Hybrid Cardiac Models Connecting Cardiac Monolayers with Two-Dimensional Computational Simulations in Real-time

Younes Valibeigi

Department of Physiology

McGill University

Montreal, Quebec, Canada

August 2021

A THESIS SUBMITTED TO MCGILL UNIVERSITY IN PARTIAL FULFILLMENT OF THE REQUIREMENTS OF THE DEGREE OF MASTER OF SCIENCE

Abstract

Cardiovascular disease is the number one cause of death worldwide. Tachycardias, which are potentially deadly rapid rhythms, are often associated with reentry. Reentry, where a wave of excitation rotates around an unexcitable obstacle, is difficult to study as the circuit length is not under experimental control. Developing closed-feedback loop devices that generate user-defined re-entrant pathways can allow us to study and generate strategies for abolishing re-entrant waves. However, these tools must overcome the challenge of dynamically responding to the wave in real-time in a realistic fashion. Until recently, the development of computational simulations that predict cardiac electrophysiological wave propagation required the use of dedicated workstations that typically took many minutes to simulate seconds of activity. With the aid of a newly developed computational library (Abubu.js) that harnesses the power of the graphics card, it is now possible to develop large-scale simulations that rapidly predict wave dynamics. Using these GPU-based simulations, we built a closed feedback loop device that connects a cultured cardiac monolayer with 2D simulations of cardiac tissue in real-time. Optogenetic methods, which use light-activated ion channels to depolarize cardiac cells, are used to trigger propagating waves in cultured tissue. Motion detection cameras, with the aid of appropriate algorithms, capture monolayer activation waves and provide information to the simulation, which in turn stimulates the tissue using microcontrollers and LEDs. This device provides us real-time control over tissue space. It opens the possibility of investigating anatomical re-entrant waves with a new perspective and has the potential to revolutionize approaches for treating arrhythmias.

Abrégé

Les maladies cardiovasculaires sont la première cause de décès dans le monde. Les tachycardies, qui sont des rythmes rapides potentiellement mortels, sont souvent associées à la rentrée cardiaque. La rentrée cardiaque, où une onde tourne autour d'un obstacle inexcitable, est difficile à étudier car la longueur du circuit n'est pas sous contrôle expérimental. Le développement de dispositifs à boucle de retroaction fermée qui génèrent des voies réentrantes définies par l'utilisateur peut nous permettre d'étudier et de générer des stratégies pour abolir les ondes réentrantes. Cependant, ces outils doivent surmonter le défi de répondre dynamiquement à la vague en temps réel de manière réaliste. Jusqu'à récemment, le développement de simulations informatiques qui prédisent la propagation des ondes électrophysiologiques cardiaques nécessitait l'utilisation de postes de travail dédiés qui prenaient généralement plusieurs minutes pour simuler des secondes d'activité. À l'aide d'une nouvelle bibliothèque de calcul (Abubu.js) qui exploite la puissance de la carte graphique, il est désormais possible de développer des simulations à grande échelle qui prédisent rapidement la dynamique des ondes. À l'aide de ces simulations basées sur les GPU, nous avons construit un dispositive à boucle de retroaction fermée qui connecte une monocouche cardiaque cultivée avec des simulations 2D de tissu cardiaque en temps réel. Ce dispositif peut contrôler le tissu cardiaque grâce à l'utilisation de méthodes optogénétiques qui sensibilisent le tissu à la lumière. Les caméras de détection de mouvement à l'aide d'algorithmes appropriés capturent les ondes d'activation des monocouches et fournissent des informations à la simulation, qui à son tour stimule le tissu à l'aide de microcontrôleurs et de LED. Cet appareil nous permet de controller en temps réel l'espace tissulaire. Il ouvre la possibilité d'étudier les ondes réentrantes anatomiques avec une nouvelle perspective et a le potentiel de révolutionner les approches de traitement des arythmies.

Acknowledgments

First and foremost, I would like to extend my gratitude to Dr. Gil Bub for taking me into his lab, for all the support, trust, and encouragement. I was honored to work with him, as well as with all the past and present members of the lab. To Dr. Leon Glass, thanks for the great conversations and for providing me suggestions for my work. I am also honored to receive hands-on training by Dr. Abouzar Kaboudian for his fascinating novel computational library, and thanks for the great advice and supports he provided me. I would like to also thank Dr. Flavio Fenton for receiving me at his lab and for all the training and support I received from his lab members. Importantly, I wish to acknowledge the undergraduate students, Edward Tu and Jake Zhao, whom I received aid for complicated computational problems. Thank you to Miguel Romero Sepulveda and Khady Diagne for their patients and time in preparing numerous tissue cultures for my experiments. Last but not least, I want to thank my wonderful family for supporting me throughout my education.

Contribution of Authors

Dr. Abouzar Kaboudian assisted in implementing his library, AbubuJS, for developing the cellular automata simulation. His three-variable simulation model was used to implement the Fenton-Karma simulation. Tissue cultures were prepared by Miguel Romero Sepulveda and Khady Diagne. The GView software, developed by Dr. Gil Bub, was used for data analysis. The *serialport* connection between the microcontroller and the NodeJS program was built by Edward Tu. This thesis was revised by the supervisor, Dr. Gil Bub.

Table of Contents

Chapter 1. Introduction

Cardiovascular disease is the number one cause of death worldwide. Investigating the dynamics of cardiac propagation can give scientists insight into cardiovascular diseases and aid in developing pharmaceutical treatments.

1.1. Cardiac rhythm:

The heart is composed of four chambers: the left and right atrium on the top of the heart and the left and right ventricles on the bottom of the heart. Atrial muscles push the blood into the ventricles, and the ventricles pump the blood throughout the body, which supplies oxygen, immune cells, nutrients, and regulatory molecules to the body's organs. The heart's rhythm is controlled by a region in the right atria called the sinoatrial (SA) node that acts as a spontaneous pacemaker (F. H. Fenton et al., 2008). This region, depicted in figure 1 (Ryan, 2020), is modulated by hormonal and neural inputs to maintain blood pressure according to the body's needs. The SA node periodically generates electrical waves that travel throughout the tissue and cause the cardiac cells to contract. Each wave propagates linearly throughout the atria before being channeled through the atrioventricular (AV) node (F. H. Fenton et al., 2008). The AV node provides the electrical bridge between the right atria and the ventricles. These electrical waves initiate intracellular calcium processes that produce contractions in the heart muscles to pump blood (F. H. Fenton et al., 2008). Synchronization of excitation waves generates a coordinated cardiac contraction which is needed for optimal blood circulation.

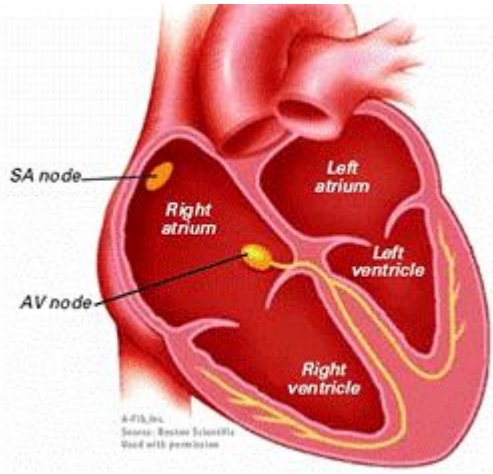


Figure 1. Heart chambers. The figure shows atria and ventricles with SA and AV nodes.

1.2. Action potentials:

Cardiac cell contraction is driven by changes in voltage across the cell's membrane. A transient rapid depolarization followed by repolarization, known as the action potential, leads to the release of calcium ions into the cell's cytoplasm, which triggers contraction. The action potential is generated by currents that travel through specialized ion channels in the cell membrane (F. H. Fenton et al., 2008). Sodium (Na^+), potassium (K^+), calcium (Ca_2^+) and chloride Cl^- are the main ions that determine the voltage state of the cell. The open/close state of various ion channels, which change depending on membrane voltage, result in different conductances for different ions. The balance between concentration and voltage

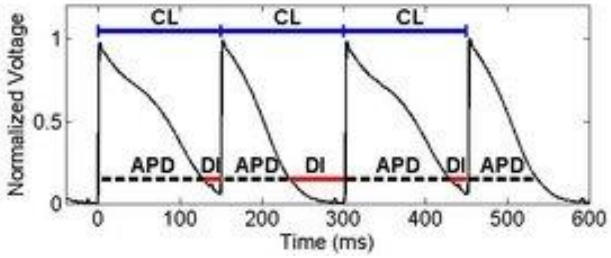


Figure 2. Action potential. The figure shows four consecutive action potentials (AP) and demonstrates examples of action potential duration (APD), diastolic interval (DI) and the cycle length (CL) within each AP.

gradients for each ion and the conductivity of each ion channel results in current flows that sum to generate an action potential. Generally, voltage-gated channels open in the initial step of the action potential and allow the influx of positive ions into the cell. This causes the membrane to depolarize. Following this rapid depolarization, the activity of other voltage-gated channels along with non-voltage-gated channels slowly repolarize the membrane, bringing back the voltage to its resting value. Voltage-gated channels cannot re-open immediately, so the cell cannot generate a new action potential until these channels reset: this time window is called the refractory period. This property allows unidirectional propagation of action potential within each cell and across the tissue. The action potential duration (APD) refers to the time period from the initial depolarization to 90% repolarization[1] (Frame & Simson, 1988)(figure 2). The APD can be associated with the depolarized state of the action potential. The diastolic interval (DI), the time between the end of the last action potential and the upstroke of the following action potential, is often associated with the refractory period for very rapid rhythms (Frame & Simson, 1988).

1.3. Cardiac waves:

The spread of electrical waves across the heart can be explained with an electrocardiogram (ECG). It measures the voltage differences between different points on the chest surface (F. H. Fenton et al., 2008). There are three peaks on the ECG recording: P, QRS, and T (Figure 3). The P wave is associated with the spread of electrical activation across the atria. The QRS complex corresponds to the travel of action potentials through Purkinje fibers and across the ventricles. The T wave is associated with the relaxation of the ventricles. The PR interval corresponds to the duration of the time for excitation to travel from the atria through the AV node to the ventricles. The duration of the ventricle excitation is associated with QT interval. A ‘cardiac arrhythmia’ is defined as a condition in which the cardiac tissue demonstrates an abnormal rhythm due to a disturbance in the electrical wave spread (F. H. Fenton et al., 2008).

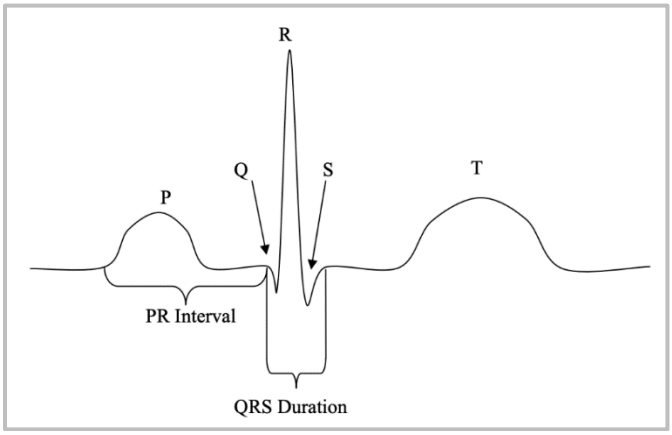


Figure 3. Electrocardiogram. P, QRS, and T waves are caused by cardiac electrical excitations.

1.4. Arrhythmias:

Abnormal wave propagation, abnormal initiation of a cardiac wave, or a combination of both can cause arrhythmias. Arrhythmias can be classified in several ways (Fenton, 2008). It can be classified according to its beating rate. A heart rate that is significantly faster than a normal rhythm (e.g., greater than 100 beats per minute in human heart) is called tachycardia. On the other hand,

[1] Action potential duration can be defined as a time window between depolarization and partial repolarization, where the degree of repolarization (50%, 70%, or 90%) is set by the researcher depending on the experimental context.

bradycardia refers to the condition where cardiac rhythm is slower than 60 beats per minute in human patients.

Arrhythmias are classified according to the electrophysiological properties of its disruption as re-entrant and non-re-entrant arrhythmias. In re-entry, electrical waves repetitively circulate around an obstacle (termed ‘anatomical re-entry’) (figure 4-C) or circulate around a pivot point resulting in the wave taking a spiral morphology (termed ‘functional re-entry’), as illustrated in figure 4-B. Fibrillation is caused when numerous spiral waves (functional re-entrant waves) are generated in the tissue (figure 4-D), and if sustained, it can terminate the patient’s life (Panfilov, 1998). In contrast, a non-re-entrant tachyarrhythmia is associated with different mechanisms. For example, an abnormal rhythm can be generated by having one or more extra pacemaking sites form in the atria or ventricles. Investigating the mechanism of arrhythmias allows us to understand the cause of abnormal cardiac rhythm and find the appropriate cure.

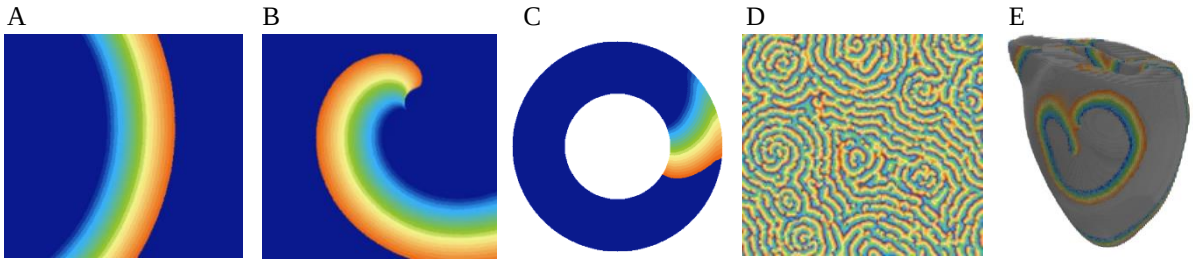


Figure 4. Unidirectional cardiac waves versus re-entrant waves. These figures show 2D cellular automata models developed using the AbubuJS library illustrating cardiac waves. (A) presents unidirectional waves in the healthy tissue. (B) shows a spiral wave in functional re-entry. (C) demonstrates anatomical re-entrant wave circulating around an obstacle. (D) shows fibrillation in the tissue, including multiple spiral waves. (E) demonstrates functional re-entrant waves in a 3D model.

1.5. Closed feedback loops and the hybrid systems:

Hybrid systems are experimental systems composed of a computer simulation coupled bidirectionally to living tissue (Kispersky et al., 2011). A well-known example of a hybrid system is the dynamic patch clamp, where an electrode injects current into an excitable cell in response to the voltage across the cell’s membrane and a computer model (Prinz & Cudmore, 2011) in order to shape the membrane’s voltage. The dynamic clamp can be utilized to provide direct answers to numerous research questions regarding neuronal (Sharp et al., 1993) and cardiac (Wilders, 2006) dynamics. A key element of hybrid systems is the use of closed feedback loops.

Closed-feedback loop systems enable novel research and therapies by controlling living tissue based on physiological parameters acquired during the experiment. In contrast to open-feedback loop systems, that rely on the application of predetermined stimuli, closed-feedback loops dynamically respond to living tissue in real-time, providing new strategies for studying in vivo cardiac dynamics (Scardigli et al., 2018). A feedback loop formed by bidirectional communication between cardiac tissue and the computer can mimic the properties of anatomical re-entrant waves (figure 5-B), while giving researchers experimental access to key parameters. For example, the length of the re-entrant circuit can be dynamically changed during an experiment leading to

situations where the excited head of the wave reaches its refractory tail and break the wave (figure 5-C). Hybrid closed feedback loop models are capable of demonstrating this phenomenon.

The first closed-loop system was developed to explore the dynamics of cardiac re-entry with variable length (Frame & Simson, 1988). Their results suggest that unstable tachycardias demonstrate alternans, ending in termination (Frame & Simson, 1988). More generally, their paper also demonstrated that closed feedback loop systems are useful experimental models for investigating anatomical re-entry. Some other closed-loop applications in cardiac dynamics include variably timed stimuli to control arrhythmias in cardiac tissue using simple models (Christini & Collins, 1996) and dynamic clamps that control physiological properties of cardiac tissue (Wilders, 2006) or neuronal cells (Sharp et al., 1993) in real-time.

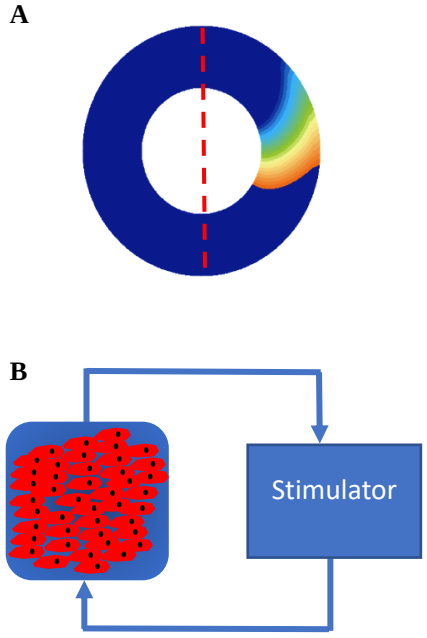


Figure 5. Anatomical re-entrant waves can be studies by closed-loop systems. (A) illustrates a model of anatomical re-entry. Image (B) indicates that a closed feedback loop that can timely stimulate the tissue will simulate re-entrant loops. (C) represents one of the mechanisms that the re-entrant wave disappears. The active head of the wave reaches the refractory tail, and the wave resolves itself.

Hybrid systems must interact with tissue very rapidly (at kilohertz rates or higher) in order to avoid artifacts. Early systems used analog circuits (Sharp et al., 1992; Tan & Joyner, 1990) to connect two excitable cells with experimentally variable resistance. Later, dedicated digital circuits running simple models were developed that could interact with cells at kilohertz rates (Robinson & Kawai, 1993; Sharp et al., 1993), eventually leading to the use of computer programs and fast acquisition boards for dynamic clamp (Christini et al., 1999; Ulrich & Huguenard, 1996). More recently, a system based on a low-latency distribution of Linux, the Real-Time eXperiment Interface (RTXI), was developed as a general platform for developing hybrid experiments (Patel et al., 2017). However, these systems were not designed to respond to complex and spatially variable 2D dynamics that characterize some arrhythmias. Researchers have used optical recordings to detect waves in closed feedback systems (Iravanian & Christini, 2007), but use static electrodes for stimulation, which eliminates the possibility of dynamically changing the wavefront's shape. Variability of the wavefront's shape can affect cardiac dynamics. Specific wavefront shapes can result in the generation of spiral waves. Static electrodes are not capable of implementing these variabilities. This problem can be solved by the emerging field of cardiac optogenetics.

Optogenetics uses tissues that have been genetically modified to express light-sensitive ion channels which, enables researchers to perturb electrical activity with high spatiotemporal resolution using light (Entcheva & Bub, 2016). Since the first optogenetic applications for mapping the origin of cardiac pacemakers and accurately pacing tissue (Arrenberg et al., 2010; Bruegmann et al., 2010), optogenetic tools have been used to study cardiac dynamics. A few studies tried to validate optogenetics as a method for clinical therapeutic approaches (Ambrosi et al., 2014), either by using multi-site illumination using fiber optics (Nussinovitch & Gepstein, 2015)., or by using projected patterns (Bruegmann et al., 2016; Crocini et al., 2016; Nyns et al., 2017). Crocini et al. 2016 used patterned optogenetic stimulations shaped according to the previously acquired data from optical mapping recordings to defibrillate the tissue. In this approach, optical manipulation is not receiving feedback from the tissue, and the approach is limited the optical manipulation's applicability to stationary dynamics (Fixed pattern for optical manipulation).

There have been several recent advances that demonstrate that a more flexible optogenetic strategy is possible. A fully optical approach has recently been developed with a high resolution that achieved optical control of cardiac waves (Burton et al., 2015; Entcheva & Bub, 2016). Welsh and colleagues demonstrated that, in principle, a complex cardiac simulation could connect to the tissue via the use of NodeJS server and microcontrollers (Welsh et al., 2019). However, they did not demonstrate the applicability of the system with living tissue or perform any cardiac experiment. Another fully optical approach with a closed-loop system used high speed cameras as the sensor combined with a digital projector as the actuator (Biasci et al., 2020; Scardigli et al., 2018). This approach used a fixed delay protocol where the tissue is stimulated after a fixed period according to its excitation cycles. These scientists showed various dynamics with different fixed delay periods. There are potential limitations with this approach: fixed delay protocols with millisecond response times require expensive specialized equipment (high temporal resolution cameras and high-speed digitizing boards with their own processors) and also face inherent issues with the fixed delay protocol, which are discussed below.

1.6. Fixed delay limitations:

In fixed delay, re-entrant wave break mostly happens at the site of stimulation, but the break can happen at any point in the real re-entrant loop. Although the re-entrant wave can have different wavelength with various velocities, fixed delay protocols also assumes that the wave travels with constant velocity in a linear shape without encountering its refractory tail (figure 6-B). Previous experiments only stimulated the tissue with static patterns (Scardigli et al., 2018), but the cardiac wave can have various shapes and patterns (figure 6-C). We suggest that replacing fixed delay with a 2D simulation could be a breakthrough in this field and eliminates some of these limitations (figure 6-D). However, conventional 2D cardiac simulations require dedicated workstations to run, and achieving real-time tissue simulation is difficult when using computational processing units (CPUs) with a limited number of cores.

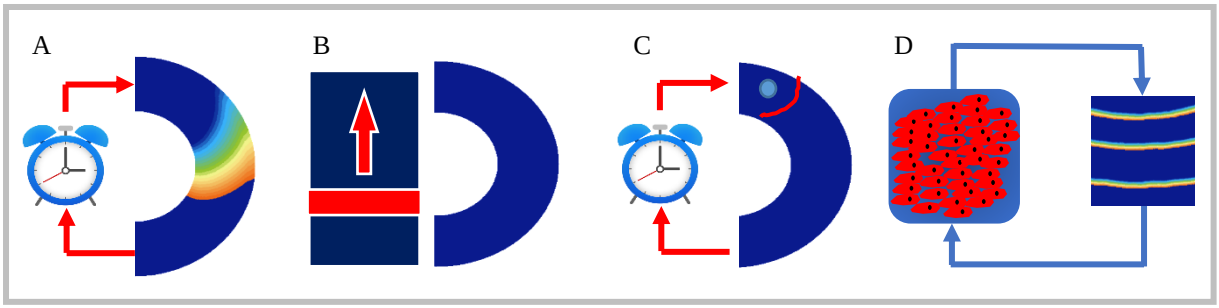


Figure 6. Closed feedback loop system with fixed delay protocol. (A) demonstrates a simple fixed delay system. Based on detected cardiac activity, the system stimulates the tissue after a fixed delay. (B) shows several limitations of fixed delays. The system assumes the waves travel unidirectionally with constant velocities and wavelengths. (C) shows the spatial limitation of the previous experiments. The static stimulation is not necessarily representative of real waves with various shapes. (D) represents our approach in replacing fixed delay with a 2D cardiac simulation.

1.7. Real-time cardiac simulations:

Dr. Kaboudian and his colleagues at the Georgia Institute of Technology developed a new simulation library using the Web Graphics Library (WebGL) programming language (Kaboudian et al., 2019a). This library (*Abubu.js*) parallelizes the computations by running on graphic processing units (GPU), allowing simulations which normally require dedicated workstations to run on normal desktop computers equipped with a modern graphics card. This new approach can develop high-performance simulations of cardiac tissue that can run fast enough to interact with living tissue in real-time. This approach has not been used before in the field of experimental cardiac electrophysiology. It was shown that *Abubu.js* is capable of developing iterative 2D models that can represent differential equation-based models such as Fenton-Karma and OVVR models that can generate 2D waves with variable wavelength and velocities (Kaboudian et al., 2019b).

1.8 Experimental requirements: the need for millisecond resolution in cardiac experiments

Cardiac dynamics are vulnerable to precisely timed pulses (Starmer, 2007), meaning that a millisecond shift in the timing between two tissue's stimulation time (also known as the S1-S2 delay) can affect tissue's excitability and change its dynamics. Cardiac stimulation protocols, therefore, have millisecond accuracy, as indicated in S1-S2 protocols in published papers (Tran et al., 2007). Previous studies indicate that small changes in fixed delay times of a few milliseconds resulted in a change in cardiac dynamics (Hall et al., 1997). Experiments investigating oscillation in conduction also demonstrate a large shift in cardiac dynamics when the value of fixed delay changes for a few milliseconds (Sun et al., 1995). For all these reasons, millisecond resolution is crucial for a closed-feedback loop system to investigate cardiac dynamics.

1.9. Our approach:

Here, we developed a hybrid system that connects real-time 2D computational simulations developed by *Abubu.js* with cardiac monolayers using a fully optical system. We controlled cardiac dynamics with the use of matrix LEDs connected to a microcontroller exciting virally infected

ChR2-expressing cells from mice by projecting user defined patterns at precise times on the monolayer based on detected cardiac activities and simulation feedbacks. Although we used inexpensive cameras with a slow frame rate, we showed that the system allowed real-time intervention within 2 ms of the detected activity, enabling tailored and tunable modification of cardiac dynamics both with 2D cardiac simulations and fixed delays. We showed that our system works both with fixed delay and 2D simulations. This new system opens the possibility of investigating anatomical re-entrant waves with a new perspective. It can replace the simple fixed delay protocol with dynamic 2D simulations, providing more realistic control over both time and tissue space.

Chapter 2. Method

2.1. Overview:

The hybrid model builds a bidirectional communication system between the cardiac monolayer and the 2D cardiac simulations (where the simulation can be replaced by a constant for a fixed delay protocol), as shown in figure 7. The Basler camera uses a motion detection technique to record excitations in the tissue. The real-time simulation runs parallel to the tissue's activity and sends its feedback to the microcontroller (Arduino Uno). The Arduino activates the LEDs to project light patterns on the ChR/2 infected tissue, which provides control over the tissue's space. NodeJS server communicates data by acting as a buffer connecting different pieces of the device. The camera and the microcontroller are synchronized to allow real-time intervention within 2ms of detected activity.

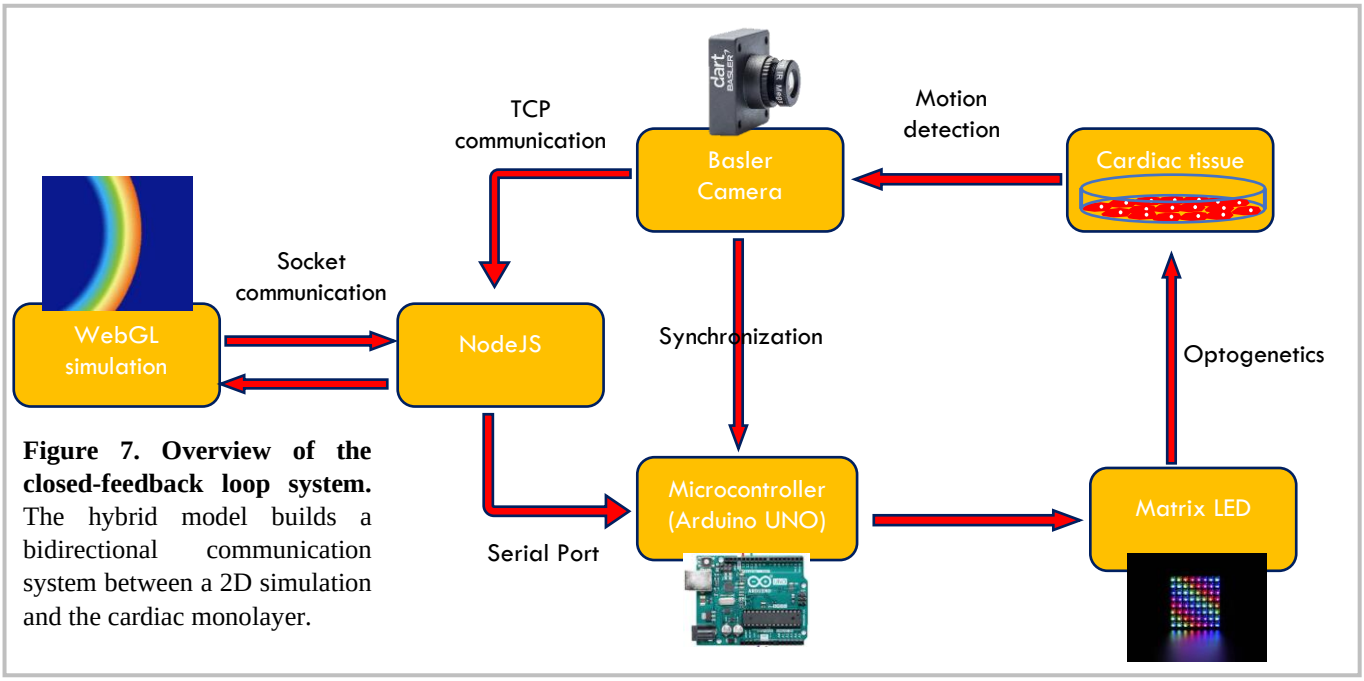


Figure 7. Overview of the closed-feedback loop system. The hybrid model builds a bidirectional communication system between a 2D simulation and the cardiac monolayer.

2.2. NodeJS and Socket communications:

NodeJS works as a buffer and connects the simulations with the camera, microcontroller and matrix LEDs. NodeJS is a back-end JavaScript runtime environment that executes JavaScript code outside a web browser: it enables us to run JavaScript programs on the local terminal (command prompt for Windows). It works as a server and communication with the client (HTML simulation) through io sockets. It provides a real-time communication system between the server file (JavaScript) and the *main.js* file in the public folder (Client). NodeJS file also communicates with the microcontroller and the camera. A Serial Port (version 9.0.7) library provides the link between NodeJS and the Arduino. The TCP (transmission control protocol) socket builds the communication system between the camera and NodeJS. The initialization and development of the NodeJS server and its corresponding socket communication is explained in detail in Appendix A.

2.3. WebGL simulation:

The cardiac cells' action potentials are modulated by excitation waves travelling in the tissue. Mathematical modeling can represent these excitation waves in 2D monolayers or 3D hearts (figure 8). However, computing 2D simulations with large arrays can be computationally very expensive and until recently required dedicated workstations (e.g., supercomputers) to reach real-time. Parallel processing is the solution to achieve the activation speed of the cardiac tissue with inexpensive computers. WebGL is a JavaScript application programming interface (API) that harnesses the power of the graphics card for parallel processing enabling computations that are not possible with previous approaches in computational modelling (Kaboudian et al., 2019c). WebGL implements shaders using the GLSL language. The shaders' code can be made for one computational cell, and GPUs can run the code parallel to other simulation cells (Gonzalez Vivo & Lowe, 2015). Unfortunately, WebGL is designed for graphical usage and implementing its feature for building cardiac simulations is complicated. AbubuJS facilitates the use of WebGL for developing computational models and allows to produce 2D and 3D cardiac simulations with a simpler approach (Kaboudian et al., 2019b).

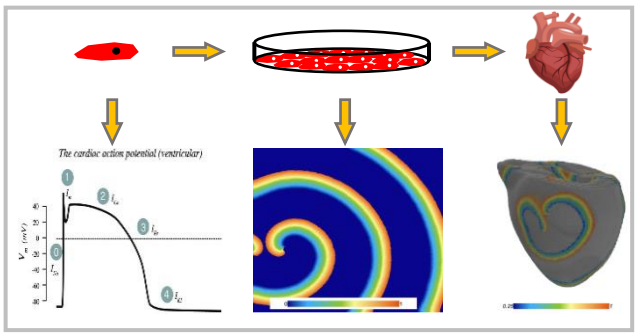


Figure 8. Modeling cardiac electrophysiology from single cells to tissue. AbubuJS library can be used to develop mathematical models demonstrating electrical activity in 2D and 3D tissue.

Every WebGL program employs two types of shaders (Vertex shader and Fragment shader) to develop a graphical pipeline for image visualization (Kaboudian, 2019/2021). The Vertex shader imports an object from the 3D physical world into a graphical cube called Clip Space (figure 9) and defines the position of every pixel on the screen according to the location of the object in the clip space. Fragment shaders fill the pixel (the plane with z=1) with a specific color according to their pixel position (location of the pixel) demonstrated in figure 9. A 4D *rbga* vector can represent every pixel to make the pixel color: vec4(red, green, blue, transparency). The pixel color would be displayed on a canvas defined in the HTML program.

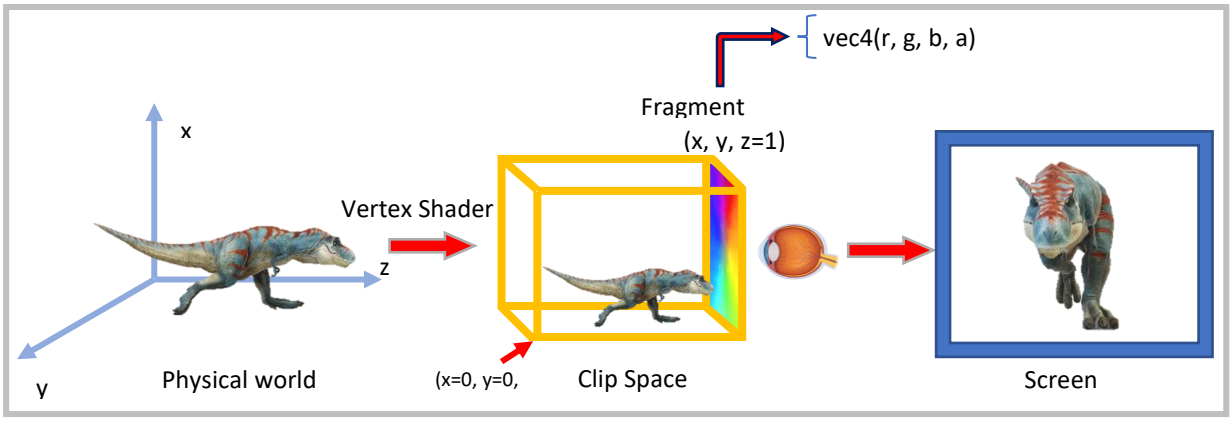


Figure 9. Graphical pipeline. WebGL uses the graphical pipeline to show images. It uses GLSL programs called shaders. The Vertex shader is a program that transfers an object from the physical world to the Clip Space. Fragment shaders fill every pixel on the Clip Space's right side with colors according to the object in the Clip Space.

The WebGL program is embedded in an HTML document. The HTML file should include a JavaScript code and at least two GLSL codes: the vertex and fragment shaders. The JavaScript code generates Textures (arrays that have 4D vectors representing pixels) and sends these textures to shaders for parallel processing. The shaders apply their algorithm for every pixel in the texture (Gonzalez Vivo & Lowe, 2015), and the GPU then runs them in parallel. The program is capable of running 2D simulations and 3D models of the heart. AbubuJS facilitates this process and removes most of the complexities involved in coding with graphic aspects of WebGL (Kaboudian, 2019/2021). AbubuJS projects organize the code in a specific folder structure displayed in figure 10 (the folder structure under the client section). The *index.html* file calls the JavaScript file (*main.js*) and the shaders (stored in the shaders folder). The *main.js* file builds the textures and sends them to shaders. Every 4D cell in the texture can store four variables such as cell voltage, channel conductivity, or time (note that in AbubuJS, vec4 variables are not treated as colors but as simulation variables). If more than four variables are required for each cell, extra textures can be defined. The *compShader.frag* (fragment shader) file includes the algorithm associated with a single computational cell in the simulation algorithm. WebGL runs the GLSL shader code in parallel for every cell in the texture. After parallel processing by shaders, *main.js* can display one of the texture variables, usually voltage, with a chosen color map on a canvas for visualization. *Require.js* program (implemented in *config.js*) manages all dependencies for each file.

NodeJS server was developed and connected to the simulation (*main.js* JavaScript file) through the *socket.io* communication system, shown in figure 10. NodeJS, running on the local terminal, sends a public folder to the web browser, including the files and libraries related to the Abubu project. The server should be run by the Node program from the terminal. The WebGL code can be accessed from the browser using a *localhost*. The web browser runs the WebGL program and displays it on the screen. A bidirectional communication system (*socket.io*) enables real-time server-client communication. A detailed explanation for developing this structure is provided in Appendix B.

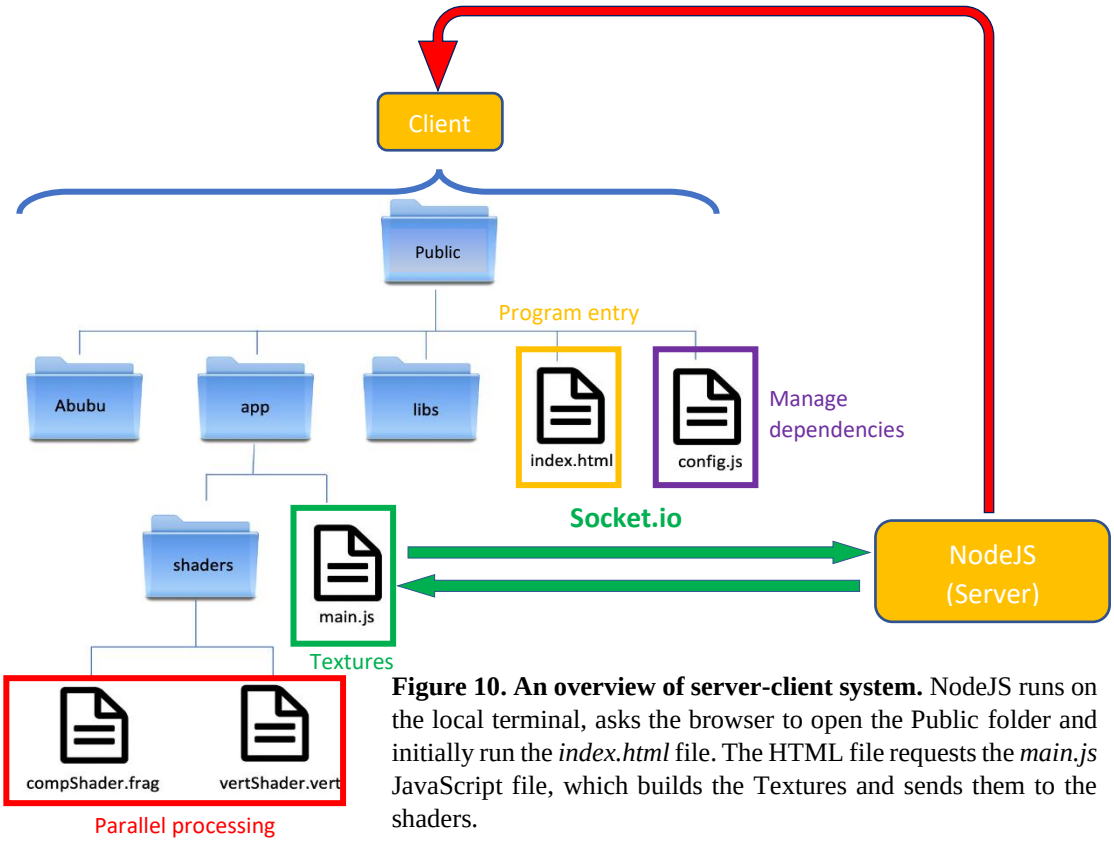


Figure 10. An overview of server-client system. NodeJS runs on the local terminal, asks the browser to open the Public folder and initially run the *index.html* file. The HTML file requests the *main.js* JavaScript file, which builds the Textures and sends them to the shaders.

2.4. Cardiac simulations:

AbubuJS library was used to implement two cardiac simulations: the cellular automata and the Fenton-Karma model. Cellular automata (CA) are discrete models that can simulate electrical wave propagation in 2D and 3D space. Cellular automata approximate electrical wave propagation in cardiac tissue by only simulating tissue-level properties, such as excitability and refractoriness (Zhu et al., 2004). The state of each cell in a cardiac cellular automata model changes only according to the activation state of each neighboring cell and its own refractory status. As a result of these simplifications, cellular automata run simulations much faster compared to the differential equation-based models. In the current CA model, every cell has a radius of excitability (Bub et al., 2002). If the ratio of active cells over total cells passes the threshold value in this radius, the cell will become active in the next iteration (figure 11-A). The threshold value is set, so that wave propagation in the model approximates wave propagation in the cardiac tissue being modelled: low threshold values result in high conduction velocities and waves that are insensitive to heterogeneities, and high thresholds result in low conduction velocities and waves that are modulated by heterogeneities (Bub et al., 2002). After each activation, the cardiac cell goes into a refractory phase, which lasts for a few iterations. The refractory period is one of the reasons that cardiac waves propagate in a well-defined direction. The above rules are mathematically modeled as follows: Each cell voltage was scaled to be in the interval of 0 and 1 (figure 11-C). The cells are initialized with zero voltage (V=0). Any cell with a voltage below 0.05 is still in the excitable

state. Any cell that goes to a voltage above 0.05 is in the refractory state and cannot be activated for the next iterations. An action potential is represented by an increased cell voltage to 0.98, and this transition can only happen if the cell is in the excitable state. The cell voltage decreases by 0.051 in each iteration for recovery. Appendix B.3. provides the GLSL shader code for the cellular automata algorithm.

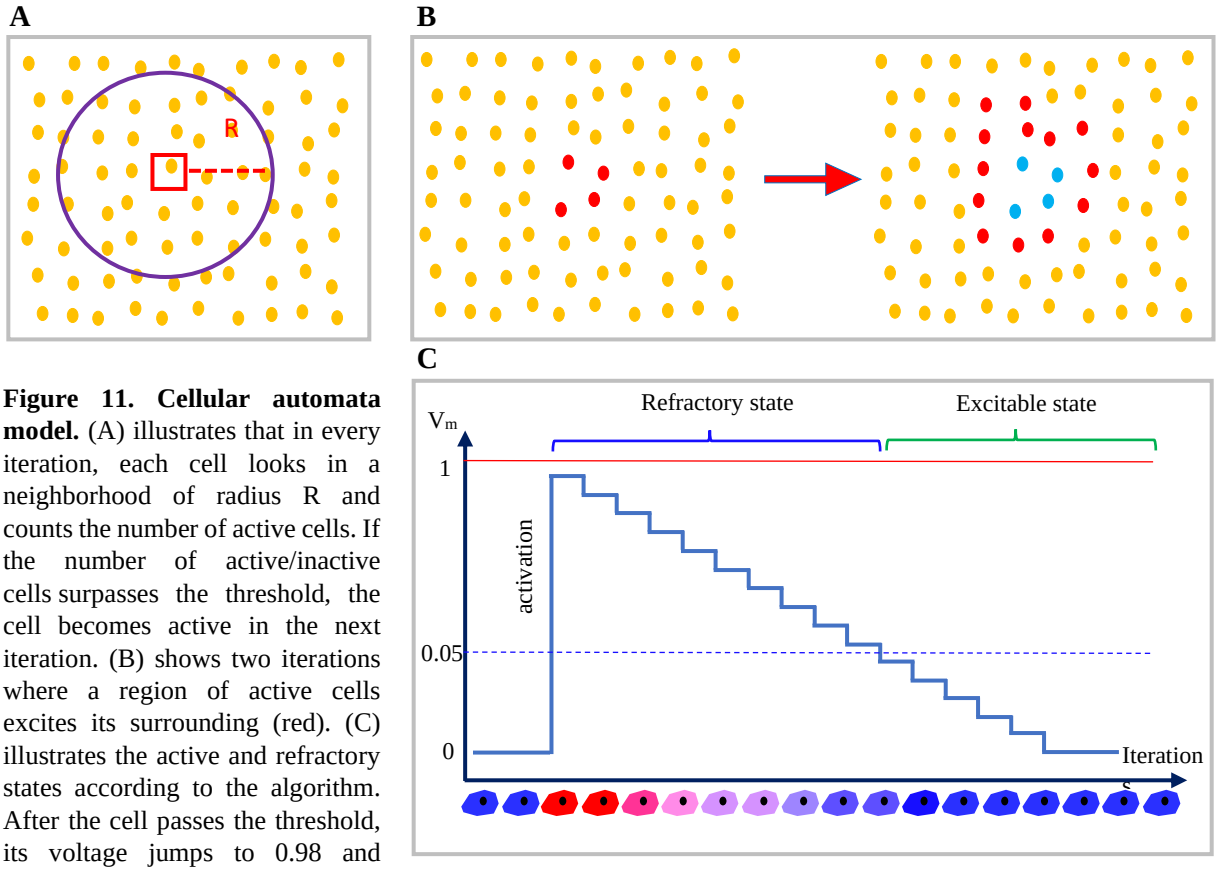


Figure 11. Cellular automata model. (A) illustrates that in every iteration, each cell looks in a neighborhood of radius R and counts the number of active cells. If the number of active/inactive cells surpasses the threshold, the cell becomes active in the next iteration. (B) shows two iterations where a region of active cells excites its surrounding (red). (C) illustrates the active and refractory states according to the algorithm. After the cell passes the threshold, its voltage jumps to 0.98 and gradually declines in each iteration.

The Fenton-Karma (FK) model is a continuous system governed by three differential equations (F. Fenton & Karma, 1998). The diffusion term expresses the spread of excitation waves. The FK model is capable of demonstrating more complex 2D dynamics such as excitation waves with various wavelengths and conduction velocities (figure 12). The FK algorithm is demonstrated in equation 1 (Tolkacheva et al., 2002):

$$\frac{dv}{dt} = -\left(J_{fast} + J_{slow} + J_{ung} + J_{stim}\right) + diffusion\ term$$

$$\frac{df}{dt} = \frac{[f_\infty(v) - f]}{\tau_f(v)} \Rightarrow J_{fast} = -\frac{fQ(v)}{\tau_{fast}}$$

$$\frac{ds}{dt} = \frac{[s_\infty(v) - s]}{\tau_s(v)} \Rightarrow J_{slow} = -\frac{sS(v)}{\tau_{slow}}$$

$$J_{ung} = P(v)/\tau_{ung}$$

$$v = \frac{dv}{dt} \times dt$$

$$f = \frac{df}{dt} \times dt \qquad (1)$$

$$s = \frac{ds}{dt} \times dt$$

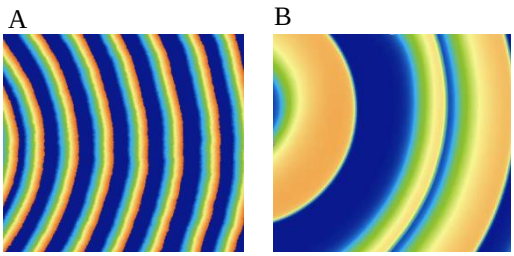


Figure 12. Comparing cellular automata model with Fenton-Karma model. Cellular Automata's (A) simplicity allows it to run the simulation with a fast speed, but it fails to demonstrate various properties of the tissue, such as ion conductivity and variable refractory periods. It can only generate waves with similar speeds and wavelengths. Although it is computationally heavier, the Fenton-Karma model (B) can generate more realistic waves with various velocities.

2.5. Cardiac monolayer:

Postnatal mice hearts were isolated and dissociated on days P0-P3. The combination of enzymatic degradation and mechanical dissociation was used to prepare a cell suspension. The enzymatic solution degrades the extracellular matrix in the cardiac tissue while maintaining cell structural integrity. The protocol for tissue culture coating and media preparation is described in Appendices F.1. and F.2. To avoid contamination of cardiac fibroblasts, ventricular cells were pre-plated in plastic dishes for one hour. Monolayers of 1 cm^2 diameter were used for the recordings. The procedure for plating cell densities is detailed in Appendix F.1. The tissue cultures were kept at 37 ^oC and 5% CO_2 humidity, and recordings were carried out in an environmental chamber with the same CO_2 and humidity levels. The guidelines for the animal handling were performed in agreement with the Canadian Council on Animal Care, and the procedure for euthanizing neonates was in agreement with McGill University SOP 301-01 under approval protocol 2018-8044.

Following the formation of the cardiac monolayer, 48 hours after plating for ventricular cells, the tissue culture is infected with adenovirus vector type 5 (dE1/D3) to express channelrhodopsin-2 mutant H134R (CHr2(H134R)) (Appendix F.2.). The multiplicity of infection (MOI) that was used is 100. After viral infection, the culture media was replaced every 48 hours.

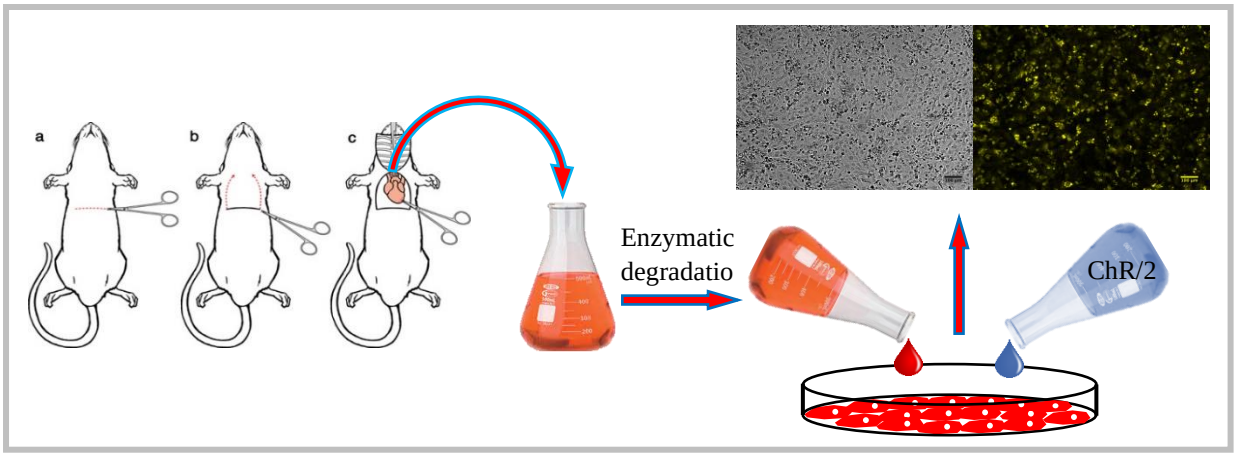


Figure 13. Schematic illustration of neonatal mice ventricular monolayer preparation. After enzymatic degradation and monolayer preparation, the cells are infected with adenovirus vector type 5 (dE1/D3), which causes them to express channelrhodopsin-2 (blue solution) which modulates the cells' membrane potential in the presence of 470 nm light. On the top right, brightfield and fluorescence images of neonatal mouse cardiac cultures tagged with YFP are shown.

2.6. Basler Camera:

A Basler Ace camera (model: acA1920-155um) was used for the recordings. To develop the necessary software, Pylon 6.0.1 Camera Software Suite (Windows 10 version) was used. The settings for the camera by Pylon Viewer 64-bit is shown below:

- Analog control ➔ Gain Auto: Continuous
- Acquisition Control ➔ Exposure Time [us]: 30000.0
- Acquisition Control ➔ Enable Acquisition Frame Rate: Check
- Acquisition Control ➔ Acquisition Frame Rate [Hz]: 30.0
- Digital I/O Control ➔ Line Selector: Line 3
- Digital I/O Control ➔ Line Mode: Output
- Digital I/O Control ➔ Line Source: Exposure Active

The camera was set for 30 FPS. Every frame took 33.33 ms. According to the developed software, the camera stores 500 frames per experiment. In order to make the device as accessible as possible, motion detection was used instead of voltage detection techniques. Motion detection has the advantage of running an experiment for a longer duration with inexpensive tools. A specific algorithm was developed to detect motion in the tissue. A C^{++} program that linked to the Pylon software development kit (SDK, version 6.0.1) was written using Microsoft Visual Studio (version *community* 2019). The unprocessed image frame from the camera would not give us any information about the excitation wave (figure 14), so for every six consecutive frames, the current frame was subtracted from its previous sixth frame, and its absolute value was taken for each pixel value. The resulted image showed motions in the tissue (figure 14). Whenever there were contractions in the tissue, it was shown as the white signal in the image. If the frames were displayed consecutively, the spread of the excitation waves throughout the tissue could be seen (figure 14).

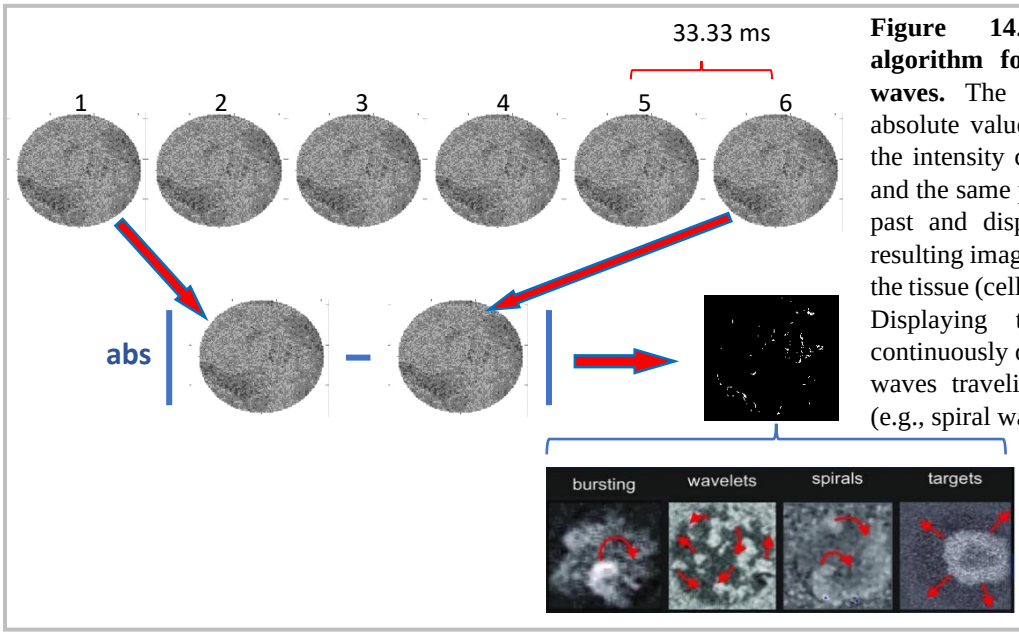


Figure 14. The camera's algorithm for detecting cardiac waves. The algorithm takes the absolute value of the difference of the intensity captured at each pixel and the same pixel six frames in the past and displays the result. The resulting image represents motion in the tissue (cells that are contracting). Displaying the filtered images continuously demonstrate excitation waves traveling in the monolayer (e.g., spiral waves).

To detect excitation, recording a single area in the tissue created large errors. It might detect waves at different locations for one frame because the device only looks at the tissue every 33.33 ms (figure 15). In order to reduce the error within 2 ms of detected activity with a low frame rate (frame period = 33.33ms), the following algorithm was developed. The program looked at two boxes in the tissue and took the pixel average of all pixels in those two boxes for every frame. If an increase in the pixel average was detected, the average of every pixel row was taken, and the program measures the row where the average passes a user defined threshold (figure 15). This threshold determines where the wave peak occurs. The procedure for determining it is described in detail in Appendix C.3. The detected row number is the pixel location (y_1) where the wave was detected. The same procedure was applied for the second box (y_2). The number of frames between detection in the first and the second box was stored for every wave (Δf). Using the distance (in pixels) between the two locations that the wave was detected and the difference in time (number of frames), the wave velocity ($Conduction\ Velocity(CV) = \frac{y_2 - y_1}{\Delta f}$) was calculated. Using this conduction velocity, the number of milliseconds it takes for the wave to travel to the end of the field of view was calculated as shown in figure 15 ($\Delta t_{cam} = \frac{\Delta y'}{CV} \times 33.33\ ms$). Thus, instead of detecting the excitation waves at a specific location, it is estimated when the wave arrives at that location. The pseudocode for this algorithm is detailed in Appendix C.

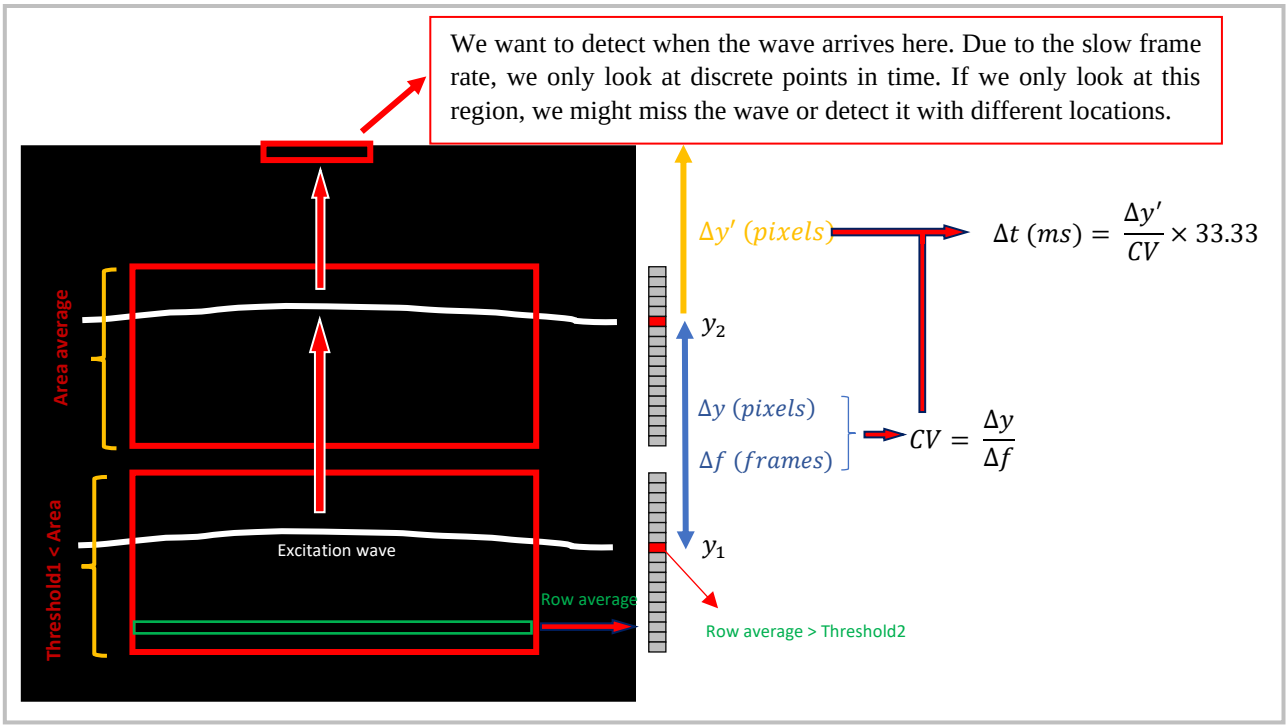


Figure 15. The algorithm for detecting the excitation waves from the filtered images. The algorithm compares the average of all pixel intensities in the two red boxes. If the area average is greater than the threshold (existence of a wave), the program looks at the row averages to detect the peak intensities (the wave location). Using the distance between the detected wave location in the boxes and the time difference, the program calculates the wave conduction velocity. Finally, the algorithm uses this data to estimate when the wave arrives at the top red box. This estimated time will be sent through the feedback loop system.

2.7. Microcontroller and Matrix LED:

Microcontrollers include a single CPU, clock, input, and output that can be used to connect LEDs with the computational models (Welsh et al., 2019). They are fast computers that are increasingly integrated into physical systems, such as toys and automobiles. Microcontrollers are single circuits that can digitally control external processes, and, due to their fast speed, can also be used for scientific computations (Welsh et al., 2019). The Arduino UNO was used and programmed with a language whose syntax is similar to C (Arduino IDE 1.8.49.0). The code was uploaded to the microcontroller from a computer through a USB cord using an Integrated Development Environment (IDE) to control the LEDs. The code had to be uploaded before the simulation runs. The Arduino IDE program is provided in Appendix D.

Matrix LEDs can be used to project light with specific wavelengths accurately on the region of interest. Blue LEDs can activate cardiac cells, which are genetically modified to express channelrhodopsin-2 (ChR2). The 8X8 Adafruit DotStar matrix LED was used for this project. The ground port (GND) on the microcontroller had to be soldered to the ground port on the matrix LED. The matrix LED required 5V voltage, which was soldered to the 5V port on the microcontroller. A program for the Serial Port was developed in the NodeJS code that converts the *socket.io* data into a buffer and sends it to the Arduino through a USB cable. To build the

communication between Arduino and matrix LEDs, the clock port (CLK) on the matrix LED had to be connected to the digital port pin 13, and the data port (DIN) had to be connected to pin 11. The complete circuity is shown in figure 16. Consistent with our experiment's requirements, the *main.js* JavaScript program can detect excitation at any location in the 2D simulation and activate any user-defined pattern of LEDs (figure 16). This feature provides spatial and temporal control of cardiac tissue.

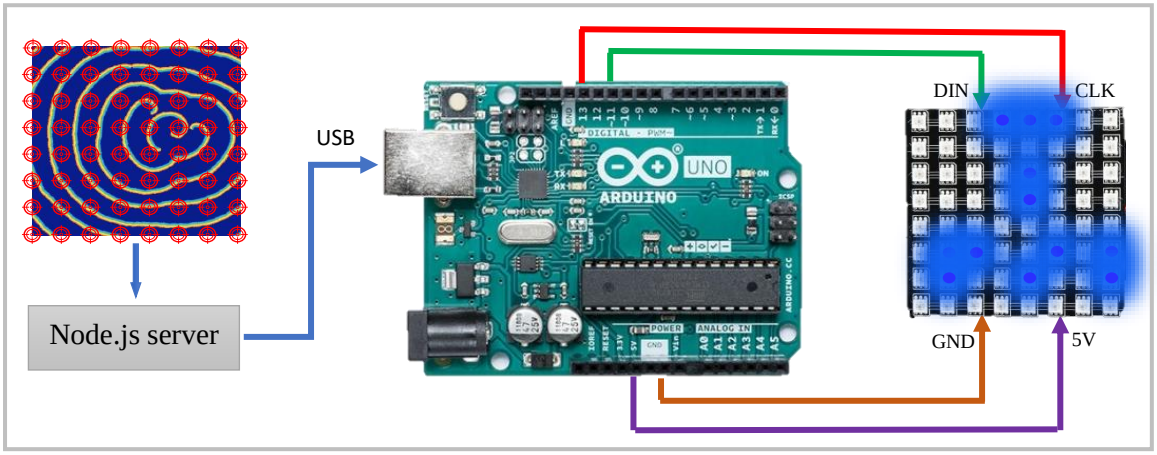


Figure 16. Demonstration of circuitry between Arduino and matrix LED. The LEDs are attached to a 3D printed object. The HTML application and Arduino can only communicate through the server with the use of a USB cable.

In order to create a base for holding the matrix LEDs, a Prusa MK2s 3D printer was used. A 3D object was developed using Google SketchUp to hold the LEDs. The 3D object was calibrated in PrusaSlicer-2.1.1 and printed using the Prusa printer.

2.8. Accuracy test 1-Photodiode recordings:

To measure the system's accruacy, a photodiode (Thorlabs SM05PD1A) connected with a preamplifier (Thorlabs AMP 110) was used. A digital oscilloscope (Picoscope) run through the PicoLog 6 application was employed to record the photodiode. The photodiode is capable of recording light intensity with millisecond precision. It was used to record the flashes of the matrix LEDs. The photodiode specifically detected the light intensity of the LEDs lights demonstrated in figure 17. The figure reflects the light intensity detected by the photodiode. When the LEDs turn on, there is a jump from 0 to -2500 in the photodiode recordings (figure 17). The cycle periods (the time slot between two peaks) were calculated and graphed on a histogram (figure 17). For example, the mentioned histogram shows that the cycle periods are equal to 1101 ms. A Python code is provided in Appendix E.2. that can read the photodiode's data and build the histogram.

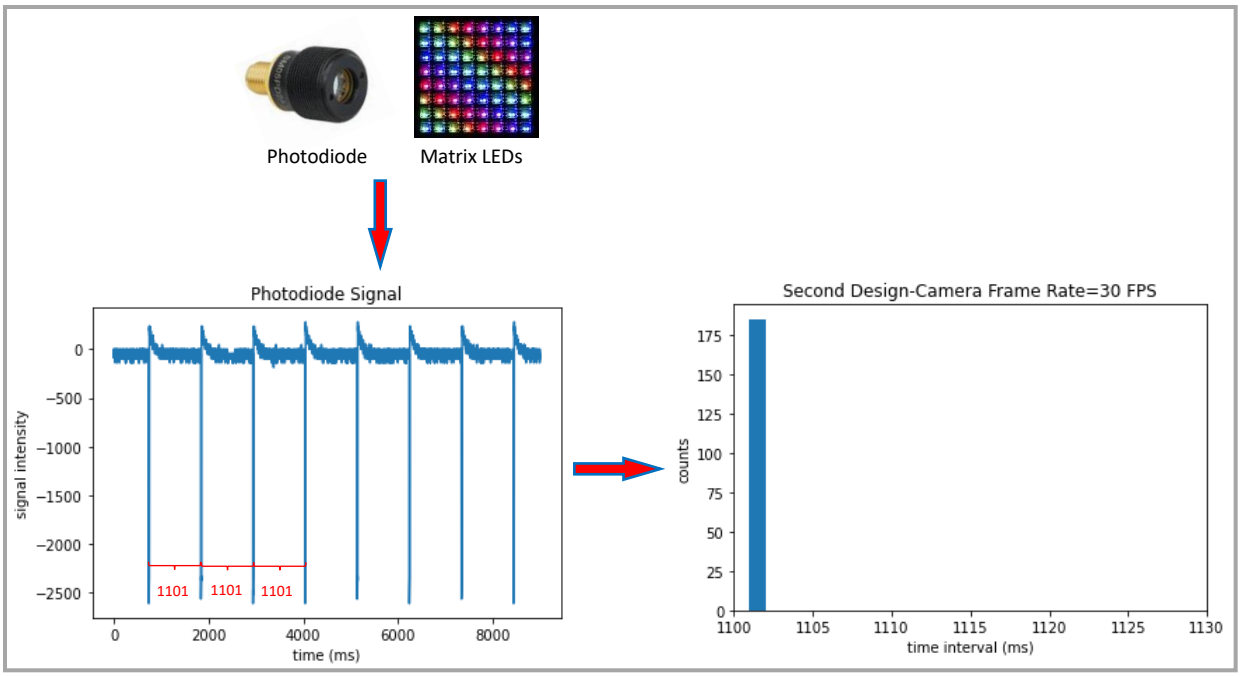


Figure 17. Photodiode records the light intensity of the matrix LEDs, which is used for determining cycle periods. The downward spikes in the photodiode signal (bottom left graph) indicate that the LEDs are turned on at that moment. The cycle periods of the LED flashes (1101 ms each) are shown in red. On the right, there is a histogram of these cycle periods. This histogram shows that all cycle periods are equal.

We developed three different designs to test the amount of error among different connections in the system. In the first design, demonstrated in figure 18-A, an algorithm was developed for the Arduino that periodically (with a cycle period of 1100 ms) turns on the LEDs. The results (Histogram's standard deviation = 0.48 ms) showed that the Arduino-LED connection has low latency and is sufficiently accurate for our experiment. In the second design, figure 18-B, the *serialport* communication system was added to the previous design. This time, a program was written in JavaScript and run using NodeJS to send a message every 1100 ms to the Arduino microcontroller to periodically activate the LEDs. The result suggested that there was a variable delay in this connection. Finally, we programmed the HTML file to send a periodic signal to our NodeJS program, which relays it to the Arduino microcontroller to activate the LEDs (figure 18-C). Adding the HTML program, which would include the WebGL simulations, further increased the error. This result suggested that there were unavoidable variable delays in *socket.io* and the *serialport* communication system. Thus, the camera and the Arduino had to be synchronized to bypass these delays in the system. The information for reproducing these three experiments is provided in Appendices A, B, and D.

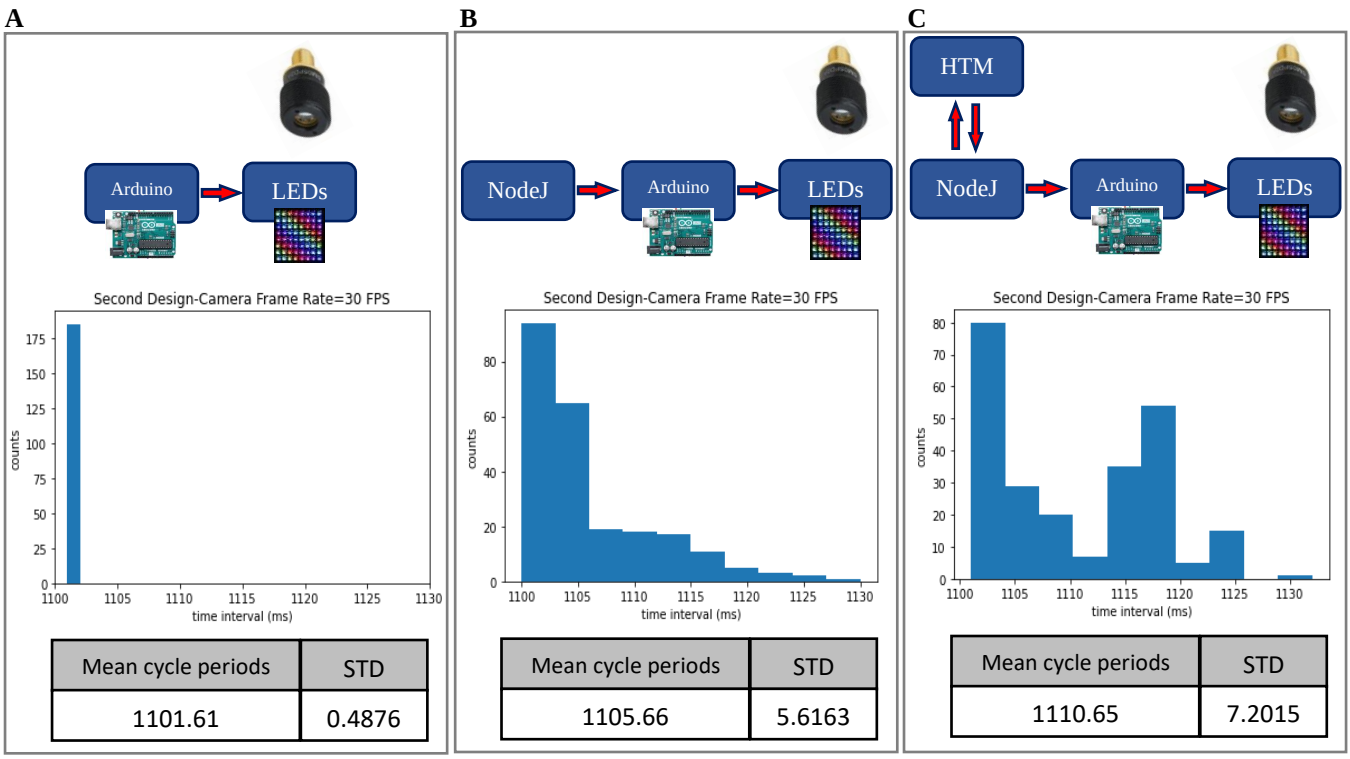


Figure 18. The result of the accuracy tests with photodiode recording. (A) most cycle periods stay within the range of 1 ms from the average (the standard deviation (STD) is less than 1 ms). This result shows that there is low or constant latency in the Arduino-LED connection. (B) indicates that there are a range of significant delays in the *serialport* communication between NodeJS and the Arduino (as there is a significant rise in STD). (C) indicates that adding socket.io communication between the simulation and NodeJS further increases the error. These results show that we need to bypass internal connections.

2.9. Camera-Arduino synchronization:

To connect the camera with the Arduino, Basler Power-I/O 6p/open 10 m Cable was used. The cable is composed of 6 wires: 1. *Brown*, 2. *Pink*, 3. *Green*, 4. *Yellow*, 5. *Grey*, 6. *White*. The brown wire was connected to the Arduino's Analog 3 (A3) pin, and the white wire was connected to the ground (GND) pin.

2.10. Synchronization mechanism:

The Basler camera generates a TTL signal every time a frame is captured. The cable enabled the camera to send this signal to the Arduino, allowing the Arduino to use this signal as a clock for synchronization. The software controlling the Basler camera calculated the time (in milliseconds) that it took for the wave to arrive in the determined location in the camera's field of view (Δt_{cam}). Δt_{cam} is added to the number of frames (F) multiplied by frame periods (33.33 ms) to give t, which is the time it took for the wave to arrive at the determined location since the beginning of the program ($t = \Delta t_{cam} + F \times 33.33ms$) in milliseconds. This value was sent to the WebGL simulation. Once the simulation receives the signal, it simulates a wave travelling from one side of the simulated domain to the other, keeping track of the number of iterations needed. The number of iterations is multiplied by dt, which is a scaling factor determined by experimental conditions ($\Delta t_{sim} = F_{sim} \times dt$, where dt is the conversion factor that scales an iteration in the simulation to

real-time in milliseconds). dt is set to scale conduction velocity in the simulation with wave speed in real tissue. It depends on the simulation length (SimLen), tissue's approximate conduction velocity (CV), and approximate number of iterations for the wave to travel the SimLen (# itr): $dt = SimLen/(CV \times \#\ itr\ ))$. Then, the calculated time for the simulation was added to the signal ($t = t + \Delta t_{sim}$), and this value is sent to the Arduino through the *serialport*. The microcontroller divides this value (t) by the frame period and determines how many camera frames this corresponds to, plus the amount of remainder in ms ($t/33.33ms = F_{total} + remainder$). The Arduino then pauses until the right number of TTL pulses from the camera is detected (F_{total}), then pauses for an additional time equal to the remainder at which point it activated the LEDs. During this process, every time the camera caught a frame, it updated Arduino's frame number ($F_{Arduino} = F_{Arduino} + 1$).

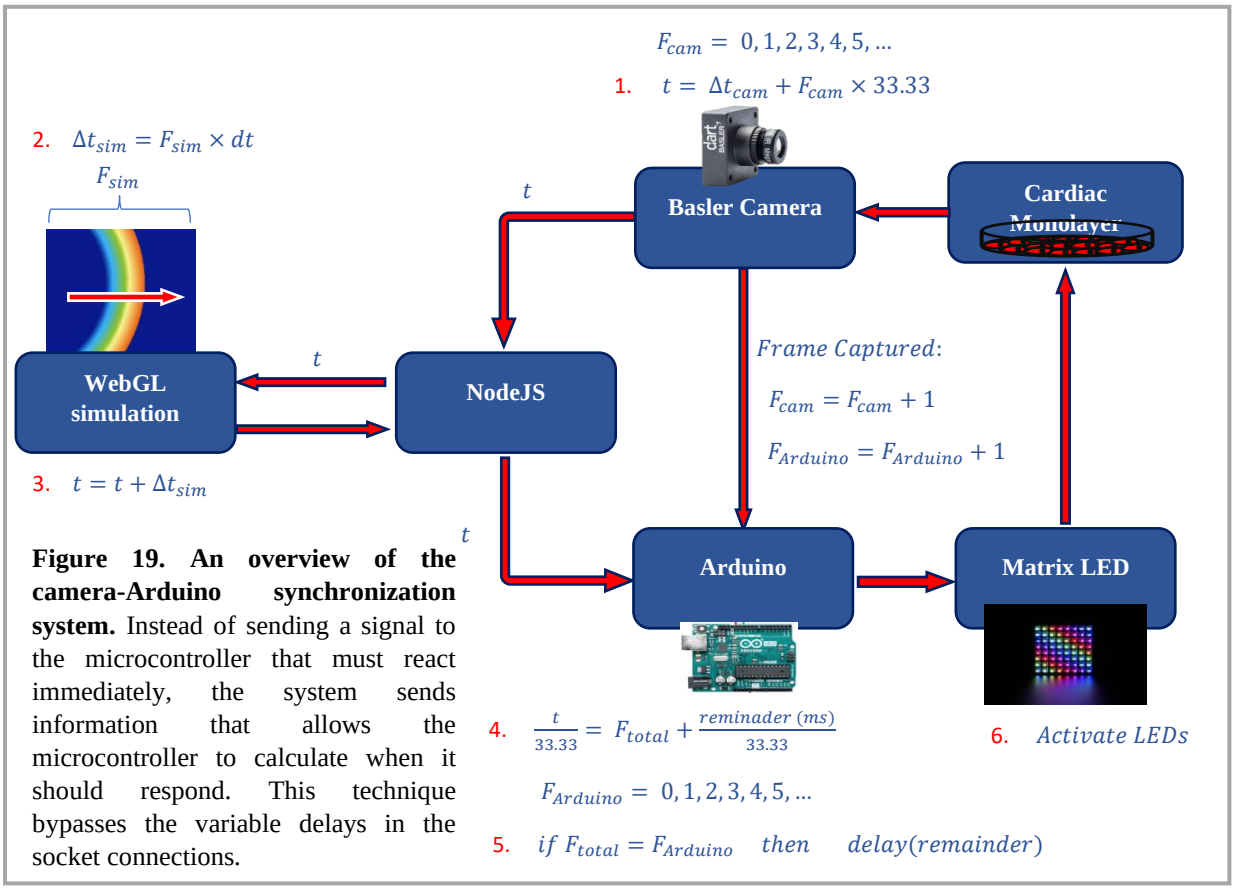


Figure 19. An overview of the camera-Arduino synchronization system. Instead of sending a signal to the microcontroller that must react immediately, the system sends information that allows the microcontroller to calculate when it should respond. This technique bypasses the variable delays in the socket connections.

For example, assume the camera detected an excitation wave at the ($F_{cam} = 2$) and predicted that it would arrive at the determined location in 3 frames ($F_{cam} = 2 + 3 = 5$). Thus, the corresponding time in milliseconds was $t = 5 \times 33.33 = 167\ ms$. If the simulation wave took 420 frames to travel ($F_{sim} = 420$) and we assume that dt is 0.5 ms, then Δt_{sim} would be 210 ms. The microcontroller would receive a message that $t = 167 + 210 = 377\ ms$. The microcontroller software then calculates the number of frames plus a remainder: ($\frac{377}{33.33} = 11\ frames + \frac{10\ ms}{33.33}$).

Therefore, the microcontroller would wait until the 11th frame, then it would delay another 10 more milliseconds to stimulate the LEDs. While the microcontroller still must receive a time value from NodeJS using *socket.io*, which introduces variable delays, any small delays in receiving the message can be ignored as long as these delays are less than the calculated value of *t*. Using this technique, instead of sending a signal to the microcontroller that has to react immediately, the system sends information that allows the microcontroller to calculate when it should turn on the LEDs. This technique bypasses the variable delays in the socket connections and increases the accuracy.

2.11. The accuracy test for the hybrid system:

To test the device's accuracy, a DMD projector (Vialux XGA1303), controlled by a computer installed with Jython (Java+Python) software, was used to make artificial wave mimicking real tissue excitation waves. The projector applied patterns of waves to the microscope stage (figure 20-A), with periods of 1100 ms. The camera recorded these patterns (as shown in figure 20-B) and sent them to the WebGL program that implemented a simple fixed delay protocol – returning a fixed time instead of running simulations. After the delay, LEDs were activated using the scheme described above (Section 2.10). Note the LEDs did not affect the artificial wave generated by the projector. Since the projectors' periods were constant and the system only added a constant delay, theoretically, the cycle periods detected by the photodiode should be constant. This test gave a measurement of the system's response accuracy.

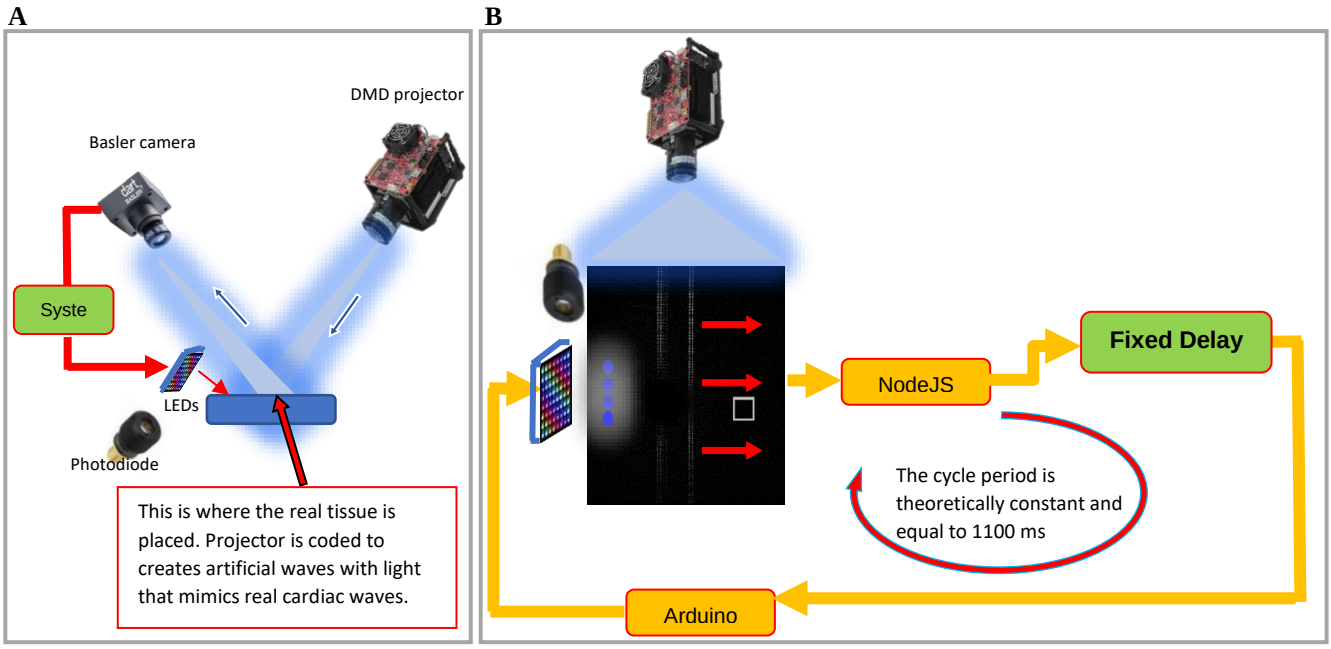


Figure 20. This accuracy test was designed to measure system's cycle periods. The DMD projector produces patterns of light mimicking cardiac waves. (A) shows that the light is projected on the plate, where the camera records it and imports it into the closed-loop system. (B) illustrates the system with more details. A frame is shown in the figure showing the artificial waves are traveling from the left to the right side.

2.12. Microscopy and the optics:

The set-up utilized Olympus MVX10 and Nikon Eclipse Ti-U (Burton et al., 2015) demonstrated in figure 21. This figure shows how the different parts of the system are physically connected.

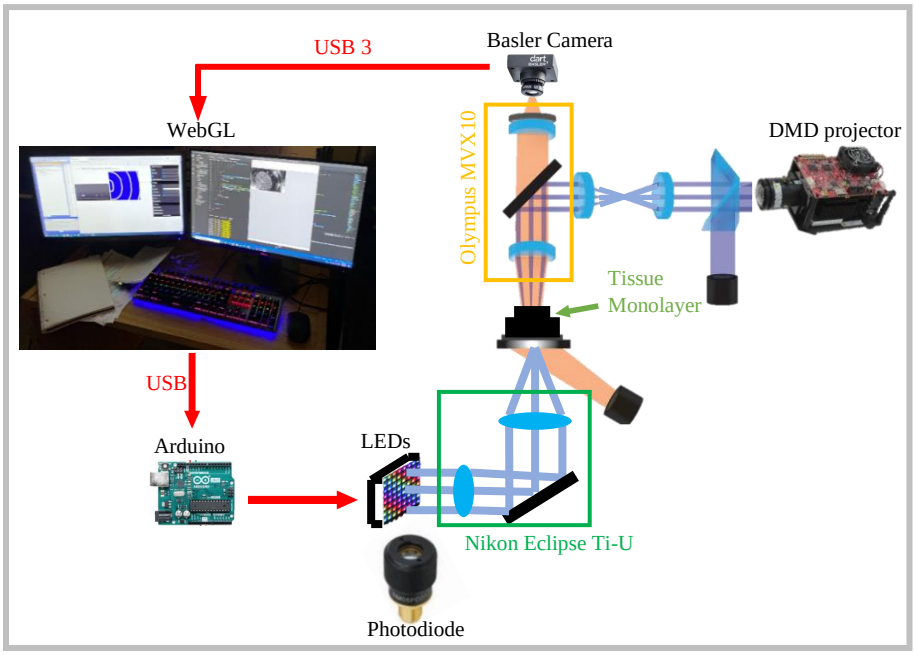


Figure 21. Microscopy set-up. The system is made up of Olympus MVX10 and Nikon Eclipse Ti-U. The LED light comes from the bottom and excites the tissue. The camera record from the top. The projector creates artificial waves and applies these pattern on the stage for accuracy tests. The closed feedback loop (red connection) allows us to interfere with the tissue.

2.13. Data analysis and statistics:

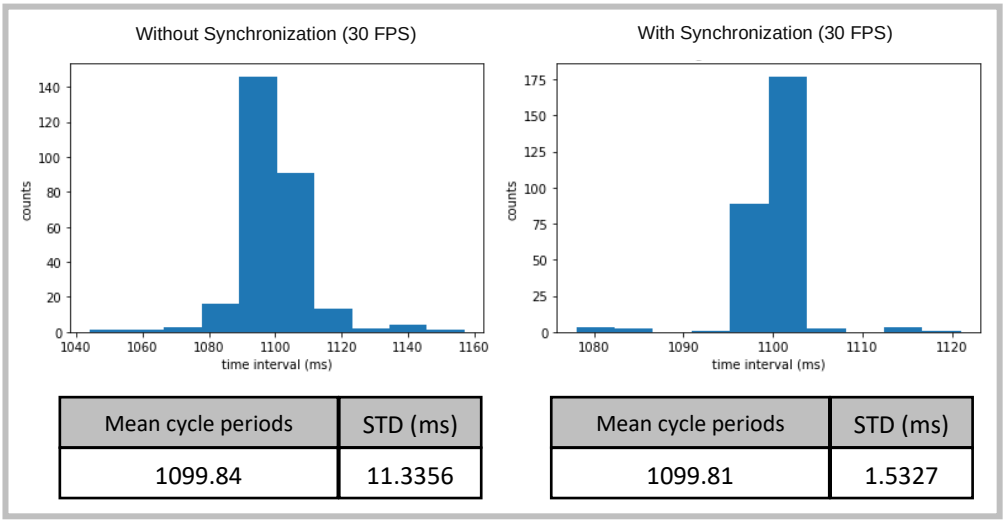
For data analysis, GView64 software developed by Dr. Gil Bub was used for analyzing motion detection in the cardiac monolayer (Burton et al., 2015).

Chapter 3: Results

Here, we show for the first time a closed-feedback loop system that connects the cardiac monolayer with 2D simulations, which replaces the use of a fixed delay. We illustrate some of the potential applications of this hybrid cardiac model in a fully optical system that can control cardiac monolayers.

3.1. Accuracy test (The impact of synchronization):

Millisecond accuracy is crucial for developing a real-time hybrid system. Cardiac waves travel with an approximate velocity of 10 micrometers per milliseconds in a monolayer (Dou et al., 2020). A few milliseconds shift in the timing of the tissue's stimulation can drastically change the tissue's dynamics (explained in section 1.8.). Therefore, millisecond accuracy is necessary to avoid artifacts in our hybrid model. Our approach to detect waves at two different locations in the camera's field of view is designed to differentiate the location of excitation waves within milliseconds. The result of the first accuracy test (Figure 18) showed that the error in the socket loops is more than a millisecond. Figure 22 demonstrates the result of the accuracy test for both the non-synchronized and synchronized systems. Both histograms show the range of system's cycle periods generated by the projector (with periods of 1100 ms). Synchronization reduced the standard deviation from 11 down to less than two milliseconds. The result suggests that the synchronization approach was successful, and the hybrid system reached the expected accuracy for the experiment with cardiac monolayers.



Mean cycle periods	STD (ms)
1099.84	11.3356

Mean cycle periods	STD (ms)
1099.81	1.5327

Figure 22. The effect of synchronization on the hybrid system's accuracy. By adding the connection between the camera and the microcontroller, the hybrid system bypasses the delays in the socket loops and changes data's standard deviation from 11 ms down to 1.5 ms. This result suggests that synchronization was a successful approach for increasing the system's accuracy.

3.2. Source of error: Signal noise

There are two potential sources of error for the temporal accuracy: resolution error and signal noise. The hybrid system used the camera's resolution to get temporal accuracy. The optics for the camera's imaging system is $X1$, and we know that every pixel occupies 5.86 microns for the camera used in these experiments. The error associated with detecting wave location ($\delta y_1, \delta y_2$) is

therefore 5.86 microns. Using the rules of the propagation of uncertainty (Rouaud, 2013) and the formulas indicated in section 2.6., the error associated with the time sent through the system is calculated. This error (0.57 ms) stays within the accuracy bounds determined for this experiment. The assumptions and calculations are as follows:

$y_1 \sim 350\ pixels \times 5.86 = 2051\ microns,$

$y_2 \sim 700\ pixels \times 5.86 = 4102\ microns \ => \ \Delta y \sim 2051\ microns, \qquad \Delta y' \sim 2930\ microns$

$\Delta f \sim 6\ frames$

$$The\ error\ associates\ with\ \Delta y\text{:}\ \ \delta(\Delta y) = \sqrt{{\delta y_1}^2 + {\delta y_2}^2} = \sqrt{5.86^2 + 5.86^2} = 8.28\ microns$$

$$Conduction\ velocity\ error\text{:}\ CV = \frac{\Delta y}{\Delta f} \times \delta CV \Rightarrow |CV| \times \sqrt{\left({\delta(\Delta y)}/{|\Delta y|}\right)^2} = 1.38\ \frac{microns}{frames}$$

$$\Delta t's\ error\text{:}\ \Delta t = \frac{\Delta y'}{CV} \times 33.33 \ => \ \delta(\Delta t) = |\Delta t| \times \sqrt{\left(\frac{\delta CV}{|CV|}\right)^2 + \left(\frac{\delta(\Delta y')}{|\Delta y'|}\right)^2} = 0.57\ ms$$

Although the hybrid system performs with high accuracy, a few exceptions show that it could be inaccurate. We hypothesized that the source of the error is the signal noise. As explained in section 2.6., to detect wave location, row averages were calculated. Figure 23 demonstrates three consecutive arrays of row averages with different colors. Every wave has two peaks because there are two types of motion in the tissue: the cell contraction following by cell relaxation. Placing the threshold of peak detection (Fig 23: dotted line) at different values can result in different distance measurements. Noise in the detected signal (Fig 23: red circle) can lead to large differences in the measured wave location. Since the difference between the two signal peaks is supposed to be 33.33 ms, the little protuberance in the signal, shown in figure 23, can create a significant difference in detecting the wave velocity and ultimately the time value sent to the microcontroller (it can be several milliseconds). We suggest multiplying the row averages with a Gaussian filter in order to avoid this error in future experimentations.

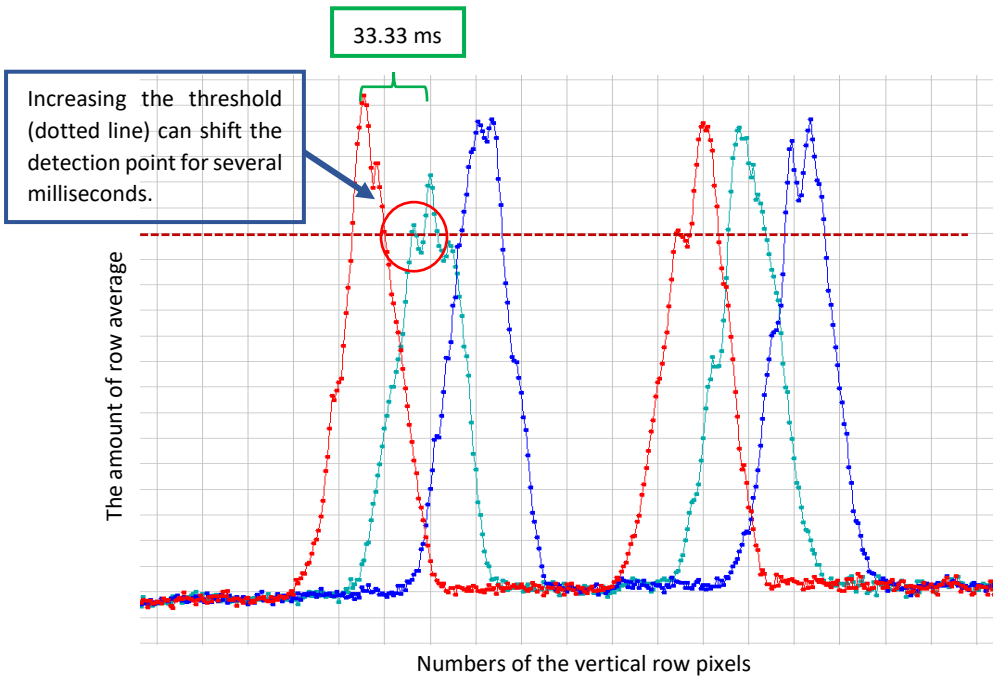


Figure 23. The source of inaccuracy: signal noise. The graph demonstrates three consecutive frames with different colors. The wave (including two peaks) is traveling in the vertical rows. Since the wave takes 33.33 ms to travel from one peak location to the next, the protuberance shown in the red circle can create significant error (several milliseconds) in detecting the wave.

3.3. Closed-loop experiment with 2D cellular automata simulation:

Closed-loop systems mostly use fixed delay (Scardigli et al., 2018) or simple mathematical models (Iravanian & Christini, 2007). We have developed a closed-feedback loop system using a 2D simulation for the first time. We have connected a tissue monolayer with a cellular automata simulation, and we have shown that we are able to modulate the tissue's dynamics in response to the activity of the model. Figure 24 shows the result of two experiments. Comparing the control experiment (figure 24-A) with closed feedback loop (figure 24-B) demonstrates that the hybrid system is affecting tissue's dynamics. The figure (Fig 24-C) shows the evolution of cycle length over the experiment. The result suggests cycle periods are alternating between two different values in response to the model's activity. Figure 24 D-F represents the same experiment with a fixed delay protocol: the fixed delay protocol results in a larger variation in wave dynamics. This proof-of-concept experiment demonstrates the capability of the hybrid system and confirms that it can be used for studying cardiac dynamics both with fixed delay protocol and 2D simulations.

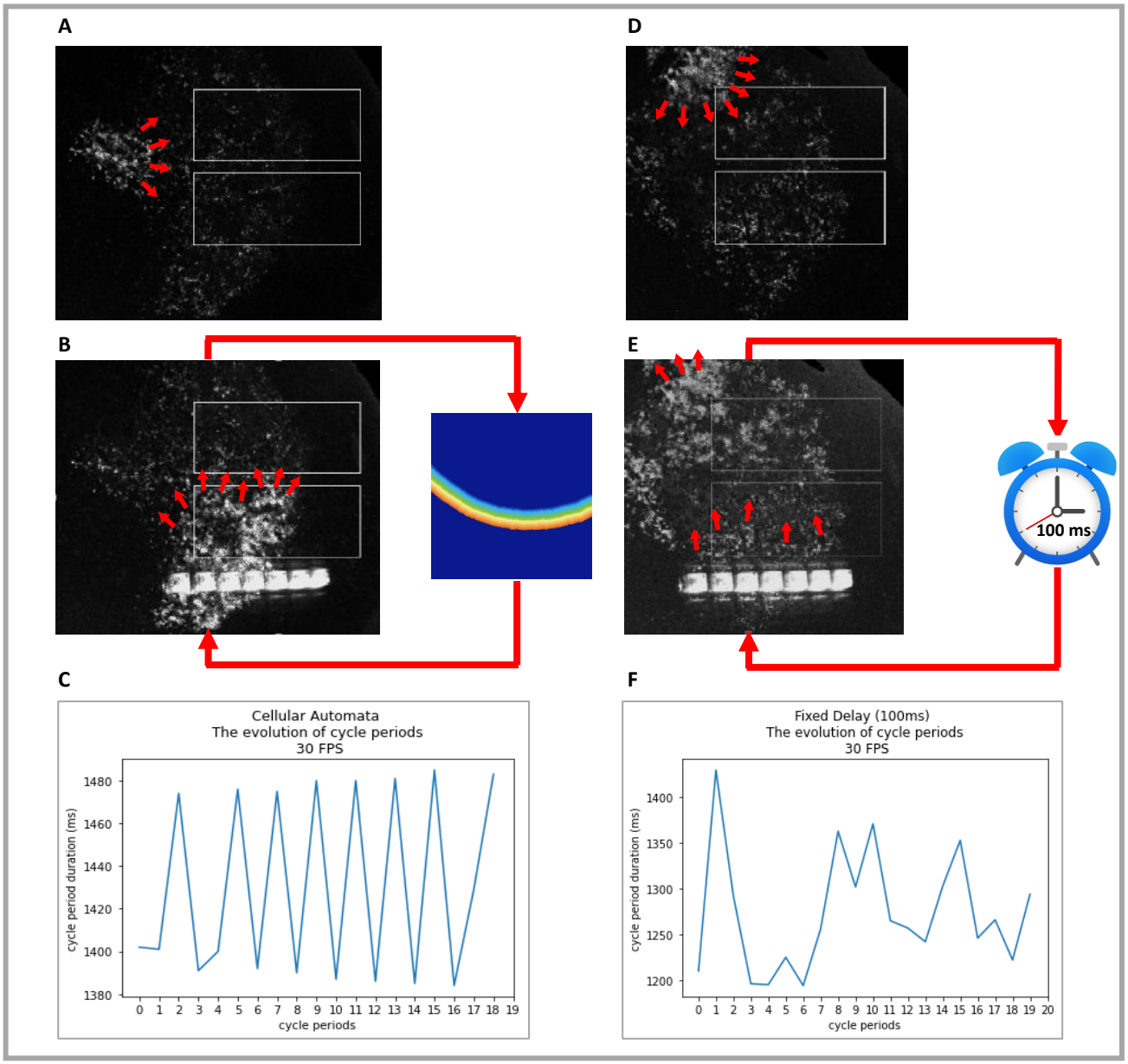


Figure 24. The hybrid system with cellular automata model and fixed delay. (A) and (D) were the control samples (the system is off). The pacemaker is represented with the red arrows. (B) shows that the hybrid system communicates with the cellular automata model and interferes with the tissue. The LED lights can be seen in the image. The row averages were calculated from the white rectangles. (C) illustrates the evolution of cycle length of the re-entrant loop. It suggests the existence of alternans for cycle lengths. (E) presents the hybrid system with a fixed delay model (100 ms delay). The cycle periods associated with this experiment are shown in panel (F).

3.4. Closed-loop experiment with 2D Fenton-Karma simulation:

The Fenton-Karma model (a differential equation-based model) was replaced by cellular automata to demonstrate the capability of the hybrid system with various models. Due to the slow speed of the FK model, the hybrid system could not respond to every wave generated by the tissue because the natural pacemaker in the tissue had fired at a high frequency (calculating wave propagation in the model took approximately 3000 milliseconds, whereas the pacemaker fired about every 1400

milliseconds). However, at some point in the experiment, the hybrid model reset the pacemaker's phase (figure 25-B). In addition, the tissue's activity modulated the model's behavior. High-frequency activity from the pacemaker resulted in wave-breaks, generating a spiral wave in the simulation in the middle of the experiment. This behavior was not seen in any experiment before. Experimental systems that use a fixed delay or a simple model without a spatial component could not generate 2D dynamics such as spiral waves.

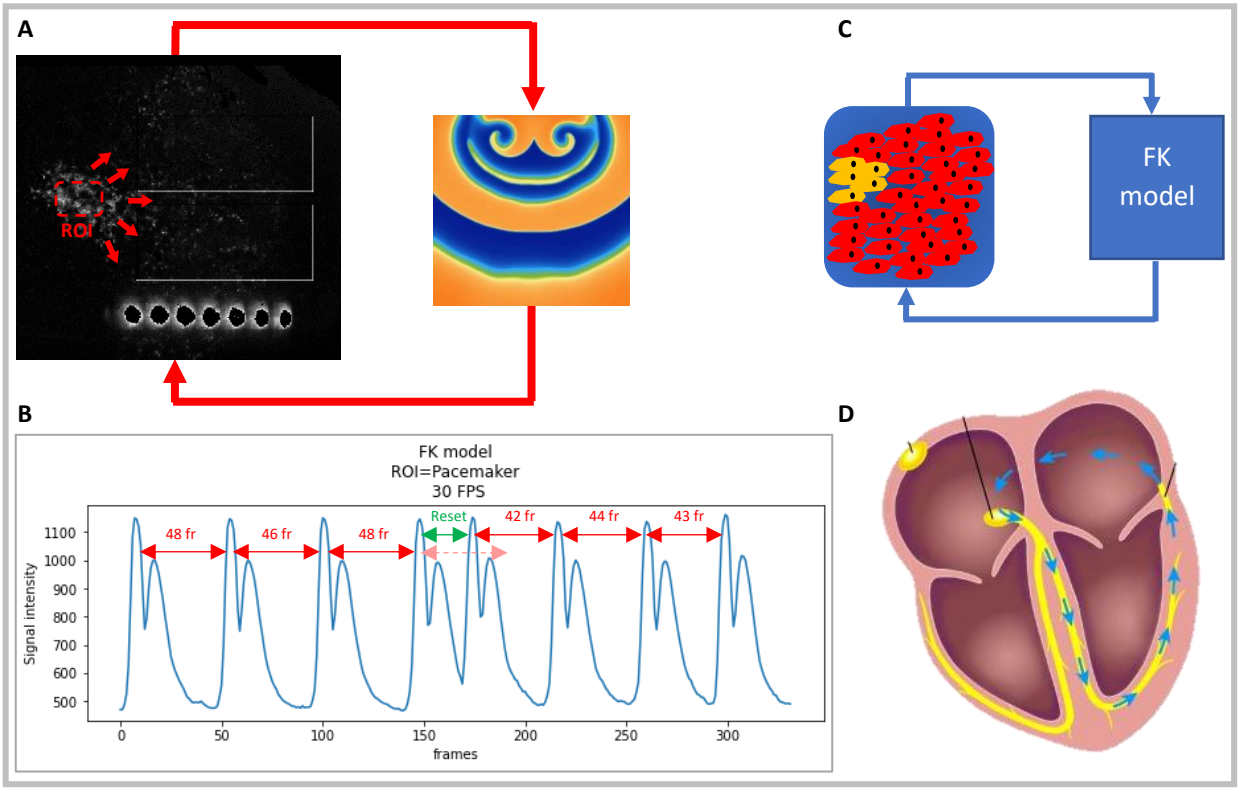


Figure 25. The hybrid system with the Fenton-Karma model. (A) shows the hybrid system implementing the FK model. Due to the slow computation speed of the FK model, LEDs could not fully overdrive the pacemaker, but they could occasionally reset its phase. (B) illustrates the number of frame differences between the peaks in signal intensity of the region of interest. (C) illustrates that this model includes a pacemaker with a long re-entrant loop. (D) shows a cartoon of WPW syndrome and suggests that our system can be an experimental model for WPW syndrome.

While the slow speed of the FK model prevents us from simulating a re-entrant circuit with a short path length, there are re-entrant arrhythmias that have longer conduction paths. For example, Wolf Parkinson's White (WPW) syndrome is caused by a conductive bridge between the ventricles and the atria. In WPW, the excitation wave in the ventricle circles back to the atria and interfere with the sinus pacemaker (figure 25-D). WPW syndrome, therefore, can be represented in a simplified system consisting of a large closed-loop circuit and a pacemaker, which is similar to the condition we have generated in our experiment in Figure 25. This experiment suggests that the hybrid system potentially can be used to investigate cardiac disease conditions as well.

Chapter 4: Discussion

We have developed a hybrid cardiac system that can connect cardiac monolayers with 2D simulations. We suggested that fixed delay can be replaced with 2D simulations for developing a closer model to a real re-entrant loop. Using *Abubu.js* (Kaboudian et al., 2019b), we implemented 2D models capable of running faster than in real-time: the model could predict the location of waves in tissue before the waves propagated to those locations. We have shown that our hybrid system can modulate the activity of a cardiac monolayer and is capable of re-generating re-entrant loops. The proof of principle experiment also suggests that the hybrid system can be used to model cardiac disease conditions such as Wolf Parkinson's White syndrome. One of the key advantages of our hybrid cardiac system is to reach a millisecond accuracy with inexpensive components, which contrasts with other systems that use specialized high-speed cameras and programmable digital acquisition boards (Scardigli et al., 2018). Also, our system is the first one that uses spatially extended models in place of a fixed delay (Biasci et al., 2020; Scardigli et al., 2018) or simple difference equation (Christini & Collins, 1996; Iravanian & Christini, 2007). Future directions might implement 3D models, developed using *Abubu.js* (Kaboudian et al., 2019b), to interact with real hearts.

4.1. Summary of hybrid cardiac model's limitations:

While our system introduces several advances to the field of real-time control of cardiac tissue, there are some important limitations. The system relies on motion detection to measure wave location. Although the motion detection method provides some advantages, such as running the experiment for a longer duration, our device fails to implement other recording methods such as voltage detection, which would result in higher accuracy for measuring wave location (Christoph et al., 2018). For example, while the cardiac contraction is driven by voltage changes, contraction in one location can cause motion in connected tissue that isn't excited, resulting in error. In addition, the dye-free system preferentially captures data from high contrast regions, which may not be representative of the activity in the monolayer.

A low camera's frame rate does not enable the system to precisely detect events between two cameras' shuts, limiting the hybrid model's capability to locate concordant waves. The device assumes all the waves are concordant. Concordant waves travel with a constant velocity throughout the tissue (Anderson et al., 1974). However, real tissue can generate discordant waves: waves that travel with variable velocity in different monolayer regions (Anderson et al., 1974). Since our system assumes that waves propagate at constant velocity when calculating the time to trigger the LEDs, our system will fail in experiments where discordant alternans are present.

Noise can highly affect the accuracy of the camera's detection system. Tissue samples generating a large amount of noise do not allow the motion detection algorithm to accurately detect waves; hence, reducing the accuracy of the hybrid model. Convoluting a gaussian filter with the camera's image data might solve this issue in future experimentations.

Finally, while we believe that replacing fixed delays with simulations will ultimately result in a better representation of a re-entrant circuit, we have not yet shown the 2D simulations necessary are a better model for replacing the fixed delay protocol.

4.2. Evaluating the hybrid system's performance against the previous models:

Compared to the previous systems, our approach has some significant advantages. Other teams developed hybrid systems using expensive high-speed cameras (Biasci et al., 2020; Iravanian & Christini, 2007; Scardigli et al., 2018), which limits its accessibility to labs with specialized equipment. We developed an algorithm (section 2.6.) that can provide highly accurate results despite low frame rate cameras, which greatly reduces the experiment's cost. Our hybrid model uses inexpensive machine vision cameras that allow labs with modest funding to replicate our system and tailor it for their research. Moreover, the fully optical setup enables our system to respond to complex and spatially variable 2D dynamics that characterize some arrhythmic properties as opposed to previous approaches (Christini & Collins, 1996; Frame & Simson, 1988; Patel et al., 2017) who uses electrode stimulation techniques. Although our hybrid model can be tailored for open-loop experiments, its main benefit is as an automated closed-loop system that allows bidirectional communication with the tissue and a simulation. This is in contrast to most cardiac optogenetic studies, which typically apply stimuli that do not vary in response to tissue dynamics (e.g., Bub & Burton, 2015; Crocini et al., 2016). While other research groups developed synchronized 2D GPU-based simulations with microcontrollers (Welsh et al., 2019), we are the first group to evaluate the model by interfacing with living cardiac tissue. Our hybrid model can perform with both fixed-delay protocol and 2D simulations. Compared to the previous approaches, which only used fixed-delay systems (Biasci et al., 2020; Scardigli et al., 2018), our model solves all the limitations of the fixed-delay system explained in section 1.6. Moreover, the use of motion detection technique can extend the life expectancy of the tissue compared to voltage detection systems (Christoph et al., 2018), hence opening the possibility of designing experiments that requires a longer duration (up to days).

Despite the potentials mentioned above, our system has some disadvantages compared to other experimental systems which use specialized high-speed cameras (e.g., Iravanian & Christini, 2007; Scardigli et al. 2018). These high-speed systems enable the detection of cardiac waves with millisecond accuracy in real-time in any part of the tissue. In contrast, our system relies on an algorithm that detects the waves at two points to calculate their speed and predict their location on the tissue. Although our system can predict wave location for a unidirectional smoothly propagating wave with milliseconds accuracy, it fails to predict wave location in cases where waves have variable velocity (e.g., discordant alternans) and direction across the tissue. This limitation reduces the possible experiments that our system can perform. Finally, the camera algorithm is only evaluated using cardiac monolayers, while previous investigations (Iravanian & Christini, 2007; Scardigli et al., 2018) have been validated in ex-vivo intact hearts.

It should also be noted that other research groups have demonstrated systems that have some of the advantages of the system described in this thesis. Both our hybrid model system and RTXI

systems (Patel et al., 2017) use easily sourced and inexpensive components, which have benefits for accessibility and dissemination. The fixed delay protocol performs similarly in our system and systems developed by some other research teams (Biasci et al., 2020; Scardigli et al., 2018). And, similar to some previous approaches (Burton et al., 2015; Entcheva & Bub, 2016), our system is fully optical.

4.3. Future direction:

Our real-time control system enables new experiments that can give insights into the dynamics of re-entry. First, comparing the result of the fixed delay model with 2D simulations can give more insight into which approach is more realistic for experiments that investigate the stability of re-entrant loops. Second, implementing programs that can activate the tissue with similar wave shapes to those generated in the simulation may increase the stability of re-entry and ultimately be more representative of what is occurring in intact tissue (figure 26-A). Finally, the system can be extended by developing computer vision software that implements AI models to detect re-entrant waves in the tissue and respond to them in real-time, which can have potential therapeutic applications in the near future (figure 26-B). Implementing these experiments will further demonstrate our hybrid system's versatility.

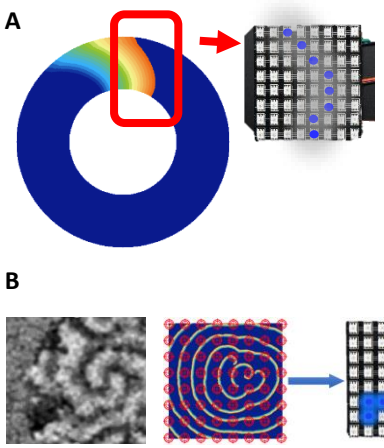


Figure 26. Future developments of the hybrid cardiac model. Cardiac waves have various shapes. Fixed delay assumes they have uniform shapes. (A) shows that the hybrid system is capable of transferring simulations' shape to the monolayer using the matrix LEDs. Developing software AI using computer vision can allow the hybrid system to detect re-entry and suppress it with specific patterns of light (B).

Appendix A. Developing the NodeJS server

A.1. Initiating the NodeJS server

NodeJS (we used v12.18.3) and npm package manager (we used 6.14.6) must be installed prior to developing a NodeJS project. Open the project folder and make a file called *server.js*. Open a terminal (Linux/Mac) or command prompt (Windows) and go to the project folder's location. The following commands will initiate the NodeJs server.

```
npm init
```

Executing this command asks you a series of questions about package name, version, author, etc. Choose the entry point as *server.js* and type *yes* at the end. The code initiates the server. The following commands install the necessary packages for developing communication systems.

```
npm install express

npm install socket.io

npm install serialport
```

The *express* package allows building the client. The *socket.io* and *serialport* package provides the tool for server-client and server-microcontroller communication systems consecutively.

A.2. Developing the *socket.io* communication system

The following code in *server.js* imports the necessary libraries to build the server-client communication system:

```
const express = require('express');
const app = express();
const server = require('http').createServer(app);
const io = require("socket.io")(server);
const HOST = 'localhost';
const PORTIO = 8081;
```

The last two lines indicate that in order to access the server, *http://localhost:8081* should be searched in the web browser. The *socket.io* communication system is developed as follows:

```
// Server is listening to localhost:8081
server.listen(PORTIO, function(){});

/ Initiate Public folder
app.use(express.static('public'));
```

The below code allows for receiving data from the client. The channel is named *led*.

```
io.on("connection", function(sockIO){
    console.log('Client is disconnected')
    //Receive data from public
    sockIO.on('led', function(data) {
        // the data can be accesed by the following code
        var received_data = data.value;
        /** The code for sending received_data to the
         * Arduino should be placed in here*/
    });
    sockIO.on('disconnect', (reason) => {
        console.log('Client is disconnected')
    });
});
```

The above code receives the data from the client. The below code sends data to the client. It should be placed in the TCP socket section (explained later) after receiving a signal from the camera.

```
io.emit('led', {value: camera_signal});
```

A.3. Developing *serialport* for NodeJS-Arduino communication system:

To build the server-microcontroller communication channel, a *port* and the *baudrate* should be defined in *server.js* as follows:

```
const SerialPort = require("serialport");
const Readline = require('@serialport/parser-readline');
const serialPort = new SerialPort("COM5", { baudRate: 9600 });
const parser = serialPort.pipe(new Readline({ delimiter: '\n' }));
// Windows: COM1, COM2, COM3, COM4 or COM5
// Mac: /dev/cu.usbmodem14101
// Ubuntu: /dev/ttyACM0
```

The *port* should be chosen according to the operating system in use (Windows, Mac, or Linux).

Only specific formats of data can be sent through the *serialport*. Thus, buffers with the appropriate length (4 bytes) should be made. The data is broken into smaller pieces and stored in each buffer's byte. The following code is placed right after the data is received from the HTML client:

```
// Building the NodeJS buffer
let buf = Buffer.allocUnsafe(4);
// Converting the variable into the buffer format
buf.writeInt32LE(received_data);
// Send
serialPort.write(buf);
```

The *serialport* also allows data to be received from the Arduino (it is not necessary to build the hybrid system):

```
serialPort.on("open", () => {
    console.log('Arduino is connected');
  });
  parser.on('data', data =>{
    // data is the byte value received from Arduino
  });
```

A.4. TCP communication between the camera and the NodeJS server:

The Transmission control protocol (TCP) socket allows connecting camera's C^{++} code with the *server.js*. The following code in *server.js* builds the NodeJS end:

```
net.createServer(function(sockNet){
    sockNet.on('data', async function(data) {
        // This works like a loop
        camera_signal = JSON.parse(data);

        /** The code for sending data to the HTML
         * client should be placed here */

        // Node should return a message to C++
        sockNet.write("Return String\n");
    });

    // Close TCP connection
    sockNet.on('close', function(data) {});
}).listen(PORTNET, HOST);

    // Close TCP connection
    sockNet.on('close', function(data) {});
}).listen(PORTNET, HOST);
```

Note that the TCP socket works like a complete closed loop, so for every piece of data received by the server, a piece of data should be sent to the client (camera). The code for the C^{++} end (TCP client: *Grab.cpp*) is as follows:

```
string ipAddress = "127.0.0.1";    // IP Address of the server
int port = 8080;                   // Listening port # on the server

// Initialize WinSock
WSAData data;
WORD ver = MAKEWORD(2, 2);
int wsResult = WSAStartup(ver, &data);
if (wsResult != 0) {
    cerr << "Can't start Winsock, Err #" << wsResult << endl;
    return 0;
}

// Create socket
SOCKET sock = socket(AF_INET, SOCK_STREAM, 0);
if (sock == INVALID_SOCKET) {
    cerr << "Can't create socket, Err #" << WSAGetLastError() << endl;
    WSACleanup();
    return 0;
}

// Fill in a hint structure
sockaddr_in hint;
hint.sin_family = AF_INET;
hint.sin_port = htons(port);
inet_pton(AF_INET, ipAddress.c_str(), &hint.sin_addr);

// Connect to server
int connResult = connect(sock, (sockaddr*)&hint, sizeof(hint));
if (connResult == SOCKET_ERROR) {
    cerr << "Can't connect to server, Err #" << WSAGetLastError() << endl;
    closesocket(sock);
    WSACleanup();
    return 0;
}
```

Note that NodeJS server uses *port 8081* for communicating with the HTML client and *port* 8080 for TCP client. Whenever a *signal* variable is needed to be sent to the server, the following code should be placed in the C^{++} client:

```
char buf[4096];
// The signal should be converted to char format
std::string s = std::to_string(signal);
char const* pchar = s.c_str();

// Send the text
int sendResult = send(sock, pchar, (int)strlen(pchar), 0);

// for every signal sent, one should be received
if (sendResult != SOCKET_ERROR) {
    // Wait for response
    ZeroMemory(buf, 4096);
    int bytesReceived = recv(sock, buf, 4096, 0);
}
```

Appendix B. AbubuJS simulation

The detailed explanation of AbubuJS simulation is explained in (Kaboudian et al., 2019b). The three-variable model designed by Dr. Kaboudian was used for the Fenton-Karma model. The code's structure, explained in section 2.3., can be accessed from the supplementary material of (Kaboudian et al., 2019b). For an AbubuJS project, all the codes and files are the same except the *main.js*, the shaders (*.frag* files), and a few lines in the *index.html* file. Any AbubuJS project from (Kaboudian et al., 2019b) can be used to develop a new project by manipulating the mentioned files.

B.1. The HTML file (*index.html*):

The HTML file should call the necessary libraries:

```
<script src="socket.io/socket.io.js"></script>
<script src='config.js'></script>
<script src='libs/stats.js'></script>
<script data-main="app/main" src="libs/require.js"></script>
```

It also needs to define a canvas for displaying the result of the simulation:

```
<canvas width=512 height=512 id='canvas'></canvas>
```

B.2. Developing *main.js* JavaScript file:

The following format in the *main.js* file allows for the correct implementation of shaders and *Abubu.js* library:

```
define(['jquery',
        'Abubu/Abubu.js',
        'shader!initShader.frag',
        'shader!compShader.frag',
        'shader!clickShader.frag'
        ],
        function($,
                Abubu,
                initShader,
                compShader,
                clickShader,
                ) {
        // The rest of the code is placed here
          // such as building Textures and displaying them
        // The socket.io code should be placed here
});
```

A detailed explanation for building fragment shaders, textures, and displaying them on the canvas can be accessed by (Kaboudian et al., 2019b). Dr. Kaboudian also demonstrated implementing these features in his tutorials (https://github.com/kaboudian/WebGLTutorials).

Note that we used this library to develop the cellular automata (CA) model from the ground up. The Fenton-Karma (FK) model is Dr. Kaboudian's three-variable model. The *socket.io* communications (explained later) only need to be added to the FK model. The detailed explanation of the CA algorithm is as follows:

Assuming a 256×256 2D numerical grid, the following AbubuJS functions defines a canvas and the necessary functions for time-stepping and iterative CA solution.

```
/*-------------------- define Canvas-----------------------*/
env.canvas_1 = document.getElementById('canvas');
env.canvas_1.width = env.width;
env.canvas_1.height= env.height;

/*------------ define computational textures----------------*/
env.txtCA1 = new Abubu.Float32RTexture(256,256) ;
env.txtCA2 = new Abubu.Float32RTexture(256,256) ;
```

Building an initial texture, which holds the cell location and the position of cell blocks (cells that never excite), allows initializing the simulations. An array (table) of $256 \times 256 \times 4$ holds these variables. Every one of 256×256 cells have four channels (red, green, blue, transparency). The red and the green channel holds the x and y perturbations for the CA computational cell's location. This slight perturbation for each cell provides heterogeneity for the tissue and avoids unrealistic wave shapes. *Abubu.random()* function is used to provide this random perturbation. The green channel holds the information for the block locations. It tells the simulation which cells mimic the scar's location.

```
/*------------------- initial texture-----------------------*/
var table = new Float32Array(256*256*4);
var idx = 0 ;
for(var j=0; j<256; j++){            // Along y-axis
    for(var i=0 ; i <256; i++){       // Along x-axis
        table[idx++] = env.psize*(Abubu.random()-0.5); //red Perturbation
        table[idx++] = env.psize*(Abubu.random()-0.5); //green Perturbation
        table[idx++] = 1. ; // blue - block
        table[idx++] = 0. ; // a
    }
}

env.txtInit1 = new Abubu.Float32Texture(env.width,env.height,{data:table}) ;
```

AbubuJS solvers send the textures to the shaders for parallel processing. Shader's codes apply its algorithms to every computational cell, which includes four channels. The initial solver initializes the simulations. Note that all solver has to be rendered with *render()* function. This solver initializes both `env.txtCA1` and `env.txtCA2` textures.

```
env.initSolver = new Abubu.Solver({
    //Choosing the shader
    fragmentShader  :   initShader,
    //Shader's outputs
    renderTargets   :{
        o_col_0 : { location : 0, target : env.txtCA1 } ,
        o_col_1 : { location : 1, target : env.txtCA2 } ,
    }
});
//All solvers must be rendered
env.initSolver.render();
```

The code for the *initShader* (initializer fragment shader) associated with the *initSolver* is provided in the next section (B.3.).

The main solver sends the textures iteratively to the *compShader*. The *compShader* is a fragment shader that implements the CA algorithm explained in section 2.4. Note that the solver has three sections: a. it chooses the shader, b. it imports the inputs (textures / parameters) into the shader, c. receives the outputs (textures) from the chosen shader.

```
env.sovlerCA1 = new Abubu.Solver({
    // a. Choosing the Shader
    fragmentShader  :   compShader,
    // b. Shader inputs
    uniforms        :   {
        input_txt = { type : 's', value : env.txtCA1 } ;
        inital_txt = { type : 's', value : env.txtInit1 } ;
        radius       = { type : 'f', value : env.radius       } ;
        threshold      = { type : 'f', value : env.threshold      } ;
        Lx       = { type : 'f', value : env.Lx       } ;
        // more inputs can be added here
    },
    // c. Shader output
    renderTargets   :   {
        out_txt : {location: 0, target : env.txtCA2},
    }
});
```

```
env.sovlerCA2 = new Abubu.Solver({
    // a. Choosing the Shader
    fragmentShader  :   compShader,
    // b. Shader inputs
    uniforms        :   {
        input_txt = { type : 's', value : env.txtCA2 } ;
        inital_txt = { type : 's', value : env.txtInit1 } ;
        radius       = { type : 'f', value : env.radius       } ;
        threshold      = { type : 'f', value : env.threshold      } ;
        Lx       = { type : 'f', value : env.Lx       } ;
        // more inputs can be added here
    },
    // c. Shader output
    renderTargets   :   {
        out_txt : {location: 0, target : env.txtCA1},
    }
});
```

The above two solvers can be rendered by the *env.march()* function.

```
env.march = function(){
    env.sovlerCA1.render();
    env.sovlerCA2.render();
};
```

Whenever this function is called, `env.sovlerCA1` is called, which sends `env.txtCA1` texture to the *compShader.frag* for parallel processing. It puts the result into `env.txtCA2`. The second solver, `env.sovlerCA1,` performs vice versa: compute with `env.txtCA2` and output the result into `env.txtCA1.` Therefore, by calling *env.march()*, the content of `env.txtCA1` texture is updated by two iterations. The GLSL shader code for the *compShader.frag*, which includes the CA algorithm, is provided in the next section (B.3.).

After every computation, the resulted texture is displayed on the canvas using the below code:

```
env.displayCA = new Abubu.Plot2D({
    target          :   env.txtCA1,
    channel         :   'r',
    minValue        :   0.,
    enableMinColor  : true,
    minColor        : [1,1,1],
    maxValue        : 1.,
    colormap        : env.colormap,
    canvas          : env.canvas_1,
});
//Initializing the display function
env.displayCA.init();
```

The detailed explanation of *`Abubu.Plot2D`*'s parameters can be accessed by (Kaboudian et al., 2019c). For every CA iteration, the two below lines must be called:

```
env.march();
env.displayCA.render();
```

All of this algorithm can be placed in a function and recursively called:

```
function run(){
       env.march();
       env.displayCA.render();
       requestAnimationFrame(run);

}
```

`requestAnimationFrame(run)` allow *run()* function to recursively call itself. Every time *env.march()* is called, the simulation is proceeded by two iteration and displayed by *env.displayCA.render()*. This function, *run()*, can be used to calculate the number of frames it takes for the excitation wave to travel from one side of the canvas to another side (F_{sim}), explained in section 2.10. This value can be multiplied by dt (specified according to the experiment) to calculate Δt_{sim} ($\Delta t_{sim} = F_{sim} \times dt$). Finally, this value is added to the signal coming from the camera ($\Delta t_{total} = t + \Delta t_{sim}$).

Note that the code for exciting the simulation or recording the cell voltage at a specific location is presented later in this section.

To communicate with the server, *main.js* can use the below code to receive data from the server:

```
const socket = io.connect('http://localhost:8081');
socket.on('led', function(data){
    t = data.value;
});
```

The camera signal `t` is added to the simulation time ($\Delta t_{total} = t + \Delta t_{sim}$) and is sent to the NodeJS server. The following line of code sends data to the server:

```
socket.emit("led", {value: delta_t_total});
```

To detect cell voltage at a specific position in the simulation, the below code provides the utility to access textures' values. Note that this code might not be available in the old versions of AbubuJS library.

```
env.txtCA1Reader = new Abubu.TextureReader(env.txtCA1);
env.txtCA1Data = env.txtCA1Reader.read();
```

`env.txtCA1Data` would be a 1D array of $256 \times 256 \times 4$, that includes all texture values and their four channels. Since the voltage is stored in the texture's red channel, every four values in `env.txtCA1Data` refers to the cell voltage.

Note that various functions in the AbubuJS library provide more capabilities for controlling the simulation, such as clicking on the canvas to create excitation waves. For more information, please refer to Dr. Kaboudian's GitHub tutorials: https://github.com/kaboudian/WebGLTutorials.

B.3. The GLSL shader code for the cellular automata algorithm

Note that AbubuJS defines a default Vertex shader. By using the below code in any fragment shader, a 2D vector representing the cell location derived by the default vertex shader can be accessed:

```
in vec2 cc ;
```

The following GLSL code is the fragment shader (*initShader*) associated with the initial solver. It initializes all cell voltages to be zero.

```
// defines the variable's precision
precision highp float ;
precision highp int ;

// the pixel location (imported from vertex shader)
in vec2 cc ;

// Shader outputs
layout (location =0) out float o_col_0 ; // output texture 1
layout (location =1) out float o_col_1 ; // output texture 2

void main(){
    float red = 0. ; // represents cell voltage
    o_col_0 = red ;
    o_col_1 = red ;
    return ;
}
```

Below is the code for the *compShader* that includes the CA algorithm. Note that the below code is associated with one single cell in the CA simulation. The code is broken down into multiple steps for simplicity.

First, the input values are imported from the *main.js*:

```
// Defining the variable precision
precision highp float ;
precision highp int ;

// inputs of the shader
uniform sampler2D   input_txt; // includes cell voltages
uniform sampler2D   inital_txt ; // include cell locations and the blocks
layout (location =0) out float out_txt; // output color of the shader

// Cell locations
in vec2 cc ;
// input constant CA variables
uniform float radius, threshold, Lx;
```

Then, the distance values are initialized for accessing surrounding cells:

```
void main() {
    vec2 size   = vec2(textureSize(input_txt,0)) ; //size of texture
    vec2 ii     = vec2(1.,0.)/size ; // Unit vector in x  //size of pixel
    vec2 jj     = vec2(0.,1.)/size ; // .............. y

    float dx = Lx/size.x ;
    float dy = dx ;

    vec4  C = texture(input_txt, cc) ;
    vec2  pert = texture(inital_txt , cc).xy ;
    float ifBlock = texture(inital_txt , cc).z; //gives me the block condition
    vec2  cellCoord = cc*Lx + pert ;
```

If the current cell is a block, the CA algorithm should not be applied. Otherwise, the cell is a computing cell and is part of the CA algorithm:

```
    if(ifBlock == 0.){   //* If the pixel is part of the block
        C.r = -1.;
    }else{
```

According to section 2.4., if the cell has a voltage lower than 0.05, the cell is in an excitable state and can generate action potential:

```
        // If the cell is not in refractory.
        if(C.r < 0.05){
```

The cell starts to count the number of active (`voltage > 0.7`) and inactive surrounding cells. Note that the red channel (C.r) includes the voltage value.

```
// Initializing the variables for counting alive and inactive cells.
float aliveNeighbors = 0.;
float deadNeighbors = 0.;

int m = int(round(radius/dx)) ;
int n = int(round(radius/dy)) ;
// Circulating around the current cell in the radius of R
// m and n define the distance for CA radius
for (int i=-m; i<(m+1) ;i++){
    for (int j=-n; j<(n+1); j++ ){
        // find the position of each surrounding cell
         vec2 currPos =  cc+float(i)*ii+float(j)*jj ;

        // if the cell is not surrounded by a barrier
         float reachBlock = texture(in_itxt, currPos).z;
             if (reachBlock == 1.){      //* No block
                  // count the number of alive and dead cells
                 float voltage = texture(in_txt,currPos).r ; //texture
                 pert = texture(in_itxt,currPos).xy ;
                 vec2 compCellCoord= currPos*vec2(Lx,Lx) + pert ;

                  // If the surrounding cell is within the radius
                       // If the cell has surpassed voltage, count it as active
                       // If the cell has low voltage, count it as dead cell
                 float distance = length(cellCoord-compCellCoord) ;
                 if(distance<radius){
                     if ( (voltage >0.7) && (voltage<0.99) ){
                         aliveNeighbors+=1.;
                     }else{
                         deadNeighbors+=1.;
                     }
                 }
```

Note that bordering areas (near blocks) have a specific algorithm different from the general CA algorithm. If the current cell is faced with a block, the algorithm looks at the mirroring computational cell (the cells on the other side of the current cell).

```
            // If the surrounding cell is a block, look at the mirroring cell
        } else if (reachBlock == 0.) {
            currPos = cc+float(i)*ii*(-1.)+float(j)*jj*(-1.) ;  //* mirroring
            float voltage = texture(in_txt,currPos).r ; //texture
            pert = texture(in_itxt,currPos).xy ;
            vec2 compCellCoord= currPos*vec2(Lx,Lx) + pert ;//physical
            float distance = length(cellCoord-compCellCoord) ;
            if(distance<radius){
                if ( (voltage >0.7) && (voltage<0.99) ){
                    aliveNeighbors+=1.;
                }else{
                    deadNeighbors+=1.;
                }
// ------------> Closing all the functions and for loops
            }
        }
    }
  }
```

The shader compares the alive/dead ratio to the threshold to decide whether the cell should generate action potential or remain in an excitable state:

```
  //if the alive/dead ratio is greater than threshold raise the voltage
   if ((aliveNeighbors/deadNeighbors)>(threshold)){
       C.r =  0.98;
   }else {
       C.r = 0.;
   }
```

If the cell is in the refractory state (`voltage > 0.05`), the CA algorithm only gradually reduce the voltage:

```
//if the cell is in refractory
}else{
   C.r = C.r - 0.051;
}
```

And finally, the resulted voltage is sent back to the *main.js* file:

```
  }
  out_txt = C.r ;
  return ;
}
```

The above GLSL code computes one iteration of the cellular automata algorithm, and it is called twice every time the *env.march()* function is called.

In order to excite the simulation, a *camera_signal* variable can be imported to the shader besides other simulation variables such as *radius* and *threshold*. This signal can tell the simulation to generate excitation waves at specific locations. The below code generates a wave at the left side of the canvas whenever the shader receives a signal:

```
if (camera_signal_available){
      if(cc.x < 0.01){
            C.r = 0.98;
      }
}
```

This code can be placed in before exporting the cell values to the *main.js* file.

Appendix C. Pylon and the camera C++ code

C.1. Installing SDK package:

A *C*++ program linked to the Pylon software development kit (SDK, version 6.0.1) was written using Microsoft Visual Studio (version *community 2019*). The following path in the SDK kit reaches a *C*++ code that can be manipulated according to the algorithm explained in section 2.6.

```
SDK-kit\Development\Samples\C++\Grab\Grab.cpp
```

Note that in order to access the *Development* folder, the development version of SDK must be installed.

C.2. Calculating filtered images:

The algorithm is explained in section 2.6. should be placed after the program successfully grabbed a frame. Please look for this line in the original *Grab.cpp* file:

```
// Image grabbed successfully?
if (ptrGrabResult->GrabSucceeded())
{
	const uint8_t* pImageBuffer = (uint8_t*)ptrGrabResult->GetBuffer();
	// The algorithms explained in section 2.6. <----------------
}
```

The following **pseudocode** explains the algorithm for creating the filtered images:

abs(current frame – previous sixth frame)

```
xdim = 1920 // Frame's weight
ydim = 1200 // Frame's height
current_frame[xdim * ydim]
six_previous_frame[xdim * ydim * 6]
z = 0 // Frame number
// Iterate throughout the frame's pixels
for (Iterate among the pixel values of each frame) {
	i = current pixel location
	current_frame[i] = pixel_value
	// Store the current pixel in an array that keeps the value for every six frames
	index = the pixel index in the current frame
	six_previous_frames[index] = pixel_value

	// After the sixth frame, start to calculate the filtered image
	if (z >= 6) {
	    // Access sixth previous frame for six_previous_frames
	    // Because all the frames
	    filtered_image[i] = abs(current_frame[i]- six_previous_frames[index-6*xdim*ydim])

	 }
}
```

The `filtered_image` is an array that represents motion in the tissue, as explained in section 2.6.

C.3. Calculating conduction velocity and Δt_{cam}:

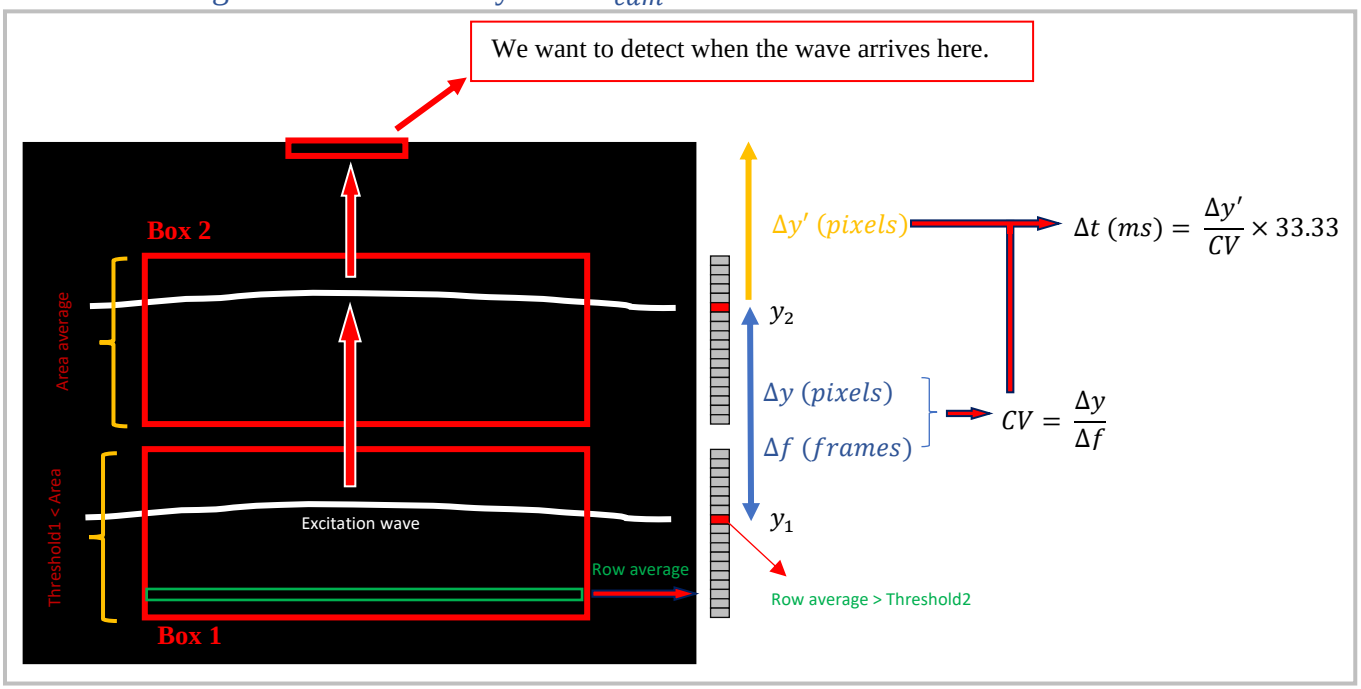


Figure 27. The algorithm for detecting the excitation waves from the filtered images.

We need to look at two boxes in the filtered image explained in figure 15 and figure 27.

The below pseudocode expresses this algorithm:

```
for (iterate among the pixels in the first box) {
       row_averages_box1 = calculate the average of each row pixels
       area_average_box1 = calculate the average of all pixels in the box
}

for (iterate among the pixels in the second box) {
       row_averages_box2 = calculate the average of each row pixels
       area_average_box2 = calculate the average of all pixels in the box
}
```

Afterward, we define the threshold for the whole box's area and the threshold for the row averages.

```
float area_threshold =  10.6;
float row_threshold = 15.0;
```

The microscopy lighting and tissue's transparency are different among each cell culture; thus, these threshold values are specified before each experiment. We can manually determine these thresholds by printing area averages and row averages using (*cout << endl)* function. If the system

cannot detect the waves, the threshold values should be lowered. A periodic raise should be seen in the printed values of the row averages or area averages. The thresholds are supposed to be determined according to those peaks.

The following **pseudocode** allows calculating the wave's conduction velocity. First, we look at the first box:

```
Refractory_value = 0
if (averages_box1 > area_threshold  and refractory_value = 0) {
       z1 = z;
       refractory_value = max_refractory
       for (iterate through row_averages_box1) {
              if row_averages_box1[i] > row_threshold
                     y1 = current vertical location of the wave
                     break the loop
       }
} refractoryAmount
```

Only after `z1` and `y1` are determined (the threshold condition is satisfied), we look at the second box:

```
if (averages_box2 > area_threshold  and z1/y1 determined) {
       z2 = z;
       for (iterate through row_averages_box2) {
              if row_averages_box2[i] > row_threshold
                     y2 = current vertical location of the wave
                     break the loop
       }

       conduction_velocity = (y2-y1)/(z2-z1)
       detla_y = distance between y2 and the point where the wave arrives
       delta_t_camera = (delta_y)/conduction_velocity * (30 FPS/1000 ms)
       delta_t_camera is sent to the NodeJS
       refresh y1, y2, z1, z2
}
```

Every time a wave is in a box, we should ignore the excitation wave in that box for aspecific number of frames afterward. This value, which is called `refractoryAmount`, prevents detecting a wave twice or detecting the relaxation signal, explained in section 3.2. The `max_refractory` should be determined according to the wave speed and the frame rate. Experimentally, `max_refractory = 20` worked the best for 30 FPS.

At the end of the experiment, we store the last 500 frames of the filtered images in a large array of $500 \times xdim \times ydim$. This large array is stored as a binary file and later is analyzed by the GView application. Since developing this code was complicated and the GView application only accepts a specific format, we provide the exact C^{++} code below:

```
int numFrames = 500;
int storeFrameSize = 500*xdim*ydim;

uint8_t* unsigShiftedFrames = new uint8_t[storeFrameSize];
	for (int i = 0; i < storeFrameSize; i++) {
		unsigShiftedFrames[i] = (unsigned short int)stored500Frames[i];
	}

uint32_t bubmode = _byteswap_ulong(2);
uint32_t bubzdim = _byteswap_ulong(numFrames);
uint32_t bubxdim = _byteswap_ulong(xdim);
uint32_t bubydim = _byteswap_ulong(ydim);

fstream storeFile;
storeFile = fstream("C:\\file location\\binary_file", ios::out | ios::binary);
storeFile.write(reinterpret_cast<char*>(&bubmode), sizeof(uint32_t));
storeFile.write(reinterpret_cast<char*>(&bubzdim), sizeof(uint32_t));
storeFile.write(reinterpret_cast<char*>(&bubydim), sizeof(uint32_t));
storeFile.write(reinterpret_cast<char*>(&bubxdim), sizeof(uint32_t));
storeFile.write((char*)unsigShiftedFrames, storeFrameSize * sizeof(uint8_t));
```

Appendix D. The Arduino

D.1. Arduino setting:

In Arduino IDE (we used version 1.8.49), in the tools section, the *Board* value is supposed to be "Arduino Uno", and the *port* should be chosen according to your device (explained in section A.3., for our device, it is COM5 since our operative system is windows 10). *Adafuit DotStarMatrix* (we used version 1.0.5) should be installed using (Tools→Manage libraries). Arduino IDE uses C syntax.

D.2. Arduino Code:

The following libraries and definitions should be set in the beginning:

```
#include <Adafruit_DotStar.h>
#include <SPI.h>
#define NUMPIXELS 64
Adafruit_DotStar matrix(NUMPIXELS, DOTSTAR_BRG);
// set a blue color for LEDs
uint32_t color = 0x0000FF;
```

We used analog pin 3 to connect the Arduino with the camera:

```
int analogPin = A3;
```

The following settings are used in the *setup()* function:

```
void setup() {
  Serial.begin(9600);
  matrix.begin(); // Initialize pins for output
  matrix.setBrightness(255);
  matrix.show();  // Turn all LEDs off ASAP
}
```

The *loop()* function runs continuously. The algorithm for receiving data from the NodeJS server as well as the camera is placed here. The following **pseudocode** demonstrates an overview:

```
void loop() {
       //a. Receive signal from the camera (F_camera ++)
       //b. If F_camera = F_Arduino → wait(remainder) → activate LEDs
       //c. Receive t from the NodeJS
       //d. Breakes t into F_Arduino and remainder
}
```

a. This code updates the Arduino about the camera's frame number:

```
val = analogRead(analogPin);
if (val > 600 && prevVal<=600){
    f_camera++;
}
```

b. If the right frame is reached, we activate the LEDs:

```
if (f_camera == f_arduino){
    //wait the remainder
    delayMicroseconds(remainder*1000);
    //activate corresponding LED
    matrix.setPixelColor(LED_number, color);
    matrix.show();
    // The above code allows activating any pattern of LEDs

    // The LEDs should stay on for about 100 ms to activate the tissue
}
```

Note that LEDs should stay on for at about 100 ms to be able to activate the tissue. They can be left on for around 3 frames (for example, receiving 3 signals from the camera: $3 \times 33.33\ ms$).

c. Receive data from the server

```
if (Serial.available()){
    byte input;
    input = Serial.read();
```

Every signal received is a byte from t. Four received signals (bytes) should be combined to produce t. The below code allows this combination and stores it in *inputInt*:

```
if (fist_signal){inputInt = inputInt + input*1;}
if (second_signal){inputInt = inputInt + (long)input*256;}
if (third_signal){inputInt = inputInt + (long)input*65536;}
if (fourth_signal){inputInt = inputInt + (long)input*16777216;}
```

d. The below code breaks down *inputInt* to the number of frames and the remainder:

```
float frP = 1000.0/30.0 // camera's frame period
int f_arduino = abs((long)((float)(inputInt)/frP));
int temp = (int)(inputInt%100);
int remainder = abs((int)round(temp-((int)((float)temp/frP))*frP));
```

Creating a middle variable (*temp*) is a technique to avoid the inaccuracy of the float variables.

Appendix E. analyzing photodiode data and the region of interest

E.1. Picolog application creates CSV files:

The PicoLog 6 creates CSV files. Every line in the file presents LEDs' signal intensity in a millisecond.

E.2. Python code for analyzing photodiode's CSV files:

We developed a python code that reads the CSV file and calculates cycle periods. The below code reads the file and stores all data in an array:

```
import csv
import numpy as np
import matplotlib.pyplot as plt

fullData = [];
with open('file_name.csv', 'r') as file:
    reader = csv.reader(file)
    for row in reader:
        fullData.append(row)
```

Then the signal intensities are stored in a new array:

```
nums = []
for row in range(1,len(fullData[:])-10):
    nums.append(float(fullData[row][1]))
```

And the timing for the signal peaks is calculated:

```
c = 0
peaks = []
while(c<len(nums)):
    if(nums[c]<-2000):
        peaks.append(c)
        c+=50
    c+=1
```

The cycle periods are calculated by subtracting the timing of the peak values:

```
cycle_periods = []
for i in range(1, len(peaks)):
    cycle_periods.append(peaks[i]-peaks[i-1])
```

Finally, the histogram, mean, and standard deviation of the cycle periods can be determined.

```
plt.hist(cycle_periods, bins=1)
plt.xlabel("time interval (ms)")
plt.ylabel("counts")
plt.title("Cycle Periods")
plt.show()

mean = np.mean(cycle_periods)
std = np.std(cycle_periods)
```

E.3. Python code for analyzing GView's *roi* files:

The GView application, developed by Dr. Gil Bub (Burton et al., 2015), can record the signal intensity of a region of interest (roi) from the camera's data (stored 500 frames) and save it as a *roi* file. The below python code reads this file and stores its content in a list:

```
roi_data = []
f = open("file_name.roi", "r")
for row in f:
    x1=0
    x2=0
    cc=0
    x1found = 0
    for x in row:
        if (x==' ' and x1found==0):
            x1=cc
            x1found=1
        if(x=='\n'):
            x2=cc
        cc = cc+1
    roi_data.append(row[x1+1:x2-1])

roi_signal_intensity = np.zeros(len(roi_data)-1)
for x in range(len(roi_data)-1):
    roi_signal_intensity[x] = (int(roi_data[x]))
```

The roi_signal_intensity vector can be displayed by a plot as demonstrated in section 3.4.

Appendix F. Monolayer preparation

The following protocol follows a step-by-step procedure from Miltenyi Biotec (Protocol No. 130-098-373, Miltenyi Biotec Inc, California, US), with additional steps from (Ambrosi et al., 2014; Burton et al., 2015). The entire protocol was first described in a submitted MSc thesis from a student in the same laboratory (Sepúlveda, 2020).

Cells were isolated using a Miltenyi gentleMACS Dissociator, which automates the cell dissociation process. Procedures for animal handling were performed in agreement with guidelines of the Canadian Council on Animal Care. Neonates were euthanized by decapitation in agreement with McGill University SOP 301-01 under approved protocol 2018-8044. Sterile techniques are followed during all procedures.

F.1. Tissue culture coating (for 24-well plates) and plating cell densities:

1. 5 µL of fibronectin (VWR, Ontario, CA) is diluted into 500 µL of PBS for a final concentration of 50 µg/µL.
2. 125 µL of thoroughly mixed dilution must be added to each well.
3. The plate should be incubated at 37 ºC and 5% CO2 for at least 1 h before use.

The densities described in the below table were used for neonatal mice cardiomyocytes.

Cell density and seeding volume (Table F1)			
Vessel size	Surface area in cm^2	Seeding volume in ml	Cell number (~156 x 10^3 cells/cm^2)
Glass ring	1	0.232	156 x 10^3
24-well	1.9	0.6	296 x 10^3
12-well	3.8	1.2	593 x 10^3

F.2. Ventricle dissociation, tissue culture, and adenoviral infection of cardiac monolayers:

i. Prepare necessary solutions from stocks Penicillin/Streptomycin Mixture (Pen/Strep, Quality Biological, Cat#: 120-095-721), Dulbecco's Modified Eagle Medium (DMEM, Wisent, REF: 319-062-CL), Phosphate Buffered Saline (PBS, Wisent, REF: 311-010-CL), and Fetal Bovine Serum (FBS, Gibco, Cat#12483020). DMEM (10%) maintenance media is prepared by adding 50 mL of FBS and 5 mL on Pen/Strep to 450 mL of DMEM, and DMEM (2%) is prepared by adding 200 µL of FBS and 100 µL of Pen/Strep to 10mL of DMEM.

ii. Harvest the heart and dissect the ventricles:
    a. Obtain a litter (6 or more animals) of postnatal 0 – 3 (P0 - P3) day old mice pups.

b. Pups are decapitated, and the heart is isolated by first opening the rib cage with sharp scissors and remove the heart with tweezers. The hearts are placed in a shallow plate containing 3 ml of PBS solution on ice.
c. Scissors are used to remove the ventricles (around the lower 70% portion of the heart) and remaining connective tissue.
d. The tissue is cleaned by swirling the plate regularly to remove blood cells from the ventricles.
e. The ventricles are each cut into 4 – 6 small pieces around 1–2 mm^3.

iii. The ventricles are dissociated into single cells using the Neonatal Heart Dissociation Kit and gentleMACS Dissociator, following protocol 130-098-373. The program used on the dissociator was "m_neoheart_01_01". The protocol takes approximately 45 minutes.
a. The resulting suspension is centrifuged at 600xg for 5 mins.
b. The pellet is resuspended in 10 mL of DMEM 10% using gentle manual agitation with a wide-mouthed pipette.

iv. Remove cardiac fibroblasts and count cells:
a. The 10 mL suspension is transferred to a shallow10 cm culture dish and incubated (at 37 degrees, 5% CO2) for 45 minutes. Fibroblasts settle and adhere to the bottom of the dish. The supernatant contains an enriched population of myocytes.
b. The supernatant (10ml) and an additional 5ml PBS for washing the plate are transferred to a 50ml Falcon tube.
c. Centrifuge the 15ml supernatant at 600xg for 5 min and resuspend the pellet in 1 mL of warmed DMEM 10%.
d. Cell concentration is determined using a standard hemocytometer, with Trypan Blue added to count dead cells.

v. Culture cardiomyocyte cells in 24-well plates:
a. Once counted, the desired concentration per well is determined per table F1 above (approximately 300,000 cells per well in a 24 well plate).
b. An additional 1 mL of DMEM 10% per well is added, and the tissue incubated for 24 hours.
c. Provide post-plating maintenance by replacing the solution with 2 mL of DMEM 10% every 24-48 hours.

vi. Infect the ventricular myocytes by Ad-CMV-hChR2(H134R)-eYFP (Ambrosi et al., 2014) 2 to 3 days post plating:
a. Replace DMEM 10% with a small amount of DMEM 2% (250 μL per plate).
b. For our desired multiplicity of infection (MOI) of 100 and a virus titer of 2.2x106, we use 13.6 μL of virus solution per well (assuming 300,000 cells/well).
c. Incubate monolayers for 48 h prior to running experiments

References

Ambrosi, C. M., Klimas, A., Yu, J., & Entcheva, E. (2014). Cardiac applications of optogenetics. *Progress in Biophysics and Molecular Biology*, *115*(2–3), 294–304. https://doi.org/10.1016/j.pbiomolbio.2014.07.001

Anderson, R. H., Shinebourne, E. A., & Gerlis, L. M. (1974). Criss-Cross Atrioventricular Relationships Producing Paradoxical Atrioventricular Concordance or Discordance. *Circulation*, *50*(1), 176–180. https://doi.org/10.1161/01.CIR.50.1.176

Arrenberg, A. B., Stainier, D. Y. R., Baier, H., & Huisken, J. (2010). Optogenetic control of cardiac function. *Science (New York, N.Y.)*, *330*(6006), 971–974. https://doi.org/10.1126/science.1195929

Biasci, V., Sacconi, L., Cytrynbaum, E. N., Pijnappels, D. A., De Coster, T., Shrier, A., Glass, L., & Bub, G. (2020). Universal mechanisms for self-termination of rapid cardiac rhythm. *Chaos: An Interdisciplinary Journal of Nonlinear Science*, *30*(12), 121107. https://doi.org/10.1063/5.0033813

Bruegmann, T., Boyle, P. M., Vogt, C. C., Karathanos, T. V., Arevalo, H. J., Fleischmann, B. K., Trayanova, N. A., & Sasse, P. (2016). Optogenetic defibrillation terminates ventricular arrhythmia in mouse hearts and human simulations. *The Journal of Clinical Investigation*, *126*(10), 3894–3904. https://doi.org/10.1172/JCI88950

Bruegmann, T., Malan, D., Hesse, M., Beiert, T., Fuegemann, C. J., Fleischmann, B. K., & Sasse, P. (2010). Optogenetic control of heart muscle in vitro and in vivo. *Nature Methods*, *7*(11), 897–900. https://doi.org/10.1038/nmeth.1512

Bub, G., & Burton, R.-A. B. (2015). Macro-micro imaging of cardiac–neural circuits in co-cultures from normal and diseased hearts. *The Journal of Physiology*, *593*(14), 3047–3053. https://doi.org/10.1113/jphysiol.2014.285460

Bub, G., Shrier, A., & Glass, L. (2002). Spiral Wave Generation in Heterogeneous Excitable Media. *Physical Review Letters*, *88*(5), 058101. https://doi.org/10.1103/PhysRevLett.88.058101

Burton, R. A. B., Klimas, A., Ambrosi, C. M., Tomek, J., Corbett, A., Entcheva, E., & Bub, G. (2015). Optical control of excitation waves in cardiac tissue. *Nature Photonics*, *9*(12), 813–816. https://doi.org/10.1038/nphoton.2015.196

Christini, D. J., & Collins, J. J. (1996). Using chaos control and tracking to suppress a pathological nonchaotic rhythm in a cardiac model. *Physical Review E*, *53*(1), R49–R52. https://doi.org/10.1103/PhysRevE.53.R49

Christini, D. J., Stein, K. M., Markowitz, S. M., & Lerman, B. B. (1999). Practical real-time computing system for biomedical experiment interface. *Annals of Biomedical Engineering*, *27*(2), 180–186. https://doi.org/10.1114/1.185

Christoph, J., Chebbok, M., Richter, C., Schröder-Schetelig, J., Bittihn, P., Stein, S., Uzelac, I., Fenton, F. H., Hasenfuß, G., Gilmour Jr., R. F., & Luther, S. (2018). Electromechanical vortex filaments during cardiac fibrillation. *Nature*, *555*(7698), 667–672. https://doi.org/10.1038/nature26001

Crocini, C., Ferrantini, C., Coppini, R., Scardigli, M., Yan, P., Loew, L. M., Smith, G., Cerbai, E., Poggesi, C., Pavone, F. S., & Sacconi, L. (2016). Optogenetics design of mechanistically-based stimulation patterns for cardiac defibrillation. *Scientific Reports*, *6*, 35628. https://doi.org/10.1038/srep35628

Dou, W., Zhao, Q., Malhi, M., Liu, X., Zhang, Z., Wang, L., Masse, S., Nanthakumar, K., Hamilton, R., Maynes, J. T., & Sun, Y. (2020). Label-free conduction velocity mapping and gap junction assessment of functional iPSC-Cardiomyocyte monolayers. *Biosensors and Bioelectronics*, *167*, 112468. https://doi.org/10.1016/j.bios.2020.112468

Entcheva, E., & Bub, G. (2016). All-optical control of cardiac excitation: Combined high-resolution optogenetic actuation and optical mapping. *The Journal of Physiology*, *594*(9), 2503–2510. https://doi.org/10.1113/JP271559

Fenton, F. H., Cherry, E. M., & Glass, L. (2008). Cardiac arrhythmia. *Scholarpedia*, *3*(7), 1665. https://doi.org/10.4249/scholarpedia.1665

Fenton, F., & Karma, A. (1998). Vortex dynamics in three-dimensional continuous myocardium with fiber rotation: Filament instability and fibrillation. *Chaos: An Interdisciplinary Journal of Nonlinear Science*, *8*(1), 20–47. https://doi.org/10.1063/1.166311

Frame, L. H., & Simson, M. B. (1988). Oscillations of conduction, action potential duration, and refractoriness. A mechanism for spontaneous termination of reentrant tachycardias. *Circulation*, *78*(5 Pt 1), 1277–1287. https://doi.org/10.1161/01.cir.78.5.1277

Gonzalez Vivo, P., & Lowe, J. (2015). *The Book of Shaders*. The Book of Shaders. https://thebookofshaders.com/

Hall, K., Christini, D. J., Tremblay, M., Collins, J. J., Glass, L., & Billette, J. (1997). Dynamic Control of Cardiac Alternans. *Physical Review Letters*, *78*(23), 4518–4521. https://doi.org/10.1103/PhysRevLett.78.4518

Iravanian, S., & Christini, D. J. (2007). Optical mapping system with real-time control capability. *American Journal of Physiology-Heart and Circulatory Physiology*, *293*(4), H2605–H2611. https://doi.org/10.1152/ajpheart.00588.2007

Kaboudian, A. (2021). *Kaboudian/abubujs* [JavaScript]. https://github.com/kaboudian/abubujs (Original work published 2019)

Kaboudian, A., Cherry, E. M., & Fenton, F. H. (2019a). Real-time interactive simulations of large-scale systems on personal computers and cell phones: Toward patient-specific heart modeling and other applications. *Science Advances*, *5*(3), eaav6019. https://doi.org/10.1126/sciadv.aav6019

Kaboudian, A., Cherry, E. M., & Fenton, F. H. (2019b). Real-time interactive simulations of large-scale systems on personal computers and cell phones: Toward patient-specific heart modeling and other applications. *Science Advances*, *5*(3). https://doi.org/10.1126/sciadv.aav6019

Kaboudian, A., Cherry, E. M., & Fenton, F. H. (2019c). Large-scale interactive numerical experiments of chaos, solitons and fractals in real time via GPU in a web browser. *Chaos, Solitons & Fractals*, *121*, 6–29. https://doi.org/10.1016/j.chaos.2019.01.005

Kispersky, T. J., Economo, M. N., Randeria, P., & White, J. A. (2011). GenNet: A Platform for Hybrid Network Experiments. *Frontiers in Neuroinformatics*, *0*. https://doi.org/10.3389/fninf.2011.00011

Nussinovitch, U., & Gepstein, L. (2015). Optogenetics for in vivo cardiac pacing and resynchronization therapies. *Nature Biotechnology*, *33*(7), 750–754. https://doi.org/10.1038/nbt.3268

Nyns, E. C. A., Kip, A., Bart, C. I., Plomp, J. J., Zeppenfeld, K., Schalij, M. J., de Vries, A. A. F., & Pijnappels, D. A. (2017). Optogenetic termination of ventricular arrhythmias in the whole heart: Towards biological cardiac rhythm management. *European Heart Journal*, *38*(27), 2132–2136. https://doi.org/10.1093/eurheartj/ehw574

Panfilov, A. V. (1998). Spiral breakup as a model of ventricular fibrillation. *Chaos: An Interdisciplinary Journal of Nonlinear Science*, *8*(1), 57–64. https://doi.org/10.1063/1.166287

Patel, Y. A., George, A., Dorval, A. D., White, J. A., Christini, D. J., & Butera, R. J. (2017). Hard real-time closed-loop electrophysiology with the Real-Time eXperiment Interface (RTXI). *PLOS Computational Biology*, *13*(5), e1005430. https://doi.org/10.1371/journal.pcbi.1005430

Prinz, A. A., & Cudmore, R. H. (2011). Dynamic clamp. *Scholarpedia*, *6*(5), 1470. https://doi.org/10.4249/scholarpedia.1470

Robinson, H. P. C., & Kawai, N. (1993). Injection of digitally synthesized synaptic conductance transients to measure the integrative properties of neurons. *Journal of Neuroscience Methods*, *49*(3), 157–165. https://doi.org/10.1016/0165-0270(93)90119-C

Rouaud, M. (2013). *Probability, statistics and estimation. Propagation of uncertainties*.

Ryan, S. S. (2020). *Ablation of the AV Node and Implanting of a Pacemaker*. Atrial Fibrillation: Resources for Patients. https://a-fib.com/treatments-for-atrial-fibrillation/av-node-with-pacemaker/

Scardigli, M., Müllenbroich, C., Margoni, E., Cannazzaro, S., Crocini, C., Ferrantini, C., Coppini, R., Yan, P., Loew, L. M., Campione, M., Bocchi, L., Giulietti, D., Cerbai, E., Poggesi, C., Bub, G., Pavone, F. S., & Sacconi, L. (2018). Real-time optical manipulation of cardiac conduction in intact hearts. *The Journal of Physiology*, 3841–3858. https://doi.org/10.1113/JP276283@10.1111/(ISSN)1469-7793.EC2018

Sepúlveda, J. R. (2020). *Optically induced heterogeneities in cardiac tissue* [McGill University]. https://escholarship.mcgill.ca/concern/theses/6t053m985

Sharp, A. A., Abbott, L. F., & Marder, E. (1992). Artificial electrical synapses in oscillatory networks. *Journal of Neurophysiology*, *67*(6), 1691–1694. https://doi.org/10.1152/jn.1992.67.6.1691

Sharp, A. A., O'Neil, M. B., Abbott, L. F., & Marder, E. (1993). Dynamic clamp: Computer-generated conductances in real neurons. *Journal of Neurophysiology*, *69*(3), 992–995. https://doi.org/10.1152/jn.1993.69.3.992

Starmer, C. F. (2007). Vulnerability of cardiac dynamics. *Scholarpedia*, *2*(11), 1847. https://doi.org/10.4249/scholarpedia.1847

Sun, J., Amellal, F., Glass, L., & Billette, J. (1995). Alternans and period-doubling bifurcations in atrioventricular nodal conduc. *Journal of Theoretical Biology*, *173*(1), 79–91. https://doi.org/10.1006/jtbi.1995.0045

Tan, R. C., & Joyner, R. W. (1990). Electrotonic influences on action potentials from isolated ventricular cells. *Circulation Research*, *67*(5), 1071–1081. https://doi.org/10.1161/01.RES.67.5.1071

Tolkacheva, E. G., Schaeffer, D. G., Gauthier, D. J., & Mitchell, C. C. (2002). Analysis of the Fenton–Karma model through an approximation by a one-dimensional map. *Chaos: An Interdisciplinary Journal of Nonlinear Science*, *12*(4), 1034–1042. https://doi.org/10.1063/1.1515170

Tran, D. X., Yang, M.-J., Weiss, J. N., Garfinkel, A., & Qu, Z. (2007). Vulnerability to re-entry in simulated two-dimensional cardiac tissue: Effects of electrical restitution and stimulation sequence. *Chaos: An Interdisciplinary Journal of Nonlinear Science*, *17*(4), 043115. https://doi.org/10.1063/1.2784387

Ulrich, D., & Huguenard, J. R. (1996). γ-Aminobutyric acid type B receptor-dependent burst-firing in thalamic neurons: A dynamic clamp study. *Proceedings of the National Academy of Sciences of the United States of America*, *93*(23), 13245–13249.

Welsh, A. J., Delgado, C., Lee-Trimble, C., Kaboudian, A., & Fenton, F. H. (2019). Simulating waves, chaos and synchronization with a microcontroller. *Chaos: An Interdisciplinary Journal of Nonlinear Science, 29*(12), 123104. https://doi.org/10.1063/1.5094351

Wilders, R. (2006). Dynamic clamp: A powerful tool in cardiac electrophysiology. *The Journal of Physiology*, *576*(2), 349–359. https://doi.org/10.1113/jphysiol.2006.115840

Zhu, H., Sun, Y., Rajagopal, G., Mondry, A., & Dhar, P. (2004). Facilitating arrhythmia simulation: The method of quantitative cellular automata modeling and parallel running. *BioMedical Engineering OnLine*, *3*, 29. https://doi.org/10.1186/1475-925X-3-29