constants)
+ PNab]*ALGEBRAIC[(offset * num_of_algebraic) + vffrt]*( STATES[(offset * num_of_states) +
nai]*exp(ALGEBRAIC[(offset * num_of_algebraic) + vfrt]) - CONSTANTS[(offset * num_of_constants)
+ nao]))/(exp(ALGEBRAIC[(offset * num_of_algebraic) + vfrt]) - 1.00000);
        ALGEBRAIC[(offset * num_of_algebraic) + JdiffNa] = (STATES[(offset * num_of_states) +
nass] - STATES[(offset * num_of_states) + nai])/2.00000;
        ALGEBRAIC[(offset * num_of_algebraic) + PhiCaNa] = ( 1.00000*ALGEBRAIC[(offset *
num_of_algebraic) + vffrt]*( 0.750000*STATES[(offset * num_of_states) +
nass]*exp( 1.00000*ALGEBRAIC[(offset * num_of_algebraic) + vfrt]) - 0.750000*CONSTANTS[(offset
* num_of_constants) + nao]))/(exp( 1.00000*ALGEBRAIC[(offset * num_of_algebraic) + vfrt]) -
1.00000);
        ALGEBRAIC[(offset * num_of_algebraic) + ICaNa] = (1.00000 - ALGEBRAIC[(offset *
num_of_algebraic) + fICaLp])*CONSTANTS[(offset * num_of_constants) + PCaNa]*ALGEBRAIC[(offset *
num_of_algebraic) + PhiCaNa]*STATES[(offset * num_of_states) + d]*( ALGEBRAIC[(offset *
num_of_algebraic) + f]*(1.00000 - STATES[(offset * num_of_states) + nca])+ STATES[(offset *
num_of_states) + jca]*ALGEBRAIC[(offset * num_of_algebraic) + fca]*STATES[(offset *
num_of_states) + nca])+ ALGEBRAIC[(offset * num_of_algebraic) + fICaLp]*CONSTANTS[(offset *
num_of_constants) + PCaNap]*ALGEBRAIC[(offset * num_of_algebraic) + PhiCaNa]*STATES[(offset *
num_of_states) + d]*( ALGEBRAIC[(offset * num_of_algebraic) + fp]*(1.00000 - STATES[(offset *
num_of_states) + nca])+ STATES[(offset * num_of_states) + jca]*ALGEBRAIC[(offset *
num_of_algebraic) + fcap]*STATES[(offset * num_of_states) + nca]);
        ALGEBRAIC[(offset * num_of_algebraic) + allo_ss] =
1.00000/(1.00000+pow(CONSTANTS[(offset * num_of_constants) + KmCaAct]/STATES[(offset *
num_of_states) + cass], 2.00000));
        ALGEBRAIC[(offset * num_of_algebraic) + h7_ss] = 1.00000+ (CONSTANTS[(offset *
num_of_constants) + nao]/CONSTANTS[(offset * num_of_constants) +
kna3])*(1.00000+1.00000/ALGEBRAIC[(offset * num_of_algebraic) + hna]);
        ALGEBRAIC[(offset * num_of_algebraic) + h8_ss] = CONSTANTS[(offset * num_of_constants)
+ nao]/( CONSTANTS[(offset * num_of_constants) + kna3]*ALGEBRAIC[(offset * num_of_algebraic) +
hna]*ALGEBRAIC[(offset * num_of_algebraic) + h7_ss]);
        ALGEBRAIC[(offset * num_of_algebraic) + k3pp_ss] = ALGEBRAIC[(offset * num_of_algebraic)
+ h8_ss]*CONSTANTS[(offset * num_of_constants) + wnaca];
        ALGEBRAIC[(offset * num_of_algebraic) + h1_ss] = 1.00000+ (STATES[(offset *
num_of_states) + nass]/CONSTANTS[(offset * num_of_constants) + kna3])*(1.00000+ALGEBRAIC[(offset
* num_of_algebraic) + hna]);
        ALGEBRAIC[(offset * num_of_algebraic) + h2_ss] = ( STATES[(offset * num_of_states) +
nass]*ALGEBRAIC[(offset * num_of_algebraic) + hna])/( CONSTANTS[(offset * num_of_constants) +
kna3]*ALGEBRAIC[(offset * num_of_algebraic) + h1_ss]);
```

```
        ALGEBRAIC[(offset * num_of_algebraic) + k4pp_ss] =  ALGEBRAIC[(offset * num_of_algebraic)
+ h2_ss]*CONSTANTS[(offset * num_of_constants) + wnaca];
        ALGEBRAIC[(offset  *  num_of_algebraic)  +  h4_ss]  =  1.00000+  (STATES[(offset  *
num_of_states) + nass]/CONSTANTS[(offset * num_of_constants) + kna1])*(1.00000+STATES[(offset *
num_of_states) + nass]/CONSTANTS[(offset * num_of_constants) + kna2]);
        ALGEBRAIC[(offset * num_of_algebraic) + h5_ss] = ( STATES[(offset * num_of_states) +
nass]*STATES[(offset  *  num_of_states)  +  nass])/(  ALGEBRAIC[(offset  *  num_of_algebraic)  +
h4_ss]*CONSTANTS[(offset  *  num_of_constants)  +  kna1]*CONSTANTS[(offset  *  num_of_constants)  +
kna2]);
        ALGEBRAIC[(offset * num_of_algebraic) + k7_ss] =  ALGEBRAIC[(offset * num_of_algebraic)
+ h5_ss]*ALGEBRAIC[(offset * num_of_algebraic) + h2_ss]*CONSTANTS[(offset * num_of_constants) +
wna];
        ALGEBRAIC[(offset * num_of_algebraic) + k8_ss] =  ALGEBRAIC[(offset * num_of_algebraic)
+ h8_ss]*CONSTANTS[(offset * num_of_constants) + h11_ss]*CONSTANTS[(offset * num_of_constants)
+ wna];
        ALGEBRAIC[(offset  *  num_of_algebraic)  +  h9_ss]  =  1.00000/ALGEBRAIC[(offset  *
num_of_algebraic) + h7_ss];
        ALGEBRAIC[(offset * num_of_algebraic) + k3p_ss] =  ALGEBRAIC[(offset * num_of_algebraic)
+ h9_ss]*CONSTANTS[(offset * num_of_constants) + wca];
        ALGEBRAIC[(offset * num_of_algebraic) + k3_ss] = ALGEBRAIC[(offset * num_of_algebraic)
+ k3p_ss]+ALGEBRAIC[(offset * num_of_algebraic) + k3pp_ss];
        ALGEBRAIC[(offset  *  num_of_algebraic)  +  h3_ss]  =  1.00000/ALGEBRAIC[(offset  *
num_of_algebraic) + h1_ss];
        ALGEBRAIC[(offset * num_of_algebraic) + k4p_ss] = ( ALGEBRAIC[(offset * num_of_algebraic)
+ h3_ss]*CONSTANTS[(offset * num_of_constants) + wca])/ALGEBRAIC[(offset * num_of_algebraic) +
hca];
        ALGEBRAIC[(offset * num_of_algebraic) + k4_ss] = ALGEBRAIC[(offset * num_of_algebraic)
+ k4p_ss]+ALGEBRAIC[(offset * num_of_algebraic) + k4pp_ss];
        ALGEBRAIC[(offset  *  num_of_algebraic)  +  h6_ss]  =  1.00000/ALGEBRAIC[(offset  *
num_of_algebraic) + h4_ss];
        ALGEBRAIC[(offset * num_of_algebraic) + k6_ss] =  ALGEBRAIC[(offset * num_of_algebraic)
+ h6_ss]*STATES[(offset * num_of_states) + cass]*CONSTANTS[(offset * num_of_constants) + kcaon];
        ALGEBRAIC[(offset * num_of_algebraic) + x1_ss] =  CONSTANTS[(offset * num_of_constants)
+ k2_ss]*ALGEBRAIC[(offset * num_of_algebraic) + k4_ss]*(ALGEBRAIC[(offset * num_of_algebraic)
+ k7_ss]+ALGEBRAIC[(offset * num_of_algebraic) + k6_ss])+ CONSTANTS[(offset * num_of_constants)
+ k5_ss]*ALGEBRAIC[(offset * num_of_algebraic) + k7_ss]*(CONSTANTS[(offset * num_of_constants)
+ k2_ss]+ALGEBRAIC[(offset * num_of_algebraic) + k3_ss]);
        ALGEBRAIC[(offset * num_of_algebraic) + x2_ss] =  CONSTANTS[(offset * num_of_constants)
+ k1_ss]*ALGEBRAIC[(offset * num_of_algebraic) + k7_ss]*(ALGEBRAIC[(offset * num_of_algebraic)
+ k4_ss]+CONSTANTS[(offset * num_of_constants) + k5_ss])+ ALGEBRAIC[(offset * num_of_algebraic)
+ k4_ss]*ALGEBRAIC[(offset * num_of_algebraic) + k6_ss]*(CONSTANTS[(offset * num_of_constants)
+ k1_ss]+ALGEBRAIC[(offset * num_of_algebraic) + k8_ss]);
        ALGEBRAIC[(offset * num_of_algebraic) + x3_ss] =  CONSTANTS[(offset * num_of_constants)
+ k1_ss]*ALGEBRAIC[(offset * num_of_algebraic) + k3_ss]*(ALGEBRAIC[(offset * num_of_algebraic)
+ k7_ss]+ALGEBRAIC[(offset * num_of_algebraic) + k6_ss])+ ALGEBRAIC[(offset * num_of_algebraic)
+ k8_ss]*ALGEBRAIC[(offset * num_of_algebraic) + k6_ss]*(CONSTANTS[(offset * num_of_constants)
+ k2_ss]+ALGEBRAIC[(offset * num_of_algebraic) + k3_ss]);
        ALGEBRAIC[(offset * num_of_algebraic) + x4_ss] =  CONSTANTS[(offset * num_of_constants)
+ k2_ss]*ALGEBRAIC[(offset * num_of_algebraic) + k8_ss]*(ALGEBRAIC[(offset * num_of_algebraic)
```

```
+ k4_ss]+CONSTANTS[(offset * num_of_constants) + k5_ss])+ ALGEBRAIC[(offset * num_of_algebraic)
+ k3_ss]*CONSTANTS[(offset * num_of_constants) + k5_ss]*(CONSTANTS[(offset * num_of_constants)
+ k1_ss]+ALGEBRAIC[(offset * num_of_algebraic) + k8_ss]);
        ALGEBRAIC[(offset * num_of_algebraic) + E1_ss] = ALGEBRAIC[(offset * num_of_algebraic)
+ x1_ss]/(ALGEBRAIC[(offset * num_of_algebraic) + x1_ss]+ALGEBRAIC[(offset * num_of_algebraic)
+ x2_ss]+ALGEBRAIC[(offset * num_of_algebraic) + x3_ss]+ALGEBRAIC[(offset * num_of_algebraic) +
x4_ss]);
        ALGEBRAIC[(offset * num_of_algebraic) + E2_ss] = ALGEBRAIC[(offset * num_of_algebraic)
+ x2_ss]/(ALGEBRAIC[(offset * num_of_algebraic) + x1_ss]+ALGEBRAIC[(offset * num_of_algebraic)
+ x2_ss]+ALGEBRAIC[(offset * num_of_algebraic) + x3_ss]+ALGEBRAIC[(offset * num_of_algebraic) +
x4_ss]);
        ALGEBRAIC[(offset * num_of_algebraic) + E3_ss] = ALGEBRAIC[(offset * num_of_algebraic)
+ x3_ss]/(ALGEBRAIC[(offset * num_of_algebraic) + x1_ss]+ALGEBRAIC[(offset * num_of_algebraic)
+ x2_ss]+ALGEBRAIC[(offset * num_of_algebraic) + x3_ss]+ALGEBRAIC[(offset * num_of_algebraic) +
x4_ss]);
        ALGEBRAIC[(offset * num_of_algebraic) + E4_ss] = ALGEBRAIC[(offset * num_of_algebraic)
+ x4_ss]/(ALGEBRAIC[(offset * num_of_algebraic) + x1_ss]+ALGEBRAIC[(offset * num_of_algebraic)
+ x2_ss]+ALGEBRAIC[(offset * num_of_algebraic) + x3_ss]+ALGEBRAIC[(offset * num_of_algebraic) +
x4_ss]);
        ALGEBRAIC[(offset * num_of_algebraic) + JncxNa_ss] = ( 3.00000*( ALGEBRAIC[(offset *
num_of_algebraic) + E4_ss]*ALGEBRAIC[(offset * num_of_algebraic) + k7_ss] -  ALGEBRAIC[(offset
* num_of_algebraic) + E1_ss]*ALGEBRAIC[(offset * num_of_algebraic) + k8_ss])+ ALGEBRAIC[(offset
*  num_of_algebraic)  +  E3_ss]*ALGEBRAIC[(offset  *  num_of_algebraic)  +  k4pp_ss])  -
ALGEBRAIC[(offset * num_of_algebraic) + E2_ss]*ALGEBRAIC[(offset * num_of_algebraic) + k3pp_ss];
        ALGEBRAIC[(offset  *  num_of_algebraic)  +  JncxCa_ss]  =  ALGEBRAIC[(offset  *
num_of_algebraic) + E2_ss]*CONSTANTS[(offset * num_of_constants) + k2_ss] -  ALGEBRAIC[(offset
* num_of_algebraic) + E1_ss]*CONSTANTS[(offset * num_of_constants) + k1_ss];
        ALGEBRAIC[(offset  *  num_of_algebraic)  +  INaCa_ss]  =  0.200000*CONSTANTS[(offset  *
num_of_constants) + Gncx]*ALGEBRAIC[(offset * num_of_algebraic) + allo_ss]*( CONSTANTS[(offset
* num_of_constants) + zna]*ALGEBRAIC[(offset * num_of_algebraic) + JncxNa_ss]+ CONSTANTS[(offset
* num_of_constants) + zca]*ALGEBRAIC[(offset * num_of_algebraic) + JncxCa_ss]);
        ALGEBRAIC[(offset * num_of_algebraic) + IpCa] = ( CONSTANTS[(offset * num_of_constants)
+  GpCa]*STATES[(offset  *  num_of_states)  +  cai])/(CONSTANTS[(offset  *  num_of_constants)  +
KmCap]+STATES[(offset * num_of_states) + cai]);
        ALGEBRAIC[(offset * num_of_algebraic) + ICab] = ( CONSTANTS[(offset * num_of_constants)
+ PCab]*4.00000*ALGEBRAIC[(offset * num_of_algebraic) + vffrt]*( STATES[(offset * num_of_states)
+ cai]*exp( 2.00000*ALGEBRAIC[(offset * num_of_algebraic) + vfrt]) -  0.341000*CONSTANTS[(offset
*  num_of_constants)  +  cao]))/(exp(  2.00000*ALGEBRAIC[(offset  *  num_of_algebraic)  +  vfrt])  -
1.00000);
        ALGEBRAIC[(offset * num_of_algebraic) + Jdiff] = (STATES[(offset * num_of_states) +
cass] - STATES[(offset * num_of_states) + cai])/0.200000;
        ALGEBRAIC[(offset * num_of_algebraic) + fJrelp] = 1.00000/(1.00000+CONSTANTS[(offset *
num_of_constants) + KmCaMK]/ALGEBRAIC[(offset * num_of_algebraic) + CaMKa]);

        //cvar starts here
        ALGEBRAIC[(offset * num_of_algebraic) + Jrel] = CONSTANTS[(offset * num_of_constants)
+ Jrel_scale] * ( (1.00000 - ALGEBRAIC[(offset * num_of_algebraic) + fJrelp])*STATES[(offset *
num_of_states)  +  Jrelnp]+  ALGEBRAIC[(offset  *  num_of_algebraic)  +  fJrelp]*STATES[(offset  *
num_of_states) + Jrelp]);
```

```
        ALGEBRAIC[(offset * num_of_algebraic) + Bcass] = 1.00000/(1.00000+( CONSTANTS[(offset
*     num_of_constants)     +     BSRmax]*CONSTANTS[(offset     *     num_of_constants)     +
KmBSR])/pow(CONSTANTS[(offset * num_of_constants) + KmBSR]+STATES[(offset * num_of_states) +
cass],  2.00000)+(  CONSTANTS[(offset  *  num_of_constants)  +  BSLmax]*CONSTANTS[(offset  *
num_of_constants) + KmBSL])/pow(CONSTANTS[(offset * num_of_constants) + KmBSL]+STATES[(offset *
num_of_states) + cass], 2.00000));
        ALGEBRAIC[(offset * num_of_algebraic) + Jupnp] = ( CONSTANTS[(offset * num_of_constants)
+ upScale]*0.00437500*STATES[(offset * num_of_states) + cai])/(STATES[(offset * num_of_states)
+ cai]+0.000920000);
        ALGEBRAIC[(offset * num_of_algebraic) + Jupp] = ( CONSTANTS[(offset * num_of_constants)
+  upScale]*2.75000*0.00437500*STATES[(offset  *  num_of_states)  +  cai])/((STATES[(offset  *
num_of_states) + cai]+0.000920000) - 0.000170000);
        ALGEBRAIC[(offset * num_of_algebraic) + fJupp] = 1.00000/(1.00000+CONSTANTS[(offset *
num_of_constants) + KmCaMK]/ALGEBRAIC[(offset * num_of_algebraic) + CaMKa]);

        ALGEBRAIC[(offset * num_of_algebraic) + Jleak] = CONSTANTS[(offset * num_of_constants)
+ Jleak_scale] * ( 0.00393750*STATES[(offset * num_of_states) + cansr])/15.0000;
        ALGEBRAIC[(offset * num_of_algebraic) + Jup] = CONSTANTS[(offset * num_of_constants) +
Jup_scale] * ( ( (1.00000 - ALGEBRAIC[(offset * num_of_algebraic) + fJupp])*ALGEBRAIC[(offset *
num_of_algebraic) + Jupnp]+ ALGEBRAIC[(offset * num_of_algebraic) + fJupp]*ALGEBRAIC[(offset *
num_of_algebraic) + Jupp]) - ALGEBRAIC[(offset * num_of_algebraic) + Jleak]);

        ALGEBRAIC[(offset * num_of_algebraic) + Bcai] = 1.00000/(1.00000+( CONSTANTS[(offset *
num_of_constants)      +      cmdnmax]*CONSTANTS[(offset      *      num_of_constants)      +
kmcmdn])/pow(CONSTANTS[(offset * num_of_constants) + kmcmdn]+STATES[(offset * num_of_states) +
cai],  2.00000)+(  CONSTANTS[(offset  *  num_of_constants)  +  trpnmax]*CONSTANTS[(offset  *
num_of_constants) + kmtrpn])/pow(CONSTANTS[(offset * num_of_constants) + kmtrpn]+STATES[(offset
* num_of_states) + cai], 2.00000));

        ALGEBRAIC[(offset * num_of_algebraic) + Jtr] = CONSTANTS[(offset * num_of_constants) +
Jtr_scale] * (STATES[(offset * num_of_states) + cansr] - STATES[(offset * num_of_states) +
cajsr])/100.000;
        //cvar ends here
        ALGEBRAIC[(offset * num_of_algebraic) + Bcajsr] = 1.00000/(1.00000+( CONSTANTS[(offset
*     num_of_constants)     +     csqnmax]*CONSTANTS[(offset     *     num_of_constants)     +
kmcsqn])/pow(CONSTANTS[(offset * num_of_constants) + kmcsqn]+STATES[(offset * num_of_states) +
cajsr], 2.00000));


        RATES[(offset * num_of_rates) + hL] = (ALGEBRAIC[(offset * num_of_algebraic) + hLss] -
STATES[(offset * num_of_states) + hL])/CONSTANTS[(offset * num_of_constants) + thL];
        RATES[(offset * num_of_rates) + hLp] = (ALGEBRAIC[(offset * num_of_algebraic) + hLssp]
- STATES[(offset * num_of_states) + hLp])/CONSTANTS[(offset * num_of_constants) + thLp];
        RATES[(offset * num_of_rates) + m] = (ALGEBRAIC[(offset * num_of_algebraic) + mss] -
STATES[(offset * num_of_states) + m])/ALGEBRAIC[(offset * num_of_algebraic) + tm];
        RATES[(offset * num_of_rates) + hf] = (ALGEBRAIC[(offset * num_of_algebraic) + hss] -
STATES[(offset * num_of_states) + hf])/ALGEBRAIC[(offset * num_of_algebraic) + thf];
```

```
        RATES[(offset * num_of_rates) + hs] = (ALGEBRAIC[(offset * num_of_algebraic) + hss] -
STATES[(offset * num_of_states) + hs])/ALGEBRAIC[(offset * num_of_algebraic) + ths];
        RATES[(offset * num_of_rates) + a] = (ALGEBRAIC[(offset * num_of_algebraic) + ass] -
STATES[(offset * num_of_states) + a])/ALGEBRAIC[(offset * num_of_algebraic) + ta];
        RATES[(offset * num_of_rates) + d] = (ALGEBRAIC[(offset * num_of_algebraic) + dss] -
STATES[(offset * num_of_states) + d])/ALGEBRAIC[(offset * num_of_algebraic) + td];
        RATES[(offset * num_of_rates) + ff] = (ALGEBRAIC[(offset * num_of_algebraic) + fss] -
STATES[(offset * num_of_states) + ff])/ALGEBRAIC[(offset * num_of_algebraic) + tff];
        RATES[(offset * num_of_rates) + fs] = (ALGEBRAIC[(offset * num_of_algebraic) + fss] -
STATES[(offset * num_of_states) + fs])/ALGEBRAIC[(offset * num_of_algebraic) + tfs];
        RATES[(offset * num_of_rates) + jca] = (ALGEBRAIC[(offset * num_of_algebraic) + fcass]
- STATES[(offset * num_of_states) + jca])/CONSTANTS[(offset * num_of_constants) + tjca];
        RATES[(offset * num_of_rates) + nca] =  ALGEBRAIC[(offset * num_of_algebraic) +
anca]*CONSTANTS[(offset * num_of_constants) + k2n] - STATES[(offset * num_of_states) +
nca]*ALGEBRAIC[(offset * num_of_algebraic) + km2n];
        RATES[(offset * num_of_rates) + xrf] = (ALGEBRAIC[(offset * num_of_algebraic) + xrss]
- STATES[(offset * num_of_states) + xrf])/ALGEBRAIC[(offset * num_of_algebraic) + txrf];
        RATES[(offset * num_of_rates) + xrs] = (ALGEBRAIC[(offset * num_of_algebraic) + xrss]
- STATES[(offset * num_of_states) + xrs])/ALGEBRAIC[(offset * num_of_algebraic) + txrs];
        RATES[(offset * num_of_rates) + xs1] = (ALGEBRAIC[(offset * num_of_algebraic) + xs1ss]
- STATES[(offset * num_of_states) + xs1])/ALGEBRAIC[(offset * num_of_algebraic) + txs1];
        RATES[(offset * num_of_rates) + xk1] = (ALGEBRAIC[(offset * num_of_algebraic) + xk1ss]
- STATES[(offset * num_of_states) + xk1])/ALGEBRAIC[(offset * num_of_algebraic) + txk1];
        RATES[(offset * num_of_rates) + j] = (ALGEBRAIC[(offset * num_of_algebraic) + jss] -
STATES[(offset * num_of_states) + j])/ALGEBRAIC[(offset * num_of_algebraic) + tj];
        RATES[(offset * num_of_rates) + ap] = (ALGEBRAIC[(offset * num_of_algebraic) + assp] -
STATES[(offset * num_of_states) + ap])/ALGEBRAIC[(offset * num_of_algebraic) + ta];
        RATES[(offset * num_of_rates) + fcaf] = (ALGEBRAIC[(offset * num_of_algebraic) + fcass]
- STATES[(offset * num_of_states) + fcaf])/ALGEBRAIC[(offset * num_of_algebraic) + tfcaf];
        RATES[(offset * num_of_rates) + fcas] = (ALGEBRAIC[(offset * num_of_algebraic) + fcass]
- STATES[(offset * num_of_states) + fcas])/ALGEBRAIC[(offset * num_of_algebraic) + tfcas];
        RATES[(offset * num_of_rates) + ffp] = (ALGEBRAIC[(offset * num_of_algebraic) + fss] -
STATES[(offset * num_of_states) + ffp])/ALGEBRAIC[(offset * num_of_algebraic) + tffp];
        RATES[(offset * num_of_rates) + xs2] = (ALGEBRAIC[(offset * num_of_algebraic) + xs2ss]
- STATES[(offset * num_of_states) + xs2])/ALGEBRAIC[(offset * num_of_algebraic) + txs2];
        RATES[(offset * num_of_rates) + CaMKt] =  CONSTANTS[(offset * num_of_constants) +
aCaMK]*ALGEBRAIC[(offset * num_of_algebraic) + CaMKb]*(ALGEBRAIC[(offset * num_of_algebraic) +
CaMKb]+STATES[(offset * num_of_states) + CaMKt]) -  CONSTANTS[(offset * num_of_constants) +
bCaMK]*STATES[(offset * num_of_states) + CaMKt];
        RATES[(offset * num_of_rates) + hsp] = (ALGEBRAIC[(offset * num_of_algebraic) + hssp]
- STATES[(offset * num_of_states) + hsp])/ALGEBRAIC[(offset * num_of_algebraic) + thsp];
        RATES[(offset * num_of_rates) + jp] = (ALGEBRAIC[(offset * num_of_algebraic) + jss] -
STATES[(offset * num_of_states) + jp])/ALGEBRAIC[(offset * num_of_algebraic) + tjp];
        RATES[(offset * num_of_rates) + mL] = (ALGEBRAIC[(offset * num_of_algebraic) + mLss] -
STATES[(offset * num_of_states) + mL])/ALGEBRAIC[(offset * num_of_algebraic) + tmL];
        RATES[(offset * num_of_rates) + fcafp] = (ALGEBRAIC[(offset * num_of_algebraic) + fcass]
- STATES[(offset * num_of_states) + fcafp])/ALGEBRAIC[(offset * num_of_algebraic) + tfcafp];
        RATES[(offset * num_of_rates) + iF] = (ALGEBRAIC[(offset * num_of_algebraic) + iss] -
STATES[(offset * num_of_states) + iF])/ALGEBRAIC[(offset * num_of_algebraic) + tiF];
```

```
        RATES[(offset * num_of_rates) + iS] = (ALGEBRAIC[(offset * num_of_algebraic) + iss] -
STATES[(offset * num_of_states) + iS])/ALGEBRAIC[(offset * num_of_algebraic) + tiS];
        RATES[(offset * num_of_rates) + iFp] = (ALGEBRAIC[(offset * num_of_algebraic) + iss] -
STATES[(offset * num_of_states) + iFp])/ALGEBRAIC[(offset * num_of_algebraic) + tiFp];
        RATES[(offset * num_of_rates) + iSp] = (ALGEBRAIC[(offset * num_of_algebraic) + iss] -
STATES[(offset * num_of_states) + iSp])/ALGEBRAIC[(offset * num_of_algebraic) + tiSp];
        RATES[(offset * num_of_rates) + Jrelnp] = (ALGEBRAIC[(offset * num_of_algebraic) +
Jrel_inf] - STATES[(offset * num_of_states) + Jrelnp])/ALGEBRAIC[(offset * num_of_algebraic) +
tau_rel];
        RATES[(offset * num_of_rates) + Jrelp] = (ALGEBRAIC[(offset * num_of_algebraic) +
Jrel_infp] - STATES[(offset * num_of_states) + Jrelp])/ALGEBRAIC[(offset * num_of_algebraic) +
tau_relp];
        RATES[(offset * num_of_rates) + ki] = ( - ((ALGEBRAIC[(offset * num_of_algebraic) +
Ito]+ALGEBRAIC[(offset * num_of_algebraic) + IKr]+ALGEBRAIC[(offset * num_of_algebraic) +
IKs]+ALGEBRAIC[(offset * num_of_algebraic) + IK1]+ALGEBRAIC[(offset * num_of_algebraic) +
IKb]+ALGEBRAIC[(offset * num_of_algebraic) + Istim]) - 2.00000*ALGEBRAIC[(offset *
num_of_algebraic) + INaK])*CONSTANTS[(offset * num_of_constants) + cm]*CONSTANTS[(offset *
num_of_constants) + Acap])/( CONSTANTS[(offset * num_of_constants) + F]*CONSTANTS[(offset *
num_of_constants) + vmyo])+( ALGEBRAIC[(offset * num_of_algebraic) + JdiffK]*CONSTANTS[(offset
* num_of_constants) + vss])/CONSTANTS[(offset * num_of_constants) + vmyo];
        RATES[(offset * num_of_rates) + kss] = ( - ALGEBRAIC[(offset * num_of_algebraic) +
ICaK]*CONSTANTS[(offset * num_of_constants) + cm]*CONSTANTS[(offset * num_of_constants) +
Acap])/( CONSTANTS[(offset * num_of_constants) + F]*CONSTANTS[(offset * num_of_constants) + vss])
- ALGEBRAIC[(offset * num_of_algebraic) + JdiffK];
        RATES[(offset * num_of_rates) + nai] = ( - (ALGEBRAIC[(offset * num_of_algebraic) +
INa]+ALGEBRAIC[(offset * num_of_algebraic) + INaL]+ 3.00000*ALGEBRAIC[(offset * num_of_algebraic)
+ INaCa_i]+ 3.00000*ALGEBRAIC[(offset * num_of_algebraic) + INaK]+ALGEBRAIC[(offset *
num_of_algebraic) + INab])*CONSTANTS[(offset * num_of_constants) + Acap]*CONSTANTS[(offset *
num_of_constants) + cm])/( CONSTANTS[(offset * num_of_constants) + F]*CONSTANTS[(offset *
num_of_constants) + vmyo])+( ALGEBRAIC[(offset * num_of_algebraic) + JdiffNa]*CONSTANTS[(offset
* num_of_constants) + vss])/CONSTANTS[(offset * num_of_constants) + vmyo];
        RATES[(offset * num_of_rates) + nass] = ( - (ALGEBRAIC[(offset * num_of_algebraic) +
ICaNa]+ 3.00000*ALGEBRAIC[(offset * num_of_algebraic) + INaCa_ss])*CONSTANTS[(offset *
num_of_constants) + cm]*CONSTANTS[(offset * num_of_constants) + Acap])/( CONSTANTS[(offset *
num_of_constants) + F]*CONSTANTS[(offset * num_of_constants) + vss]) - ALGEBRAIC[(offset *
num_of_algebraic) + JdiffNa];
        RATES[(offset * num_of_rates) + V] = - (ALGEBRAIC[(offset * num_of_algebraic) +
INa]+ALGEBRAIC[(offset * num_of_algebraic) + INaL]+ALGEBRAIC[(offset * num_of_algebraic) +
Ito]+ALGEBRAIC[(offset * num_of_algebraic) + ICaL]+ALGEBRAIC[(offset * num_of_algebraic) +
ICaNa]+ALGEBRAIC[(offset * num_of_algebraic) + ICaK]+ALGEBRAIC[(offset * num_of_algebraic) +
IKr]+ALGEBRAIC[(offset * num_of_algebraic) + IKs]+ALGEBRAIC[(offset * num_of_algebraic) +
IK1]+ALGEBRAIC[(offset * num_of_algebraic) + INaCa_i]+ALGEBRAIC[(offset * num_of_algebraic) +
INaCa_ss]+ALGEBRAIC[(offset * num_of_algebraic) + INaK]+ALGEBRAIC[(offset * num_of_algebraic) +
INab]+ALGEBRAIC[(offset * num_of_algebraic) + IKb]+ALGEBRAIC[(offset * num_of_algebraic) +
IpCa]+ALGEBRAIC[(offset * num_of_algebraic) + ICab]+ALGEBRAIC[(offset * num_of_algebraic) +
Istim]);
        RATES[(offset * num_of_rates) + cass] = ALGEBRAIC[(offset * num_of_algebraic) +
Bcass]*((( - (ALGEBRAIC[(offset * num_of_algebraic) + ICaL] - 2.00000*ALGEBRAIC[(offset *
num_of_algebraic) + INaCa_ss])*CONSTANTS[(offset * num_of_constants) + cm]*CONSTANTS[(offset *
```

```
num_of_constants)    +    Acap])/(    2.00000*CONSTANTS[(offset    *    num_of_constants)    +
F]*CONSTANTS[(offset  *  num_of_constants)  +  vss])+(  ALGEBRAIC[(offset  *  num_of_algebraic)  +
Jrel]*CONSTANTS[(offset  *  num_of_constants)  +  vjsr])/CONSTANTS[(offset  *  num_of_constants)  +
vss]) - ALGEBRAIC[(offset * num_of_algebraic) + Jdiff]);
        RATES[(offset  *  num_of_rates)  +  cai]  =   ALGEBRAIC[(offset  *  num_of_algebraic)  +
Bcai]*((( - ((ALGEBRAIC[(offset * num_of_algebraic) + IpCa]+ALGEBRAIC[(offset * num_of_algebraic)
+  ICab])  -   2.00000*ALGEBRAIC[(offset  *  num_of_algebraic)  +  INaCa_i])*CONSTANTS[(offset  *
num_of_constants)       +       cm]*CONSTANTS[(offset       *       num_of_constants)       +
Acap])/( 2.00000*CONSTANTS[(offset * num_of_constants) + F]*CONSTANTS[(offset * num_of_constants)
+ vmyo]) - ( ALGEBRAIC[(offset * num_of_algebraic) + Jup]*CONSTANTS[(offset * num_of_constants)
+ vnsr])/CONSTANTS[(offset * num_of_constants) + vmyo])+( ALGEBRAIC[(offset * num_of_algebraic)
+ Jdiff]*CONSTANTS[(offset * num_of_constants) + vss])/CONSTANTS[(offset * num_of_constants) +
vmyo]);
        RATES[(offset * num_of_rates) + cansr] = ALGEBRAIC[(offset * num_of_algebraic) + Jup]
-  (  ALGEBRAIC[(offset  *  num_of_algebraic)  +  Jtr]*CONSTANTS[(offset  *  num_of_constants)  +
vjsr])/CONSTANTS[(offset * num_of_constants) + vnsr];
        RATES[(offset  *  num_of_rates)  +  cajsr]  =   ALGEBRAIC[(offset  *  num_of_algebraic)  +
Bcajsr]*(ALGEBRAIC[(offset * num_of_algebraic) + Jtr] - ALGEBRAIC[(offset * num_of_algebraic) +
Jrel]);
        }

        __device__ void solveEuler( double *STATES, double *RATES, double dt, int offset){

          int num_of_states = 41;
          int num_of_rates = 41;

          STATES[(offset  *  num_of_states)  +V]  =  STATES[(offset  *  num_of_states)  +  V]  +
RATES[(offset * num_of_rates) + V] * dt;
          STATES[(offset * num_of_states) + CaMKt] = STATES[(offset * num_of_states) + CaMKt]
+ RATES[(offset * num_of_rates) + CaMKt] * dt;
          STATES[(offset * num_of_states) + cass] = STATES[(offset * num_of_states) + cass] +
RATES[(offset * num_of_rates) + cass] * dt;
          STATES[(offset * num_of_states) + nai] = STATES[(offset * num_of_states) + nai] +
RATES[(offset * num_of_rates) + nai] * dt;
          STATES[(offset * num_of_states) + nass] = STATES[(offset * num_of_states) + nass] +
RATES[(offset * num_of_rates) + nass] * dt;
          STATES[(offset  *  num_of_states)  +  ki]  =  STATES[(offset  *  num_of_states)  +  ki]  +
RATES[(offset * num_of_rates) + ki] * dt;
          STATES[(offset * num_of_states) + kss] = STATES[(offset * num_of_states) + kss] +
RATES[(offset * num_of_rates) + kss] * dt;
          STATES[(offset * num_of_states) + cansr] = STATES[(offset * num_of_states) + cansr]
+ RATES[(offset * num_of_rates) + cansr] * dt;
          STATES[(offset * num_of_states) + cajsr] = STATES[(offset * num_of_states) + cajsr]
+ RATES[(offset * num_of_rates) + cajsr] * dt;
          STATES[(offset * num_of_states) + cai] = STATES[(offset * num_of_states) + cai] +
RATES[(offset * num_of_rates) + cai] * dt;
          STATES[(offset  *  num_of_states)  +  m]  =  STATES[(offset  *  num_of_states)  +  m]  +
RATES[(offset * num_of_rates) + m] * dt;
```

```
          STATES[(offset * num_of_states) + hf] = STATES[(offset * num_of_states) + hf] +
RATES[(offset * num_of_rates) + hf] * dt;
          STATES[(offset * num_of_states) + hs] = STATES[(offset * num_of_states) + hs] +
RATES[(offset * num_of_rates) + hs] * dt;
          STATES[(offset * num_of_states) + j] = STATES[(offset * num_of_states) + j] +
RATES[(offset * num_of_rates) + j] * dt;
          STATES[(offset * num_of_states) + hsp] = STATES[(offset * num_of_states) + hsp] +
RATES[(offset * num_of_rates) + hsp] * dt;
          STATES[(offset * num_of_states) + jp] = STATES[(offset * num_of_states) + jp] +
RATES[(offset * num_of_rates) + jp] * dt;
          STATES[(offset * num_of_states) + mL] = STATES[(offset * num_of_states) + mL] +
RATES[(offset * num_of_rates) + mL] * dt;
          STATES[(offset * num_of_states) + hL] = STATES[(offset * num_of_states) + hL] +
RATES[(offset * num_of_rates) + hL] * dt;
          STATES[(offset * num_of_states) + hLp] = STATES[(offset * num_of_states) + hLp] +
RATES[(offset * num_of_rates) + hLp] * dt;
          STATES[(offset * num_of_states) + a] = STATES[(offset * num_of_states) + a] +
RATES[(offset * num_of_rates) + a] * dt;
          STATES[(offset * num_of_states) + iF] = STATES[(offset * num_of_states) + iF] +
RATES[(offset * num_of_rates) + iF] * dt;
          STATES[(offset * num_of_states) + iS] = STATES[(offset * num_of_states) + iS] +
RATES[(offset * num_of_rates) + iS] * dt;
          STATES[(offset * num_of_states) + ap] = STATES[(offset * num_of_states) + ap] +
RATES[(offset * num_of_rates) + ap] * dt;
          STATES[(offset * num_of_states) + iFp] = STATES[(offset * num_of_states) + iFp] +
RATES[(offset * num_of_rates) + iFp] * dt;
          STATES[(offset * num_of_states) + iSp] = STATES[(offset * num_of_states) + iSp] +
RATES[(offset * num_of_rates) + iSp] * dt;
          STATES[(offset * num_of_states) + d] = STATES[(offset * num_of_states) + d] +
RATES[(offset * num_of_rates) + d] * dt;
          STATES[(offset * num_of_states) + ff] = STATES[(offset * num_of_states) + ff] +
RATES[(offset * num_of_rates) + ff] * dt;
          STATES[(offset * num_of_states) + fs] = STATES[(offset * num_of_states) + fs] +
RATES[(offset * num_of_rates) + fs] * dt;
          STATES[(offset * num_of_states) + fcaf] = STATES[(offset * num_of_states) + fcaf] +
RATES[(offset * num_of_rates) + fcaf] * dt;
          STATES[(offset * num_of_states) + fcas] = STATES[(offset * num_of_states) + fcas] +
RATES[(offset * num_of_rates) + fcas] * dt;
          STATES[(offset * num_of_states) + jca] = STATES[(offset * num_of_states) + jca] +
RATES[(offset * num_of_rates) + jca] * dt;
          STATES[(offset * num_of_states) + ffp] = STATES[(offset * num_of_states) + ffp] +
RATES[(offset * num_of_rates) + ffp] * dt;
          STATES[(offset * num_of_states) + fcafp] = STATES[(offset * num_of_states) + fcafp]
+ RATES[(offset * num_of_rates) + fcafp] * dt;
          STATES[(offset * num_of_states) + nca] = STATES[(offset * num_of_states) + nca] +
RATES[(offset * num_of_rates) + nca] * dt;
          STATES[(offset * num_of_states) + xrf] = STATES[(offset * num_of_states) + xrf] +
RATES[(offset * num_of_rates) + xrf] * dt;
```

```
          STATES[(offset * num_of_states) + xrs] = STATES[(offset * num_of_states) + xrs] +
RATES[(offset * num_of_rates) + xrs] * dt;
          STATES[(offset * num_of_states) + xs1] = STATES[(offset * num_of_states) + xs1] +
RATES[(offset * num_of_rates) + xs1] * dt;
          STATES[(offset * num_of_states) + xs2] = STATES[(offset * num_of_states) + xs2] +
RATES[(offset * num_of_rates) + xs2] * dt;
          STATES[(offset * num_of_states) + xk1] = STATES[(offset * num_of_states) + xk1] +
RATES[(offset * num_of_rates) + xk1] * dt;
          STATES[(offset * num_of_states) + Jrelnp] = STATES[(offset * num_of_states) + Jrelnp]
+ RATES[(offset * num_of_rates) + Jrelnp] * dt;
          STATES[(offset * num_of_states) + Jrelp] = STATES[(offset * num_of_states) + Jrelp]
+ RATES[(offset * num_of_rates) + Jrelp] * dt;
        }

        __device__ void solveAnalytical(double *CONSTANTS, double *STATES, double *ALGEBRAIC,
double *RATES, double dt, int offset)
        {
          int num_of_constants = 145;
          int num_of_states = 41;
          int num_of_algebraic = 199;
          int num_of_rates = 41;

        // #ifdef EULER // moved as its own function

        // #else
        ////==============
          ////Exact solution
          ////==============
          ////INa
          STATES[(offset * num_of_states) + m] = ALGEBRAIC[(offset * num_of_algebraic) + mss]
- (ALGEBRAIC[(offset * num_of_algebraic) + mss] - STATES[(offset * num_of_states) + m]) * exp(-
dt / ALGEBRAIC[(offset * num_of_algebraic) + tm]);
          STATES[(offset * num_of_states) + hf] = ALGEBRAIC[(offset * num_of_algebraic) + hss]
- (ALGEBRAIC[(offset * num_of_algebraic) + hss] - STATES[(offset * num_of_states) + hf]) * exp(-
dt / ALGEBRAIC[(offset * num_of_algebraic) + thf]);
          STATES[(offset * num_of_states) + hs] = ALGEBRAIC[(offset * num_of_algebraic) + hss]
- (ALGEBRAIC[(offset * num_of_algebraic) + hss] - STATES[(offset * num_of_states) + hs]) * exp(-
dt / ALGEBRAIC[(offset * num_of_algebraic) + ths]);
          STATES[(offset * num_of_states) + j] = ALGEBRAIC[(offset * num_of_algebraic) + jss]
- (ALGEBRAIC[(offset * num_of_algebraic) + jss] - STATES[(offset * num_of_states) + j]) * exp(-
dt / ALGEBRAIC[(offset * num_of_algebraic) + tj]);
          STATES[(offset * num_of_states) + hsp] = ALGEBRAIC[(offset * num_of_algebraic) + hssp]
- (ALGEBRAIC[(offset * num_of_algebraic) + hssp] - STATES[(offset * num_of_states) + hsp]) *
exp(-dt / ALGEBRAIC[(offset * num_of_algebraic) + thsp]);
          STATES[(offset * num_of_states) + jp] = ALGEBRAIC[(offset * num_of_algebraic) + jss]
- (ALGEBRAIC[(offset * num_of_algebraic) + jss] - STATES[(offset * num_of_states) + jp]) * exp(-
dt / ALGEBRAIC[(offset * num_of_algebraic) + tjp]);
```

```
        STATES[(offset * num_of_states) + mL] = ALGEBRAIC[(offset * num_of_algebraic) + mLss]
- (ALGEBRAIC[(offset * num_of_algebraic) + mLss] - STATES[(offset * num_of_states) + mL]) *
exp(-dt / ALGEBRAIC[(offset * num_of_algebraic) + tmL]);
        STATES[(offset * num_of_states) + hL] = ALGEBRAIC[(offset * num_of_algebraic) + hLss]
- (ALGEBRAIC[(offset * num_of_algebraic) + hLss] - STATES[(offset * num_of_states) + hL]) *
exp(-dt / CONSTANTS[(offset * num_of_constants) + thL]);
        STATES[(offset * num_of_states) + hLp] = ALGEBRAIC[(offset * num_of_algebraic) + hLssp]
- (ALGEBRAIC[(offset * num_of_algebraic) + hLssp] - STATES[(offset * num_of_states) + hLp]) *
exp(-dt / CONSTANTS[(offset * num_of_constants) + thLp]);
        ////Ito
        STATES[(offset * num_of_states) + a] = ALGEBRAIC[(offset * num_of_algebraic) + ass]
- (ALGEBRAIC[(offset * num_of_algebraic) + ass] - STATES[(offset * num_of_states) + a]) * exp(-
dt / ALGEBRAIC[(offset * num_of_algebraic) + ta]);
        STATES[(offset * num_of_states) + iF] = ALGEBRAIC[(offset * num_of_algebraic) + iss]
- (ALGEBRAIC[(offset * num_of_algebraic) + iss] - STATES[(offset * num_of_states) + iF]) * exp(-
dt / ALGEBRAIC[(offset * num_of_algebraic) + tiF]);
        STATES[(offset * num_of_states) + iS] = ALGEBRAIC[(offset * num_of_algebraic) + iss]
- (ALGEBRAIC[(offset * num_of_algebraic) + iss] - STATES[(offset * num_of_states) + iS]) * exp(-
dt / ALGEBRAIC[(offset * num_of_algebraic) + tiS]);
        STATES[(offset * num_of_states) + ap] = ALGEBRAIC[(offset * num_of_algebraic) + assp]
- (ALGEBRAIC[(offset * num_of_algebraic) + assp] - STATES[(offset * num_of_states) + ap]) *
exp(-dt / ALGEBRAIC[(offset * num_of_algebraic) + ta]);
        STATES[(offset * num_of_states) + iFp] = ALGEBRAIC[(offset * num_of_algebraic) + iss]
- (ALGEBRAIC[(offset * num_of_algebraic) + iss] - STATES[(offset * num_of_states) + iFp]) *
exp(-dt / ALGEBRAIC[(offset * num_of_algebraic) + tiFp]);
        STATES[(offset * num_of_states) + iSp] = ALGEBRAIC[(offset * num_of_algebraic) + iss]
- (ALGEBRAIC[(offset * num_of_algebraic) + iss] - STATES[(offset * num_of_states) + iSp]) *
exp(-dt / ALGEBRAIC[(offset * num_of_algebraic) + tiSp]);
        ////ICaL
        STATES[(offset * num_of_states) + d] = ALGEBRAIC[(offset * num_of_algebraic) + dss]
- (ALGEBRAIC[(offset * num_of_algebraic) + dss] - STATES[(offset * num_of_states) + d]) * exp(-
dt / ALGEBRAIC[(offset * num_of_algebraic) + td]);
        STATES[(offset * num_of_states) + ff] = ALGEBRAIC[(offset * num_of_algebraic) + fss]
- (ALGEBRAIC[(offset * num_of_algebraic) + fss] - STATES[(offset * num_of_states) + ff]) * exp(-
dt / ALGEBRAIC[(offset * num_of_algebraic) + tff]);
        STATES[(offset * num_of_states) + fs] = ALGEBRAIC[(offset * num_of_algebraic) + fss]
- (ALGEBRAIC[(offset * num_of_algebraic) + fss] - STATES[(offset * num_of_states) + fs]) * exp(-
dt / ALGEBRAIC[(offset * num_of_algebraic) + tfs]);
        STATES[(offset * num_of_states) + fcaf] = ALGEBRAIC[(offset * num_of_algebraic) +
fcass] - (ALGEBRAIC[(offset * num_of_algebraic) + fcass] - STATES[(offset * num_of_states) +
fcaf]) * exp(-dt / ALGEBRAIC[(offset * num_of_algebraic) + tfcaf]);
        STATES[(offset * num_of_states) + fcas] = ALGEBRAIC[(offset * num_of_algebraic) +
fcass] - (ALGEBRAIC[(offset * num_of_algebraic) + fcass] - STATES[(offset * num_of_states) +
fcas]) * exp(-dt / ALGEBRAIC[(offset * num_of_algebraic) + tfcas]);
        STATES[(offset * num_of_states) + jca] = ALGEBRAIC[(offset * num_of_algebraic) + fcass]
- (ALGEBRAIC[(offset * num_of_algebraic) + fcass] - STATES[(offset * num_of_states) + jca]) *
exp(- dt / CONSTANTS[(offset * num_of_constants) + tjca]);
```

```
        STATES[(offset * num_of_states) + ffp] = ALGEBRAIC[(offset * num_of_algebraic) + fss]
- (ALGEBRAIC[(offset * num_of_algebraic) + fss] - STATES[(offset * num_of_states) + ffp]) *
exp(-dt / ALGEBRAIC[(offset * num_of_algebraic) + tffp]);
        STATES[(offset * num_of_states) + fcafp] = ALGEBRAIC[(offset * num_of_algebraic) +
fcass] - (ALGEBRAIC[(offset * num_of_algebraic) + fcass] - STATES[(offset * num_of_states) +
fcafp]) * exp(-d / ALGEBRAIC[(offset * num_of_algebraic) + tfcafp]);
        STATES[(offset * num_of_states) + nca] = ALGEBRAIC[(offset * num_of_algebraic) + anca]
* CONSTANTS[(offset * num_of_constants) + k2n] / ALGEBRAIC[(offset * num_of_algebraic) + km2n]
- (ALGEBRAIC[(offset * num_of_algebraic) + anca] * CONSTANTS[(offset * num_of_constants) + k2n]
/ ALGEBRAIC[(offset * num_of_algebraic) + km2n] - STATES[(offset * num_of_states) + nca]) *
exp(-ALGEBRAIC[(offset * num_of_algebraic) + km2n] * dt);
        ////IKr
        STATES[(offset * num_of_states) + xrf] = ALGEBRAIC[(offset * num_of_algebraic) + xrss]
- (ALGEBRAIC[(offset * num_of_algebraic) + xrss] - STATES[(offset * num_of_states) + xrf]) *
exp(-dt / ALGEBRAIC[(offset * num_of_algebraic) + txrf]);
        STATES[(offset * num_of_states) + xrs] = ALGEBRAIC[(offset * num_of_algebraic) + xrss]
- (ALGEBRAIC[(offset * num_of_algebraic) + xrss] - STATES[(offset * num_of_states) + xrs]) *
exp(-dt / ALGEBRAIC[(offset * num_of_algebraic) + txrs]);
        ////IKs
        STATES[(offset * num_of_states) + xs1] = ALGEBRAIC[(offset * num_of_algebraic) + xs1ss]
- (ALGEBRAIC[(offset * num_of_algebraic) + xs1ss] - STATES[(offset * num_of_states) + xs1]) *
exp(-dt / ALGEBRAIC[(offset * num_of_algebraic) + txs1]);
        STATES[(offset * num_of_states) + xs2] = ALGEBRAIC[(offset * num_of_algebraic) + xs2ss]
- (ALGEBRAIC[(offset * num_of_algebraic) + xs2ss] - STATES[(offset * num_of_states) + xs2]) *
exp(-dt / ALGEBRAIC[(offset * num_of_algebraic) + txs2]);
        ////IK1
        STATES[(offset * num_of_states) + xk1] = ALGEBRAIC[(offset * num_of_algebraic) + xk1ss]
- (ALGEBRAIC[(offset * num_of_algebraic) + xk1ss] - STATES[(offset * num_of_states) + xk1]) *
exp(-dt / ALGEBRAIC[(offset * num_of_algebraic) + txk1]);
        ////INaCa
        ////INaK
        ////IKb
        ////INab
        ////ICab
        ///IpCa
        ////Diffusion fluxes
        ////RyR receptors
        STATES[(offset * num_of_states) + Jrelnp] = ALGEBRAIC[(offset * num_of_algebraic) +
Jrel_inf] - (ALGEBRAIC[(offset * num_of_algebraic) + Jrel_inf] - STATES[(offset * num_of_states)
+ Jrelnp]) * exp(-dt / ALGEBRAIC[(offset * num_of_algebraic) + tau_rel]);
        STATES[(offset * num_of_states) + Jrelp] = ALGEBRAIC[(offset * num_of_algebraic) +
Jrel_infp] - (ALGEBRAIC[(offset * num_of_algebraic) + Jrel_infp] - STATES[(offset * num_of_states)
+ Jrelp]) * exp(-dt / ALGEBRAIC[(offset * num_of_algebraic) + tau_relp]);
        ////SERCA Pump
        ////Calcium translocation
        //
        ////============================
        ////Approximated solution (Euler)
        ////============================
```

```
          ////ICaL
          //STATES[jca] = STATES[jca] + RATES[jca] * dt;
          ////CaMK
          STATES[(offset * num_of_states) + CaMKt] = STATES[(offset * num_of_states) + CaMKt]
+ RATES[(offset * num_of_rates) + CaMKt] * dt;
          ////Membrane potential
          STATES[(offset * num_of_states) + V] = STATES[(offset * num_of_states) + V] +
RATES[(offset * num_of_rates) + V] * dt;
          ////Ion Concentrations and Buffers
          STATES[(offset * num_of_states) + nai] = STATES[(offset * num_of_states) + nai] +
RATES[(offset * num_of_rates) + nai] * dt;
          STATES[(offset * num_of_states) + nass] = STATES[(offset * num_of_states) + nass] +
RATES[(offset * num_of_rates) + nass] * dt;
          STATES[(offset * num_of_states) + ki] = STATES[(offset * num_of_states) + ki] +
RATES[(offset * num_of_rates) + ki] * dt;
          STATES[(offset * num_of_states) + kss] = STATES[(offset * num_of_states) + kss] +
RATES[(offset * num_of_rates) + kss] * dt;
          STATES[(offset * num_of_states) + cai] = STATES[(offset * num_of_states) + cai] +
RATES[(offset * num_of_rates) + cai] * dt;
          STATES[(offset * num_of_states) + cass] = STATES[(offset * num_of_states) + cass] +
RATES[(offset * num_of_rates) + cass] * dt;
          STATES[(offset * num_of_states) + cansr] = STATES[(offset * num_of_states) + cansr]
+ RATES[(offset * num_of_rates) + cansr] * dt;
          STATES[(offset * num_of_states) + cajsr] = STATES[(offset * num_of_states) + cajsr]
+ RATES[(offset * num_of_rates) + cajsr] * dt;
        // #endif
        }

        __device__ double set_time_step(double TIME,
          double time_point,
          double max_time_step,
          double *CONSTANTS,
          double *RATES,
          double *STATES,
          double *ALGEBRAIC,
          int offset) {
          double time_step = 0.005;
          int num_of_constants = 145;
          int num_of_rates = 41;

          if (TIME <= time_point || (TIME - floor(TIME / CONSTANTS[BCL + (offset *
num_of_constants)]) * CONSTANTS[BCL + (offset * num_of_constants)]) <= time_point) {
            //printf("TIME <= time_point ms\n");
            return time_step;
            //printf("dV = %lf, time_step = %lf\n",RATES[V] * time_step, time_step);
          }
          else {
            //printf("TIME > time_point ms\n");
```

```
            if (std::abs(RATES[V + (offset * num_of_rates)] * time_step) <= 0.2) {//Slow changes
in V
                // printf("dV/dt <= 0.2\n");
                time_step = std::abs(0.8 / RATES[V + (offset * num_of_rates)]);
                //Make sure time_step is between 0.005 and max_time_step
                if (time_step < 0.005) {
                    time_step = 0.005;
                }
                else if (time_step > max_time_step) {
                    time_step = max_time_step;
                }
                //printf("dV = %lf, time_step = %lf\n",std::abs(RATES[V] * time_step),
time_step);
            }
            else if (std::abs(RATES[V + (offset * num_of_rates)] * time_step) >= 0.8) {//Fast
changes in V
                // printf("dV/dt >= 0.8\n");
                time_step = std::abs(0.2 / RATES[V + (offset * num_of_rates)]);
                while (std::abs(RATES[V + (offset * num_of_rates)] * time_step) >= 0.8 &&
                       0.005 < time_step &&
                       time_step < max_time_step) {
                    time_step = time_step / 10.0;
                    // printf("dV = %lf, time_step = %lf\n",std::abs(RATES[V] * time_step),
time_step);
                }
            }
            // __syncthreads();
            return time_step;
          }
        }
```

c. Cellmodel.hpp

This file contains the general interface for the cell models. It declares common functions that all cellmodel should have. Cellmodel.hpp implemented as below:

```
        #ifndef CELL_HPP
        #define CELL_HPP

        class Cellmodel
        {
        protected:
          Cellmodel(){}
        public:
          unsigned short algebraic_size;
          unsigned short constants_size;
          unsigned short states_size;
          unsigned short gates_size;
```

```
        unsigned short current_size;
        unsigned short concs_size;
        double ALGEBRAIC[255];
        double CONSTANTS[255];
        double RATES[255];
        double STATES[255];
        char gates_header[255];
        unsigned short gates_indices[255];
        char current_header[255];
        unsigned short current_indices[255];
        char concs_header[255];
        unsigned short concs_indices[255];
        virtual ~Cellmodel() {}
        virtual void initConsts() = 0;
        virtual void initConsts(double type){}
        virtual void initConsts(double type, double conc, double *hill){}
        virtual void initConsts(double type, double conc, double *hill, bool is_dutta){}
        virtual  void  computeRates(double  TIME,  double  *CONSTANTS,  double  *RATES,  double
*STATES, double *ALGEBRAIC) = 0;
        virtual void solveAnalytical(double dt) {};
      };
      #endif
```

d. enums/enum_Ohara_rudy_2011.hpp

This header act as a 'translation dictionary' for each variable in the cell model file. Enumeration required to easily track each variable, so instead of looking at numbers, I looked at pre-defined variable names corresponds to each correct values in the cell model. The enumeration follows CellML's description of each value and their function, as in the list below:

```
    #ifndef
EN_OHARA_RUDY_2011_HPP
    #define
EN_OHARA_RUDY_2011_HPP

    enum E_ALGEBRAIC_T{
      vffrt = 29,
      vfrt = 39,
      INa = 62,
      INaL = 64,
      Ito = 70,
      ICaL = 81,
      ICaNa = 82,
      ICaK = 85,
      IKr = 96,
      IKs = 98,
      IK1 = 100,
      INaCa_i = 132,
      INaCa_ss = 162,
      INaK = 181,
      tjp = 41,
      fINap = 61,
      mLss = 32,
      tmL = 42,
      hLss = 2,
      hLssp = 3,
      fINaLp = 63,
      ass = 4,
      ta = 17,
      iss = 5,
      delta_epi = 18,
      tiF_b = 33,
      tiS_b = 43,
      tiF = 46,
      tiS = 48,
      AiF = 65,
      AiS = 66,
      i = 67,
      assp = 34,
      dti_develop = 50,
      Axrf = 88,
      Axrs = 91,
      xr = 94,
      rkr = 95,
      xs1ss = 10,
      xs2ss = 26,
      txs1 = 27,
      KsCa = 97,
      txs2 = 38,
      xk1ss = 11,
      txk1 = 28,
      rk1 = 99,
      hna = 102,
      hca = 101,
      h1_i = 103,
      h2_i = 104,
      h3_i = 105,
      h4_i = 106,
      h5_i = 107,
      h6_i = 108,
```

```
INab = 184,
IKb = 183,
IpCa = 188,
ICab = 186,
Istim = 12,
CaMKb = 45,
CaMKa = 47,
JdiffNa = 187,
Jdiff = 189,
Jup = 196,
JdiffK = 185,
Jrel = 191,
Jtr = 197,
Bcai = 49,
Bcajsr = 53,
Bcass = 51,
ENa = 56,
EK = 57,
EKs = 58,
mss = 0,
tm = 13,
hss = 1,
thf = 14,
ths = 15,
h = 59,
jss = 16,
tj = 30,
hssp = 31,
thsp = 40,
hp = 60,
h8_ss = 140,
h9_ss = 141,
k3p_ss = 142,
k3pp_ss = 143,
k3_ss = 144,
k4_ss = 147,
k4p_ss = 145,
k4pp_ss = 146,
k6_ss = 148,
k7_ss = 149,
k8_ss = 150,
x1_ss = 151,
x2_ss = 152,
x3_ss = 153,
x4_ss = 154,
E1_ss = 155,
E2_ss = 156,
E3_ss = 157,
E4_ss = 158,
allo_ss = 159,
JncxNa_ss = 160,
JncxCa_ss = 161,
Knai = 163,
Knao = 164,
P = 165,
a1 = 166,
b2 = 167,
a3 = 168,
b3 = 169,
b4 = 170,
x1 = 171,
x2 = 172,
x3 = 173,
x4 = 174,
E1 = 175,
E2 = 176,
E3 = 177,
E4 = 178,
JnakNa = 179,
JnakK = 180,
xkb = 182,
Jrel_inf = 86,
tau_rel = 92,
dti_recover = 52,
tiFp = 54,
tiSp = 55,
ip = 68,
fItop = 69,
dss = 6,
fss = 7,
f = 71,
fcass = 19,
Afcaf = 72,
Afcas = 73,
fca = 74,
fp = 75,
fcap = 76,
km2n = 8,
anca = 20,
PhiCaL = 77,
PhiCaNa = 78,
PhiCaK = 79,
fICaLp = 80,
td = 21,
tff = 22,
tfs = 23,
tfcaf = 35,
tfcas = 36,
tffp = 37,
tfcafp = 44,
xrss = 9,
txrf = 24,
txrs = 25,
    vmyo = 118,
    vnsr = 119,
    vjsr = 120,
    vss = 121,
    amp = 12,
    duration = 13,
    KmCaMK = 14,
    aCaMK = 15,
    bCaMK = 16,
    CaMKo = 17,
    KmCaM = 18,
    cmdnmax_b = 19,
    cmdnmax = 93,
    kmcmdn = 20,
    trpnmax = 21,
    kmtrpn = 22,
    BSRmax = 23,
    KmBSR = 24,
    BSLmax = 25,
    KmBSL = 26,
    csqnmax = 27,
    kmcsqn = 28,
    cm = 29,
    PKNa = 30,
    mssV1 = 31,
    mssV2 = 32,
    mtV1 = 33,
    mtV2 = 34,
    mtD1 = 35,
    mtD2 = 36,
    mtV3 = 37,
    mtV4 = 38,
    hssV1 = 39,
    hssV2 = 40,
    Ahs = 94,
    Ahf = 41,
    GNa = 42,
    thL = 43,
    thLp = 95,
    GNaL_b = 44,
    GNaL = 96,
    Gto_b = 45,
    Gto = 97,
h7_i = 109,
h8_i = 110,
h9_i = 111,
k3p_i = 112,
k3pp_i = 113,
k3_i = 114,
k4_i = 117,
k4p_i = 115,
k4pp_i = 116,
k6_i = 118,
k7_i = 119,
k8_i = 120,
x1_i = 121,
x2_i = 122,
x3_i = 123,
x4_i = 124,
E1_i = 125,
E2_i = 126,
E3_i = 127,
E4_i = 128,
allo_i = 129,
JncxNa_i = 130,
JncxCa_i = 131,
h1_ss = 133,
h2_ss = 134,
h3_ss = 135,
h4_ss = 136,
h5_ss = 137,
h6_ss = 138,
h7_ss = 139,
    Gncx_b = 64,
    Gncx = 128,
    h10_i = 122,
    h11_i = 123,
    h12_i = 124,
    k1_i = 125,
    k2_i = 126,
    k5_i = 127,
    h10_ss = 129,
    h11_ss = 130,
    h12_ss = 131,
    k1_ss = 132,
    k2_ss = 133,
    k5_ss = 134,
    k1p = 65,
    k1m = 66,
    k2p = 67,
    k2m = 68,
    k3p = 69,
    k3m = 70,
    k4p = 71,
    k4m = 72,
    Knai0 = 73,
    Knao0 = 74,
    delta = 75,
    Kki = 76,
    Kko = 77,
    MgADP = 78,
    MgATP = 79,
    Kmgatp = 80,
    H = 81,
    eP = 82,
    Khp = 83,
    Knap = 84,
    Kxkur = 85,
    Pnak_b = 86,
    Pnak = 138,
    b1 = 135,
    a2 = 136,
    a4 = 137,
    GKb_b = 87,
    GKb = 105,
    PNab = 88,
```

```
  Jrel_infp = 87,
  Jrel_temp = 84,
  tau_relp = 93,
  Jrel_inf_temp = 83,
  fJrelp = 190,
  tau_rel_temp = 89,
  tau_relp_temp = 90,
  Jupnp = 192,
  Jupp = 193,
  fJupp = 194,
  Jleak = 195,
  step_low = 196,
  step_high = 197,
};
enum E_CONSTANTS_T{
  celltype = 0,
  nao = 1,
  cao = 2,
  ko = 3,
  R = 4,
  T = 5,
  F = 6,
  zna = 7,
  zca = 8,
  zk = 9,
  L = 10,
  rad = 11,
  vcell = 104,
  Ageo = 113,
  Acap = 117,

  cai = 9,
  m = 10,
  hf = 11,
  hs = 12,
  j = 13,
  hsp = 14,
  jp = 15,
  mL = 16,
  hL = 17,
  hLp = 18,
  a = 19,
  Kmn = 46,
  k2n = 47,
  PCa_b = 48,
  Aff = 98,
  Afs = 109,
  PCa = 99,
  PCap = 110,
  PCaNa = 111,
  PCaK = 112,
  PCaNap = 115,
  PCaKp = 116,
  tjca = 100,
  GKr_b = 49,
  GKr = 101,
  GKs_b = 50,
  GKs = 102,
  GK1 = 103,
  GK1_b = 51,
  knal = 52,
  kna2 = 53,
  kna3 = 54,
  kasymm = 55,
  wna = 56,
  wca = 57,
  wnaca = 58,
  kcaon = 59,
  kcaoff = 60,
  qna = 61,
  qca = 62,
  KmCaAct = 63,
iF = 20,
  iS = 21,
  ap = 22,
  iFp = 23,
  iSp = 24,
  d = 25,
  ff = 26,
  fs = 27,
  fcaf = 28,
  fcas = 29,
  jca = 30,
      PCab = 89,
      GpCa = 90,
      KmCap = 91,
      bt = 92,
      a_rel = 106,
      btp = 107,
      a_relp = 114,
      upScale = 108,
      stim_start = 139,
      BCL = 140,
      //stim_end  = 141,

      // Additional constants
for cvar
      Jrel_scale = 141,
      Jup_scale = 142,
      Jtr_scale = 143,
      Jleak_scale = 144,
      //KCaMK_scale = 144,
    };
    enum E_STATES_T{
      V = 0,
      CaMKt = 1,
      cass = 2,
      nai = 3,
      nass = 4,
      ki = 5,
      kss = 6,
      cansr = 7,
      cajsr = 8,
    ffp = 31,
      fcafp = 32,
      nca = 33,
      xrf = 34,
      xrs = 35,
      xs1 = 36,
      xs2 = 37,
      xk1 = 38,
      Jrelnp = 39,
      Jrelp = 40,
    };

    #endif
```

E. ‘modules’ Folder

The modules folder is a vital component of the project, containing some essential files that implement the core functionalities of the simulation. These files are modularised into source (.cpp or .cu) and header (.hpp or .cuh) formats, ensuring that the codebase is both organised and extensible. Each module serves a specific purpose, contributing to the overall framework of the simulation pipeline. As in the writing of this thesis, I found that some scripts were redundant and will be removed in the next updates.

a. cipa_t.cu and cipa_t.cuh

These files handle data type related to the formatting of CiPA (Comprehensive in vitro Proarrhythmia Assay) metrics, which are critical and become standard for assessing drug-induced proarrhythmia risks. All implementation in cipa_t.cu has been migrated to its header file (cipa_t.cuh) and will be removed in the next updates. Header file of cipa_t.cuh contains commented old vectors and naturalised 1D arrays as such:

```
#ifndef CIPA_T_HPP
#define CIPA_T_HPP

#include <map>
#include <string>

#include <cuda_runtime.h>

// using std::multimap;
// using std::string;
__global__ struct cipa_t{
  double qnet_ap;
  double qnet4_ap;
  double inal_auc_ap;
  double ical_auc_ap;
  double qnet_cl;
  double qnet4_cl;
  double inal_auc_cl;
  double ical_auc_cl;

  double dvmdt_repol;
  double vm_peak;
  double vm_valley;
  // multimap<double, double> vm_data;
  // multimap<double, double> dvmdt_data;
  // multimap<double, double> cai_data;
  // multimap<double, string> ires_data;

  // multimap<double, string> inet_data;
  // multimap<double, string> qnet_data;
  // multimap<double, string> inet4_data;
  // multimap<double, string> qnet4_data;

  // multimap<double, string> time_series_data;
```

```
  // temporary fix for this
  double vm_data[7000];
  double vm_time[7000];

  double dvmdt_data[7000];
  double dvmdt_time[7000];

  double cai_data[7000];
  double cai_time[7000];

  double ires_data[7000];
  double ires_time[7000];

  double inet_data[7000];
  double inet_time[7000];

  double qnet_data[7000];
  double qnet_time[7000];

  double inet4_data[7000];
  double inet4_time[7000];

  double qnet4_data[7000];
  double qnet4_time[7000];

  // double time_series_data[7000];
  // double time_series_time[7000];



  // __device__ cipa_t();
  // __device__ cipa_t( const cipa_t &source );
  // cipa_t& operator=(const cipa_t & source);
  // __device__ void copy(const cipa_t &source);
  // __device__ void init(const double vm_val);
  // __device__ void clear_time_result();


};


#endif
```

b. drug_conc.cpp and drug_conc.hpp

These files manage the drug concentration data used in the simulations. These act as utility file to lookup if user did not declare the

concentration value and drugs used is in the scope of CiPA. Lookup process is happening in the CPU domain. Below is the list of known drug by this utility function and its implementation using map:

```
#include "drug_conc.hpp"

float getValue(const std::unordered_map<std::string, float>& drugConc, const std::string& key, float defaultValue) {
    auto it = drugConc.find(key);
    if (it != drugConc.end()) {
        return it->second;
    }
    return defaultValue;
}

// Instantiate the dictionary with keys and values
std::unordered_map<std::string, float> drugConcentration = {
    {"azimilide", 70.0f},
    {"bepridil", 33.0f},
    {"disopyramide", 742.0f},
    {"dofetilide", 2.0f},
    {"ibutilide", 100.0f},
    {"quinidine", 3237.0f},
    {"sotalol", 14690.0f},
    {"vandetanib", 255.0f},
    {"astemizole", 0.26f},
    {"chlorpromazine", 38.0f},
    {"cisapride", 2.6f},
    {"clarithromycin", 1206.0f},
    {"clozapine", 71.0f},
    {"domperidone", 19.0f},
    {"droperidol", 6.3f},
    {"ondansetron", 139.0f},
    {"pimozide", 0.431f},
    {"risperidone", 1.81f},
    {"terfenadine", 4.0f},
    {"diltiazem", 122.0f},
    {"loratadine", 0.45f},
    {"metoprolol", 1800.0f},
    {"mexiletine", 4129.0f},
    {"nifedipine", 7.7f},
    {"nitrendipine", 3.02f},
    {"ranolazine", 1948.2f},
    {"tamoxifen", 21.0f},
```

```
    {"verapamil", 81.0f}
};
```

And header file:

```
#ifndef DRUG_CONC_HPP
#define DRUG_CONC_HPP

#include <unordered_map>
#include <string>

// Declare the dictionary function template
float getValue(const std::unordered_map<std::string, float>& drugConc, const std::string& key, float defaultValue = 0.0);

// Declare the dictionary extern
extern std::unordered_map<std::string, float> drugConcentration;
#endif  // DRUG_CONC_HPP
```

c. glob_funct.cpp and glob_funct.hpp

Global function or glob_funct encapsulate global utility functions that are used across various parts of the codebase. At the time this thesis was written, this script regulates the reading of flags and input deck. There are some unused functions that will be removed in the future. The global function script is implemented as below:

```
#include "glob_funct.hpp"
// #include "../libs/zip.h"

#include <cstdarg>
#include <cstdio>
#include <cstdlib>
#include <cstring>

// to make it more "portable" between OSes.
#if defined _WIN32
  #include <direct.h>
  #define snprintf _snprintf
  #define vsnprintf _vsnprintf
  #define strcasecmp _stricmp
  #define strncasecmp _strnicmp
```

```
#else
  #include <dirent.h>
#endif
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>


void mpi_printf(unsigned short node_id, const char *fmt, ...)
{

  va_list args;
  va_start(args, fmt);
  vprintf(fmt, args);
  va_end(args);
}

void mpi_fprintf(unsigned short node_id, FILE *stream, const char *fmt, ...)
{
#ifndef _WIN32
  if(mympi::rank == node_id){
    va_list args;
    va_start(args, fmt);
    vfprintf(stream, fmt, args);
    va_end(args);
  }
#else
  va_list args;
  va_start(args, fmt);
  vfprintf(stream, fmt, args);
  va_end(args);
#endif
}

void edison_assign_params(int argc, char *argv[], param_t *p_param)
{
  bool is_default;
  char buffer[100];
  char key[100];
  char value[100];
  char file_name[150];
  FILE *fp_inputdeck;

  // parameters from arguments
  for (int idx = 1; idx < argc; idx += 2) {
    if (!strcasecmp(argv[idx], "-input_deck"))
      strncpy(file_name, argv[idx + 1], sizeof(file_name));
    else if (!strcasecmp(argv[idx], "-hill_file"))
      strncpy(p_param->hill_file, argv[idx + 1], sizeof(p_param->hill_file));
```

```
      else if (!strcasecmp(argv[idx], "-cvar_file"))
        strncpy(p_param->cvar_file, argv[idx + 1], sizeof(p_param->cvar_file));
    }

    is_default = false;
    fp_inputdeck = fopen( file_name, "r");
    if(fp_inputdeck == NULL){
      fprintf(stderr, "Cannot open input deck file %s!!!\nUse default value as the
failsafe.\n", file_name);
      is_default = true;
    }

    // read input_deck line by line
    // and store each line to the buffer
    while ( is_default == false && fgets( buffer, 100, fp_inputdeck ) != NULL ) {
      sscanf( buffer, "%s %*s %s", key, value );
      if (strcasecmp(key, "Simulation_Mode") == 0) {
        p_param->simulation_mode = strtod( value, NULL );
      }
      else if (strcasecmp(key, "Celltype") == 0) {
        p_param->celltype = strtod( value, NULL );
      }
      else if (strcasecmp(key, "Is_Dutta") == 0) {
        p_param->is_dutta = strtol( value, NULL, 10 );
      }
      else if (strcasecmp(key, "Use_Conductance_Variability") == 0) {
        p_param->is_cvar = strtol( value, NULL, 10 );
      }
      else if (strcasecmp(key, "Pace_Find_Steepest") == 0) {
        p_param->find_steepest_start = strtod( value, NULL);
      }
      else if (strcasecmp(key, "GPU_Index") == 0) {
        p_param->gpu_index = strtod( value, NULL);
      }
      else if (strcasecmp(key, "Basic_Cycle_Length") == 0) {
        p_param->bcl = strtod( value, NULL );
      }
      else if (strcasecmp(key, "Number_of_Pacing") == 0) {
        p_param->pace_max = strtod( value, NULL );
      }
      else if (strcasecmp(key, "Time_Step") == 0) {
        p_param->dt = strtod( value, NULL );
      }
      //TODO: #Automation 1. eliminate drug_name and concentration on drug_name
      // else if (strcasecmp(key, "Drug_Name") == 0) {
      //   strncpy( p_param->drug_name, value, sizeof(p_param->concs) );
      // }
      // else if (strcasecmp(key, "Concentrations") == 0) {
      //   p_param->conc = strtod( value, NULL );
```

```
    // }

  }

  if( is_default == false ) fclose( fp_inputdeck );
}


int make_directory(const char* dirname )
{
#if defined _WIN32
  return _mkdir(dirname);
#else
  return mkdir(dirname, 0775);
#endif
}

int is_file_existed(const char* pathname)
{
#if defined _WIN32
  struct _stat buf;
  return _stat( pathname, &buf );
#else
  struct stat st = {0};
  return stat(pathname, &st);
#endif
}
```

d. glob_type.cpp and glob_type.hpp

The glob_type files define and manage the global data types and structures used throughout the project. For now, this script is being used to store IC50 data. The glob_type.hpp is found unused and will be removed in the next update. As glob_type.cpp is still being used, here is our implementation into it:

```
#ifndef GLOB_TYPE_HPP
#define GLOB_TYPE_HPP

#include <vector>

// global variable for MPI.
struct mympi
{
  static char host_name[255];
```

```
  static int host_name_len;
  static int rank;
  static int size;
};

// data structure for IC50
typedef struct row_data { double data[14]; } row_data;
typedef std::vector< row_data > drug_t;

// data structure to store
// ICaL/INaL control value
// for calculating qinward
// control means 0 concentration
// otherwise, drugs
typedef struct{
  double ical_auc_control;
  double inal_auc_control;
  double ical_auc_drug;
  double inal_auc_drug;
} qinward_t;

#endif
```

e. gpu.cu and gpu.cuh

The gpu files represent the core computational engine of the simulation, leveraging the parallel processing capabilities of NVIDIA GPUs to achieve significant speedups in simulation tasks. These files are essential for the framework's computational efficiency, enabling the processing of thousands of samples in parallel.

gpu.cu is the implementation file that contains CUDA-specific kernels and device functions. CUDA kernels are at the heart of the GPU simulation, responsible for solving the differential equations of the cell models, processing drug interactions, and performing large-scale computations for multiple samples concurrently. The kernels in this file are designed to optimise memory usage and minimise latency, ensuring the efficient execution of simulations. Header gpu.cuh serves as the header file that defines the interface for the GPU-specific functionalities implemented

in gpu.cu. It declares the prototypes of CUDA kernels and device functions, providing a structured entry point for other parts of the code to interact with GPU-based operations. The main gpu.cu code will look as this:

```
#include "../cellmodels/Ohara_Rudy_2011.hpp"
#include <stdio.h>
#include <cuda_runtime.h>
#include <cuda.h>

#include "glob_funct.hpp"
#include "glob_type.hpp"
#include "gpu_glob_type.cuh"
#include "gpu.cuh"


/*
all kernel function has been moved. Unlike the previous GPU code, now we seperate everything
into each modules.
all modules here are optimised for GPU and slightly different than the original component based
code
differences are related to GPU offset calculations
*/

__device__  void  kernel_DoDrugSim(double  *d_ic50,  double  *d_cvar,  double  d_conc,  double
*d_CONSTANTS, double *d_STATES, double *d_RATES, double *d_ALGEBRAIC,
                                      double *d_STATES_RESULT,
                                    //  double *time, double *states, double *out_dt,  double
*cai_result,
                                    //  double *ina, double *inal,
                                    //  double *ical, double *ito,
                                    //  double *ikr, double *iks,
                                    //  double *ik1,
                                     double  *tcurr,  double  *dt,  unsigned  short  sample_id,
unsigned int sample_size,
                                     cipa_t *temp_result, cipa_t *cipa_result,
                                     param_t *p_param
                                     )
    {

    unsigned int input_counter = 0;

    int num_of_constants = 145;
    int num_of_states = 41;
    int num_of_algebraic = 199;
    int num_of_rates = 41;
```

```
// INIT STARTS
temp_result[sample_id].qnet_ap = 0.;
temp_result[sample_id].qnet4_ap = 0.;
temp_result[sample_id].inal_auc_ap = 0.;
temp_result[sample_id].ical_auc_ap = 0.;

temp_result[sample_id].qnet_cl = 0.;
temp_result[sample_id].qnet4_cl = 0.;
temp_result[sample_id].inal_auc_cl = 0.;
temp_result[sample_id].ical_auc_cl = 0.;

temp_result[sample_id].dvmdt_repol = -999;
temp_result[sample_id].vm_peak = -999;
temp_result[sample_id].vm_valley = d_STATES[(sample_id * num_of_states) +V];

cipa_result[sample_id].qnet_ap = 0.;
cipa_result[sample_id].qnet4_ap = 0.;
cipa_result[sample_id].inal_auc_ap = 0.;
cipa_result[sample_id].ical_auc_ap = 0.;

cipa_result[sample_id].qnet_cl = 0.;
cipa_result[sample_id].qnet4_cl = 0.;
cipa_result[sample_id].inal_auc_cl = 0.;
cipa_result[sample_id].ical_auc_cl = 0.;

cipa_result[sample_id].dvmdt_repol = -999;
cipa_result[sample_id].vm_peak = -999;
cipa_result[sample_id].vm_valley = d_STATES[(sample_id * num_of_states) +V];
// INIT ENDS
bool is_peak = false;
// to search max dvmdt repol

tcurr[sample_id] = 0.0;
dt[sample_id] = p_param->dt;
double tmax;
double max_time_step = 1.0, time_point = 25.0;
double dt_set;

int cipa_datapoint = 0;

const double bcl = p_param->bcl;

// const double inet_vm_threshold = p_param->inet_vm_threshold;
// const unsigned short pace_max = 300;
// const unsigned short pace_max = 1000;
const unsigned short pace_max = p_param->pace_max;
const unsigned short last_drug_check_pace = p_param->find_steepest_start;
double conc = d_conc; //mmol
```

```
    double type = p_param->celltype;
    double epsilon = 10E-14;

    // eligible AP shape means the Vm_peak > 0.
    bool is_eligible_AP;
    // Vm value at 30% repol, 50% repol, and 90% repol, respectively.
    double vm_repol30, vm_repol50, vm_repol90;
    double t_peak_capture = 0.0;
    unsigned short pace_steepest = 0;

    bool init_states_captured = false;

    // qnet_ap/inet_ap values
          double inet_ap, qnet_ap, inet4_ap, qnet4_ap, inet_cl, qnet_cl, inet4_cl, qnet4_cl;
          double inal_auc_ap, ical_auc_ap,inal_auc_cl, ical_auc_cl;
    // qinward_cl;

    // char buffer[255];

    // static const int CALCIUM_SCALING = 1000000;
          // static const int CURRENT_SCALING = 1000;

    // printf("Core %d:\n",sample_id);
    initConsts(d_CONSTANTS, d_STATES, type, conc, d_ic50, d_cvar, p_param->is_dutta, p_param-
>is_cvar, sample_id);


    applyDrugEffect(d_CONSTANTS, conc, d_ic50, epsilon, sample_id);
    d_CONSTANTS[BCL + (sample_id * num_of_constants)] = bcl;

    // generate file for time-series output

    tmax = pace_max * bcl;
    int pace_count = 0;


    // printf("%d,%lf,%lf,%lf,%lf\n", sample_id, dt[sample_id], tcurr[sample_id], d_STATES[V +
(sample_id * num_of_states)],d_RATES[V + (sample_id * num_of_rates)]);
    // printf("%lf,%lf,%lf,%lf,%lf\n", d_ic50[0 + (14*sample_id)], d_ic50[1+ (14*sample_id)],
d_ic50[2+ (14*sample_id)], d_ic50[3+ (14*sample_id)], d_ic50[4+ (14*sample_id)]);

    while (tcurr[sample_id]<tmax)
    {
        computeRates(tcurr[sample_id], d_CONSTANTS, d_RATES, d_STATES, d_ALGEBRAIC, sample_id);

        dt_set = set_time_step( tcurr[sample_id], time_point, max_time_step,
        d_CONSTANTS,
        d_RATES,
        d_STATES,
```

```
        d_ALGEBRAIC,
        sample_id);

        //euler only
        // dt_set = 0.005;

        // printf("tcurr at core %d: %lf\n",sample_id,tcurr[sample_id]);
        if (floor((tcurr[sample_id] + dt_set) / bcl) == floor(tcurr[sample_id] / bcl)) {
          dt[sample_id] = dt_set;
          // printf("dt : %lf\n",dt_set);
          // it goes in here, but it does not, you know, adds the pace,
        }
        else{
          dt[sample_id] = (floor(tcurr[sample_id] / bcl) + 1) * bcl - tcurr[sample_id];

          // new part starts
          if( is_eligible_AP && pace_count >= pace_max-last_drug_check_pace) {
            temp_result[sample_id].qnet_ap = qnet_ap;
            temp_result[sample_id].qnet4_ap = qnet4_ap;
            temp_result[sample_id].inal_auc_ap = inal_auc_ap;
            temp_result[sample_id].ical_auc_ap = ical_auc_ap;
            temp_result[sample_id].qnet_cl = qnet_cl;
            temp_result[sample_id].qnet4_cl = qnet4_cl;
            temp_result[sample_id].inal_auc_cl = inal_auc_cl;
            temp_result[sample_id].ical_auc_cl = ical_auc_cl;
            //         fprintf(fp_vmdebug,         "%hu,%.2lf,%.2lf,%.2lf,%.2lf,%.2lf,%.2lf\n",
pace_count,t_peak_capture,temp_result.vm_peak,vm_repol30,vm_repol50,vm_repol90,temp_result.dvm
dt_repol);
            // replace result with steeper repolarization AP or first pace from the last 250
paces
            // if( temp_result->dvmdt_repol > cipa_result.dvmdt_repol ) {
            //   pace_steepest = pace_count;
            //   cipa_result = temp_result;
            //   }
            if( temp_result[sample_id].dvmdt_repol > cipa_result[sample_id].dvmdt_repol ) {
              pace_steepest = pace_count;
              //            printf("Steepest            pace            updated:            %d
dvmdt_repol: %lf\n",pace_steepest,temp_result[sample_id].dvmdt_repol);
              // cipa_result = temp_result;
              cipa_result[sample_id].qnet_ap = temp_result[sample_id].qnet_ap;
              cipa_result[sample_id].qnet4_ap = temp_result[sample_id].qnet4_ap;
              cipa_result[sample_id].inal_auc_ap = temp_result[sample_id].inal_auc_ap;
              cipa_result[sample_id].ical_auc_ap = temp_result[sample_id].ical_auc_ap;

              cipa_result[sample_id].qnet_cl = temp_result[sample_id].qnet_cl;
              cipa_result[sample_id].qnet4_cl = temp_result[sample_id].qnet4_cl;
              cipa_result[sample_id].inal_auc_cl = temp_result[sample_id].inal_auc_cl;
              cipa_result[sample_id].ical_auc_cl = temp_result[sample_id].ical_auc_cl;
```

```
            cipa_result[sample_id].dvmdt_repol = temp_result[sample_id].dvmdt_repol;
            cipa_result[sample_id].vm_peak = temp_result[sample_id].vm_peak;
            cipa_result[sample_id].vm_valley = d_STATES[(sample_id * num_of_states) +V];
            is_peak = true;
            init_states_captured = false;
            }
          else{
            is_peak = false;
          }
        };
        inet_ap = 0.;
        qnet_ap = 0.;
        inet4_ap = 0.;
        qnet4_ap = 0.;
        inal_auc_ap = 0.;
        ical_auc_ap = 0.;
        inet_cl = 0.;
        qnet_cl = 0.;
        inet4_cl = 0.;
        qnet4_cl = 0.;
        inal_auc_cl = 0.;
        ical_auc_cl = 0.;
        t_peak_capture = 0.;

        // temp_result->init( p_cell->STATES[V]);
        temp_result[sample_id].qnet_ap = 0.;
        temp_result[sample_id].qnet4_ap = 0.;
        temp_result[sample_id].inal_auc_ap = 0.;
        temp_result[sample_id].ical_auc_ap = 0.;

        temp_result[sample_id].qnet_cl = 0.;
        temp_result[sample_id].qnet4_cl = 0.;
        temp_result[sample_id].inal_auc_cl = 0.;
        temp_result[sample_id].ical_auc_cl = 0.;

        temp_result[sample_id].dvmdt_repol = -999;
        temp_result[sample_id].vm_peak = -999;
        temp_result[sample_id].vm_valley = d_STATES[(sample_id * num_of_states) +V];
        // end of init

        pace_count++;
        input_counter = 0; // at first, we reset the input counter since we re gonna only take
one, but I remember we don't have this kind of thing previously, so do we need this still?
        cipa_datapoint = 0; // new pace? reset variables related to saving the values,

        is_eligible_AP = false;
        // new part ends
         if(sample_id == 0 || (sample_id % 1000 == 0 && sample_id>999)){
```

```
            printf("core: %d pace count: %d, steepest: %d, dvmdt_repol: %lf, conc: %lf,
celltype: %lf\n",sample_id,pace_count, pace_steepest, cipa_result[sample_id].dvmdt_repol, conc,
d_CONSTANTS[(sample_id * num_of_constants) + celltype]);
          }
          //  printf("core: %d pace count: %d t: %lf, steepest: %d, dvmdt_repol: %lf,
t_peak:          %lf\n",sample_id,pace_count,          tcurr[sample_id],          pace_steepest,
cipa_result[sample_id].dvmdt_repol,t_peak_capture);
          // writen = false;
        }

        solveAnalytical(d_CONSTANTS, d_STATES, d_ALGEBRAIC, d_RATES,  dt[sample_id], sample_id);
        // solveEuler(d_STATES, d_RATES, dt[sample_id], sample_id);

        if (pace_count >= pace_max-last_drug_check_pace)
        {

                            if( tcurr[sample_id] > ((d_CONSTANTS[(sample_id * num_of_constants)
+BCL]*pace_count)+(d_CONSTANTS[(sample_id * num_of_constants) +stim_start]+2)) &&
                                        tcurr[sample_id]    <    ((d_CONSTANTS[(sample_id    *
num_of_constants)      +BCL]*pace_count)+(d_CONSTANTS[(sample_id      *      num_of_constants)
+stim_start]+10)) &&
                                        abs(d_ALGEBRAIC[(sample_id * num_of_algebraic) +INa]) <
1)
          {
            // printf("check 1\n");
            if( d_STATES[(sample_id * num_of_states) +V] > temp_result[sample_id].vm_peak )
            {
              temp_result[sample_id].vm_peak = d_STATES[(sample_id * num_of_states) +V];
              if(temp_result[sample_id].vm_peak > 0)
              {
                vm_repol30      =      temp_result[sample_id].vm_peak      -      (0.3      *
(temp_result[sample_id].vm_peak - temp_result[sample_id].vm_valley));
                vm_repol50      =      temp_result[sample_id].vm_peak      -      (0.5      *
(temp_result[sample_id].vm_peak - temp_result[sample_id].vm_valley));
                vm_repol90      =      temp_result[sample_id].vm_peak      -      (0.9      *
(temp_result[sample_id].vm_peak - temp_result[sample_id].vm_valley));
                is_eligible_AP = true;
                t_peak_capture = tcurr[sample_id];
              }
              else is_eligible_AP = false;
            }
                            }
                            else   if(   tcurr[sample_id]   >   ((d_CONSTANTS[(sample_id   *
num_of_constants)      +BCL]*pace_count)+(d_CONSTANTS[(sample_id      *      num_of_constants)
+stim_start]+10)) && is_eligible_AP )
          {

                                      if(   d_RATES[(sample_id   *   num_of_rates)   +V]   >
temp_result[sample_id].dvmdt_repol &&
```

```
                                                d_STATES[(sample_id  *  num_of_states)  +V]  <=
vm_repol30 &&
                                                d_STATES[(sample_id  *  num_of_states)  +V]  >=
vm_repol90 )
              {
                                                temp_result[sample_id].dvmdt_repol          =
d_RATES[(sample_id * num_of_rates) +V];
                // printf("check 4\n");
                                        }
          }
                              // calculate AP shape
                              if(is_eligible_AP  &&  d_STATES[(sample_id  *  num_of_states)  +V]  >
vm_repol90)
          {

          inet_ap = (d_ALGEBRAIC[(sample_id * num_of_algebraic) +INaL]+d_ALGEBRAIC[(sample_id *
num_of_algebraic)          +ICaL]+d_ALGEBRAIC[(sample_id          *          num_of_algebraic)
+Ito]+d_ALGEBRAIC[(sample_id     *     num_of_algebraic)     +IKr]+d_ALGEBRAIC[(sample_id     *
num_of_algebraic) +IKs]+d_ALGEBRAIC[(sample_id * num_of_algebraic) +IK1]);
          inet4_ap = (d_ALGEBRAIC[(sample_id * num_of_algebraic) +INaL]+d_ALGEBRAIC[(sample_id
*        num_of_algebraic)        +ICaL]+d_ALGEBRAIC[(sample_id        *        num_of_algebraic)
+IKr]+d_ALGEBRAIC[(sample_id * num_of_algebraic) +INa]);
          qnet_ap += (inet_ap * dt[sample_id])/1000.;
          qnet4_ap += (inet4_ap * dt[sample_id])/1000.;
          inal_auc_ap += (d_ALGEBRAIC[(sample_id * num_of_algebraic) +INaL]*dt[sample_id]);
          ical_auc_ap += (d_ALGEBRAIC[(sample_id * num_of_algebraic) +ICaL]*dt[sample_id]);
                              }
          inet_cl = (d_ALGEBRAIC[(sample_id * num_of_algebraic) +INaL]+d_ALGEBRAIC[(sample_id *
num_of_algebraic)          +ICaL]+d_ALGEBRAIC[(sample_id          *          num_of_algebraic)
+Ito]+d_ALGEBRAIC[(sample_id     *     num_of_algebraic)     +IKr]+d_ALGEBRAIC[(sample_id     *
num_of_algebraic) +IKs]+d_ALGEBRAIC[(sample_id * num_of_algebraic) +IK1]);
          inet4_cl = (d_ALGEBRAIC[(sample_id * num_of_algebraic) +INaL]+d_ALGEBRAIC[(sample_id
*        num_of_algebraic)        +ICaL]+d_ALGEBRAIC[(sample_id        *        num_of_algebraic)
+IKr]+d_ALGEBRAIC[(sample_id * num_of_algebraic) +INa]);
          qnet_cl += (inet_cl * dt[sample_id])/1000.;
          qnet4_cl += (inet4_cl * dt[sample_id])/1000.;
          inal_auc_cl += (d_ALGEBRAIC[(sample_id * num_of_algebraic) +INaL]*dt[sample_id]);
          ical_auc_cl += (d_ALGEBRAIC[(sample_id * num_of_algebraic) +ICaL]*dt[sample_id]);

          // save temporary result -> ALL TEMP RESULTS IN, TEMP RESULT != WRITTEN RESULT
          if((pace_count   >=   pace_max-last_drug_check_pace)   &&   (is_peak   ==   true)   &&
(pace_count<pace_max) )
          {
            // printf("input_counter: %d\n",input_counter);
            // datapoint_at_this_moment = tcurr[sample_id] - (pace_count * bcl);
            temp_result[sample_id].cai_data[cipa_datapoint]    =       d_STATES[(sample_id    *
num_of_states) +cai] ;
            temp_result[sample_id].cai_time[cipa_datapoint] =  tcurr[sample_id];
```

```
            temp_result[sample_id].vm_data[cipa_datapoint]     =     d_STATES[(sample_id     *
num_of_states) +V];
            temp_result[sample_id].vm_time[cipa_datapoint] = tcurr[sample_id];

            temp_result[sample_id].dvmdt_data[cipa_datapoint]    =    d_RATES[(sample_id    *
num_of_rates) +V];
            temp_result[sample_id].dvmdt_time[cipa_datapoint] = tcurr[sample_id];

            if(init_states_captured == false){
              for(int counter=0; counter<num_of_states; counter++){
                d_STATES_RESULT[(sample_id * num_of_states) + counter] = d_STATES[(sample_id *
num_of_states) + counter];
              }
              init_states_captured = true;
            }

            // time series result

            // time[input_counter + sample_id] = tcurr[sample_id];
            // states[input_counter + sample_id] = d_STATES[V + (sample_id * num_of_states)];

            // out_dt[input_counter + sample_id] = d_RATES[V + (sample_id * num_of_states)];


            //  cai_result[input_counter  +  sample_id]  =  d_ALGEBRAIC[cai  +  (sample_id  *
num_of_algebraic)];

            //   ina[input_counter   +   sample_id]   =   d_ALGEBRAIC[INa   +   (sample_id   *
num_of_algebraic)] ;
            //   inal[input_counter   +   sample_id]   =   d_ALGEBRAIC[INaL   +   (sample_id   *
num_of_algebraic)] ;

            //   ical[input_counter   +   sample_id]   =   d_ALGEBRAIC[ICaL   +   (sample_id   *
num_of_algebraic)] ;
            //   ito[input_counter   +   sample_id]   =   d_ALGEBRAIC[Ito   +   (sample_id   *
num_of_algebraic)] ;

            //   ikr[input_counter   +   sample_id]   =   d_ALGEBRAIC[IKr   +   (sample_id   *
num_of_algebraic)] ;
            //   iks[input_counter   +   sample_id]   =   d_ALGEBRAIC[IKs   +   (sample_id   *
num_of_algebraic)] ;

            //   ik1[input_counter   +   sample_id]   =   d_ALGEBRAIC[IK1   +   (sample_id   *
num_of_algebraic)] ;

            input_counter = input_counter + sample_size;
            cipa_datapoint = cipa_datapoint + 1; // this causes the resource usage got so mega
and crashed in running
```

```
            } // temporary guard ends here

                    } // end the last 250 pace operations
        tcurr[sample_id] = tcurr[sample_id] + dt[sample_id];
        //printf("t after addition: %lf\n", tcurr[sample_id]);

    } // while loop ends here
    // __syncthreads();
}



__global__ void kernel_DrugSimulation(double *d_ic50, double *d_cvar, double *d_conc, double
*d_CONSTANTS, double *d_STATES, double *d_RATES, double *d_ALGEBRAIC,
                                      double *d_STATES_RESULT,
                                      // double *time, double *states, double *out_dt,  double
*cai_result,
                                      // double *ina, double *inal,
                                      // double *ical, double *ito,
                                      // double *ikr, double *iks,
                                      // double *ik1,
                                      unsigned int sample_size,
                                      cipa_t *temp_result, cipa_t *cipa_result,
                                      param_t *p_param
                                      )
  {
    unsigned short thread_id;
    thread_id = blockIdx.x * blockDim.x + threadIdx.x;
    if (thread_id >= sample_size) return;
    double time_for_each_sample[10000];
    double dt_for_each_sample[10000];
    // cipa_t temp_per_sample[2000];
    // cipa_t cipa_per_sample[2000];
    // printf("in\n");
    // printf("Calculating %d\n",thread_id);
    kernel_DoDrugSim(d_ic50,  d_cvar,  d_conc[thread_id],  d_CONSTANTS,  d_STATES,  d_RATES,
d_ALGEBRAIC,
                           d_STATES_RESULT,
                           // time, states, out_dt, cai_result,
                           // ina, inal,
                           // ical, ito,
                           // ikr, iks,
                           // ik1,
                           time_for_each_sample, dt_for_each_sample, thread_id, sample_size,
                           temp_result, cipa_result,
                           p_param
                           );
                           // __syncthreads();
```

```
    // printf("Calculation for core %d done\n",sample_id);
  }
```

The header file of gpu.hpp declared as follows:

```
#ifndef GPU_CUH
#define GPU_CUH
#include <cuda_runtime.h>
#include <cuda.h>
#include "cipa_t.cu"
__global__ void kernel_DrugSimulation(double *d_ic50, double *d_cvar, double *d_conc, double
*d_CONSTANTS, double *d_STATES, double *d_RATES, double *d_ALGEBRAIC,
                                      double *d_STATES_RESULT,
                                    //   double *time, double *states, double *out_dt,  double
*cai_result,
                                    //   double *ina, double *inal,
                                    //   double *ical, double *ito,
                                    //   double *ikr, double *iks,
                                    //   double *ik1,
                                      unsigned int sample_size,
                                      cipa_t *temp_result, cipa_t *cipa_result,
                                      param_t *p_param
                                      );


__device__  void  kernel_DoDrugSim(double  *d_ic50,  double  *d_cvar,  double  d_conc,  double
*d_CONSTANTS, double *d_STATES, double *d_RATES, double *d_ALGEBRAIC,
                                      double *d_STATES_RESULT,
                                    //    double *time, double *states, double *out_dt,  double
*cai_result,
                                    //    double *ina, double *inal,
                                    //    double *ical, double *ito,
                                    //    double *ikr, double *iks,
                                    //    double *ik1,
                                       double  *tcurr,  double  *dt,  unsigned  short  sample_id,
unsigned int sample_size,
                                       cipa_t *temp_result, cipa_t *cipa_result,
                                       param_t *p_param
                                       );


#endif
```

In the post-processing mode (the mode that only runs the simulation once and take all biomarkers calculation), these comments were uncommented to let more data flow in and out of the GPU kernel.

f. param.cpp and param.hpp

These scripts manages the simulation parameters. It reads the input deck file and put them into the code as variables. The param.cpp implements functions to read the input file, load, parse, validate parameter files, and act as a failsafe if one or more parameters or parameter file is not readable. This script has the default values and path of each required numbers and files. This file ensures that the simulation runs with accurate and user-defined settings. Header of param.hpp declares variables used to store these parameter values and ensuring that they are well defined in the simulation. The header file goes as a following struct:

```
#ifndef PARAM_HPP
#define PARAM_HPP

struct param_t
{
  unsigned short simulation_mode; // toggle between sample-based or full-pace simulations
  bool is_dutta; // TRUE if using Dutta scaling
  unsigned short gpu_index;
  bool is_print_graph; // TRUE if we want to print graph
  bool is_using_output; // TRUE if using last output file
  bool is_cvar;
  double bcl; // basic cycle length
  // unsigned int max_samples;
  unsigned short pace_max; // maximum pace
  unsigned short find_steepest_start;
  unsigned short celltype;  // cell types
  double dt;        // time step
  double dt_write;  // writing step
  double inet_vm_threshold; // Vm threshold for calculating inet
  char hill_file[1024];
  char cvar_file[1024];
  char drug_name[100];
  char concs[100];
  float conc;
  void init();
```

```
  void show_val();
};

#endif
```

And the main file reads parameter and their default values to the whole project with following way:

```
#include "param.hpp"
#include <cstdio>
#include "glob_funct.hpp"
void param_t::init()
{
  simulation_mode = 0;
  // max_samples = 10000;
  is_dutta = true;
  gpu_index = 0;

  is_cvar = false;
  bcl = 2000.;
  pace_max = 10;

  find_steepest_start = 5;

  celltype = 0.;
  dt = 0.005;
  // dt = 0.1;

  conc = 99.0;
  // dt_write = 2.0;
  // inet_vm_threshold = -88.0;

  snprintf(hill_file, sizeof(hill_file), "%s", "./drugs/bepridil/IC50_samples.csv");
  snprintf(cvar_file, sizeof(cvar_file), "%s", "./drugs/10000_pop.csv");
  snprintf(drug_name, sizeof(drug_name), "%s", "bepridil");
  // snprintf(concs, sizeof(concs), "%s", "99.0");
}

void param_t::show_val()
{
  //change this to printf somehow
  mpi_printf( 0, "%s -- %s\n", "Simulation mode", simulation_mode ? "full-pace" :
"sample-based" );
  mpi_printf( 0, "%s -- %s\n", "Hill File", hill_file );
  mpi_printf( 0, "%s -- %hu\n", "Celltype", celltype);
  mpi_printf( 0, "%s -- %s\n", "Is_Dutta", is_dutta ? "true" : "false" );
```

```
    mpi_printf( 0, "%s -- %s\n", "Is_Cvar", is_cvar ? "true" : "false" );
    mpi_printf( 0, "%s -- %lf\n", "Basic_Cycle_Length", bcl);
    mpi_printf( 0, "%s -- %d\n", "GPU_Index", gpu_index);
    mpi_printf( 0, "%s -- %hu\n", "Number_of_Pacing", pace_max);
    mpi_printf( 0, "%s -- %hu\n", "Pace_Find_Steepest", find_steepest_start);
    mpi_printf( 0, "%s -- %lf\n", "Time_Step", dt);
    mpi_printf( 0, "%s -- %s\n", "Drug_Name", drug_name);
    mpi_printf( 0, "%s -- %lf\n\n\n", "Concentrations", conc);
}
```

F. Critical Scripts

This section highlights the key scripts within the framework, focusing on their roles and significance in executing and customising simulations. These critical scripts form the backbone of the codebase, enabling efficient management of computations, GPU operations, and parameter configurations. If there is any issue with the code, 80% of the issue can be solved by looking at these three script first. Most of debugging, adjustement, and modification happen in these scripts. Below is a detailed breakdown:

a. Main script

The main script (main.cu) serves as the central controller of the simulation framework. It is the entry point of the program, responsible for managing the simulation's overall workflow, from initialisation to the final output. Below are its primary responsibilities and modifications related to main.cu:

- Simulation Parameter Parsing
    - Reads user-provided inputs, such as input_deck.txt, initial state files, and IC50/Hill coefficient files.
    - Validates the correctness and presence of required input files.
    - Flags any missing or improperly formatted files, preventing the simulation from starting.
- Simulation Setup

    - Loads and processes input parameters from the provided files.
    - Sets up the pacing protocol, drug concentrations, and simulation-specific configurations.
    - Handles initialisation for multiple samples and conditions.
- Memory Management
    - Allocates and deallocates memory for the CPU and GPU, ensuring efficient resource utilisation using core (thread) per block calculation.
    - Transfers data such as parameters, initial states, and configurations between host (CPU) and device (GPU).
- GPU Kernel Launch
    - Initiates GPU computations by invoking the kernel functions for parallel execution.
    - Ensures proper thread and block configurations to optimise performance for the given workload.
- Result Handling
    - Collects output data from the GPU and processes it for storage.
    - Saves simulation results to the designated output directory.
    - Generates logs for debugging and validation purposes.
- Error Handling and Debugging
    - Includes checks to identify and report common errors, such as missing inputs, memory allocation failures, or kernel launch issues. If something is wrong with the kernel function in the GPU, the code always goes back to this script, assuming the kernel function has done even with no output.
    - Provides detailed logs to help trace and resolve errors during simulation execution.

b. GPU control script

The GPU control script (`gpu.cu` and `gpu.cuh`) is the core component that drives the high-performance computations. It includes all GPU-specific operations, such as kernel implementations, and parallel execution logic. The kernel simulates the electrical behaviour of cardiomyocyte cells over

time, under the influence of a specific drug at a given concentration. This script manages the distribution of workloads across GPU threads and blocks, ensuring that simulations are processed efficiently.

Key responsibilities of this script include:

- Calling the numerical solvers (e.g., Forward Euler, Rush-Larsen) for simulating cardiac cell dynamics.
- Managing steps and loops of simulation, such as calling drug effect applying function.
- Calculating metrics for biomarkers, time series data, and such related to simulation, or calculated during simulation loops.

Most performance issues or GPU errors, such as kernel launch failures or memory overflows, can be traced back to this script. It is also the primary location for implementing modifications to the numerical methods or adding new solvers.

c. Parameters

The parameter script (param.cpp and param.hpp) defines and manages the input parameters required for the simulation. It acts as a configuration layer, ensuring that all essential variables are correctly initialised and passed to the simulation workflow. This script is also responsible for providing default values for parameters when they are not explicitly defined in the input files. It ensures that the simulation remains robust against incomplete or inconsistent configurations.

Adjustments to this script are necessary when:

- Adding new features or variables to the simulation.
- Customising the simulation for different biological models or experimental setups.

- Debugging errors related to undefined or mismatched parameters.

G. Commands and Flag Usages

This section outlines the essential commands and flags used to execute, customise, and manage the simulation framework. This part tend to face human error as in wrong path. The error usually shown as the file being read as 0, or file not found. Also the code being run without proper compilation previously. The first phase of the simulation's command and flags run as below:

```
./drug_sim -input_deck input_deck.txt -hill_file drug/IC50_file.csv
```

Second phase of the simulation can be run as below:

```
./drug_sim -input_deck input_deck.txt -hill_file drug/IC50_file.csv -init_file
initfile.csv
```

The commands above represent two primary phases of the simulation process. In the first phase, the simulation is initiated with the essential input deck and Hill coefficient file. The -input_deck flag specifies the configuration file (input_deck.txt), which contains simulation parameters such as pacing details, cell models to be used, and output settings. The -hill_file flag points to a CSV file (drug/IC50_file.csv) containing IC50 and Hill coefficient values for the drug being simulated. This step generates initial conditions and baseline data required for further simulations.

In the second phase, the simulation proceeds with additional parameters specified by the -init_file flag. This flag allows the user to include an initial state file (initfile.csv), which represents the system's state at the end of the first phase. By leveraging this initial state, the second phase can bypass reinitialisation and focus on computing drug effects or specific pacing

protocols. This phase is particularly useful for testing drug interactions or extended pacing simulations without repeating initial computations.

In other cell models, more input is required and uses the same method to be registered in the simulation. ORd 2017 requires additional -herg_file as input, since this newer cell model needs herg fitting value from CiPA. This update increases the simulation accuracy. A common ORd 2017 simulation will be called as such:

```
./drug_sim -input_deck input_deck.txt -hill_file drug/IC50_file.csv -herg_file drug/herg.csv
```

While herg.csv is structured as:

```
Kmax,Ku,n,halfmax,Vhalf,slope
5594000,0.0001719,0.9374,147200000,-61.34,NA
```

Each drug will have their own hERG file.

Also in three of the cell models, future update related to inter-individual variability will be applied. In the future, these cell models will require additional -cvar_file (conductance variability) as input. This update will enrich the simulation capability to simulate drug effect in different population segments such as healthy people compared with people living with heart failure history. This update will increase the simulation accuracy and variability. A common ORd 2017 simulation with inter-individual variability will be called as such:

```
./drug_sim -input_deck input_deck.txt -hill_file drug/IC50_file.csv -herg_file drug/herg.csv -cvar_file cvar.csv
```

H. Troubleshooting

This section provides guidance on addressing frequent issues encountered when using the simulation framework. These troubleshooting steps are designed to assist users in diagnosing and resolving problems effectively. Especially in the situation where CUDA debugger were not informative enough due to the uniqueness of the parallelisation.

- File Not Found Errors

Problem: The simulation fails with an error indicating that a required file is missing or cannot be accessed.

Cause:

- Incorrect file path provided in the command-line arguments.
- The file does not exist in the specified directory.

Solution:

Verify the file path in the command. Ensure the paths match the directory structure (e.g., drug/IC50_file.csv). Check if the file exists in the specified folder. If not, create or move the file to the correct location. Ensure file permissions allow read access.

- Compilation Errors

Problem: Compilation fails, showing errors related to missing libraries or incompatible flags.

Cause:

- Required CUDA or GCC/G++ versions are not installed.
- Incorrect flags in the Makefile.

Solution:

Verify that the system has the correct version of CUDA (e.g., version 11.x or later).Check that nvcc and gcc are installed and added to the system's PATH. Review and update the Makefile to match the system configuration. For instance, ensure paths to CUDA libraries and headers are correct.

- Zeroes or Incomplete Output

Problem: The output files contain no data or are incomplete.

Cause:

- Incorrect input parameters or missing initial state files.
- Errors in the numerical solver or GPU execution.

Solution:

Ensure the input deck (input_deck.txt) is correctly formatted and contains all required parameters.Verify that the initial state file (initfile.csv) exists and is properly formatted. Check the GPU kernel execution logs (printed in the terminal) for errors (e.g., memory overflows or segmentation faults).

- GPU Kernel Launch Failures,

Problem: The simulation crashes during execution, or right just before interacting with GPU kernel function.

Cause:

- Insufficient GPU memory to handle the workload.

- Incompatible GPU architecture flags in the Makefile.

Solution:

Reduce the number of samples or concentrations in the input parameters to fit within available GPU memory. For information, in average it takes 2 MB of GPU memory to simulate one sample. Verify the GPU's compute capability and ensure the Makefile includes the correct -arch flag (e.g., -arch=sm_86 for NVIDIA RTX 4090).

- Runtime Errors Due to Command-Line Arguments

Problem: The program crashes or produces unexpected results when running with specific arguments.

Cause:

- Missing or incorrectly formatted command-line arguments.

Solution:

Ensure all required arguments (-input_deck, -hill_file, -init_file, etc.) are included and point to valid files. Refer to the section on "Commands and Flag Usages" for examples of valid commands.

- Mismatched Results

Problem: The GPU simulation results do not match the expected outputs from the CPU simulation.

Cause:

- Numerical instability or precision issues in the GPU solver.
- Issue with drug effect implementation

Solution:

Double-check the numerical solver configurations to ensure consistency between CPU and GPU implementations. Compare the input data files to confirm they are identical for both simulations. Start comparison from non-drug, control data first. If control from both CPU and GPU shows no noticeable difference in action potential shape, it is confirmed that the issue comes from drug effect implementation.

Check the IC50 input, current development applies these values to the ratio of the gate instead of the gate. Previous implementation may implement directly to the gate. These two kinds of IC50 files can be distinguished by looking at the orde of the numbers. Old implementation method usually uses thousands or hundred of thousands, while more recent version usually stays at most around the thousands.

If the IC50 uses a more recent version, ensure in the drug effect implementation function, it does not change the gate, but instead the '_b' of the gate (GNaL_b for example).

- Slow Performance on GPU

Problem: The GPU simulation runs slower than expected, or it runs more than 24 even 48 hours for 1000 pacing or less.

Cause:

- Inappropriate workload size or insufficient parallelisation.
- Suboptimal GPU memory management.
- Less compatible GPU clock speed.

Solution:

Increase the workload size to fully utilise GPU cores (e.g., simulate more samples or concentrations). Profile the simulation using tools like nvprof or Nsight to identify bottlenecks in memory transfers or kernel executions. Ensure the simulation runs on a more recent GPU generation from NVIDIA, specifically the RTX series (RTX 3080Ti, RTX 4090, etc.). Also for information, this simulation tested and optimised for a gaming grade PC GPU with clock speed around 900-1200 MHz. Some of server-grade GPU has a lot of GPU memory but lacking of clockspeed. I conducted a test on a server grade GPU, NVIDIA T4 with 81 GB of GPU memory, but it was considered too slow. It finished the job simulating 100 pacing of ORd 2011 with drug effect in around 36-40 hours.

- Memory Overflows or Leaks

Problem: The simulation crashes with strange, unreadable error message, makes us cannot access the GPU, or hangs due to excessive memory usage.

Cause:

  - Improper memory allocation or deallocation in GPU kernels.

Solution:

Inspect the GPU control script (gpu.cu) for memory management issues.
Use tools like cuda-memcheck to debug memory leaks or access violations.

- Compatibility Issues

Problem: The code does not work on certain hardware or software configurations. Signed by a fail in running the kernel script with no error at all.

Cause:

- Dependency mismatches, such as outdated libraries or drivers.

Solution:

Update the GPU drivers to the latest version recommended by NVIDIA. Ensure the CUDA version matches the code's requirements (CUDA 11.x). Also ensure the Makefile includes the correct -arch flag (e.g., -arch=sm_86 for NVIDIA RTX 4090). The code is not tested on other than GTX or RTX family. The oldest GPU I have ever tested the code was GTX 1660 Ti.

Enhancing the efficiency of animal-alternative *in silico* drug cardiotoxicity prediction through CUDA-based parallel processing		Thesis for Master of Engineering	December 2024	Iga Narendra Pramawijaya