Department of Precision and Microsystems Engineering

Bi-threshold Logic Gates for Cellular Automata-based Mechanical Computing

EOINLEE BLEY

Report no : 2023.105
Coach : Malte ten Wolde
Professor : Dr. D. Farhadi Machekposhti
Specialisation : MSD
Type of report : Master Thesis
Date : 21/12/2023

Preface

This thesis is the culmination of my journey in High Tech Mechanical Engineering at Delft University of Technology. The road has been long and challenging, and I am grateful for the love and guidance I have received along the way. I would like to thank my supervisors, Dr. Davood Farhadi Machekposhti and Malte ten Wolde, for their guidance and patience. I would also like to thank my friends and family for their unending support and encouragement. Finally, I would like to thank the reader for taking the time to read this thesis.

Eoinlee Bley
Delft, The Netherlands
December 8, 2023

Delft University of Technology

Bi-threshold Logic Gates for Cellular Automata-based Mechanical Computing

Authors:
Eoinlee Bley (5216737)

Supervisors:
Dr. Davood Farhadi Machekposhti
Malte ten Wolde

in partial fulfillment of the requirements for the degree of

Master of Science
in Mechanical Engineering

December 8, 2023

Abstract

Mechanical computing systems have historically faced challenges in efficiently integrating computation and memory functions, often leading to complex designs and limited scalability in applications ranging from micro-electromechanical systems (MEMS) to programmable matter. This study aims to address this issue by introducing a mechanical system based on cellular automata (CA) principles, utilizing a bi-threshold tristable mechanism for implementing nonlinear Boolean functions. The system's design translates Elementary Cellular Automata (ECA) rules, such as Rule 110, into mechanical motion using compliant multistable mechanisms.

Finite Element Analysis (FEA) and pseudo-rigid body simulations validate the operational feasibility of this approach. The results confirm the system's capability to process complex computational tasks by mechanically embodying specific ECA rules. This development marks a step forward in mechanical computing, offering a more integrated approach to computational material design.

The research establishes a methodical design approach, enabling the embodiment of a wide range of Elementary Cellular Automata (ECA) rules within the mechanical framework. This approach extends beyond linearly separable Boolean functions, illustrating a significant advancement in mechanical computing capabilities. The findings provide a basis for future work in scaling the system and exploring its applicability in fields such as programmable matter and intelligent mechanical systems.

Contents

1 Introduction

The pursuit of integrating computational capabilities into materials has garnered significant interest, particularly in the development of intelligent mechanical systems. These systems, which encompass soft devices[1], MEMS[2], and robotic materials[3], stand to benefit from the incorporation of sensing, actuation, computation, and communication functionalities within their very substrate[4, 5, 6, 7]. The field of mechanical computation, wherein data storage and computation are executed through deformation and mechanical motion, has seen a resurgence in interest, with applications in programmable matter[8, 9], programmable mechanical metamaterials[10, 11, 12, 13], and development of intelligent mechanical structures[14].

Mechanical computation, distinct from its electronic counterpart, presents unique challenges and opportunities. The seminal works of Roukes[15] and Reif[16] have demonstrated that mechanical systems can indeed process information, and recent advancements have led to the realisation of mechanical logic gates through innovative approaches such as origami[17, 18] and buckled beams[19]. These gates, the fundamental building blocks of computation, are typically interconnected through mechanical[20], mechano-electronic[13], or mechanical-fluidic interfaces[1], propagating signals across the computational system. However, one of the primary limitations in mechanical computing–and Von Neumann computational architectures in general–is the inefficiency in the data exchange between memory and computing modules. This bottleneck has prompted the exploration of in-memory mechanical computation, though such solutions often necessitate complex peripheral systems or intricate energy landscapes to function effectively[21].

Mechanical memory plays a pivotal role in these systems, with the purpose of encoding, storing, and retrieving data. The creation of non-volatile mechanical memory has been a significant step forward, using mechanisms like multistable structures, origami-based metamaterials, and buckled beam configurations[21, 22, 23, 24]. Despite these advances, the challenge of signal transmission[25, 26] and data exchange between memory and processing elements persists, leading to a search for more efficient computational architectures[27].

In light of these challenges, this paper investigates the application of cellular automata (CA) as an alternative computational architecture. Cellular automata are discrete, grid-based models where each cell's state evolves based on the states of its neighbours, governed by simple rules[28]. They offer a rich domain for exploring computational phenomena and have been shown to be capable of universal computation[29, 30, 31], and have been investigated as a potential computer[32] and neural network architecture[33]. The study of CA in the context of mechanical systems presents an opportunity to address the data exchange bottleneck, with the potential to harness the natural dynamics of CA for efficient information processing[34]. To date, a direct mechanical implementation of such a system has not been developed.

This paper posits a novel approach to mechanical computing by utilising multistable compliant mechanisms as bi-threshold perceptron gates. These gates can enable the implementation of various Elementary Cellular Automata (ECA) rules within a single mechanical system. The exploration of such a system could lead to a new class of intelligent metamaterials that can perform computation in memory, reducing the need for separate data storage and processing modules[27, 22, 23].

Firstly, we investigate the potential of using cellular automata as a template for mechanical computation. Cellular automata operate on the premise of local interactions and state transitions governed by a set of rules. Translating this concept into a mechanical medium involves designing elements that can change state based on the configuration of their neighboring elements[29].

Secondly, we introduce the concept of bi-threshold gates within the mechanical domain. These gates are inspired by the bi-threshold perceptron, a type of neural node that can implement a wide range of nonlinear Boolean functions[35, 36, 37]. These gates weigh and aggregate input signals and compare the result to an upper and lower threshold, outputting a binary signal. These gates have shown improved density for implementation of a given logical function over a single threshold perceptron, or traditional logic gate implementations[38, 39, 40]. In mechanical terms, these thresholds are manifested through the force thresholds in a multistable mechanism, wherein the state of equilibrium changes only when the applied force exceeds certain values[23]. These equilibrium states are analogous to the states in cellular automaton, and the transitions between these states, induced by mechanical forces, correspond to the rule-based evolution of cellular automata.

This paper aims to explore the viability of these bi-threshold gates as fundamental units of a novel mechanical computing system, seeking to answer whether they can effectively emulate the behaviour of cellular automata and overcome the challenges of data exchange inefficiency in mechanical computing systems.

Section 2 of the paper delves into the theoretical framework of Elementary Cellular Automata (ECA), outlining their mathematical formalism and how these rules can be represented and implemented in a mechanical context through bi-threshold gates and geometric modelling. Next, the core innovation of this study is introduced: the detailed design and functionality of a mechanical system's unit cell, serving as the building block for our computational architecture. Section 3, presents the results of pseudo-rigid body model simulations, showcasing

the dynamic computational capabilities of the system and its adherence to specified ECA rules. Lastly, the practicality and potential applications of the system are discussed. The constraints on the range of ECA rules it can implement, design and manufacturing limitations, scalability challenges, and the implications in various technological domains are explicated.

2 Method

2.1 Theoretical Foundations

2.1.1 Elementary Cellular Automata Formalism

Cellular automata (CA) are grid-based computational models where each cell evolves over time according to a rule set R. In Elementary Cellular Automata (ECA), the domain is one-dimensional and the state space is binary, $S = \{0, 1\}$. Each cell's future state is determined by its current state and those of its immediate neighbours.
Mathematically, for cell i at time t, the next state u_i^{t+1} is governed by a rule function $f : S^3 \to S$:

$$u_i^{t+1} = f(u_{i-1}^t, u_i^t, u_{i+1}^t)$$

With a binary state and 3-cell neighbourhood, The rule set R has $2^8 = 256$ unique ECA rules. These are indexed from 0 to 255, following Wolfram's convention, which assigns a number to each rule based on the binary outcomes of the eight possible neighborhood states, read in reverse order. Starting from the neighborhood configuration 111 down to 000, the binary sequence formed by the resulting states is converted into a decimal number. This decimal number is the Wolfram rule number, providing a unique identifier for each possible set of cellular automaton behaviours based on local interactions. For example, Rule 110 is defined explicitly as:

$$\begin{aligned} f_{110}: \quad & (0,0,0) \to 0,\ (0,0,1) \to 1,\ (0,1,0) \to 1,\ (0,1,1) \to 1, \\ & (1,0,0) \to 0,\ (1,0,1) \to 1,\ (1,1,0) \to 1,\ (1,1,1) \to 0 \end{aligned}$$

If we list these outputs in order, from $(1, 1, 1)$ to $(0, 0, 0)$, we get the binary sequence 01101110. Converting this binary sequence to a decimal number, where 01101110_2 is equivalent to 110_{10}, we get the Wolfram rule number 110. This rule number effectively summarises the state-transition logic of the cellular automaton and is a part of a broader system known as Wolfram's classification. We will take Rule 110 as a case study for the remainder of this paper, as it is amongst the most widely studied ECA rules and is known to be Turing complete[30].
Consult Figure 1A for a graphical depiction of Rule 110's eight possible neighbourhoods and their respective output states. Also shown is the time evolution of the rule, starting from a single 'on' cell at the left edge of the domain.

2.1.2 Hypercube Representation of Cellular Automata Rules

Consider a cube in $\mathbb{R}^3$ as the domain D, with each vertex representing a unique neighbourhood configuration $N = (N_{-1}, N_0, N_1)$, where $N_{-1}, N_0, N_1 \in \{0, 1\}$. The cube's vertices are colored based on a rule function $f : \{0, 1\}^3 \to \{0, 1\}$, thereby geometrically realising the Boolean truth table of an Elementary Cellular Automaton (ECA).
To introduce the concept of linear separability, consider separating parallel hyperplanes P defined by a normal vector $\mathbf{n}$ and offsets $\{d_1, d_2, \ldots, d_n\}$. These hyperplanes partition D into regions where the vertices share the same output state as determined by f.
We define a domain classification function $\Delta : D \to \{0, 1, \ldots, n\}$ as $\Delta(x) = \sum_{i=1}^{n} H(\mathbf{n} \cdot \mathbf{x} - d_i)$, where $H(z)$ is the Heaviside step function. Each value of $\Delta(x)$ corresponds to one of the $n + 1$ regions formed by P.
Finally, a mapping function $M : \{0, 1, \ldots, n\} \to \{0, 1\}$ translates the region identifier $\Delta(x)$ into the 'on' or 'off' state for each neighbourhood configuration. Refer to Figure 1B for a graphical representation of the cube and separating planes for Rule 110.

2.2 Design Methodology

One consequence of formulating 3-input Boolean functions as linearly separable regions in a hypercube in input space is the observation that most 3-input Boolean functions can be represented by a pair of parallel planes. This means we can translate a potentially complex Boolean algebraic expression composed of several AND, OR, XNOR, etc., gates into a single bi-threshold perceptron gate.
The bi-threshold perceptron for this ECA context can be formally represented as:

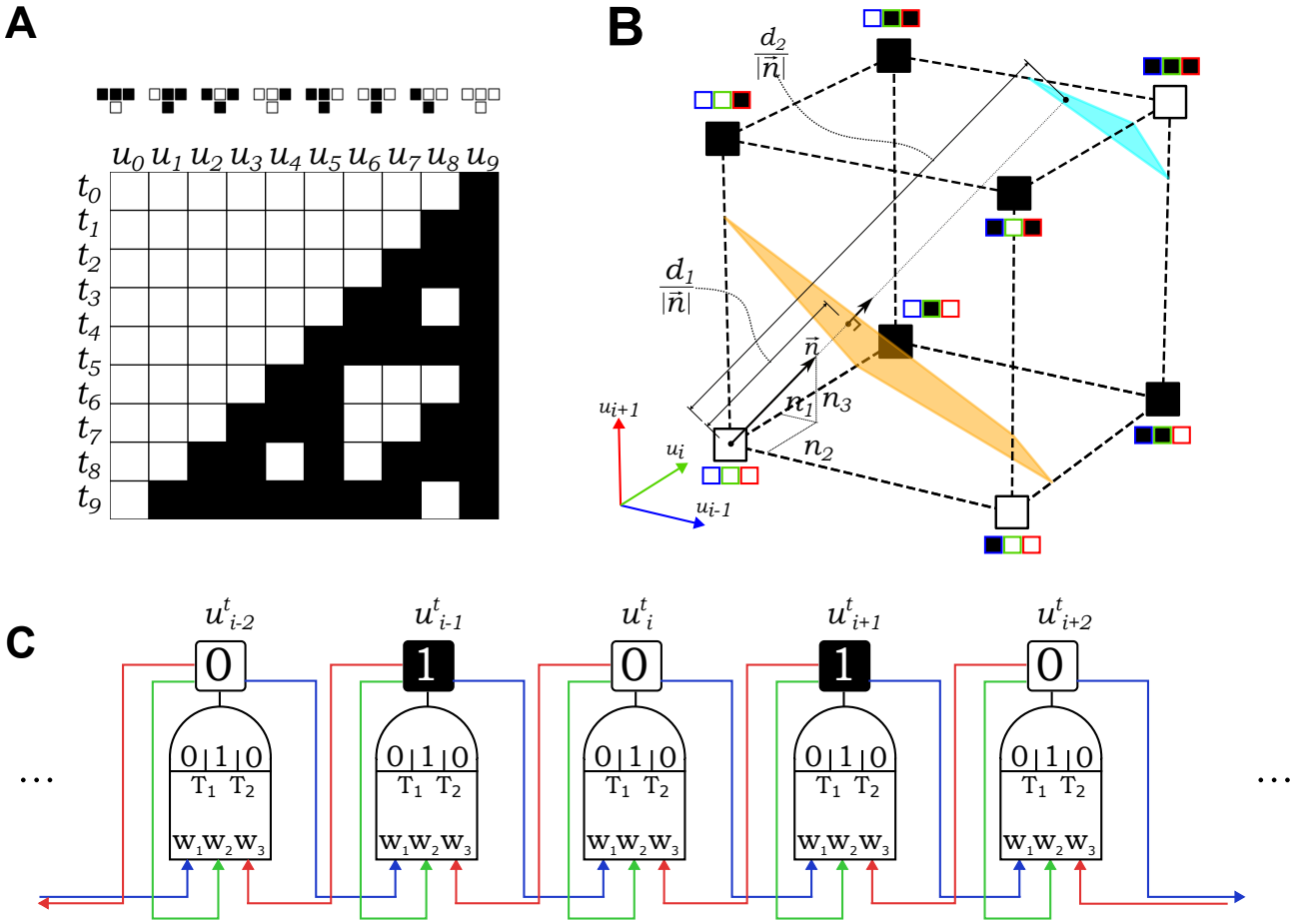


Figure 1: A. The transition rule and time evolution of the Rule 110 cellular automata, showing emergent complexity from simple initial state. B. Cube representation Rule 110 with separating planes defined by normal vector $\vec{n}$ and offset constants d_1 and d_2. The red, green and blue colouring corresponds to left neighbour, middle, and right neighbour cells of the neighbourhood respectively. C. Proposed Bi-threshold gate circuit representation of an ECA architecture. Each cell consists of a decision element and a memory element, and transmits and receives its state information with its neigbours.

$$f(N) = \begin{cases} 0 & \text{if } \sum_{i=-1}^{1} w_i \cdot N_i > T_1 \\ 1 & \text{if } T_2 \leq \sum_{i=-1}^{1} w_i \cdot N_i \leq T_1 \\ 0 & \text{if } \sum_{i=-1}^{1} w_i \cdot N_i < T_2 \end{cases} \tag{1}$$

Where $N = (N_{-1}, N_0, N_1)$ is the neighbourhood configuration, w_i are the weights, and T_1 and T_2 are the two thresholds. The two thresholds T_1 and T_2 define the three regions of the cube, which correspond to the different output states of the ECA rule. The weights w_i are the coefficients of the neighbourhood configuration N in the linear equation $\sum_{i=-1}^{1} w_i \cdot N_i$. The weights and thresholds are the parameters of the perceptron, and their values determine the behaviour of the rule.

In light of the mathematical formalism presented, we introduce a conceptual mechanical metamaterial designed to embody the logic and behaviour of Elementary Cellular Automata (ECAs). This metamaterial is constructed from an array of interconnected unit cells, each serving as a mechanical analog to the bi-threshold pergeptron gate.

The core of each unit cell is a tristable element, functioning as the decision-making component. The tristable element has three stable states, akin to the three regions separated by the two parallel planes in the cube of our geometric representation. This element is responsible for holding the output state of the cell, dictated by the weighted sum of its inputs.

Each unit cell is interconnected via coupling springs, which transmit mechanical signals between adjacent cells. The stiffness values of the coupling springs k_i act as the weights w_i in the bi-threshold perceptron equation. These values determine the force interactions and state transitions between adjacent unit cells. The tristable

elements have multiple stable states, analogous to the regions separated by planes in the cube of our geometric ECA representation.
An input clock signal introduces a temporal dimension to the mechanical system, enabling dynamic state evolution similar to time-stepping in ECAs. This clock signal sets the computational cycle and synchronises the unit cells.
Thus, the mechanical properties of the springs and tristable elements correspond directly to the mathematical constructs of the bi-threshold perceptron, providing a means to implement ECA rules in a mechanical system. The specific embodiment of this concept is detailed in the following section.

2.3 Detailed Implementation

2.3.1 Unit Cell Architecture

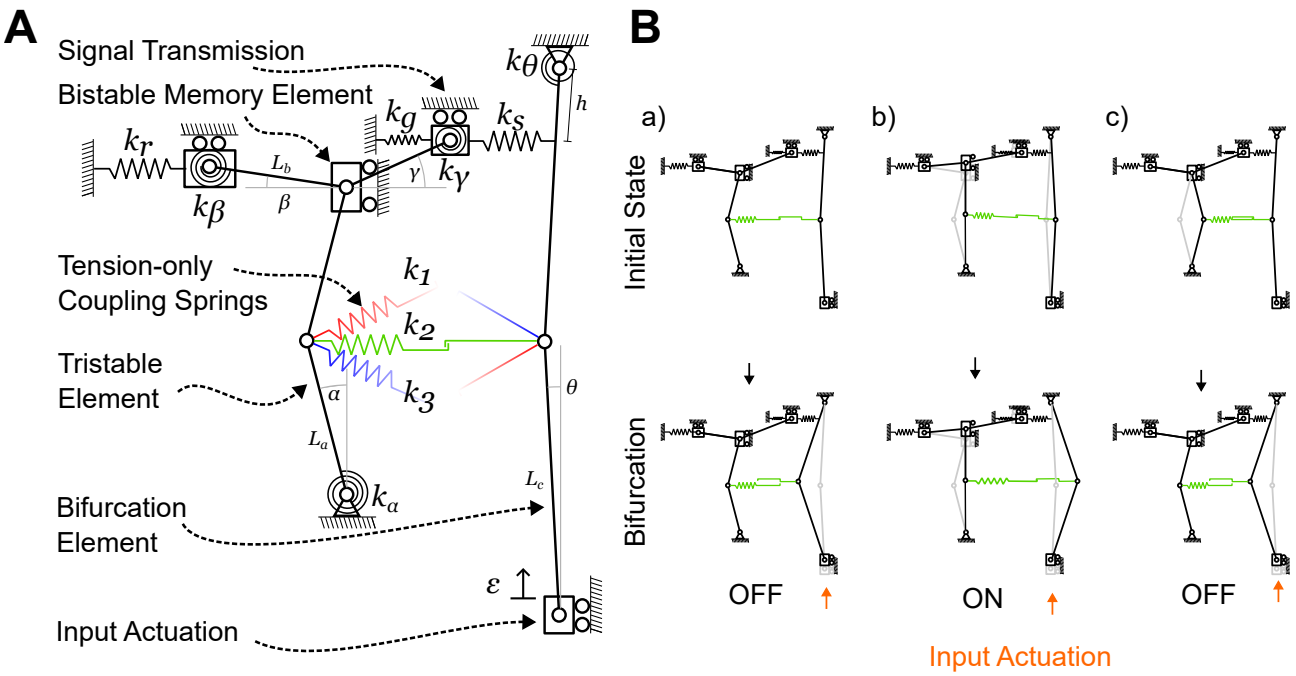


Figure 2: A. Pseudo-rigid body model of unit cell. B. Bifurcation element displacement under actuation for each possible starting state configuration.

The unit cell is designed to be planar and monolithic for scalability, operate under a single shared clock signal for synchronisation, transmit forces between adjacent cells, hold state in the absence of input, and transition states according to neighboring conditions. Figure 2A depicts the unit cell and its key components in schematic form: the tristable element, the bistable element, and the signal transmission connecting it to the input bifurcation element. Coupling springs, coloured in red, green, and blue, link the tristable element out-of-plane to the bifurcation elements of adjacent unit cells and the cell itself. The configuration of each unit cell is fully defined by two displacements: d^t in the $\hat{e}_1$ direction for the tristable element and d^b in the $\hat{e}_1$ direction for the bifurcation element due to the parallel links constraining rotational degrees of freedom, as shown in the compliant schematic representation in Figure 3. The input bifurcation element is so named because under actuation by an input clock signal ϵ, its shuttle block will displace a distance δ in one of two directions, pushing or pulling on the coupling springs according to the state of the unit cell, "on" or "off" respectively. In order for the the bifurcation element to bifurcate only due to the configuration of the bistable element and not the tristable element, the operation of the unit cell requires the coupling springs to be *tension-only* (Figure 2B). The details of this feature is elaborated in the next section.
The bistable element acts as a mechanical binary memory element for the unit cell, while the tristable element's two snap-through force thresholds act as decision boundaries corresponding to the thresholds of the threshold gate, or the separating planes of the boolean function. The signal transmission and bifurcation elements facilitate the temporal clocking and informational interconnection of the unit cells.
A simplified pseudo-rigid body model of the unit cell is depicted in Figure 2A. In this model, the contributions of numerous short-length flexure joints are aggregated into four torsional springs with angular stiffnesses

$(k_\alpha, k_\beta, k_\gamma, k_\theta)$. These springs are subject to characteristic angular displacements $(\alpha, \beta, \gamma, \theta)$, which represent the angles of the tristable, bistable, signal transmission, and bifurcation links, respectively. The angles α, β, and γ are part of a single-degree-of-freedom kinematic chain, while θ and the input displacement ϵ form another single-degree-of-freedom kinematic chain. The bistable element is simplified using symmetry and modeled as a single-link slider with torsional stiffness k_β and a reaction/support stiffness k_r. The transmission element is represented as a single link connecting the bistable element's shuttle block to a horizontal slider block. This link has a torsional stiffness k_γ and is guided by flexures with a support stiffness k_g. The signal transmission block is connected to the bifurcation element via a spring with stiffness k_s. This spring is attached at a point located a distance h from the anchor pivot of the bifurcation element.
Figure 2B shows the effect of the position of the tristable element on the behaviour of the bifurcation element under actuation. For each possible starting state configuration, the bifurcation element buckles accordingly to the left or right, depending on the position of the tristable element. This behaviour is analogous to the decision boundaries of the bi-threshold perceptron, where the tristable element's position determines the state of the unit cell, and the bifurcation element's displacement determines the output state of the cell.

2.3.2 Compliant Embodiment and FEA Modelling

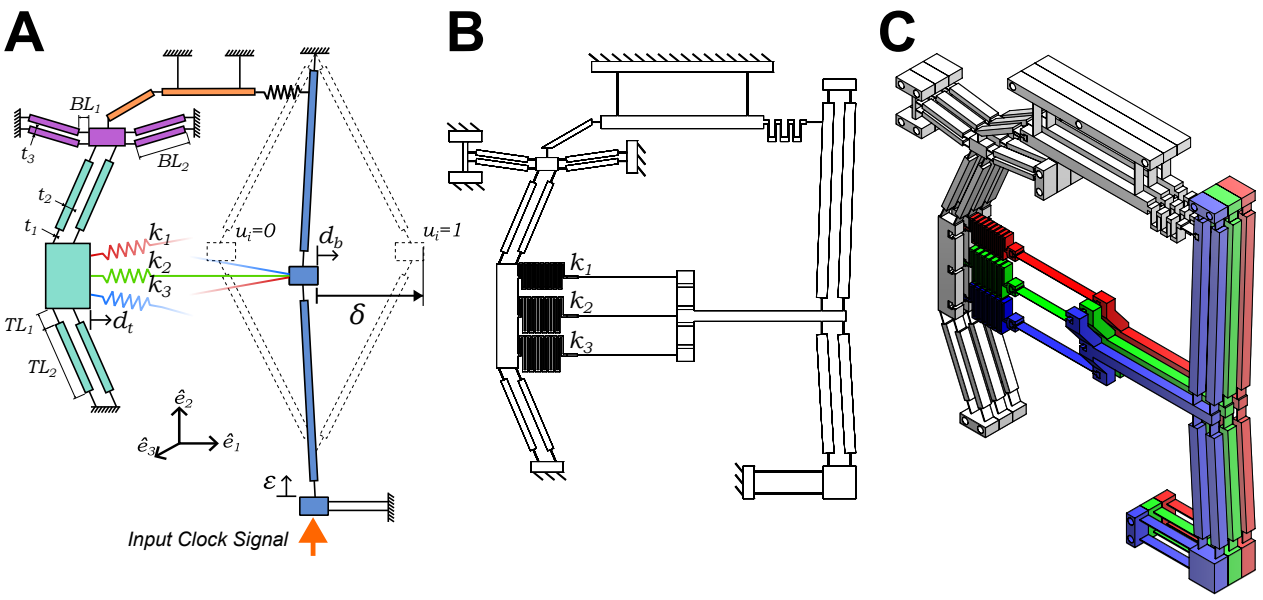


Figure 3: A. Schematic representation of compliant embodiment unit cell with tristable (teal), bistable (magenta), signal (orange) and bifurcation (blue) elements highlighted. Dashed configurations show extent of bifurcation deflection under actuation B. Actual compliant representation of single unit cell, showing tension only springs as thin buckling lamina. Also shown is bistable support, which can be tuned to affect bistable force thresholds. C. 3D CAD representation of periodic arrangement of the prototype. The coloured bifurcation elements of the neighbourhood all connect to the central cell. The coupling springs of the neighbours are not shown for clarity.

As the previous section showed, the force displacement behaviour of the tristable mechanism is crucial to the performance of the unit cell as a logical element. Gou et al. [23] developed a methodical design approach for monolithic compliant multistable structures. The approach employs a single bistable element with additional end effectors in series–which act effectively as frequency multipliers–to add extra stable points. In this paper, we use the same approach, selecting the architecture with a single additional end effector, which can result in a mechanism with 1, 2, 3, or 4 stable states, depending on the design parameters. First, a suitable bistable mechanism is selected and characterised. In our case, this is modelled as a single link-slider, with initial angle β_0, link length L_b, bending stiffness k_β, and compressive reaction-support stiffness k_r. To convert the model to a compliant embodiment, these values are then converted to dimensional parameters of flexures and reinforced sections, based on the Young's modulus and flexural strength of the material to be used, as well as the constraints of the manufacturing method and the desired scale of the mechanism. Appropriate values for these parameters

can be found in literature. We then fine-tuned the values with parameter sweeps in FEA and PRBM simulations. The values used in this study are shown in Table 1.
The end-effector compliant embodiment comprises of two pairs of parallel, lumped-compliance links and a moving end-effector block. The submechanism is parameterised by an initial link angle α_0, link length L_a and combined flexural stiffness k_α. These are calculated to satisfy the following conditions:

- The bistable element is at its second equilibrium position when the end effector is in the central configuration, i.e. $\alpha = 0°$.
- The retaining force of the bistable element in its second equilibrium position is greater than the peak reaction force generated by the end effector flexures in the central configuration.
- The snap-through force of the bistable element is greater than the peak reaction force generated by the end effector flexures in the right-most configuration.
- The buckling load of the end effector flexures is greater than the peak reaction force of the bistable element.

These conditions impose upper and lower bounds on the design parameters, which can be used to determine the appropriate values for the end effector parameters. The full design process can be found in the cited paper[23].
The tension-only coupling springs are modelled as linear springs with stiffness k_i. The stiffness values are calculated using the parametric design strategy outlined in the previous section. The springs are designed to be tension-only, i.e. they are slack when the bifurcation mechanism buckles to the left, signalling "off". This can be achieved in embodiment by several methods, such as a contact-based latch system [41], or by using a buckling flexure with a negligible buckling load. The latter approach is assumed in this paper, for simplicity and reducing contact effects. The buckling force of a leaf flexure can be calculated by $F_b = \frac{\pi^2 EI}{L_b^2}$, where E is the Young's modulus of the material, I is the second moment of area of the flexure cross-section, and L_b is the length of the flexure. The buckling force can be arbitrarily reduced by increasing the length of the flexure and keeping I sufficiently small.
The coupling springs were designed as serpentine springs with a single buckling flexure. The spring is defined by the following parameters: spring leg length L_s, number of legs N, and the moment of inertia of the spring leg cross section I The spring must be designed such that at maximum extension Δ, the stress does not exceed the yield stress of the material $\sigma_{\max}$, which can be calculated by $\sigma_{\max} = \frac{k_i \Delta L t}{4I}$. Rearranging this gives an upper bound on the length of each leg: $L_{\max} = \frac{\sigma_{\max} 4I}{k_i \Delta}$. The total stiffness of the spring is given by $k_i = \frac{12EI}{L^3 N}$, as the legs act as springs in series. This formula gives a lower bound on N to satisfy the range of motion constraint using $N = \frac{12EI}{L_{\max}^3 k_i}$. Together these allow the selection of a suitable spring design, that satisfies any form factor constraints. $L_s = \sqrt[3]{\frac{12EI}{N k_i}}$ The table below shows the calculated parameters for one possible embodiment of the tristable element and coupling springs to implement Rule 110:

Parameter	w	$t1$	$t2$	$t3$	$BL1$	$BL2$	k_β	k_r	$TL1$	$TL2$	β_0	α_0
Value	5	0.4	2.5	2.5	4	24	0.0146	407.7	6	35	8	22
Units (mm or °)	mm	mm	mm	mm	mm	mm	Nm/rad	kN/m	mm	mm	°	°

Table 1: Design parameters for tristable element and coupling springs

2.3.3 FEA Validation of Tristable Element & Coupling Spring

Ansys APDL was used to validate the design of the tristable element and coupling springs determined by the PRB model. The sub-mechanisms were modelled using BEAM188 elements with linear elastic material to represent the flexures. Fixed boundary conditions were imposed where the mechanism would be anchored to the substrate. The coupling springs stiffness was validated by applying a fixed extension to the spring and measuring the reaction force. The tristable element was validated using a displacement boundary condition on the tristable shuttle, and measuring the reaction force, as the snap-through behaviour prevents convergence when a force boundary condition is applied. The snap-through and bi-threshold behaviour was validated by applying a displacement boundary condition to all combinations of coupling springs using arc-length loading method and determining the equilibrium states. The bifurcation and signal transmission components were also validated to demonstrate the state-dependent buckling behaviour of the input mechanism. Additionally, the complete unit cell was modelled and simulated to validate the dependence of deflection of the bifurcation element under actuation on the state of the tristable element. This was done by initialising the tristable element in each of its stable states and applying a displacement boundary condition to the input mechanism. The results of the FEA validation are shown later in Figure 8.

2.3.4 Pseudo-Rigid Body Modelling

The culmination and main result of the theoretical framework and concept mechanism developed thus far is a pseudo-rigid body model simulation of the unit cell and complete system, implemented in a custom Python script. The simulation implements the kinematics and kinetics of the simplified unit cell in Figure 2B, and models the nonlinear interaction between cells, and the simultanoeous actuation of the bifurcation mechanisms. The simulation's objectives are twofold:

1. To produce the force-displacement graphs that enable the parametric design strategy for implementing a specific ECA rule.

2. To provide a computational testing ground for the system's time evolution, thereby serving as a preliminary validation of the parametric design strategy for embodying a specific ECA rule.

The simulation considers each cell having two degrees of freedom corresponding to the displacements of the tristable shuttle d^t and the bifurcation shuttle d^b, modelled as link angles α and θ respectively. Intermediate angular and linear displacements of the spring elements are calculated using the forward kinematics of the PRBM. The total energy of each unit cell and the coupling springs, as well as the reaction torques on the degrees of freedon are calculated using the pseudo-rigid body model, using dimensions derived from the analysis and FEA in the preceding section. The energy of each spring is calculated using the linear spring force-displacement relationship. A triangular wave function represents the clock signal *varepsilon*, with amplitude derived from the bifurcation element's maximum displacement δ as per the chosen ECA rule. The simulation advances in 1000 timesteps per actuation cycle. Equilibrium states are resolved at each timestep by minimising the system's total energy. This is achieved using Sequential Least Squares Programming with derivative information, with each new timestep initialised to the previous state's equilibrium.
The comprehensive codebase for the simulation can be found in Appendix C.

2.3.5 Force-Displacement Characterisation of Unit Cell

Figure 4A shows the theoretical force-displacement behaviour of the tristable element, graphing the reaction force on the tristable shuttle as a function of its horizontal displacement d^t. The three stable equilibria and corresponding configuration of the mechanism are shown. The specific behaviour is a function of all the joint stiffnesses and the chosen geometric dimensions and proportions of the mechanism. These design parameters must be precisely calculated and calibrated to achieve the precise desired behaviour, as variation can lead to mono-stable, bi-stable, or quad-stable behaviour. The specific derivation of the force-displacement response is detailed in Appendix
The theoretical force-displacement behaviour of the coupling tension-only springs is depicted in Figure 4B. Specifically, the spring behaves as a linear element with stiffness k_i when its elongation is positive. For negative elongation, the spring is slack, resulting in zero force. The stiffness k_i serves as a design parameter for selecting the specific ECA rule to be implemented.
Figure 4C illustrates a free-body diagram detailing the forces acting upon the tristable shuttle element. Here, F_r represents the total reaction force exerted by the tristable mechanism on the shuttle, while F_s denotes the cumulative force from all non-slack coupling springs. Equilibrium is achieved when the reaction and spring forces balance: $F_r(d^t) = F_s(d^t, d^b)$.
In Figure 5, the force-displacement behaviour of the unit cell is depicted under varying conditions of coupling spring activations. When the bifurcation shuttle is fixed at a displacement δ, denoted as $d^b = \delta$, the cumulative force from the activated coupling springs is plotted alongside the force-displacement curve of the tristable element, $F_r(d^t)$. b Equilibrium points emerge where these two force-displacement curves intersect and are represented as red dots on the difference plot. The equilibrium points correspond to the configurations where the cumulative force F_s from the coupling springs is equal to the reaction force F_r of the tristable element: $F_r(d^t) = F_s(d^t, d^b)$.
An equilibrium point "disappears" when the force-displacement line of the activated coupling springs no longer intersects with specific regions of the F_r curve—specifically, the regions around its local maxima. This occurs when the slope of the force-displacement curve for the activated springs, pivoting about the point $d^b = \delta$, not only fails to intersect but actually surpasses these local maxima regions of $F_r(d^t)$.
In this context, snap-through events happen as follows: the tristable element transitions from its current equilibrium state to an adjacent one depending on whether $F_r < F_s$ or $F_r > F_s$. This transition is triggered when the equilibrium state corresponding to the current F_r and F_s values no longer exists. This mechanical action serves as the physical embodiment of the decision boundaries in the bi-threshold perceptron.

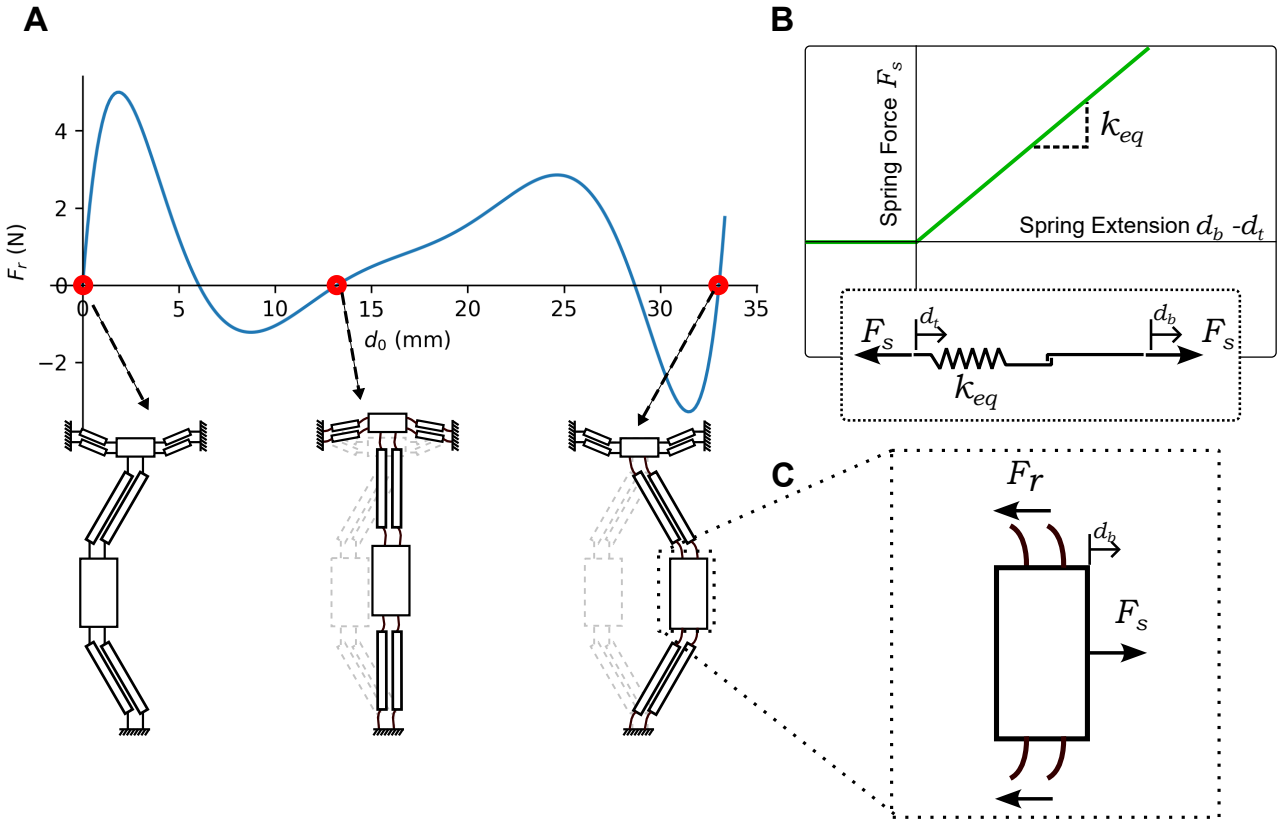


Figure 4: A. Force-displacement response of the three stable equilibria and corresponding configurations of the state element. B. Force-displacement response of the tension only spring

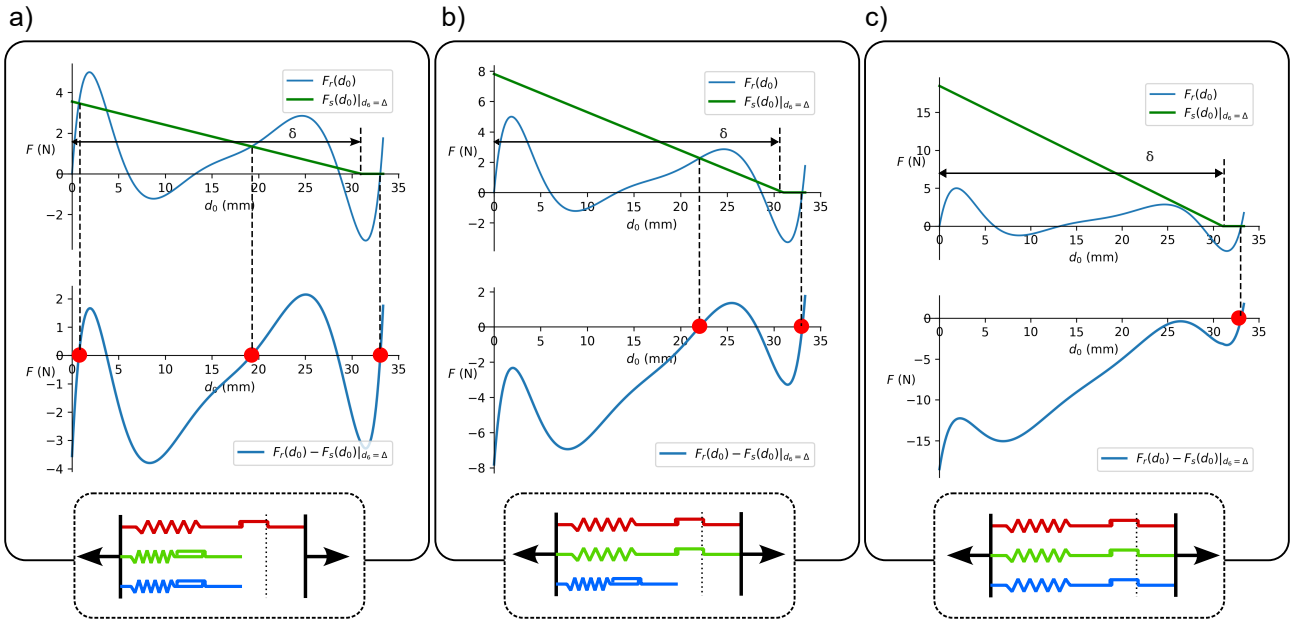


Figure 5: Force-displacement behaviour with varying coupling spring activations. Equilibrium points, shown as red dots, emerge where the tristable element's reaction force F_r intersects with the cumulative spring force F_s. Equilibria disappear when the spring force curve, anchored at $d^b = \delta$, surpasses F_r's local maxima, triggering a snap-through event. This models the decision boundaries of a bi-threshold perceptron.

Recalling the mathematical formalism of ECA rules as bi-threshold perceptrons, we can now establish a direct correspondence between the mechanical and computational domains. In the bi-threshold perceptron model, the output state is determined by evaluating a weighted sum of inputs and comparing it against two threshold

values T_1 and T_2:
Here, w_i are the weights, and x_i are the input states from the neighborhood.
In the mechanical system, these weights w_i are analogous to the stiffness k_i of the coupling springs. The input states x_i correspond to the displacements d^b of the neighboring cells, a function of the states of their bistable elements. The thresholds T_1 and T_2 correspond to the critical effective stiffnesses of the cumulative active coupling springs at which the tristable element undergoes snap-through transitions. These critical effective stiffnesses are determined by the slopes of the lines that are tangent to the specific maxima regions on the $F_r(d^t)$ force-displacement curve. These tangent lines are anchored at the point where $d^b = \delta$ on the d^t axis.

2.3.6 Parametric Strategy for ECA Rule Embodiment

Here we outline a parametric strategy for physically embodying a specific ECA rule, leveraging its geometric representation as parallel planes. The aim is to precisely calibrate the stiffness values k_i of the coupling springs and the maximum displacement δ of the bifurcation element to manifest the desired ECA rule.
A pivotal insight is that modifying the bifurcation element's maximum displacement δ alters the slopes of the lines tangent to the $F_r(d^t)$ curve, as shown in Figure 6. These tangent lines are anchored at the point $d^b = \delta$ on the d^t axis. This adjustment effectively varies the ratio between the bi-threshold perceptron's T_1 and T_2 values. Therefore, by selecting a specific δ, we can control the orientation of the separating planes, aligning them with the desired ECA rule.
Subsequently, we can determine the coupling spring stiffnesses k_i, which correspond to the slopes, which in turn correspond to the separating plane orientations, thereby completing the physical embodiment of the selected ECA rule.
However, the curve describing the critical stiffness ratio $\frac{k_2}{k_1}$ as a function of δ depends on the specific force displacement curve of the tristable element.

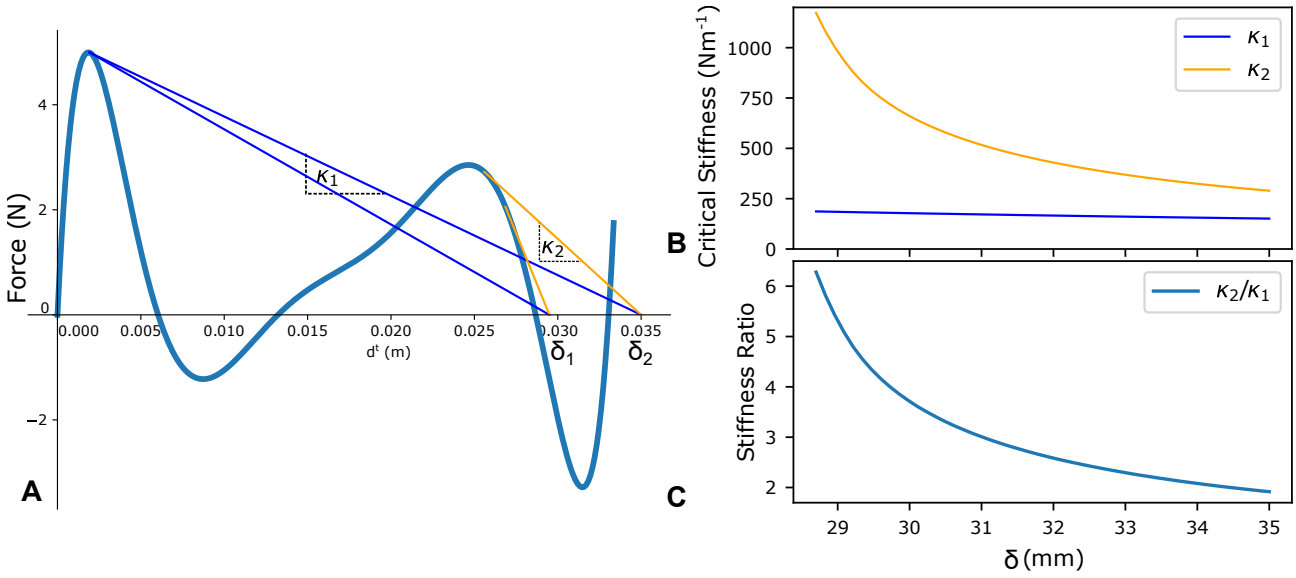


Figure 6: A. Force-displacement curve of the tristable with two different bifurcation element displacements δ_1, δ_2 showing the change in the slope of the tangent lines at the local maxima. B. Curves showing the critical stiffnesses k_1 and k_2 as a function of the bifurcation element displacement δ. C. Curve showing the critical stiffness ratio $\frac{k_2}{k_1}$ as a function of the bifurcation element displacement δ

2.4 Case Study: Implementation of Rule 110

To demonstrate the efficacy of this parametric design strategy, we implement Rule 110 in a physical prototype. The design process is as follows:

Define Rule Characteristics Each Elementary Cellular Automata (ECA) rule can be geometrically characterised by a normal vector $\mathbf{n} = [n_1, n_2, n_3]$ and threshold values T_1 and T_2. For example, for Rule 110, $\mathbf{n} = [1, 2, 2]$ and $T_1 = 1.5, T_2 = 4.5$.

Compute Critical Stiffness Ratio The ratio of the threshold values, $\frac{T_2}{T_1}$, serves as a critical parameter in the design. For Rule 110, $\frac{T_2}{T_1} = 3$.

Determine Bifurcation Displacement The bifurcation displacement δ corresponding to the critical stiffness ratio is determined by consulting Figure 6C. In the case of Rule 110, $\delta \approx 31\,\text{mm}$.

Calculate Critical Stiffnesses The critical effective stiffnesses κ_1 and κ_2 are obtained from Figure 6B, based on the selected bifurcation displacement δ.

Compute Coupling Spring Stiffness The stiffness k_i of each coupling spring is then derived using:

$$k_i = \frac{n_i \times \kappa_1}{T_1}$$

The bifurcation displacement is bounded below by the $2L_a \cos(\alpha_0)$, in order for the input to fully actuate the tristable element. It is bounded above by the stress limits of the flexures on the bifurcation element.
The final mechanical parameters for a given ECA rule, such as Rule 110, are summarised in Table 2.

	κ_1	κ_2	k_1	k_2	k_3	δ
Value	171.81	515.39	114.54	229.08	229.08	31
Units	N/m	N/m	N/m	N/m	N/m	mm

Table 2: Summary of Parametric Design Values for Rule 110

Figure 7 demonstrates the one-to-one correspondence between the tristable element's equilibrium configurations and the cube representation of Rule 110. Each equilibrium state is directly tied to a specific combination of activated coupling springs. The relative effective stiffness of these springs, when compared to two critical stiffness thresholds, serves as the mapping M that locates each vertex relative to the separating planes in the cube representation.

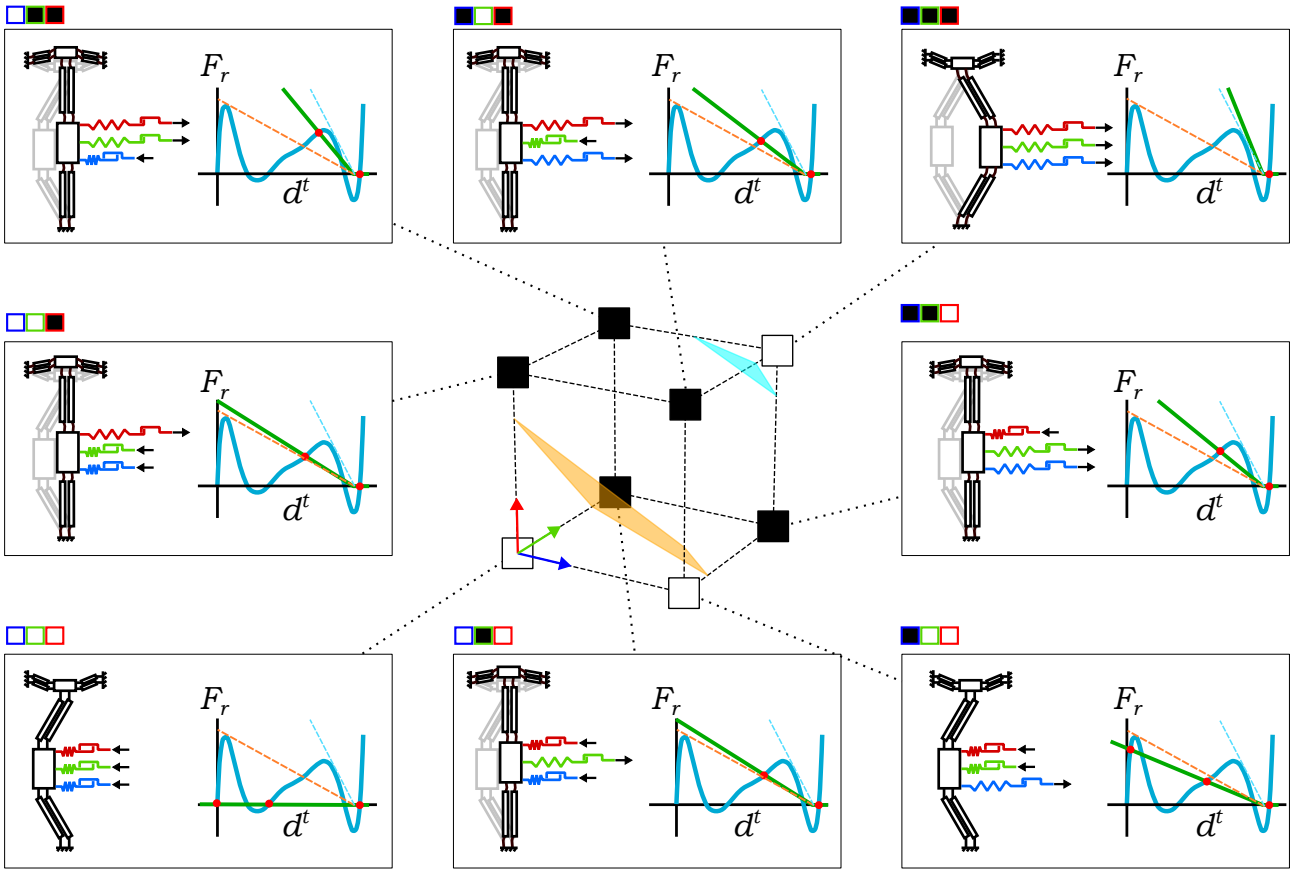


Figure 7: Equilibria of the tristable element corresponding to Rule 110. Each equilibrium state is directly tied to a specific combination of activated coupling springs. The relative effective stiffness of these springs, when compared to two critical stiffness thresholds, serves as the mapping M that locates each vertex relative to the separating planes in the cube representation.

3 Results & Discussion

3.1 FEA Validation of Tristable Element & Coupling Spring

Figure 8A shows the snap-through behaviour of the tristable element under four different equivalent stiffness configurations in the Ansys APDL FEA model. It shows that under a displacement boundary condition on a subset of the springs corresponding to a certain configuration of states of the neighbouring cells, the tristable element will snap-through to the corresponding equilibrium state. The tristable element will not snap-through if the input springs are slack. This result validates the ECA behaviour of the tristable element and coupling springs.

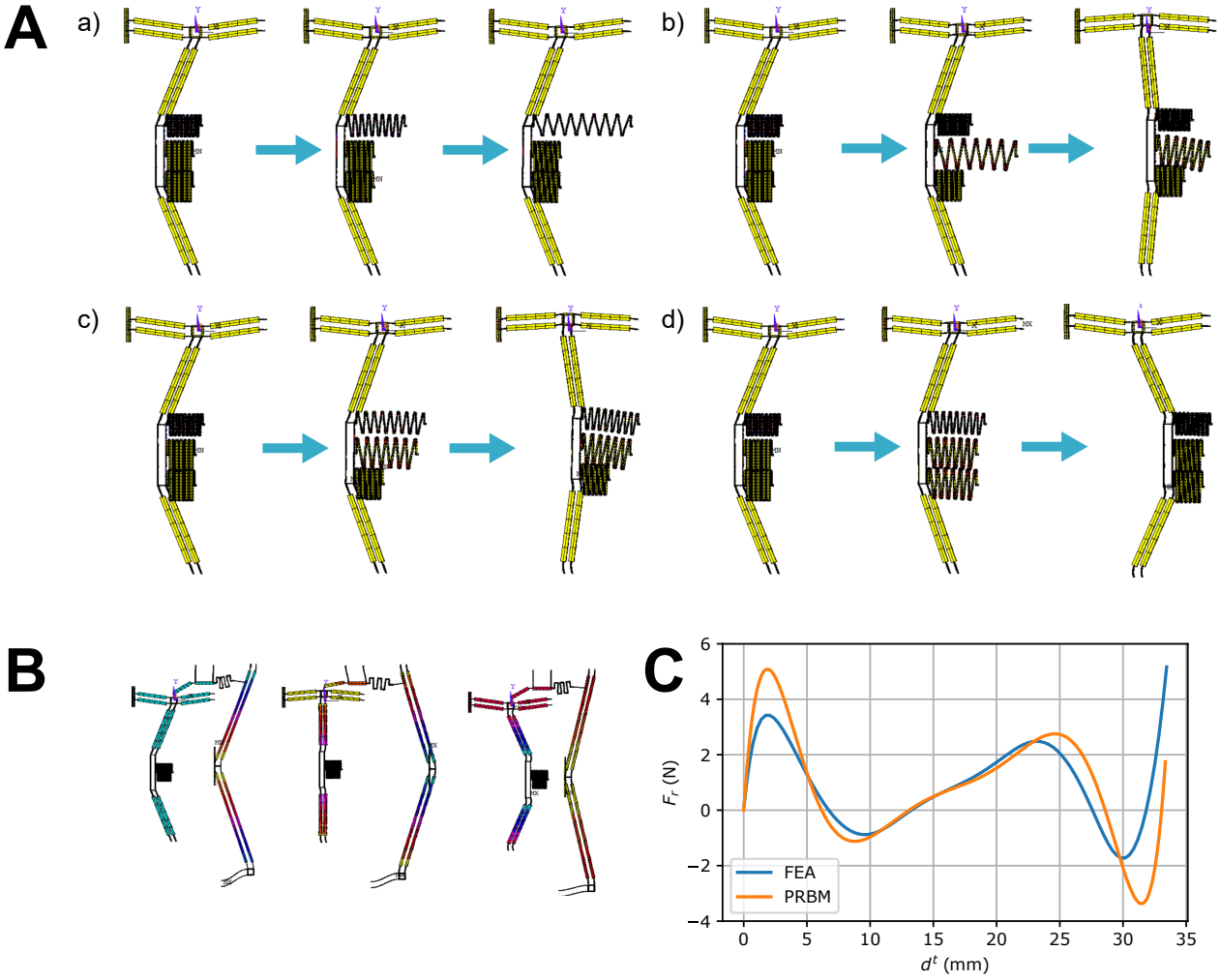


Figure 8: A. Ansys APDL FEA validation of snap-through behaviour of the tristable element under with resultant input force a) below T_1, b) above T_1 but below T_2 and c) above T_2B. Validation of the bifurcation state dependent behaviour. Input buckles to "ON" position only if state element is "ON" C. FEA Validation of the derived force displacement behaviour of the tristable element shows close agreement.

3.2 Pseudo-Rigid Body Model Simulation

Figure 9A shows a representation of the initial states and final of a a 10-unit cell system at t_0 and t_{10}. The system is initialised with a single 'on' cell at the rightmost edge of the domain Figure 9B portrays a time series simulation of the system. The states of the cells evolve in accordance with transition Rule 110. The clock signal ϵ is represented in orange. Each unit cell's bifurcation element orientation is indicative of its state—positive angle θ equates to 'off', while negative θ signifies 'on', as a negative angle implies pulling on the tension only spring.

Figure 9C shows the total potential energy E of the system over time, and the reaction force F calculated as the derivative of the potential energy with respect to the clock signal displacement ε. The energy graph is discontinuous due to the non-conservative snap-through events of the tristable elements. The magnitudes

of both the energy and reaction force are proportional to the number of unit cells in the system, as well as a funciton of the number of "on" cells, as this determines the number of active coupling springs. The reaction force is also proportional to the stiffness of the coupling springs, which depends on the ECA rule being implemented. In timestep 8, when the maximum number of cells are "on", the reaction force is at its maximum value, and the peak energy also occurs due to the energy stored in the bistable elements.

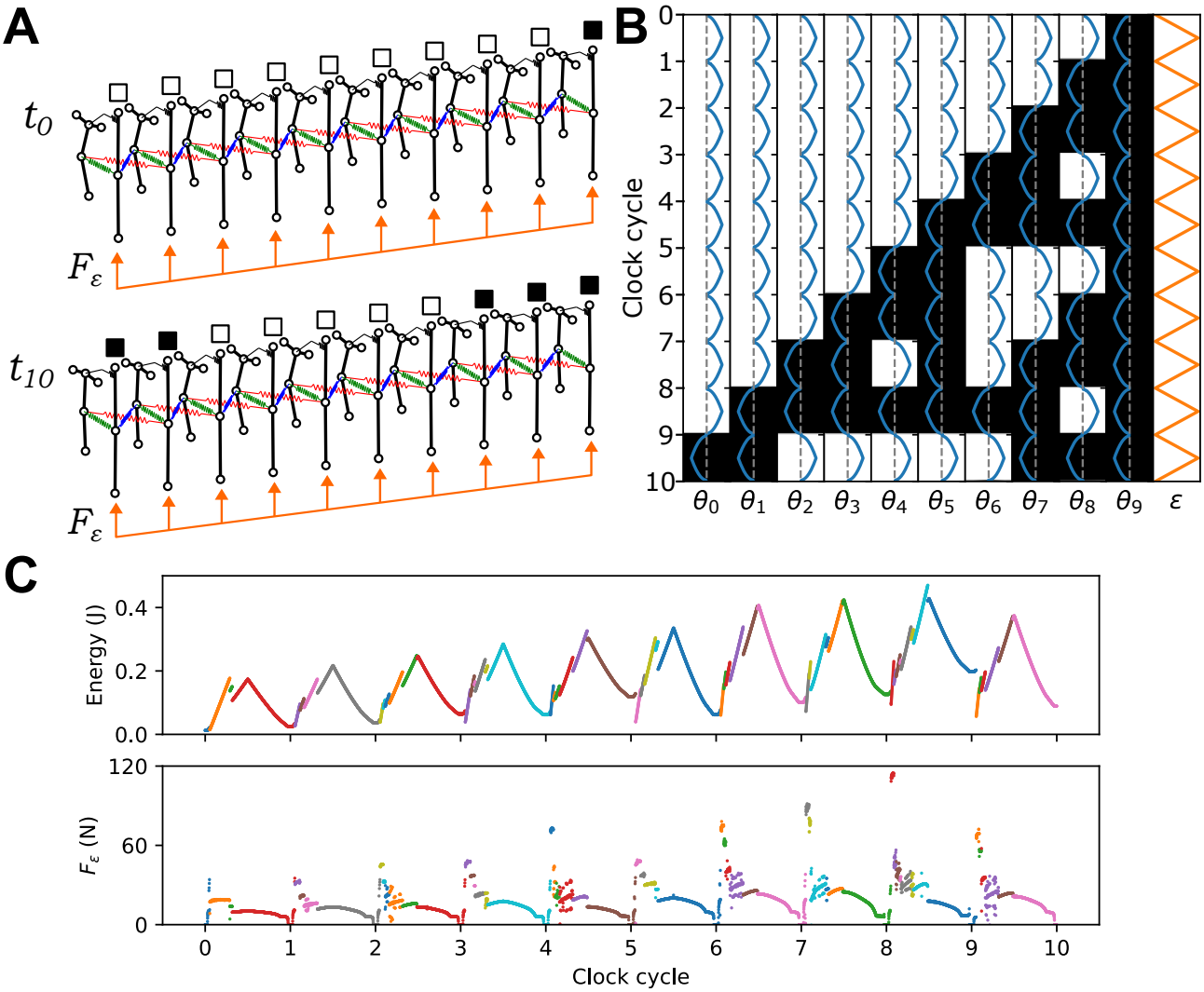


Figure 9: A. Rendering of pseudo-rigid body model at t_0 and t_{10} showing arrangement of unit cells and interconnecting springs, and the input actuation force F_ε B.Time series simulation of the system with 10 unit cells. Positive angle θ correspons to the "off" state of the unit cell, while negative angle θ corresponds to the "on" state. The clock signal ε is shown in orange. The system evolves from a single "on" cell at the right edge of the domain according to Rule 110. C. The total energy of the system and reaction force felt by the input displacement boundary condition ε.

While this paper has shown the calculation procedure for the implementation of Rule 100, the design strategy presented here can be applied to any ECA rule that is either linearly or bi-linearly separable, with some additional constraints. The normal vector of the separating planes must must have only non-negative components as the weights represent the physical property of spring stiffness, which cannot be negative. Additionally, with the current mechanism design, the rule must also satisfy the condition of null quiescence, it must output 'off' for an entirely 'off' neighbourhood i.e. $f(0,0,0) = 0$. Figure 10 shows the hypercubes of 10 bi-threshold ECA rules that satisfy these conditions. Not shown are rules with only a single threshold, as they are trivially linearly separable, and the rules which are equivalent to those shown under permutation of the inputs. The complete set of possible rules is tabulated in Appendix

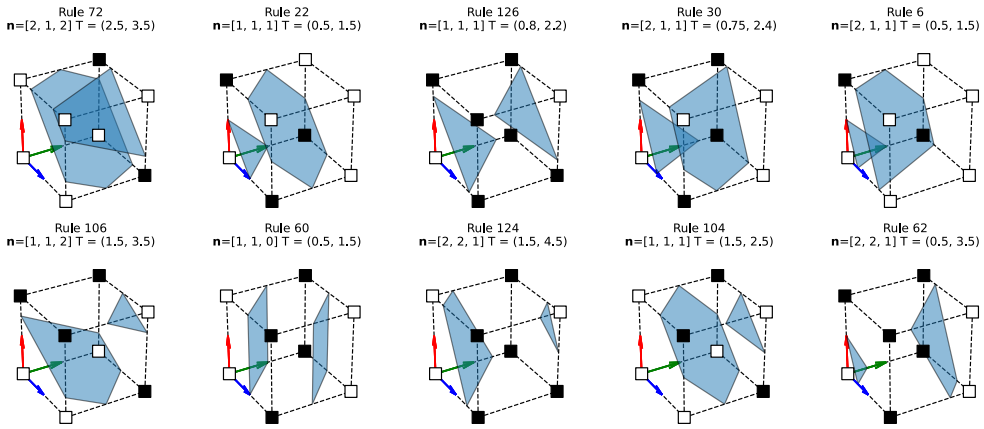


Figure 10: Cube representation of representative rules from the equivalence classes of bi-threshold ECA rules that can be implemented using the proposed design strategy. This is not exhaustive of all rules that can be implemented.

Potential applications of this research include the design of passive computing devices that operate without electricity, such as mechanical sensors that respond to environmental stimuli through changes in their structure. In soft robotics, these mechanisms could lead to the development of actuators that autonomously adjust to different loads based on pre-programmed mechanical responses. In materials science, the principles could be applied to create self-assembling structures that change configuration under specific physical conditions, serving functions similar to those of shape-memory alloys but with programmable mechanical properties.
The current embodiment of mechanical cellular automata presents several limitations that warrant attention. The complexity of the system is notable, as the intricate design and interaction of mechanical parts can lead to challenges in manufacturing and maintenance. The reliance on tension-only spring elements necessitates a mechanism for resetting, which adds to the system's complexity and may impact its reliability over time.
Additionally, as observed in Figure 9 the force requirements in the system are state-dependent, which could lead to scalability issues as larger arrays of unit cells might demand higher input forces for state transitions. This could also influence the system's energy efficiency and response times. Additionally, the computational capacity in its current form is limited; while Rule 110 has been shown to be computationally universal, the system by which this is implemented would be dramatically larger than the scale of the prototype presented here, requiring thousands of unit cells to perform basic computations. Nevertheless, the system's potential for scalability is promising.

4 Conclusion

In this research, we developed a mechanical system to implement Elementary Cellular Automata (ECA) rules, using interconnected unit cells with bi-threshold perceptron gates. This approach transforms the conceptual framework of CA into a mechanical format, focusing on the practical aspects of state transition and stability. Through finite element analysis and pseudo-rigid body simulations, we validated the mechanical system's ability to replicate the designated ECA rules. The results confirm the system's potential for mechanical computation, particularly in applications where traditional electronic components are impractical. This study contributes a methodological foundation for further investigation into mechanical systems capable of basic computational functions.

References

[1] Daniel J. Preston et al. "Digital logic for soft devices". In: *Proceedings of the National Academy of Sciences of the United States of America* 116 (16 Apr. 2019), pp. 7750–7759. ISSN: 10916490. DOI: 10.1073/PNAS.1820672116/SUPPL_FILE/PNAS.1820672116.SM02.MP4. URL: https://www.pnas.org/doi/abs/10.1073/pnas.1820672116.

[2] Ralph C. Merkle et al. "Mechanical Computing Systems Using Only Links and Rotary Joints". In: *Journal of Mechanisms and Robotics* 10 (6 Dec. 2018). ISSN: 1942-4302. DOI: 10.1115/1.4041209. URL: https://asmedigitalcollection.asme.org/mechanismsrobotics/article/10/6/061006/477620/Mechanical-Computing-Systems-Using-Only-Links-and.

[3] Katherine S. Riley et al. "Neuromorphic Metamaterials for Mechanosensing and Perceptual Associative Learning". In: *Advanced Intelligent Systems* 4 (12 Dec. 2022), p. 2200158. ISSN: 2640-4567. DOI: 10.1002/AISY.202200158. URL: https://onlinelibrary.wiley.com/doi/full/10.1002/aisy.202200158%20https://onlinelibrary.wiley.com/doi/abs/10.1002/aisy.202200158%20https://onlinelibrary.wiley.com/doi/10.1002/aisy.202200158.

[4] M. A. McEvoy and N. Correll. "Materials that couple sensing, actuation, computation, and communication". In: *Science* 347 (6228 Mar. 2015). ISSN: 10959203. DOI: 10.1126/SCIENCE.1261689.

[5] Hiromi Yasuda et al. *Mechanical computing*. Oct. 2021. DOI: 10.1038/s41586-021-03623-y.

[6] Alexandra Ion et al. "Digital Mechanical Metamaterials". In: *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems* (). DOI: 10.1145/3025453. URL: http://dx.doi.org/10.1145/3025453.3025624.

[7] Yan Fang et al. "Pattern recognition with "materials that compute"". In: *Science Advances* 2 (9 Sept. 2016). ISSN: 23752548. DOI: 10.1126/SCIADV.1601114/SUPPL_FILE/1601114_SM.PDF. URL: https://www.science.org/doi/10.1126/sciadv.1601114.

[8] Norman Margolus. "Programmable matter: Concepts and realization". In: ().

[9] C. Kaspar et al. *The rise of intelligent matter*. June 2021. DOI: 10.1038/s41586-021-03453-y.

[10] Bastiaan Florijn, Corentin Coulais, and Martin Van Hecke. "Programmable Mechanical Metamaterials". In: (2014).

[11] Katia Bertoldi et al. *Flexible mechanical metamaterials*. Oct. 2017. DOI: 10.1038/natrevmats.2017.66.

[12] Jia Xin Wang et al. "A novel programmable composite metamaterial with tunable Poisson's ratio and bandgap based on multi-stable switching". In: *Composites Science and Technology* 219 (Mar. 2022), p. 109245. ISSN: 0266-3538. DOI: 10.1016/J.COMPSCITECH.2021.109245.

[13] Tian Chen, Mark Pauly, and Pedro M. Reis. "A reprogrammable mechanical metamaterial with stable memory". In: *Nature* 589 (7842 Jan. 2021), pp. 386–390. ISSN: 14764687. DOI: 10.1038/s41586-020-03123-5.

[14] Mohammad Usman Waheed. "Functional mechanical metamaterials - development of programmable mechanical structures". In: (2022). DOI: 10.25560/95429. URL: http://spiral.imperial.ac.uk/handle/10044/1/95429.

[15] M L Roukes. "Mechanical Computation, Redux?" In: ().

[16] John H. Reif. "Mechanical Computing: The Computational Complexity of Physical Devices". In: *Encyclopedia of Complexity and Systems Science* (2017), pp. 1–21. DOI: 10.1007/978-3-642-27737-5_325-4. URL: https://link.springer.com/referenceworkentry/10.1007/978-3-642-27737-5_325-4.

[17] Jesse L. Silverberg et al. "Using origami design principles to fold reprogrammable mechanical metamaterials". In: *Science* 345 (6197 2014). ISSN: 10959203. DOI: 10.1126/science.1252876.

[18] Z. Y. Wei et al. "Geometric mechanics of periodic pleated origami". In: *Physical Review Letters* 110 (21 May 2013). ISSN: 00319007. DOI: 10.1103/PHYSREVLETT.110.215501. URL: http://arxiv.org/abs/1211.6396%20http://dx.doi.org/10.1103/PhysRevLett.110.215501.

[19] R. C. Merkle. "Two types of mechanical reversible logic". In: *Nanotechnology* 4 (2 1993), pp. 114–131. ISSN: 09574484. DOI: 10.1088/0957-4484/4/2/007.

[20] Yuanping Song et al. "Additively manufacturable micro-mechanical logic gates". In: *Nature Communications 2019 10:1* 10 (1 Feb. 2019), pp. 1–6. ISSN: 2041-1723. DOI: 10.1038/s41467-019-08678-0. URL: https://www.nature.com/articles/s41467-019-08678-0.

[21] Tie Mei et al. "A mechanical metamaterial with reprogrammable logical functions". In: *Nature Communications 2021 12:1* 12 (1 Dec. 2021), pp. 1–11. ISSN: 2041-1723. DOI: 10.1038/s41467-021-27608-7. URL: https://www.nature.com/articles/s41467-021-27608-7.

[22] Zuolin Liu et al. "Cellular automata inspired multistable origami metamaterials for mechanical learning". In: (May 2023). URL: https://arxiv.org/abs/2305.19856v1.

[23] Yanjie Gou, Guimin Chen, and Larry L. Howell. "A design approach to fully compliant multistable mechanisms employing a single bistable mechanism". In: *Mechanics Based Design of Structures and Machines* 49 (7 2021), pp. 986–1009. ISSN: 15397742. DOI: 10.1080/15397734.2019.1707685. URL: https://www.tandfonline.com/doi/abs/10.1080/15397734.2019.1707685.

[24] Guimin Chen et al. "A tristable mechanism configuration employing orthogonal compliant mechanisms". In: *Journal of Mechanisms and Robotics* 2 (1 Feb. 2010), pp. 1–6. ISSN: 19424302. DOI: 10.1115/1.4000529.

[25] Jordan R. Raney et al. "Stable propagation of mechanical signals in soft media using stored elastic energy". In: *Proceedings of the National Academy of Sciences of the United States of America* 113 (35 Aug. 2016), pp. 9722–9727. ISSN: 10916490. DOI: 10.1073/PNAS.1604838113/SUPPL_FILE/PNAS.1604838113.SM07.WMV. URL: https://www.pnas.org/doi/abs/10.1073/pnas.1604838113.

[26] Jiangnan Ding and Martin Van Hecke. "Sequential Snapping and Pathways in a Mechanical Metamaterial". In: ().

[27] Tie Mei and Chang Qing Chen. "In-memory mechanical computing". In: *Nature Communications 2023 14:1* 14 (1 Aug. 2023), pp. 1–11. ISSN: 2041-1723. DOI: 10.1038/s41467-023-40989-1. URL: https://www.nature.com/articles/s41467-023-40989-1.

[28] Andrew Ilachinski. *Cellular Automata*. WORLD SCIENTIFIC, July 2001. ISBN: 978-981-02-4623-5. DOI: 10.1142/4702. URL: https://books.google.com/books/about/Cellular_Automata.html?hl=nl&id=3Hx2lx_pEF8C.

[29] Melanie Mitchell. "Computation in Cellular Automata: A Selected Review". In: (1996).

[30] Matthew Cook. "Universality in Elementary Cellular Automata". In: (1985).

[31] Stephen Wolfram. *A New Kind of Science*. Wolfram Media, 2002. URL: https://www.wolframscience.com/nks/.

[32] R Ichard, K Squier, and Ken Steiglitz. "Programmable Parallel Arithmetic in Cellular Automata Using a Particle Model". In: *Complex Systems* 8 (1994), pp. 311–323.

[33] Fangyue Chen et al. "Implementation of arbitrary boolean functions via CNN". In: *Proceedings of the IEEE International Workshop on Cellular Neural Networks and their Applications* (2006). DOI: 10.1109/CNNA.2006.341641.

[34] Lennard J Kwakernaak and Martin Van Hecke. "Counting and Sequential Information Processing in Mechanical Metamaterials". In: ().

[35] Vladyslav Kotsovsky, Fedir Geche, and Anatoliy Batyuk. "Bithreshold Neural Network Classifier". In: *International Scientific and Technical Conference on Computer Sciences and Information Technologies* 1 (Sept. 2020), pp. 32–35. ISSN: 27663639. DOI: 10.1109/CSIT49958.2020.9321883.

[36] Vasken Bohossian and Jehoshua Bruck. "Multiple Threshold Neural Logic". In: *Advances in Neural Information Processing Systems* (1997).

[37] Donald R. Haring. "Multi-Threshold Threshold Elements". In: *IEEE Transactions on Electronic Computers* EC-15 (1 1966), pp. 45–65. ISSN: 03677508. DOI: 10.1109/PGEC.1966.264375.

[38] Saburo Muroga, Iwao Toda, and Satoru Takasu. "THEORY OF MAJORITY DECISION ELEMENTS". In: ().

[39] Ingo Wegener. "The complexity of the parity function in unbounded fan-in, unbounded depth circuits". In: *Theoretical Computer Science* 85 (1991), pp. 155–170.

[40] Vladyslav Kotsovsky and Anatoliy Batyuk. "Feed-forward Neural Network Classifiers with Bithreshold-like Activations". In: *International Scientific and Technical Conference on Computer Sciences and Information Technologies* 2022-November (2022), pp. 9–12. ISSN: 27663639. DOI: 10.1109/CSIT56902.2022.10000739.

[41] Yang Gao et al. "A Planar Single-Actuator Bi-Stable Switch Based on Latch-Lock Mechanism". In: *2019 20th International Conference on Solid-State Sensors, Actuators and Microsystems and Eurosensors XXXIII, TRANSDUCERS 2019 and EUROSENSORS XXXIII* (June 2019), pp. 705–708. DOI: 10.1109/TRANSDUCERS.2019.8808375.

Supplementary Material

A Complete Set of Bi-threshold ECA Rules

The complete set of bi-threshold ECA rules is shown in Figure 11. The rules are numbered according to the Wolfram numbering scheme.

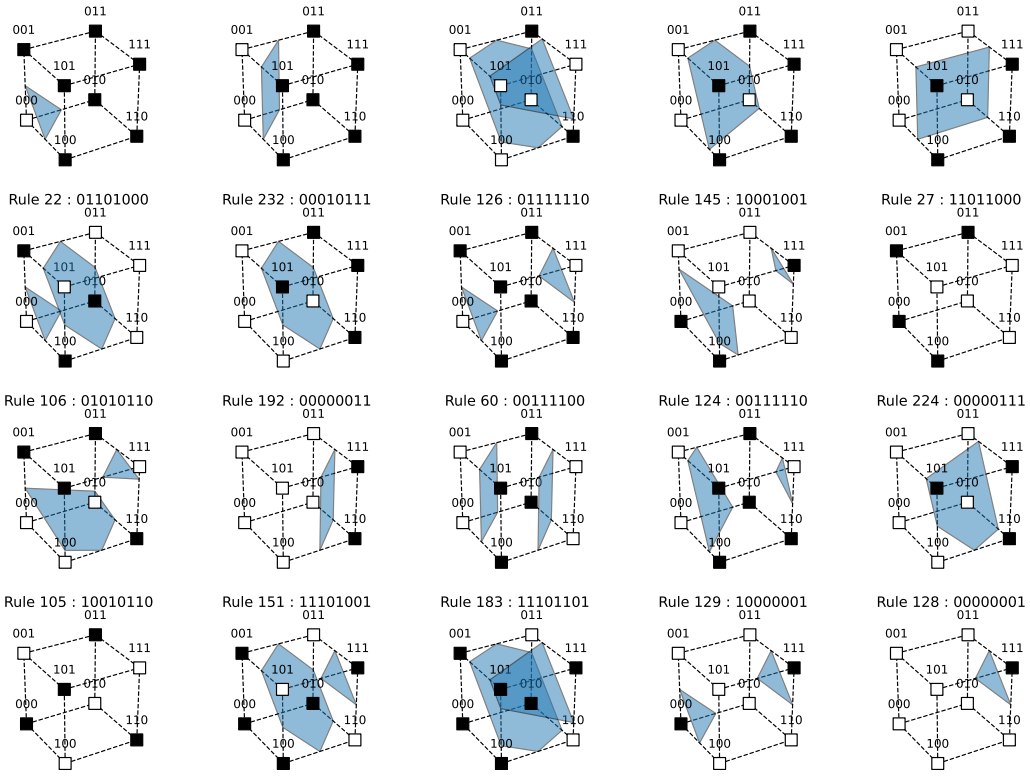


Figure 11: Complete set of implementable bi-threshold ECA rules.

B Compliant Mechanism Kinematics and Kinetics Derivation

Below is the derivation for the kinematics and kinetics of the pseudo rigid-body model used in the implementation of the simulation.

B.1 Kinematics

The kinematics are derived by calculating a series of intermediate displacements and angles. The equations are derived by considering the geometry of the mechanism and the constraints imposed by the rigid links and slider elements in the mechanism according to Figure 2A. The intermediate displacements and angles are defined as follows:
The horizontal displacement of the tristable element end effector is defined as d_0 and is given by:

$$d_0 = L_1(\sin\alpha_0 - \sin\alpha)$$

where L_1 is the length of one leg of the tristable element, α_0 is the angle of the tristable element in the initial state, and α is the angle of the tristable element in the current state. The vertical displacement of the tristable element bistable shuttle is defined as d_1 and is given by:

$$d_1 = 2L_1(\cos\alpha - \cos\alpha_0)$$

The angle of the bistable shuttle with respect to the horizontal axis is defined as β, which depends on the vertical displacement of the bistable shuttle d_1 and is given by:

$$\beta = \sin^{-1}\left(\frac{L_2 \sin\beta_0 - d_1}{L_2}\right)$$

where L_2 is the length of the bistable link and β_0 is the angle of the bistable shuttle in the initial state. The horizontal displacement of the bistable support is defined as d_2 and is given by:

$$d_2 = \frac{L_2 \sin\beta_0 - d_1}{\tan\beta} - L_2 \cos\beta_0$$

The angle of the signal router with respect to the horizontal axis is defined as γ, which depends on the vertical displacement of the bistable shuttle d_1 and is given by:

$$\gamma = \sin^{-1}\left(\frac{L_3 \sin\gamma_0 - d_1}{L_3}\right)$$

where L_3 is the length of the signal router link and γ_0 is the angle of the signal router in the initial state. The resulting horizontal displacement of the signal router end point is defined as d_3 and is given by:

$$d_3 = \frac{L_3 \sin\gamma_0 - d_1}{\tan\gamma} - L_3 \cos\gamma_0$$

The lever arm of the signal on the bifurcation element is defined as L_5 and is given by:

$$L_5 = \frac{L_4 \cos\theta_0 - (L_1 \cos\alpha_0 + L_3 \sin\gamma_0)}{\cos\theta_0}$$

where L_4 is the length of the bifurcation element, θ_0 is the angle of the bifurcation element in the initial state, and θ is the angle of the bifurcation element in the current state. This is derived from the assumption that the end effector of the tristable element and the signal router mid point are at the same height. The horizontal displacement of the point of connection of the signal on the bifurcation element is defined as d_4 and is given by:

$$d_4 = L_5(\sin\theta_0 - \sin\theta)$$

The input displacement of the bifurcation element is defined as d_5 and is given by:

$$d_5 = 2L_4(\cos\theta - \cos\theta_0)$$

The horizontal displacement of the bifurcation element end effector of unit cell i is defined as d_6^i and is given by:

$$d_6^i = L_4(\sin\theta_0 - \sin\theta)$$

As the unit cells are interconnected, and affected by the horzontal displacement of the bifurcation element end effector of the neighboring unit cells, the horizontal displacement of the bifurcation element end effector of the previous unit cell is defined as d_6^{i-1} and the horizontal displacement of the bifurcation element end effector of the next unit cell is defined as d_6^{i+1}.

Kinetics

The total mechanical energy of the system is composed of the individual energies of the flexures and tension only springs. Each angular stiffness is modelled as a torsional spring with stiffness K_f and each linear displacement is modelled as a compression only spring with stiffness C. The equivalent stiffness of the torsional spring consists of a number flexures, the equivalent stiffness of which is given by:

$$K_f = \frac{Ebt^3}{12L_f}$$

where E is the Young's modulus of the material, b is the width of the flexure, t is the thickness of the flexure, and L_f is the length of the flexure. As there may be various flexures, they are referred to as K_f1, K_f2, K_f3, and K_f4 for the flexures in the tristable element, bistable shuttle, signal router, and bifurcation element respectively. The bistable support stiffness is modelled as a fixed-fixed beam with stiffness given by:

$$C_b = \frac{Ebt_s^3}{L_s^3}$$

where t_s is the thickness of the bistable support and L_s is the length of the bistable support. The signal router connecting stiffness is modelled as a serpentine spring with 6 legs, the equivalent stiffness of which is given by:

$$C_s = \frac{Ebt^3}{6L_{signal}}$$

where t is the thickness of the serpentine flexure and L_{signal} is the length of the serpentine flexure. A small guiding stiffness of the signal router was also modelled as a pair of leaf flexures in parallel with stiffness given by:

$$C_g = \frac{2Ebt^3}{L_{guide}^3}$$

where t is the thickness of the leaf flexure and L_{guide} is the length of the leaf flexure. Additionally, the input displacement boundary condition is modelled as a compression only spring with a very high stiffness $C_i$to act as a rigid constraint. The coupling springs between unit cells are modelled as tension only springs, with stiffness determined by the rule being implemented, and referred to as K_1, K_2, and K_3 for the coupling springs between unit cells $i-1$, i, and $i+1$ respectively. The tension-only and compression-only springs are modelled using a Heaviside step function to ensure that the springs only exert force in the correct direction. The total energy of the system is the summation of the energies of the individual components and is given by:

$$\begin{aligned} E = \frac{1}{2}\Big(& 8K_{f1}(\alpha-\alpha_0)^2 + 8K_{f2}(\beta-\beta_0)^2 + 2K_{f3}(\gamma-\gamma_0)^2 \\ & + 8K_{f4}(\theta-\theta_0)^2 + C_b d_2^2 + C_s d_3^2 + C_g(d_4-d_3)^2 \\ & + C_i H(d_5+\varepsilon)(d_5+\varepsilon)^2 + K_1 H(d_6^{i-1}-d_0)(d_6^{i-1}-d_0)^2 \\ & + K_2 H(d_6^{i}-d_0)(d_6^{i}-d_0)^2 + K_3 H(d_6^{i+1}-d_0)(d_6^{i+1}-d_0)^2 \Big) \end{aligned}$$

where ε is the input displacement boundary condition, H is the Heaviside step function, and d_6^{i-1}, d_6^i, and d_6^{i+1} are the horizontal displacements of the bifurcation element end effector of the previous, current, and next unit cells respectively.
The force/moment displacement responses of the unit cell can be derived by taking the derivative of the energy with respect to the any of the displacements.

C Python Script for Pseudo-Rigid Body Simulation

The simulation models the kinematics of the rigid bodies using the forward kinematics of the mechanism. Each unit cell has two degrees of freedom corresponding to the displacements d^t and d^b, modelled as link angles α and θ respectively. The energy of the system is calculated using the pseudo-rigid body model, with the energy of each spring calculated using the linear spring force-displacement relationship. The input is modelled as the triangular wave function of the clock signal ϵ with amplitude calculated from the maximum displacement of the bifurcation element δ according to the selected ECA rule with 1000 timesteps per actuation cycle. This boundary condition is modeled as a compression only spring with sufficiently high stiffness to act as a rigid constraint. The equilibrium of the system is determined each timestep by minimizing the total energy of the system using Sequential Least Squares Programming with derivative information with initial state being the state of the previous time step.

```
import sympy as sm
import numpy as np
import matplotlib.pyplot as plt
import scipy.optimize as opt
import scipy.signal as sig
from mpl_toolkits import mplot3d
from mpl_toolkits.mplot3d.art3d import Line3DCollection
from matplotlib.collections import LineCollection
import prbm_helper_functions as phf



class UnitCell:
    """
    A class representing a unit cell in the system.

```

```
    Attributes:
    - alpha (float): the angle of the left leg with respect to the vertical axis
    - theta (float): the angle of the right leg with respect to the vertical axis
    - params (dict): a dictionary containing the mechanism parameters
    - i (int): the index of the unit cell in the system
    - left_neighbor (UnitCell): the left neighbor of the unit cell
    - right_neighbor (UnitCell): the right neighbor of the unit cell
    - F_d_func (function): a function that computes the force residual in the left leg
    - F_b_func (function): a function that computes the force residual in the right leg
    """
    def __init__(self, alpha_init, theta_init, params,i):
        self.alpha = alpha_init
        self.theta = theta_init
        self.params = params
        self.i = i


        self.left_neighbor: UnitCell = None
        self.right_neighbor: UnitCell = None

        self.define_symbols_and_parameters()

        self.expressions = self.define_symbolic_functions()

        self.create_lambda_functions()


    def define_symbols_and_parameters(self):
        self.alpha_sym, self.theta_sym, self.u_sym, self.theta_prev_sym, self.theta_next_sym =
     sm.symbols('alpha, theta, u, theta_prev, theta_next')
        self.params_symbols = {
            'alpha0': sm.symbols('alpha0'),
            'beta0': sm.symbols('beta0'),
            'gamma0': sm.symbols('gamma0'),
            'theta0': sm.symbols('theta0'),
            'L1': sm.symbols('L1'),
            'L2': sm.symbols('L2'),
            'L3': sm.symbols('L3'),
            'L4': sm.symbols('L4'),
            'C1': sm.symbols('C1'),
            'C2': sm.symbols('C2'),
            'C3': sm.symbols('C3'),
            'C4': sm.symbols('C4'),
            'Kf1': sm.symbols('Kf1'),
            'Kf2': sm.symbols('Kf2'),
            'Kf3': sm.symbols('Kf3'),
            'Kf4': sm.symbols('Kf4'),
            'K1': sm.symbols('K1'),
            'K2': sm.symbols('K2'),
            'K3': sm.symbols('K3'),
            'tog_offset': sm.symbols('tog_offset'),
        }

    def define_symbolic_functions(self):

        # Extract symbols
        alpha = self.alpha_sym
        theta = self.theta_sym
        u = self.u_sym
        theta_prev = self.theta_prev_sym
        theta_next = self.theta_next_sym
        params = self.params_symbols.copy()
        params.update(self.params) # Merge with numerical parameters

        alpha0 = params['alpha0']
        beta0 = params['beta0']
        gamma0 = params['gamma0']
        theta0 = params['theta0']
        L1 = params['L1']
        L2 = params['L2']
        L3 = params['L3']
        L4 = params['L4']
        C1 = params['C1']
```

```
        C2 = params['C2']
        C3 = params['C3']
        C4 = params['C4']
        Kf1 = params['Kf1']
        Kf2 = params['Kf2']
        Kf3 = params['Kf3']
        Kf4 = params['Kf4']
        K1 = params['K1']
        K2 = params['K2']
        K3 = params['K3']
        tog_offset = params['tog_offset']


        d0 = L1*(sm.sin(alpha0) - sm.sin(alpha)) #horizontal displacement of the decision element end effector
        d1 = 2*L1*(sm.cos(alpha) - sm.cos(alpha0)) #vertical displacement of the decision element bistable shuttle
        beta = sm.asin((L2*sm.sin(beta0)-d1)/L2) #angle of bistable shuttle with respect to the horizontal axis
        d2 = (L2*sm.sin(beta0)-d1)/(sm.tan(beta))-L2*sm.cos(beta0) #horizontal displacement of the bistable support
        gamma = sm.asin((L3*sm.sin(gamma0)-d1)/L3) #angle of the signal router with respect to the horizontal axis
        d3 = (L3*sm.sin(gamma0)-d1)/(sm.tan(gamma))-L3*sm.cos(gamma0)  #horizontal displacement of the signal router
        L5 = (L4*sm.cos(theta0)-(L1*sm.cos(alpha0) + #lever arm of the signal on the bifurcation element
                L3*sm.sin(gamma0)))/sm.cos(theta0)
        d4 = L5*(sm.sin(theta0)-sm.sin(theta)) #horizontal displacement of the signal on the bifurcation element
        d5 = 2*L4*(sm.cos(theta) - sm.cos(theta0))+u #vertical displacement of the bifurcation element input point
        d6 = L4*(sm.sin(theta0) - sm.sin(theta)) #horizontal displacement of the bifurcation element end effector
        d6_prev = L4*(sm.sin(theta0) - sm.sin(theta_prev)) #horizontal displacement of the bifurcation element end effector of the previous cell
        d6_next = L4*(sm.sin(theta0) - sm.sin(theta_next)) #horizontal displacement of the bifurcation element end effector of the next cell

        self.L5 = L5

        #define offset to introduce slack in the system
        K1_tog = sm.Heaviside(d6_prev-d0 - tog_offset)
        K2_tog = sm.Heaviside(d6-d0 - tog_offset)
        K3_tog = sm.Heaviside(d6_next-d0 - tog_offset)
        C4_tog = sm.Heaviside(d5)

        # define the energy of the system
        E = 0.5*(8*Kf1*(alpha-alpha0)**2
                 + 8*Kf2*(beta-beta0)**2
                 + 2*Kf3*(gamma-gamma0)**2
                 + 8*Kf4*(theta-theta0)**2
                 + C1*(d2)**2
                 + C2*(d3)**2
                 + C3*(d4-d3)**2
                 + C4*C4_tog*(d5)**2
                 + K1*K1_tog*(d6_prev-d0-tog_offset)**2
                 + K2*K2_tog*(d6-d0-tog_offset)**2
                 + K3*K3_tog*(d6_next-d0-tog_offset)**2
                 )

        # define the reaction moments of the system

        M_d = sm.diff(E, alpha)
        M_b = sm.diff(E, theta)



        M_b_prev = sm.diff(E, theta_prev)
        M_b_next = sm.diff(E, theta_next)
        #put all expressions in a dictionary to return including intermediate expressions
        expressions = {
            'E': E,
```

```
            'M_d': M_d,
            'M_b': M_b,
            'M_b_prev': M_b_prev,
            'M_b_next': M_b_next,
            'beta': beta,
            'gamma': gamma,
            'd0': d0,
            'd1': d1,
            'd2': d2,
            'd3': d3,
            'd4': d4,
            'd5': d5,
            'd6': d6,
            'd6_prev': d6_prev,
            'd6_next': d6_next,
            'K1_tog': K1_tog,
            'K2_tog': K2_tog,
            'K3_tog': K3_tog,
            'C4_tog': C4_tog,
        }

        return expressions

    def create_lambda_functions(self):
        #Define the symbolic expression for the dirac delta function so that it evaluates to zero for physical reasons
        dirac_delta_zero = lambda x: 0

        self.num_funcs = {}
        for key, expression in self.expressions.items():
            self.num_funcs[key] = sm.lambdify(
                (self.alpha_sym, self.theta_sym, self.u_sym,self.theta_prev_sym, self.theta_next_sym), expression, {"DiracDelta": dirac_delta_zero})

    def update_state(self, alpha, theta):
        self.alpha = alpha
        self.theta = theta

    def update_neighbors_state(self):
        self.left_neighbor_theta = self.left_neighbor.theta if self.left_neighbor is not None else self.params['theta0']
        self.right_neighbor_theta = self.right_neighbor.theta if self.right_neighbor is not None else self.params['theta0']

    def compute_cell_energy(self, u):
        self.update_neighbors_state()
        # Evaluate the lambdified functions
        E = self.num_funcs['E'](self.alpha, self.theta, u,self.left_neighbor_theta,self.right_neighbor_theta)
        return E

    def compute_force_residuals(self, u):

        self.update_neighbors_state()

        M_d = self.num_funcs['M_d'](self.alpha, self.theta, u,self.left_neighbor_theta,self.right_neighbor_theta)
        M_b = self.num_funcs['M_b'](self.alpha, self.theta, u,self.left_neighbor_theta,self.right_neighbor_theta)
        M_b_prev = self.num_funcs['M_b_prev'](self.alpha, self.theta, u,self.left_neighbor_theta,self.right_neighbor_theta)
        M_b_next = self.num_funcs['M_b_next'](self.alpha, self.theta, u,self.left_neighbor_theta,self.right_neighbor_theta)

        return M_d, M_b, M_b_prev, M_b_next

    def compute_quantities(self, u):
        self.update_neighbors_state()
        quantities = {}
        for key, _ in self.expressions.items():
            quantities[key] = self.num_funcs[key](self.alpha, self.theta, u,self.left_neighbor_theta,self.right_neighbor_theta)
        return quantities
```

```
213
214
215
216     def compute_points(self):
217
218         BE_DE_gap = self.params['L4']
219
220         beta = self.num_funcs['beta'](self.alpha, self.theta, 0,self.left_neighbor_theta,self.
    right_neighbor_theta)
221         gamma = self.num_funcs['gamma'](self.alpha, self.theta, 0,self.left_neighbor_theta,
    self.right_neighbor_theta)
222
223
224         end_effector = np.array([-self.params['L1']*np.sin(self.alpha), self.params['L1']*np.
    cos(self.alpha)])
225         bistable_shuttle = np.array([0,2*self.params['L1']*np.cos(self.alpha)])
226         bistable_anchor = bistable_shuttle + np.array([self.params['L2']*np.cos(beta),self.
    params['L2']*np.sin(beta)])
227         bistable_anchor2 = bistable_shuttle + np.array([-self.params['L2']*np.cos(beta),self.
    params['L2']*np.sin(beta)])
228         signal_router = bistable_shuttle + np.array([self.params['L3']*np.cos(gamma),self.
    params['L3']*np.sin(gamma)])
229         signal_router2 = signal_router + np.array([BE_DE_gap/4,0])
230         bifurcation_anchor = np.array([BE_DE_gap, self.params['L1']*np.cos(self.params['alpha0
    '])+self.params['L4']*np.cos(self.params['theta0'])])
231         bifurcation_shuttle = bifurcation_anchor + np.array([-self.params['L4']*np.sin(self.
    theta),-self.params['L4']*np.cos(self.theta)])
232         bifurcation_signal = bifurcation_anchor+np.array([-self.L5*np.sin(self.theta),-self.L5
    *np.cos(self.theta)])
233         bifurcation_input = bifurcation_anchor+np.array([0,-2*self.params['L4']*np.cos(self.
    theta)])
234
235
236         points = np.array([
237             np.array([0, 0]), #origin
238             end_effector, #end effector
239             bistable_shuttle, #bistable shuttle
240             bistable_anchor, #right bistable anchor
241             bistable_anchor2, #left bistable anchor
242             signal_router, #signal router
243             signal_router2, #signal router 2
244             bifurcation_signal, #bifurcation signal
245             bifurcation_anchor, #bifurcation anchor
246             bifurcation_shuttle, #bifurcation shuttle
247             bifurcation_input, #bifurcation input
248         ])
249         #convert points to dtype float
250         points = points.astype(float)
251         return points
252
253     def compute_lines(self):
254         points = self.compute_points()
255         tristable_lines = [
256             [points[0], points[1]], #origin to end effector
257             [points[1], points[2]], #end effector to bistable shuttle
258             [points[2], points[3]], #bistable shuttle to right bistable anchor
259             [points[2], points[4]], #bistable shuttle to left bistable anchor
260         ]
261         signal_router_lines = [
262             [points[2], points[5]], #bistable shuttle to signal router
263             [points[5], points[6]], #signal router to signal router 2
264         ]
265         bifurcation_lines = [
266             [points[8], points[9]], #bifurcation anchor to bifurcation shuttle
267             [points[9], points[10]], #bifurcation shuttle to bifurcation input
268         ]
269         spring_length = np.linalg.norm(points[7]-points[6])
270         signal_spring = Spring(ne=4, a=spring_length, r0=3e-3)
271         ss_xs, ss_ys= signal_spring.compute(*points[7], *points[6])
272         #compute the lines of the signal spring,
273         signal_spring_lines = [[(ss_xs[i],ss_ys[i]),(ss_xs[i+1],ss_ys[i+1])] for i in range(
    len(ss_xs)-1)]
274
```

```
        return tristable_lines, signal_router_lines, bifurcation_lines, signal_spring_lines



class System:
    def __init__(self, init_config, params):
        self.n = len(init_config)
        self.u_init = -1e-3
        self.u = self.u_init
        self.cell_spacing = 10e-3
        self.threshold = 1e-3
        self.params = params

        # Initialize unit cells
        alpha_init = params['alpha0']
        theta_init = params['theta0']

        self.unit_cells = [UnitCell(alpha_init, theta_init, params,i) for i in range(self.n)]
        self.state_vector = np.zeros(2*self.n)
        # Connect unit cells
        for i,cell in enumerate(self.unit_cells):
            if i > 0:
                cell.left_neighbor = self.unit_cells[i - 1]
            if i < self.n - 1:
                cell.right_neighbor = self.unit_cells[i + 1]

            self.state_vector[2*i] = cell.alpha
            self.state_vector[2*i+1] = cell.theta


        self.alphaub = np.deg2rad(30)
        self.alphalb = np.deg2rad(-30)

        self.thetaub = np.deg2rad(60)
        self.thetalb = np.deg2rad(-60)

        #define lower and upper bounds by repeating the upper and lower bounds for each unit
    cell
        self.lb = np.array([self.alphalb,self.thetalb]*self.n)
        self.ub = np.array([self.alphaub,self.thetaub]*self.n)
        self.bounds = opt.Bounds(self.lb,self.ub)

        self.set_configuration(init_config)

    def set_bounds(self, alb, aub, tlb, tub):

        self.lb = np.array([alb, tlb]*self.n)
        self.ub = np.array([aub, tub]*self.n)
        self.bounds.lb = self.lb
        self.bounds.ub = self.ub

    def update_state(self, state_vector):
        """
        Updates the state of the system given the current state vector.

        Args:
        state_vector (numpy.ndarray): The current state vector.
        """
        self.state_vector = state_vector
        for cell in self.unit_cells:
            cell.update_state(state_vector[2*cell.i], state_vector[2*cell.i+1])

    def compute_energy(self, state_vector=None):
        """
        Computes the energy of the system given the current state vector.

        Args:
        state_vector (numpy.ndarray): The current state vector.

        Returns:
        energy (float): The energy of the system.
        """
        # Update state in each cell
```

```
        if state_vector is not None:
            self.update_state(state_vector)
        else:
            state_vector = self.state_vector

        # Compute energy
        energy = 0
        for cell in self.unit_cells:
            energy += cell.compute_cell_energy(self.u)

        return energy

    def compute_residuals(self, state_vector=None):
        """
        Computes the residuals of the system given the current state vector.

        Args:
        state_vector (numpy.ndarray): The current state vector.

        Returns:
        residuals (numpy.ndarray): The residuals of the system.
        """
        # Update state in each cell
        if state_vector is not None:
            self.update_state(state_vector)
        else:
            state_vector = self.state_vector
        # Compute residuals
        moment_residuals = np.zeros(2*self.n)
        force_residuals = np.zeros(2*self.n)
        for cell in self.unit_cells:
            M_d, M_b,M_b_prev,M_b_next = cell.compute_force_residuals(self.u)
            moment_residuals[2*cell.i] = M_d
            moment_residuals[2*cell.i+1] = M_b

            if cell.left_neighbor is not None:
                moment_residuals[2*cell.i-1] += M_b_prev
            if cell.right_neighbor is not None:
                moment_residuals[2*cell.i+3] += M_b_next
        return moment_residuals

    def compute_force_residuals(self, state_vector=None):
        # Update state in each cell
        if state_vector is not None:
            self.update_state(state_vector)
        else:
            state_vector = self.state_vector

        force_residuals = np.zeros(2*self.n)
        for cell in self.unit_cells:
            M_d, M_b,M_b_prev,M_b_next = cell.compute_force_residuals(self.u)
            force_residuals[2*cell.i] = M_d/(self.params['L1']*np.cos(cell.alpha))
            force_residuals[2*cell.i+1] = M_b/(self.params['L4']*np.cos(cell.theta))
            if cell.left_neighbor is not None:
                force_residuals[2*cell.i-1] += M_b_prev/(self.params['L4']*np.cos(cell.
left_neighbor.theta))
            if cell.right_neighbor is not None:
                force_residuals[2*cell.i+3] += M_b_next/(self.params['L4']*np.cos(cell.
right_neighbor.theta))
        return force_residuals

    def get_quantity(self, quantity, state_vector=None):

        # Update state in each cell
        if state_vector is not None:
            self.update_state(state_vector)
        else:
            state_vector = self.state_vector

        # Compute quantity
        result = np.zeros(self.n)
        for cell in self.unit_cells:
```

```
            result[cell.i] = cell.num_funcs[quantity](cell.alpha, cell.theta, self.u,cell.left_neighbor_theta,cell.right_neighbor_theta)

        return result

    def get_intermediate_quantities(self, state_vector=None):

        # Update state in each cell
        if state_vector is not None:
            self.update_state(state_vector)
        else:
            state_vector = self.state_vector

        # Compute quantities
        results = {key: [cell.compute_quantities(self.u)[key] for cell in self.unit_cells]
                for key in self.unit_cells[0].expressions.keys()}
        return results




    def solve_equilibrium(self):
        """
        Solves the system of equations to find the equilibrium state of the unit cell.
        This method collects all force residuals and solves the system of equations using a suitable numerical method.
        """
        # Collect all force residuals
        equilibria = opt.minimize(self.compute_energy, self.state_vector,bounds=self.bounds, method='SLSQP', jac=self.compute_residuals, options={'ftol': 1e-12, 'disp': False})


        # Solve the system of equations here using a suitable numerical method
        return equilibria.x

    def simulate_actuation(self, d6max = 33e-3,n_cycles = 1,frames_per_cycle = 1000):

        umax = 2*(self.params['L4']-np.sqrt(self.params['L4']**2-d6max**2))
        t = np.linspace(0, n_cycles, int(frames_per_cycle*n_cycles))
        u_range = (sig.sawtooth(2*np.pi*t, width=0.5)+1)/2*(umax-self.u_init)+self.u_init

        #initialize the states and quantities
        states = np.zeros((len(u_range), 2*self.n))
        quantities = {key: np.ndarray((self.n,len(u_range))) for key in self.unit_cells[0].expressions.keys()}


        for i,u in enumerate(u_range):
            prev_u = self.u
            self.u = u

            #resetting mechanism
            if prev_u<self.threshold and u>self.threshold:
                self.set_bounds(self.params['alpha0'],self.params['alpha0'],self.thetalb,self.thetaub)
            else:
                self.set_bounds(self.alphalb,self.alphaub,self.thetalb,self.thetaub)

            self.state_vector = self.solve_equilibrium()
            states[i] = self.state_vector

            current_quantities = self.get_intermediate_quantities()
            for key in quantities.keys():
                quantities[key][:,i] = current_quantities[key]




        return states, quantities, t

    def get_force_response(self, n=250):
        """
        Computes the force response of the system for a given range of angles of the left leg.
```

```
        """
        alpha0 = self.params['alpha0']
        alpha_range = np.linspace(alpha0, -alpha0, n)
        force_residuals = np.zeros((len(alpha_range), 2))
        d0 = self.params['L1']*(np.sin(self.params['alpha0'])-np.sin(alpha_range))
        bub = np.copy(self.bounds.ub)
        blb = np.copy(self.bounds.lb)
        original_ub = np.copy(self.bounds.ub)
        original_lb = np.copy(self.bounds.lb)

        for i, alpha in enumerate(alpha_range):
        # update the bounds
            blb[::2] = alpha
            bub[::2] = alpha

            self.bounds.ub = bub
            self.bounds.lb = blb
        # solve for the equilivrium state
            self.solve_equilibrium()

        # get the force residuals
            force_residuals[i] = self.compute_force_residuals()[0]
        force_response = -force_residuals[:, 0]

        #restore the bounds
        self.bounds.ub = original_ub
        self.bounds.lb = original_lb

        return force_response, d0

    def get_threshold_stiffnesses(self, d6max = 33e-3):
        force_response, d0 = self.get_force_response()

        results = phf.find_tangent_lines(np.vstack((d0,force_response)).T,[d6max,0])
        return results

    def plot_system(self, fig = None, ax = None, plot_indicators = False):
        if fig is None:
            fig = plt.figure()
        if ax is None:
            ax = plt.axes(projection='3d')

        xlim = [-0.025, 0.1]
        ylim = [-0.05, self.cell_spacing*self.n+0.05]
        zlim = [-0.03, 0.1]
        ax.set_xlim3d(xlim)
        ax.set_ylim3d(ylim)
        ax.set_zlim3d(zlim)
        ax.set_box_aspect((xlim[1]-xlim[0], ylim[1]-ylim[0], zlim[1]-zlim[0]))
        ax.set_axis_off()
        ax.set_xticks([])
        ax.set_yticks([])
        ax.set_zticks([])
        ax.view_init(0, -90)
        ax.set_proj_type('ortho')


        self.tristable_lines = []
        self.tristable_joints = []
        self.signal_router_lines = []
        self.signal_router_joints = []
        self.bifurcation_lines = []
        self.bifurcation_joints = []
        self.signal_spring_lines = []
        self.self_spring_lines = []
        self.left_spring_lines = []
        self.right_spring_lines = []
        self.indicators = []

        for cell in self.unit_cells:
            points = cell.compute_points()
            tristable_lines, signal_router_lines, bifurcation_lines, signal_spring_lines =
    cell.compute_lines()
```

```
557
558             z_loc = cell.i*self.cell_spacing
559             tristable_mech = ax.add_collection3d(LineCollection(tristable_lines,color='k'), zs
    =z_loc, zdir='y')
560             self.tristable_lines.append(tristable_mech)
561             signal_router_mech = ax.add_collection3d(LineCollection(signal_router_lines,color=
    'k'), zs=z_loc, zdir='y')
562             self.signal_router_lines.append(signal_router_mech)
563             bifurcation_mech = ax.add_collection3d(LineCollection(bifurcation_lines,color='k')
    , zs=z_loc, zdir='y')
564             self.bifurcation_lines.append(bifurcation_mech)
565             signal_spring_mech = ax.add_collection3d(LineCollection(signal_spring_lines,color=
    'k'), zs=z_loc, zdir='y')
566             self.signal_spring_lines.append(signal_spring_mech)
567
568             z_locs = np.array([z_loc]*points.shape[0])
569             tristable_joints = ax.scatter(points[:5, 0], z_locs[:5], points[:5, 1], c='w',
    marker='o',depthshade=False,edgecolors='k')
570             self.tristable_joints.append(tristable_joints)
571             signal_router_joints = ax.scatter(points[5:8, 0], z_locs[5:8], points[5:8, 1], c='
    w', marker='o',depthshade=False,edgecolors='k')
572             self.signal_router_joints.append(signal_router_joints)
573             bifurcation_joints = ax.scatter(points[8:, 0], z_locs[8:], points[8:, 1], c='w',
    marker='o',depthshade=False,edgecolors='k')
574             self.bifurcation_joints.append(bifurcation_joints)
575
576
577             cell.self_spring = Spring(ne=10, a=0.05, r0=3e-3)
578             cell.self_spring.initialize_leads(*points[9],z_loc, *points[1],z_loc)
579             ss_xs, ss_zs, ss_ys = cell.self_spring.compute_with_leads(*points[9],z_loc, *
    points[1],z_loc)
580             self_spring_lines = [[(ss_xs[i],ss_ys[i],ss_zs[i]),(ss_xs[i+1],ss_ys[i+1],ss_zs[i
    +1])] for i in range(len(ss_xs)-1)]
581             self_spring_mech = ax.add_collection3d(Line3DCollection(self_spring_lines,color='g
    '))
582             self.self_spring_lines.append(self_spring_mech)
583
584             if cell.left_neighbor is not None:
585                 cell.left_spring = Spring(ne=10, a=0.05, r0=3e-3)
586                 left_points = cell.left_neighbor.compute_points()
587                 cell.left_spring.initialize_leads(*points[1],z_loc, *left_points[9],z_loc-self
    .cell_spacing)
588                 ss_xs, ss_zs, ss_ys = cell.left_spring.compute_with_leads(*points[1],z_loc, *
    left_points[9],z_loc-self.cell_spacing)
589                 left_spring_lines = [[(ss_xs[i],ss_ys[i],ss_zs[i]),(ss_xs[i+1],ss_ys[i+1],
    ss_zs[i+1])] for i in range(len(ss_xs)-1)]
590                 left_spring_mech = ax.add_collection3d(Line3DCollection(left_spring_lines,
    color='b'))
591                 self.left_spring_lines.append(left_spring_mech)
592             else:
593                 self.left_spring_lines.append(None)
594             if cell.right_neighbor is not None:
595                 cell.right_spring = Spring(ne=10, a=0.05, r0=3e-3)
596                 right_points = cell.right_neighbor.compute_points()
597                 cell.right_spring.initialize_leads(*points[1],z_loc, *right_points[9],z_loc+
    self.cell_spacing)
598                 ss_xs, ss_zs, ss_ys = cell.right_spring.compute_with_leads(*points[1],z_loc, *
    right_points[9],z_loc+self.cell_spacing)
599                 right_spring_lines = [[(ss_xs[i],ss_ys[i],ss_zs[i]),(ss_xs[i+1],ss_ys[i+1],
    ss_zs[i+1])] for i in range(len(ss_xs)-1)]
600                 right_spring_mech = ax.add_collection3d(Line3DCollection(right_spring_lines,
    color='r'))
601                 self.right_spring_lines.append(right_spring_mech)
602             else:
603                 self.right_spring_lines.append(None)
604
605             #plot a point above the bifurcation element with a box marker
606             indicator = ax.scatter(points[7,0], z_loc, points[7,1]+30e-3, marker='s', s=100,
    edgecolors='k', zorder=100,color='k')
607             self.indicators.append(indicator)
608             # color the box marker white if the bifurcation element is in the off state, which
     is when theta is positive
609             if cell.theta > 0:
```

```
                plt.setp(indicator, color=[1,1,1],edgecolors='k')
            if not plot_indicators:
                #hide the indicator
                plt.setp(indicator, visible=False)

        ax.set_xlim3d(-0.025, 0.1)
        ax.set_ylim3d(-0.05, self.cell_spacing*self.n+0.05)
        ax.set_zlim3d(-0.03, 0.1)
        ax.set_position((0,0,1,1))

        return fig, ax

    def update_plot(self, frame, states):
        self.update_state(states[frame])

        def update_lines(lines):
            return [[(x[0][0], z_loc, x[0][1]), (x[1][0], z_loc, x[1][1])]
            for x in lines
            ]


        for i, cell in enumerate(self.unit_cells):
            z_loc = cell.i * self.cell_spacing
            points = cell.compute_points()
            tristable_lines, signal_router_lines, bifurcation_lines, signal_spring_lines =
    cell.compute_lines()
            tristable_lines = update_lines(tristable_lines)
            signal_router_lines = update_lines(signal_router_lines)
            bifurcation_lines = update_lines(bifurcation_lines)
            signal_spring_lines = update_lines(signal_spring_lines)

            # Update tristable mechanism lines
            self.tristable_lines[i].set_segments(tristable_lines)

            # Update signal router mechanism lines
            self.signal_router_lines[i].set_segments(signal_router_lines)

            # Update bifurcation mechanism lines
            self.bifurcation_lines[i].set_segments(bifurcation_lines)

            # Update signal spring mechanism lines
            self.signal_spring_lines[i].set_segments(signal_spring_lines)

            # Update joints (scatter points)
            # Update tristable joints
            self.tristable_joints[i]._offsets3d = (points[:5, 0], z_loc * np.ones(5), points
    [:5, 1])

            # Update signal router joints
            self.signal_router_joints[i]._offsets3d = (points[5:8, 0], z_loc * np.ones(3),
    points[5:8, 1])

            # Update bifurcation joints
            self.bifurcation_joints[i]._offsets3d = (points[8:, 0], z_loc * np.ones(points.
    shape[0]-8), points[8:, 1])
            # Update self spring mechanism

            ss_xs, ss_zs, ss_ys = cell.self_spring.compute_with_leads(*points[9], z_loc, *
    points[1], z_loc)
            self_spring_lines = [[(ss_xs[j], ss_ys[j], ss_zs[j]), (ss_xs[j+1], ss_ys[j+1],
    ss_zs[j+1])] for j in range(len(ss_xs)-1)]
            self.self_spring_lines[i].set_segments(self_spring_lines)

            # Update left spring mechanism
            if cell.left_neighbor is not None:
                left_points = cell.left_neighbor.compute_points()
                ls_xs, ls_zs, ls_ys = cell.left_spring.compute_with_leads(*points[1], z_loc, *
    left_points[9], z_loc-self.cell_spacing)
                left_spring_lines = [[(ls_xs[j], ls_ys[j], ls_zs[j]), (ls_xs[j+1], ls_ys[j+1],
     ls_zs[j+1])] for j in range(len(ls_xs)-1)]
                self.left_spring_lines[i].set_segments(left_spring_lines)

            # Update right spring mechanism
```

```
            if cell.right_neighbor is not None:
                right_points = cell.right_neighbor.compute_points()
                rs_xs, rs_zs, rs_ys = cell.right_spring.compute_with_leads(*points[1], z_loc, *right_points[9], z_loc+self.cell_spacing)
                right_spring_lines = [[(rs_xs[j], rs_ys[j], rs_zs[j]), (rs_xs[j+1], rs_ys[j+1], rs_zs[j+1])] for j in range(len(rs_xs)-1)]
                self.right_spring_lines[i].set_segments(right_spring_lines)

            # Update indicator colour
            if cell.theta>0:
                plt.setp(self.indicators[i], color='w')
            else:
                plt.setp(self.indicators[i], color='k')

        return self.tristable_lines + self.signal_router_lines + self.bifurcation_lines + self.signal_spring_lines + self.tristable_joints + self.signal_router_joints + self.bifurcation_joints + self.self_spring_lines + self.left_spring_lines + self.right_spring_lines + self.indicators



    def get_configuration(self):
        thetas = [cell.theta for cell in self.unit_cells]

        return [1 if theta<0 else 0 for theta in thetas]

    def set_configuration(self, configuration):
        #check that configuration is valid
        assert len(configuration) == self.n, "Configuration must be a list of length n"
        assert all([c==0 or c==1 for c in configuration]), "Configuration must be a list of 0s and 1s"

        # print("Setting configuration to {}".format(configuration))
        for i in range(self.n):
            self.state_vector[2*i] = 0 if configuration[i] else self.params['alpha0']
        self.solve_equilibrium()

class Spring:
    """
    A class representing a spring.

    Attributes:
    -----------
    ne : int
        Number of elements in the spring.
    a : float
        Length of the spring.
    r0 : float
        Natural radius of the spring.
    Li_2 : float
        Square of the length of each element of the spring.
    ei : numpy.ndarray
        vector of longitudinal coordinates of the spring elements.
    b : numpy.ndarray
        vector of transverse coordinates of the spring elements.
    """

    def __init__(self, ne=None, a=None, r0=None):
        self.ne = ne
        self.a = a
        self.r0 = r0
        self.Li_2 = None
        self.ei = None
        self.b = None
        if all([ne, a, r0]):
            self._initialize()

    def _initialize(self):
        """
        Initializes the spring parameters.
        """
        self.Li_2 = (self.a / (4 * self.ne))**2 + self.r0**2
        self.ei = np.arange(2 * self.ne + 2)
```

```
        j = np.arange(2 * self.ne)
        self.b = np.concatenate(([0], (-1)**j, [0]))

    def compute(self, xa, ya, xb, yb):
        """
        Computes the position of the spring.

        Parameters:
        -----------
        xa : float
            x-coordinate of the starting point of the spring.
        ya : float
            y-coordinate of the starting point of the spring.
        xb : float
            x-coordinate of the ending point of the spring.
        yb : float
            y-coordinate of the ending point of the spring.

        Returns:
        --------
        xs : numpy.ndarray
            Array of x-coordinates of the spring elements.
        ys : numpy.ndarray
            Array of y-coordinates of the spring elements.
        """
        if self.ne is None or self.a is None or self.r0 is None:
            raise ValueError("Spring parameters not initialized!")

        R = np.array([xb, yb]) - np.array([xa, ya])
        mod_R = np.linalg.norm(R)
        L_2 = (mod_R / (4 * self.ne))**2

        if L_2 > self.Li_2:
            raise ValueError("Initial conditions cause pulling the spring beyond its maximum
    large. Try reducing these conditions.")
        else:
            r = np.sqrt(self.Li_2 - L_2)

        c = r * self.b
        u1 = R / mod_R
        u2 = np.array([-u1[1], u1[0]])

        xs = xa + u1[0] * (mod_R / (2 * self.ne + 1)) * self.ei + u2[0] * c
        ys = ya + u1[1] * (mod_R / (2 * self.ne + 1)) * self.ei + u2[1] * c

        return xs, ys

    def compute3d(self, xa, ya, za, xb, yb, zb):
        if self.ne is None or self.a is None or self.r0 is None:
            raise ValueError("Spring parameters not initialized!")

        R = np.array([xb, yb, zb]) - np.array([xa, ya, za])
        mod_R = np.linalg.norm(R)
        L_2 = (mod_R / (4 * self.ne))**2

        if L_2 > self.Li_2:
            raise ValueError("Initial conditions cause pulling the spring beyond its maximum
    length. Try reducing these conditions.")
        else:
            r = np.sqrt(self.Li_2 - L_2)

        u1 = R / mod_R
        # Define the yv vector perpendicular to the xz-plane
        yv = np.array([0,1,0])


        u3 = np.cross(yv, u1)
        # Compute the u3 vector which is the cross product of yv and u2
        u2 = np.cross(u1, u3)


        xs = xa + u1[0] * (mod_R / (2 * self.ne + 1)) * self.ei + u2[0] * r * self.b
        ys = ya + u1[1] * (mod_R / (2 * self.ne + 1)) * self.ei + u2[1] * r * self.b
```

```
        zs = za + u1[2] * (mod_R / (2 * self.ne + 1)) * self.ei + u2[2] * r * self.b

        return xs, ys, zs

    def initialize_leads(self, xa, ya, za, xb, yb, zb):
        # Calculate the total distance between points A and B
        R = np.array([xb, yb, zb]) - np.array([xa, ya, za])
        mod_R = np.linalg.norm(R)

        # Calculate lead length (buffer on each side)
        self.lead_length = (mod_R - self.a) / 2

        # Calculate the direction vector between points A and B
        self.u1_lead = R / mod_R

    def compute_with_leads(self, xa, ya, za, xb, yb, zb):
        R = np.array([xb, yb, zb]) - np.array([xa, ya, za])
        mod_R = np.linalg.norm(R)
        self.u1_lead = R / mod_R
        # Calculate the new start and end points for the spring using lead lengths
        spring_start = np.array([xa, ya, za]) + self.u1_lead * self.lead_length
        spring_end = np.array([xb, yb, zb]) - self.u1_lead * self.lead_length

        # Get spring points using the 3D compute method
        xs, ys, zs = self.compute3d(spring_start[0], spring_start[1], spring_start[2],
                                    spring_end[0], spring_end[1], spring_end[2])

        # Add the lead sections to the spring points
        xs = np.concatenate(([xa], xs, [xb]))
        ys = np.concatenate(([ya], ys, [yb]))
        zs = np.concatenate(([za], zs, [zb]))

        return xs, ys, zs
```

Delft University of Technology

LITERATURE RESEARCH & PROJECT PROPOSAL

ME56010-22

Bi-threshold Gates for Mechanical Logic in Intelligent Metamaterials

Authors:
Eoinlee Bley (5216737)

February 28, 2023

Abstract

This literature review and project proposal explores the emerging field of intelligent metamaterials, focusing on recent advancements in the design and fabrication of mechanical metamaterials with computational functionality. The review provides an overview of the state-of-the-art in this rapidly evolving field, including various design strategies and material systems. It also identifies gaps in the literature and proposes future research directions, such as the use of cellular automata as a basis for mechanical metamaterials.

The proposed project aims to develop a new class of intelligent metamaterials that can perform complex computations and information processing tasks in a parallel and distributed manner. The approach involves using networked multistable mechanisms interconnected logical responses to cyclic actuation, enabling the storage and manipulation of information. The proposed research will investigate the design, fabrication, and characterisation of these cellular automata-based metamaterials, by the use of bi-threshold elements to embody the logical functions. The project has the potential to significantly advance the field of intelligent metamaterials and open up new avenues for the development of advanced materials with unprecedented functionality.

Contents

1 Introduction

Metamaterials are architected periodic structures that exhibit unusual or unnatural material properties[1, 2], such as negative Poisson's ratio, negative refractive indices and more. The field of flexible mechanical metamaterials uses deformation and motion of mechanisms to create functional materials beyond the capabilities of a homogeneous continuous material[3].

Intelligent metamaterials have garnered renewed interest in recent times, with the blue-sky goal of developing materials that couple sensing, actuation, computation and communication[4]. When the material itself has information processing as a material property, truly distributed cognition and computation becomes possible. Such materials have numerous potential applications, such as morphing wing structures for aircraft that can change their shape mid-flight[5], robust robotic materials resilient to electromagnetic interference for aerospace, buildings with self-healing capabilities, or medical implants that can monitor and respond to changes in the body. Learning materials that can recognise patterns and adapt to novel situations[6] or materials congruent with the structure of the human brain are being pursued in the fields of neuromorphic metamaterials and morphological computing[7].

This literature review and project proposal explores the emerging field of intelligent metamaterials, focusing on recent advancements in the design and fabrication of mechanical metamaterials with computational functionality. A new approach of taking insights from the fields of cellular automata and threshold gates and a feasibility study in the embodiment of such a device as a research demonstrator is presented.

1.1 Problem Statement

In order to satisfy the description of an Intelligent Mechanical Metamaterial as we define them, such as system would need to fulfil the following requirements:

Tessellated Structure In order to be considered as a material and not a differentiated and integrated system of components, the metamaterial must tessellate and be complete in its tiled form. Boundary structures may be necessary but a material must be mostly homogeneous and consisting of a tiled unit cell structure.

State Information or Memory The metamaterial must be capable of encoding and storing information in a state element. This may be volatile or nonvolatile, but some aspect of the structure of the unit cell must function to remember the information it is processing.

Processing Capability To consider a metamaterial "intelligent" we require some sort of meaningful processing of the state information. Simple linear elastic deformation in response to stimulus from the environment would not suffice. Some meaningful, specifiable computation or logical 'decision making' must be granted by the structure of the unit cell.

Information Transmission Information that is processed by a single unit cell is very limited by the processing power of a single unit cell. Networking the cells to connect and process information collectively in parallel is necessary to transcend the power of a single unit cell. The processing capability of a single unit cell must be simple enough to scale down to make a macro scale material feasible. As such, the intelligence must be emergent from the complex connections of simple processing unit cells.

Mechanical While the overall field of metamaterials uses geometrical structure to define unusual, *mechanical* metamaterials specifically exploit motion, deformation, instabilities and elastic non-linearities to embody more exotic properties. As such, the scope of this project will be limited to this branch of metamaterials.

2 State of the Art

2.1 Search Approach & Strategy

Based on the requirements of a feasible intelligent metamaterial outlined in 1.1, a search strategy was developed. Each requirement has a area of literature that maps naturally to it. The literature search in each of these areas was seeded by widely cited review papers in each of those fields. From these seed papers, two strategies were taken to explore the literature further.

Keywords Keywords that were widely used and relevant to the goals of the project were documented and used to perform systematic searches on Google Scholar, ResearchGate and ScienceDirect.

Citation Network The online citation network visualiser CitationGecko[8] was used to explore and analyze citation networks based on the seed papers and subsequent bibliography. It works by importing a bibliography in BibTeX format and then retrieving citation data from Google Scholar. Once the citation data is retrieved, Citation Gecko constructs and visualises the citation network, where papers are represented as nodes and citations as directed edges between the nodes, as shown in

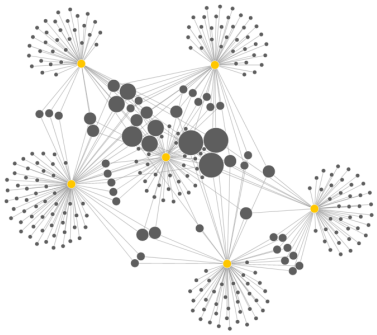


Figure 1: Citation Gecko example bibliography network. Yellow nodes are seed papers and grey nodes are citations. Larger grey nodes are cited by more seed papers.

2.2 Adaptable Mechanical Metamaterials

The tessellation requirement of course maps naturally to the field of flexible mechanical metamaterials. A recent comprehensive review of this area of the literature was conducted by Bertoldi et al[3]. A key insight from this paper that defines the fundamental reason that architecting materials leads to new properties is that *the heterogeneity of the structure breaks down the affine assumption.* In other words, designing geometric non-linearity into the structure of each component unit cell leads to non-linear elastic behaviour of the overall metamaterial. They also emphasise the potential of elastic instabilities (e.g. buckling or snapping) and large deformations for creating highly nonlinear behaviour. The outlook presented by this review concludes with several encouraging recommendations:

Complex Energy Landscapes Exploration of complex energy landscapes and frustrated materials and their rational design could allow for information storage and retrieval, and could be used to create truly useful robotic structures.

Actuation Strategies "The deep integration of actuation and the amplification of mechanical information are crucial to overcome the inevitable dissipative processes, and if combined with information processing (for example, using logic gates) would open the door to truly smart metamaterials"

Rational Design Challenges The review ends with a comment and challenge: "the rational design of metamaterials with a target property or functionality remains fiendishly difficult, and many designs so far have relied on luck and intuition." It lists many serendipitous discoveries that can be leveraged to create intelligent metamaterials, such as origami, but that rational first principles design is an unsolved problem and novel approaches are needed.

2.2.1 Rational Design

One example of such a "lucky" or intuitive framework for the rational creation of mechanical metamaterials that has recently been the focus of a section of the field is the ancient art of origami. Origami, and its less well known counterpart kirigami (which permits cutting of the sheet) are particularly interesting as they promise lamina-emergent structures[9]. Lamina emergent mechanisms are inherently easy to manufacture as they are 2D and non-subtractive. They are therefore easy to scale down in size (graphene kirigami[10]) and scale up in production volume.
Particularly the Miura-Ori pleat fold offers many avenues for exploration due to its multistable unit cells[11, 12]. However, I believe the lack of simple models of these structures and complexity of the lamina mechanics involved makes designing functional intelligent metamaterials with this approach intractable in the near future. The closest attempt has been in the recent work of Liu et al. with the exploitation of stacked Miura-ori sheets non-linear transition between stable states for the realisation of mechanical logic[13]. By designating a unique state to a permutation of states of three elements within each unit cell, and designing the transition map between those states by changing design parameters. This resulted in the ability to create simple logic gates, though through a fairly convoluted encoding-decoding procedure, as shown in Figure 2. Origami, while overall promising, does not seem to be a viable or pragmatic approach given the scope of this project.

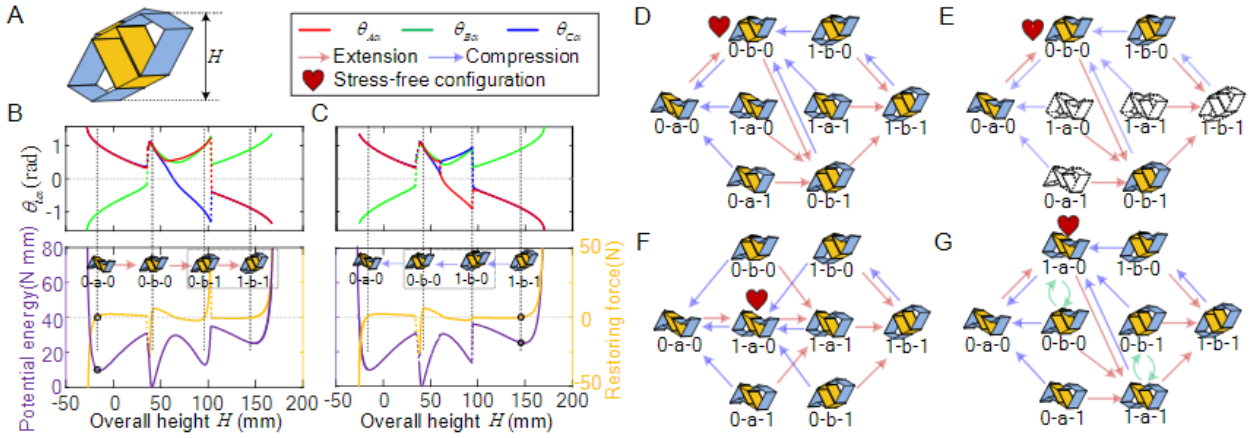


Figure 2: Architecture of logic gates (A) with and (B) without transition procedures. (C) Comparison to electronic logic circuits and (D) the completeness of the approach to embodying all of the 2^8 3-input Boolean functions.

2.2.2 Actuation Strategies

Different authors have taken different approaches to the external actuation of adaptable metamaterials. For the sake of cost and complexity, it is often desirable to have fewer and simpler actuators if possible. The actuators may also serve different roles in the functional control of the metamaterial. One approach was to embed piezoelectric actuators at the connection points of the unit cells[14]. In this work the metamaterial exhibits controllable shape morphing capabilities my the mechanical amplification of piezo-actuators. This allowed active tuning of the shape and properties of the material. One potential downside of this approach is the need for many individual actuators, and a bus and centralised controller, power supply and other auxiliary electronics.
Similarly, active locking and unlocking of electromagnetic latches was used as an actuation strategy[15]. This allowed granular control over the overall stiffness and Poisson's ratio of the metamaterial. While this strategy allowed more dramatic control over the materials properties, it also required dramatically more power, as well as an inability to scale down, as the power and efficiency of electromagnetic actuators diminishes rapidly as the dimensions decrease.
Other authors used active electromagnetic actuators at the links of the metamaterials to determine the stiffness of each joint such that it would have a programmable deformation shape[16]. They combined this with offline machine learning to find the stiffness network that would produce a desired shape morphing behaviour. However, all of these approaches of active actuators show promise for reprogrammable metamaterials, they do not do anything to imbue inherent intelligence or informational processing properties to metamaterials, as they all require external electronic computers to provide the intelligence.
A more global actuation approach was taken in the work of Van Hecke et al. on the effect of lateral confinement

on "holey sheet" metamaterials. In this approach, an elastomeric sheet with an architected periodic pattern of circular holes. They show that the lateral confinement of these sheets determines their nonlinear uniaxial compression response due to unstable collapse of the holes. They show that hysteretic behaviour can be generated in this manner. Hysteresis and multistability are necessary requirements of an inherently logical metamaterial. They develop so called hysterons that had transition pathways that could modify based on a skew in the loading[17, 18]. The "holey sheet" hysteron is shown in Figure 3b. This approach is similar to the work of Liu et al. in subsubsection 2.2.1, and aims to create logic systems in elastomeric metamaterials. Likewise, this approach is promising but intractable in the scope of this project.

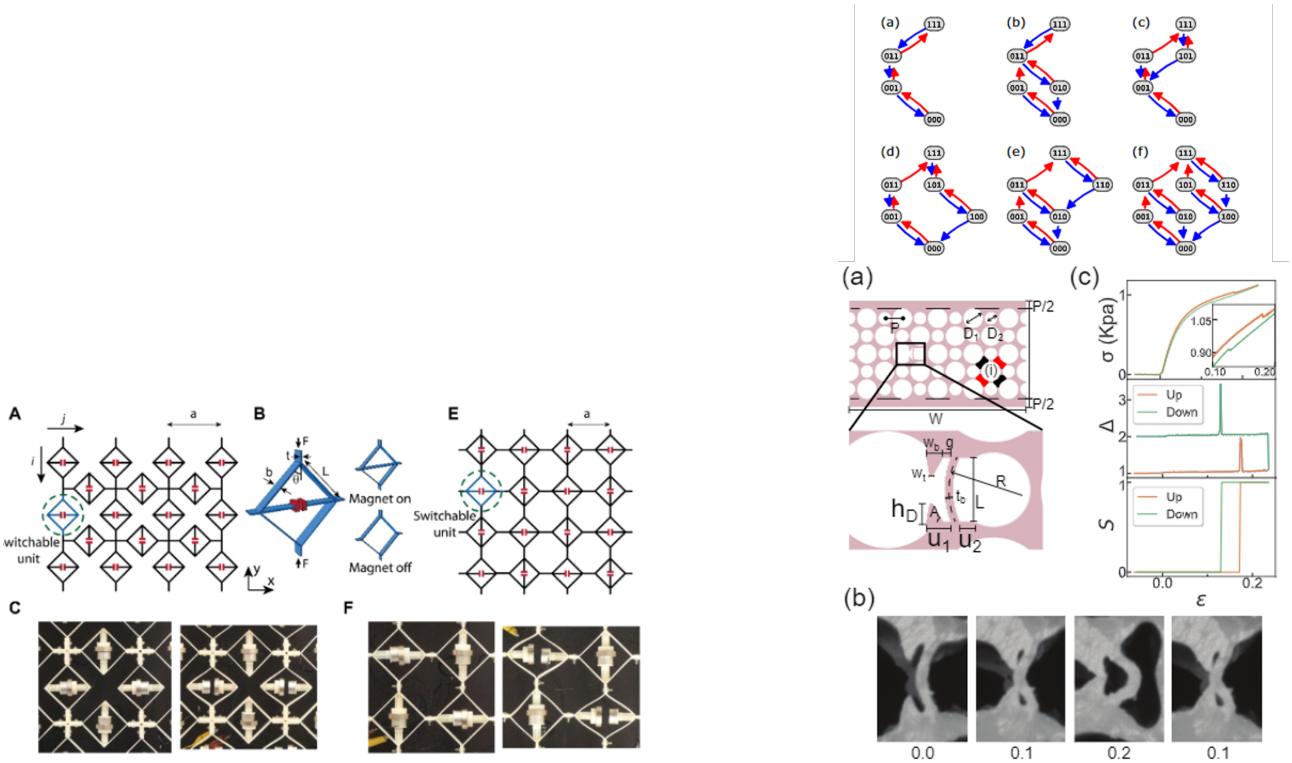
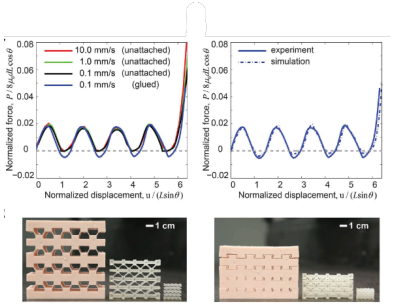


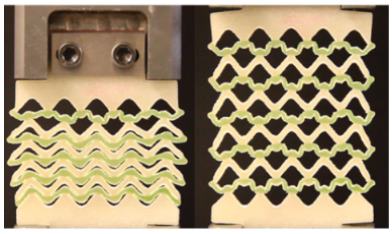
Figure 3: (a) electromagnetic locks that restructure the material by their configuratio (b) hysterons in holey elastomeric sheet metamaterial

2.2.3 Energy Landscaping

Intentional design of complex, highly nonlinear, and multistable energy landscapes could lead to useful, dense mechanical information storage and processing capabilities. Local minima in the landscape could represent states, and the directed transition flow between these states could be used to implement logical and computational processes. The majority of researchers have focused on the energy storage (or trapping) capabilities of multistable metamaterials[19, 20] without acknowledging the potential applicability of these arrays of bistable elements to intelligent metamaterials. Th They are primarily focused on energy disspation applications, observable by a sawtooth force-displacement response as in Figure 4a. However, Restrepo et al. introduce the concept of "Phase transforming cellular material" (Figure 4) and suggest that programmable structures where each phase of the material had a distinct property could be used for wave guiding, filtering and shape morphing[21]. There has also been work done to vary the geometric parameters of the bistable elements in the material to ensure the deformation sequence of the layers of the material is deterministic. Nevertheless, this approach taken by many in the field is, in my view, insufficiently developed to be fruitful within the scope of the project.



(a) Multistable force-displacement plot. Resting and compressed configurations



(b) Tension based multistability where the compressed configuration is the resting state.

(c) Multi phase structure

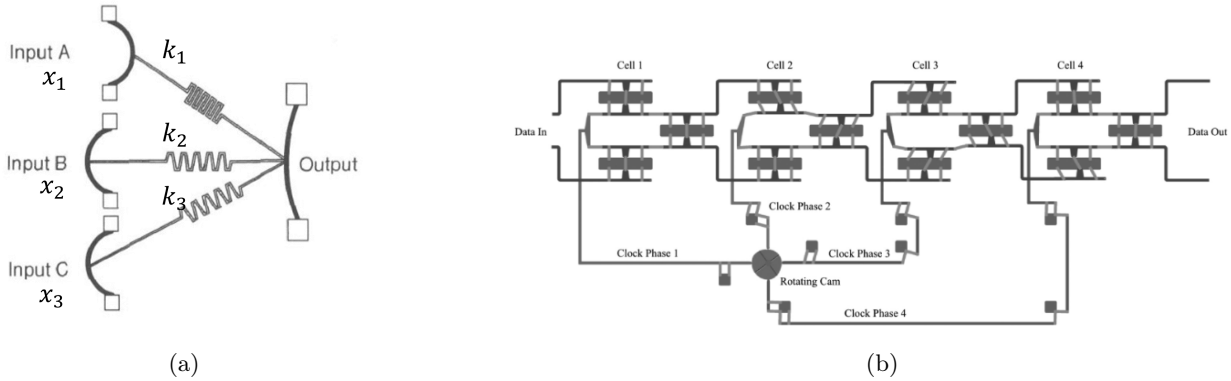
Figure 4

2.3 Mechanical Computing

The requirement of information processing for an intelligent metamaterial requires the exploration of another area of literature: mechanical computing. A 2021 perspective on the field[22] was the seed paper for the literature search. Mechanical computers are an ancient technology, from the Antikythera mechanism that made astronomical predictions in ancient Greece, to the designs of Charles Babbage's first automatic computer. According to the authors there has been a resurgence and renewed interest in mechanical computation since the domination of silicon semiconductor microelectronics and the Von Neumann architecture. They owe this emergence to advances in additive manufacturing, materials sciences, and the approaching limits of the current paradigm. They say that this problem poses challenges that require new theoretical and practical tools in various fields, including materials science, information theory, computer science, additive manufacturing, and robotics. They suggest that solutions to these challenges are likely to be found at the intersections of these fields.

Roukes[23] suggests that opportunity lies at the nanoscale, and that nanoelectromechanical systems (NEMS) may form a new paradigm once the limits of electronic devices is reached. He proposes that any realistic form of mechanical computing must use ultra-low dissipative connections between elements, i.e. compliant joints. Any losses to friction would be unacceptable at the nanoscale. To overcome the fan-out problem, the interconnection should ideally have the possibility of "gain" in the sense that external energy reservoirs, such as the elastic energy of springs, can be used to continually recharge the system by external sources. However, he does not provide concrete recommendations for implementing these systems, just that these are the requirements of a realistic, feasible system.

The inventor of public key encryption, Merkle, proposes two systems by which mechanical computing can not only be low power, but actually reversible[24]. A fundamental limitation of electronic computing is that logic gates creates entropy when information is destroyed by an irreversible logic operation. He proposes two systems: rod logic and buckled beam logic as shown in Figure 13a, for methods of computation that do not require inherent dissipation. These two proposed schemes show up in almost every subsequent paper on mechanical computation, regardless if their goal is to be reversible. Merkle himself published a study developing mechanical computing using only mechanical links and rotary joints[25]. In it they develop a theoretically Turing-complete computer using combinatorial logic, an example of the device is shown in Figure 13b. According to the authors, it is to date the simplest Turing complete mechanical computer. However, its architecture of combinatorial logic and sequential programming are incompatible with the requirements set out to embody an intelligent metamaterial, though its principles and base mechanisms could be used as constitutive elements. The following section goes into detail on current state of the art of development of mechanical Boolean elements



(a) (b)

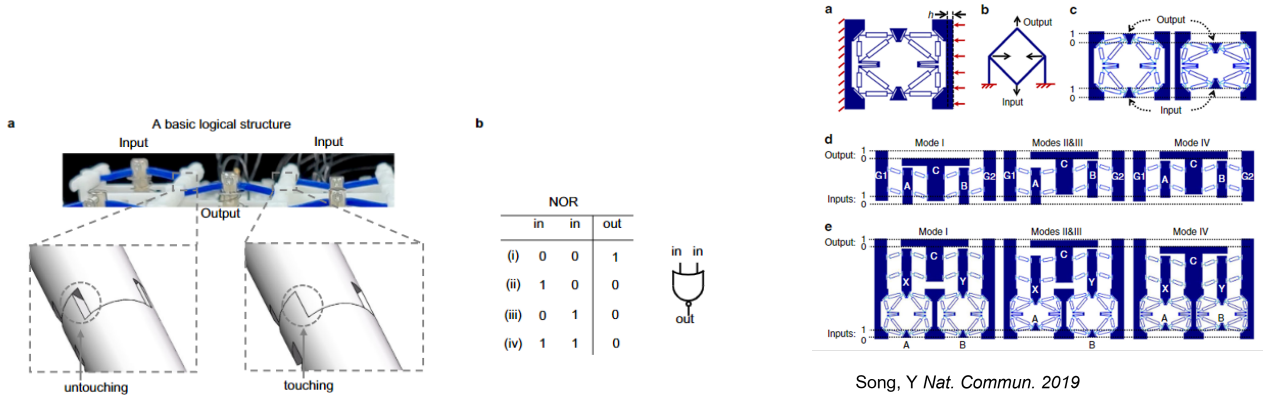
Figure 5: (a) buckled beam majority voter concept (b) bit-shift register embodied with links and rotary joints

2.3.1 Boolean Logic

Many researchers have explored the implementation of Boolean logic in mechanical metamaterials. Boolean logic is a fundamental approach to computation, describing the operations of logical AND, OR, and NOT on binary values of 0 and 1. One concrete attempt to construct a mechanical metamaterial with the rod logic from Merkle[24] used 3-D printed bistable "bit cells" to propagate information by mechanical wave propagation. Rod logic elements implemented a simple AND gate that locked or unlocked a compliant door lock as a demonstration. Their approach focused on the mechanical signal transmission, and did not go very far into the logical function embodiment. Additionally, their approach also required resetting every time the signal fired, as energy is dissipated each time. And while their resultant device is cellular, it is specifically designed for a single purpose, so would more accurately be classified as a machine than a material, a classification that the authors themselves prefer, despite the title of the paper. Other works have done similar studies into signal propagation and simple linear logic functions[26, 27], addressing many issues with stable propagation and managing the energy landscape. However, the logical functionality seems like an afterthought in most papers and the low density of processing capability with this approach makes it unfeasible for intellegent metamaterials.
Mei et al. focused on creating a reprogrammable mechanical metamaterial (ReMM) capable of universal combinatorial and sequential logic, which has a multifunctional, programmable structure that can realise both NOR and NAND gates (necessary for more complex logical architectures. The ReMM can be reset and reprogrammed, and can also show signal transmission and bifurcation, and storage of information. The paper also demonstrates more complex logical functionality, such as a half-adder, crossover and S-R latch. According to the authors:

> the ReMM is expected to serve as a platform for constructing reusable, multi-functional, and reprogrammable robotic material with robust sensing-analyzing-response function, which can benefit the development of mechanical systems with embedded intelligence.

The authors accomplish this with simple bistable beams, and contact based logic. It is a very recent paper and further research is needed to show the scalability of this approach. As will be further discussed later, this architecture can be viewed as a type of cellular automata, where the state of each buckled beam is a function of the states of its predecessors, subject to sequential excitation, or "clocking". This idea is explored more fully in section 3.



(a) (b)

Figure 6: (a)NOR gate implemented with bistable beams with revolute joint and contact based system. (b) Planar compliant mechanical logic gates

Several other authors have designed similarly complex logical functionality with different approaches. Song et al. designed a planar system that enables micro-mechanical logic gates to be manufactures with micro-additive technologies[28]. NOT, AND, OR, NOR and NAND gates were made using a system of planar buckled multi-stable mechanisms as shown in Figure 6b. The proposed benefits of this approach is that they can be integrated scaled down into micro-architected metamaterials. The compliant nature of the elements eliminates friction and allows for scalability. The paper only goes as far as constructing the gates and does not demonstrate integration into an interconnected system. I believe this approach has merit and should be expanded into a periodic system that can exhibit complex computation. Another work that takes another approach altogether is the doctoral thesis of M. Waheed[29], which gives a very comprehensive overview of the field of functional metamaterials. Logical building blocks are here build using 3-dimensional building blocks, as an effort to remove the limits of planar mechanisms, and create true 3-D materials. He also uses the rod logic system of Merkle, and creates some basic building blocks for logical systems using biased Von Mises trusses, basic bi-stable elements. However, this work leans on multi-material additive manufacturing technologies and does not seem scalable for microscale applications.

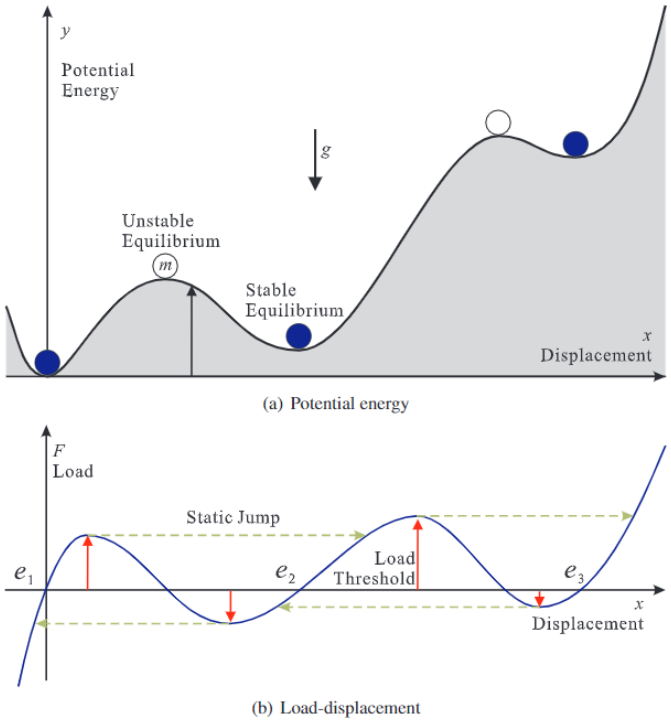
This review of mechanical computation strategies shows that multi-stability is one key aspect to generating both the nonlinear response necessary for embodying most logical functions, but also the storage and memory of state information. The next section reviews the state of the art and literature on compliant multi-stable mechanisms.

2.4 Multi-stable Compliant Mechanisms

This section aims to review the current literature on multi-stable compliant mechanisms, discussing the various design approaches, modelling techniques, and applications of these mechanisms.

Overview A 2008 doctoral thesis by Y. Oh gives a great overview of the field and foundational mathematics and mechanics necessary to synthesise multistable compliant mechanisms[30]. Two main approaches are covered, taking advantage of the instability of buckled configurations and of the bi-stability of a clamped-pinned beam. A particularly useful insight provided by this work is the canonical forms of the energy landscape and force-displacement graph of a tristable mechanism, as shown below in 7. The feasibility study in section 4 shows how this profile was achieved with a new tristable mechanism design.

Contact-based approaches Another approach to bi-stability is by use of contact-based approaches[31]. This design used a latch-lock mechanism to store the state information. Many MEMS devices use this approach but it is not practical for large periodic metamaterials. Contact based mechanisms exhibit fundamental energy loss and wear, as they rely on friction and sliding contact. Other works use magnetic systems instead of elastic materials to create multistable structures[32]. This approach suffers from the same scaling laws that make the electromagnetic active metamaterials did in the previous section. Contact also remains a problem, and the construction of such a unit cell would require assembly, impractical for large scale metamaterials.



H

Figure 7: Potential energy and load-displacement curves of a multistable system.

Environment Reactive While most of the literature in this area focuses on purely mechanically triggered bi-stability, one work developed a bistable unit element whose bistability was dependent on external chemical factors[33]. They used an absorbent polymer to create a buckled beam element that loses bistability once a sufficient quantity of liquid is absorbed from the environment. While the mechanics in this paper are fairly rudimentary, it does highlight the potential of metamaterials to be actuated and adapted to more than simple mechanical stimulus.

Binary Mechanical Properties Translating the displacement of a bi-stable element into tangible properties of the metamaterial has not been fully addressed in most works so far. Most papers are satisfied that the information can be "encoded" in the states of bi-stable elements, without regard for the "decoding" of utilisation of the state for realisation of novel properties. The work of Kuppens et al. develops an element wherein the stiffness is toggled between stiff and near-zero stiffness[34]. It does this by a balanced parallel arrangement of positive and negative stiffness elements. The negative stiffness element is toggled by the buckling of a beam. Recent work [35] used these elements to create a rudimentary metamaterial that learns. It learns a specific arrangement of lattice stiffnesses to attain a desired deformation under a given load. This work is still in early stages but is very promising outlook. Currently offline learning is used to determine the stiffnesses, but if onboard mechanical learning and information processing is built in, this would be a good first step towards realising functional intelligent metamaterials.

Pre-curved Beams The planar logic gates mentioned previously[28] and many other multistable elements that rely on buckling, require external actuation after manufacture. While this is unavoidable if a symmetric energy landscape is required, a relaxation of this requirement allows for the use of pre-curved beams[36] which are an inherently much more manufacturable and scalable solution. They do suffer the drawback of reduced bi-stability, and range of motion.

2.4.1 Tri-stable and multistable mechanisms

While this review has thus far focused on bistable elements, similar logical capabilities are also found in higher orders of multi-stability. A recent work by Zhang et al. that used kirigami techniques to make a highly non-linear spring created a tristable metamaterial structure capable of ternary logic operations, signal filtering and

more[37]. The work, illustrated in Figure 8 is recent and shows promise, as the metamaterial is programmable and capable of relatively complex computation, with a large number of possible states. However, the manufacturing techniques used are not inherently scalable at they use multimaterial 3-D printing, and kirigami- contact based nonlinear springs. This approach adapted to overcome these shortcomings would show great promise in accomplishing an intelligent metamaterial.

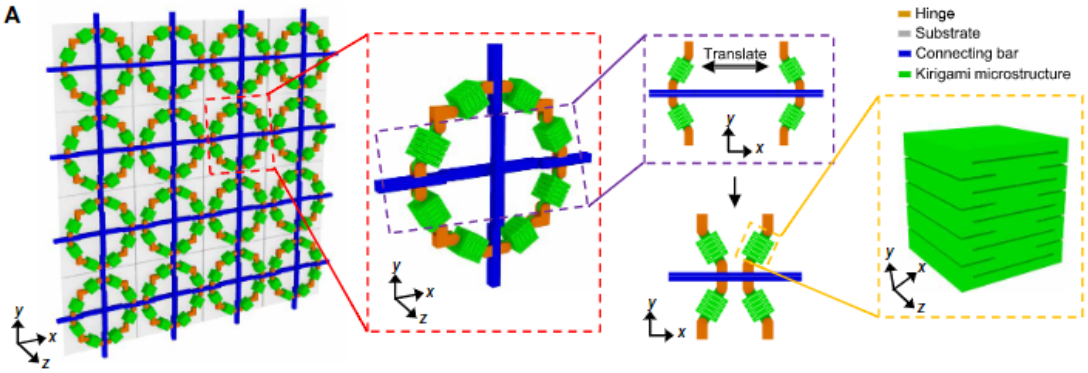


Figure 8: Schematic representaion of the kirigami based tri-stable mechanism and resultant metamaterial.

Orthogonal beam tri-stable mechanism The approach proposed by G. Chen et al. is a promising solution for achieving higher orders of multistability using a single bistable beam. By adding orthogonal bistable mechanisms in series with the original beam, a higher number of stable positions can be achieved, as shown in Figure 9. Importantly, this technique is both compact and does not require contact, making it highly desirable for many applications. Chen et al. demonstrated the feasibility of this approach for manufacturing at several length scales, suggesting that it has broad potential for use in a variety of settings. Notably, this approach also allows for the creation of multistable mechanisms with a desired number of stable positions, depending on the design requirements.

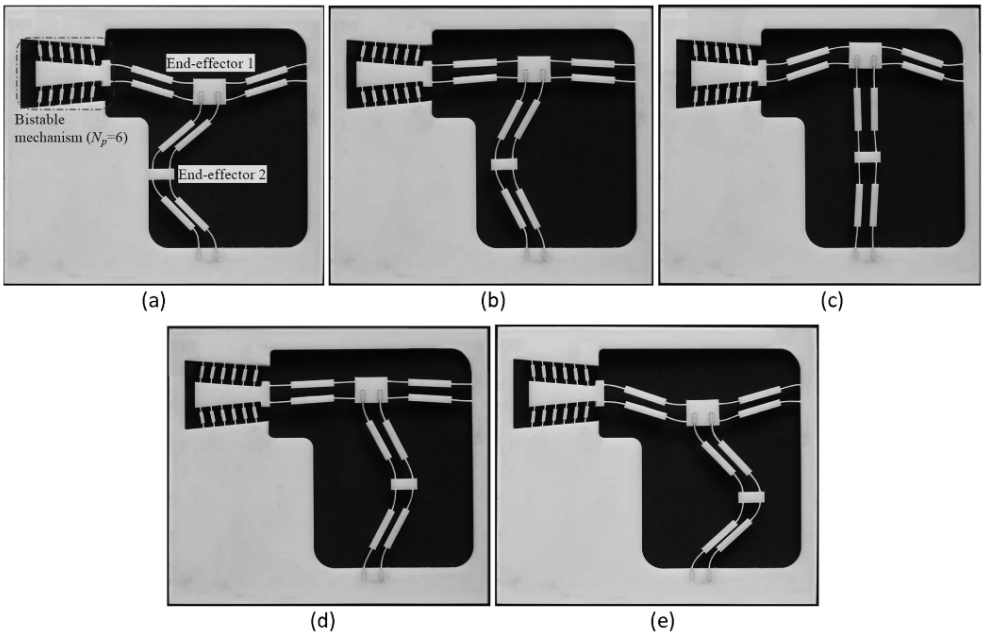


Figure 9: Multistability achieved by single bistable beam chained with successive orthogonal mechanisms.

Parallel Interconnected Bistable Mechanisms One of the most promising architectures for intelligent metamaterials takes the approach of simply arraying bistable beams in parallel[38]. This recent work by Kwakernaak et al. accomplishes counting and sequential information processing by architecting the structure and interconnections to create an irreversible response to cyclic actuation, as shown in Figure 10. They store information in the orientation of the buckled beams and create logical sequences by having beams transition states

depending on its neghbours on each clocking cycle. They conclude the paper with the insight that this approach when expanded opens up

> "routes to create systems that are Turing-complete, such as 'rule 110'or Conway's game of life. Such 'cellular automata materials' would allow massively parallel computations *in materia*."

. This approach I believe is very promising, and addresses issues faced by many of the works presented in this review. In the following sections I will present an introduction and exploration into cellular automata as a basis for mechanical metamaterials, and the necessary elements to embody them.

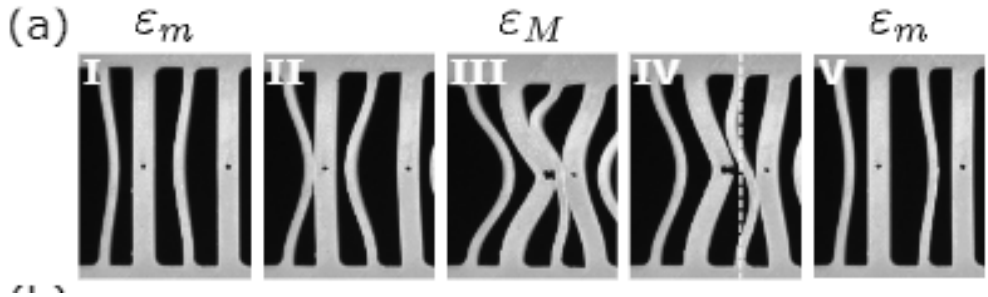


Figure 10: Evolution of the unit cell of counting metamaterial using simple cellular automata rules.

2.5 Gap of knowledge/ Research Question

Based on the literature review, a research question that arises is: How can the principles of cellular automata be used to design mechanical metamaterials with massively parallel computation capabilities?
This question aims to explore the potential of using cellular automata as a basis for mechanical metamaterials and how this approach could lead to the development of systems that are Turing-complete, such as rule 110 or Conway's Game of Life, allowing for massively parallel computations in materia. This approach could address the limitations of traditional logical and system architectures and open up new routes for the design of intelligent metamaterials and programmable matter[39]. These topics are further explored in the following sections.

3 Cellular Automata

This section introduces and explores the concept of cellular automata, for the purposes of exploring a new approach to the development of intelligent metamaterials. As mentioned in subsection 2.3,this approach has been suggested in literature before, though not explored in detail. Most of the information in this section has been adapted from the textbook Cellular Automata: A Discrete Universe[40].

3.1 Introduction to Cellular Automata

Cellular automata (CA) are mathematical models for simulating complex systems using a grid of discrete spatial cells that are updated in discrete time steps according to a set of rules that determine the state of each cell based on the states of its neighbouring cells.
The grid of cells is usually arranged in a regular lattice, such as a two-dimensional square or hexagonal grid, but can also be in just one or more dimensions. Each cell in the grid can be in one of a finite number of states, though most commonly a binary set such as "on" or "off", "alive" or "dead", or "black" or "white". The state of each cell is updated at each time step based on the states of its neighbouring cells, which can be defined using a neighborhood function that specifies which cells are considered neighbors of a given cell. The update rule for each cell is typically a deterministic function that maps the states of its neighboring cells to a new state for the cell itself. In summary, the key characteristics of a CA are as follows:

Discrete lattice of cells: The system substrate is composed of a one-, two- or three-dimensional lattice of cells.

Homogeneity: All cells within the lattice are equivalent.

Discrete states: Each cell is capable of taking on one of a finite number of possible discrete states.

Local interactions: Each cell is only able to interact with cells that are located within its local neighborhood.

Discrete dynamics: At each discrete unit time, each cell updates its current state according to a transition rule that takes into account the states of cells in its neighborhood.

It is clear that these characteristics map very clearly onto the requirements laid out in 1.1 and explored in Section 2.
Below, two examples of different types of CAs are detailed as examples. They are then explored as potential models for novel implementations of intelligent metamaterials.

3.2 Conway's Game of Life

John Conway's Game of Life is the most famous example of a cellular automaton. it was published in 1970 as an attempt to simplify Von Neumann's CA. It consists of a two-dimensional grid of cells, where each cell can be in one of two states, either alive or dead.

3.2.1 Rules of Life

The game begins with an initial configuration of living cells on the grid. At each time step of the game, the state of each cell is updated based on the states of its eight neighbouring cells. The update rules are based on a simple set of three conditions:

- Any live cell with fewer than two live neighbours dies, as if by underpopulation.
- Any live cell with two or three live neighbours lives on to the next generation.
- Any live cell with more than three live neighbours dies, as if by overpopulation.
- Any dead cell with exactly three live neighbours becomes a live cell, as if by reproduction.

The Game of Life, or just Life, is an example of an emergent complex and self-organising system. From an initial state of random noise, many patterns and structures emerge over time. Unchanging static patterns (also called "still lifes"), and periodically repeating structures are the simplest terminating configurations of game.
However, even given the simplest of rules it is impossible to determine whether any given initial configuration will grow indefinitely or eventually terminate without directly simulating it.

3.3 Elementary Cellular Automata

3.3.1 Introduction to Elementary Cellular Automata

Elementary Cellular Automata (ECA) are a class of one-dimensional cellular automata where each cell can have only two possible states, usually labelled 0 and 1, and the updating state of each cell depends only on its own state and the state of its immediate neighbours. ECAs have been extensively studied since the 1980s, especially by Stephen Wolfram. They have proven to be a rich source of complex and interesting behavior, despite their simple rules. In his book "A New Kind of Science,"[41] Wolfram argues that ECAs are a fundamental model of computation and that they can be used to explain many natural phenomena.

3.3.2 Overview of 1D Cellular Automata

The cells are arranged in a line or lattice, and each cell has a finite number of possible states. At each time step, the state of each cell is updated based on a fixed rule, which is defined as a 3-input Boolean function. The rules are often given in the form of a lookup table or transition function, which specifies the next state of a cell based on its current state and the states of its neighbours. 1D CA have been used to model a wide variety of physical, biological, and social phenomena, and they have been studied extensively in the fields of mathematics, physics, and computer science. They are particularly useful for modelling systems that exhibit self-organisation, pattern formation, and emergent behaviour.

3.3.3 Explanation of the Wolfram Code

The Wolfram code is a method of encoding the rules for an ECA using a binary number. Each of the 256 possible rules for an ECA is assigned a unique code, which corresponds to a binary number between 0 and 255. The code is obtained by interpreting the eight possible configurations of a cell and its two neighbours as binary digits, and then converting the resulting binary number to decimal. The digits are ordered from left to right, with the first digit representing the neighbourhood where all three cells are alive, and the last digit representing the neighbourhood where all three cells are dead. The resulting binary number corresponds to one of 256 possible ECA rules, each with its own distinct behaviour and pattern formation. The Wolfram code has the advantage of being simple and compact, and it allows for easy comparison and analysis of different ECAs. It also highlights the fact that even though ECAs have simple rules, they can exhibit a wide variety of complex and interesting behaviours, as shown in Figure 11.

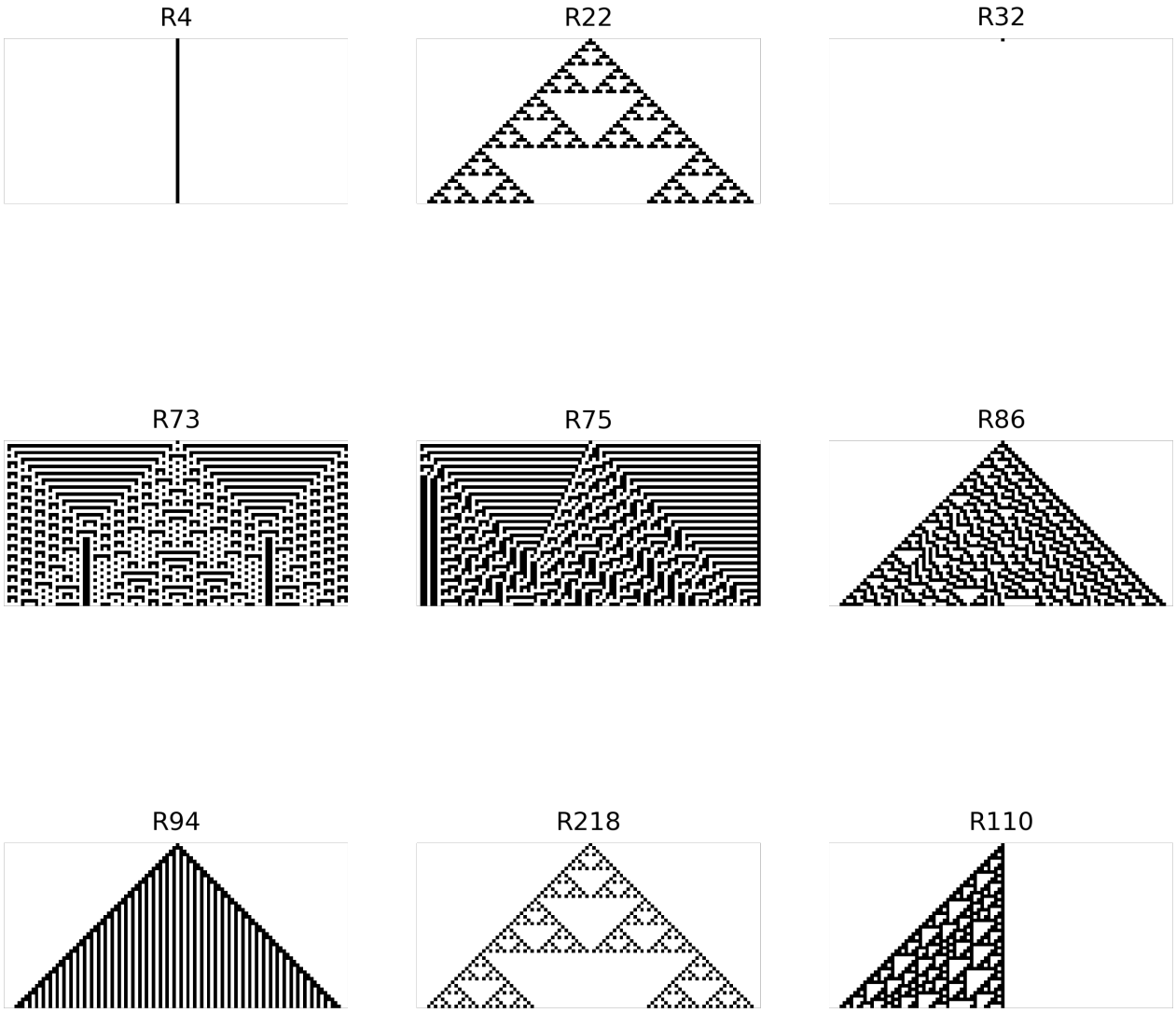


Figure 11: Various ECAs and their evolution after a single point is turned on. Complex emergent patterns and chaos can be observed from very simple rules

The properties and behaviour of ECAs depend on the specific rule that is being used. Some ECAs exhibit regular and repetitive behaviour, such as simple alternating patterns or periodic structures, while others exhibit complex and unpredictable behaviour, such as chaotic patterns or complex structures that evolve over time.

3.3.4 Mapping of ECAs to Boolean functions

The mapping of the transition function of an ECA to a 3-input Boolean function can be done by considering the state of each cell and its two neighbouring cells as the three inputs to the Boolean function. The output of the function corresponds to the state of the cell in the next generation of the ECA.

3.4 Computation in CA

Cellular automata and related architectures have been extensively studied in natural sciences, mathematics, and computer science because they have the potential to perform complex computations efficiently and robustly, as well as model the behaviour of complex systems in nature. They have been used as models of physical and biological phenomena, such as fluid flow, galaxy formation, earthquakes, and biological pattern formation. They have also been studied as mathematical objects with formal properties that can be proven, and used as parallel computing devices for the high-speed simulation of scientific models and computational tasks such as image processing. Additionally, cellular automata have been used as abstract models to study emergent cooperative or collective behaviour in complex systems[42]. During the 1980s and 1990s, there was a concerted effort to

implement large scale supercomputers built on massively parallel cellular automata hardware architectures[43]. The aim was to implement "programmable matter" that could be used to simulate an arbitrary 3-D material that could be modelled as simple interactions between cells, such as a lattice gas model[39]. The result was the CAM-8 computer, in the authors words the:

> "product of over a decade of Cellular Automata (CA) machine and modelling research by the Information Mechanics group at the MIT Lab for Computer Science. CAM8 is a parallel, uniform, scalable architecture offering unprecedented performance in the fine-grained modelling of spatially-extended systems. It provides a general-purpose instrument for the systematic exploration of a new band of the computational spectrum."

Unfortunately, the work on CAM-8 ended in 2001 due to setbacks, but showed great promise and the concept of programmable matter has since been picked up by researchers in many fields, from synthetic biology to metamaterials.
Both the Game of Life and ECAs have been shown to be capable of universal computation, meaning they both can theoretically compute anything that a Turing machine can compute. Rule 110 has long been shown to be capable of universal computation, by the use of an architecture called the cyclic tag system[44, 45]. John Conway proved in 1982 that the Game of Life can implement a universal computer and a universal constructor. In the meantime there has been a a lot of work done by enthusiasts to develop more and more advanced computers within life itself. Most recently in 2021, an improved, scalable version of an 8-bit programmable computer pattern was announced[46]. However, in some ways the goal of reproducing universal serial computation from a massively parallel architecture is a flawed approach from the start. It is a strong but flawed instinct to use a novel architecture to simply implement an already well known system. Others have tried to implement a unique method of computation using elementary cellular automata directly. A particle model was used to represent bits and processing instructions, which collide and output the result as more particles. This allowed for massively parallel arithmetic to be implemented directly in one-dimensional cellular automata.[47]. In this case, the single dimension was a constraint on the system that could be circumvented by higher dimensionality for greater functionality.Nevertheless, this work demonstrated the potential of very simple systems with clever use of their emergent properties.
More recently the focus has moved past the direct implementation of CAs to neuromorphic computing morphologies and cellular neural networks (CNNs). CNNs are similar to CAs, with the same spatial structures and connectivity, but operate as first-order dynamical systems, instead of discrete time "clocked" systems. One work demonstrates the capacity of these systems to implement arbitrary Boolean functions by the tuning of the weights of the transition function and connectivities[48].More modern trends in this field include memristor-based neuromorphic architectures for Cellular Neural Networks[49].However, this topic branches away from metamaterial implementations and is thus beyond the scope of this project, but it does demonstrate the future potential of this architecture in materials.

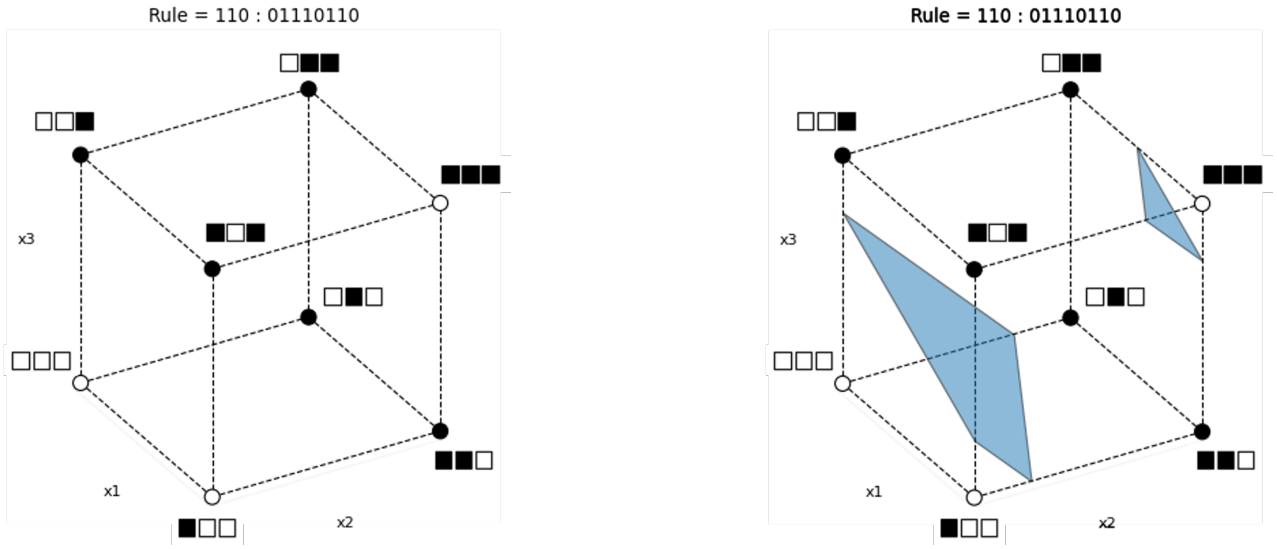
3.5 ECA as Boolean functions

This section has elucidated the potential of the cellular automata architechture for useful computational systems that also fulfil the requirements set out in Section 1.1. The question remains; what type of logical element could be implemented mechanically in order to embody a cellular automata into a metamaterial? To answer this question it is useful to make the observation that the transition rules for ECAs are simply 3-input, or "ternary" Boolean functions, one input for the state of each neighbour and the state of the cell itself. As we have shown in Section 2.4, tristable mechanisms have previously been used to implement ternary logic[37, 13] using origami and kirigami principles.

Figure 12: Caption

3.5.1 Ternary Boolean functions as unit cubes

While ternary Boolean functions can be represented as truth table, a more visual and useful representation is as a mapping from the unit cube in 3-dimensional space (where each axis represents an input variable) to the set 0, 1, which corresponds to the output of the function. The unit cube has eight corners, each corresponding to a unique combination of inputs, and each of these corners can be assigned a value of 0 or 1, depending on the output of the ternary Boolean function for that particular combination of inputs.
By assigning a colour (e.g. black for 1 and white for 0) to each corner of the cube, a visual representation of the Boolean function can be obtained. Each point within the cube corresponds to a unique combination of input variables, and the value of the Boolean function for that combination can be determined by looking at the colour of the corresponding corner.



(a) (b)

Figure 13: (a) hypercube representation of Rule 110 ternary function (b)corresponding bi-threshold representation of the function.

3.5.2 Linear Separation

The benefit of this abstraction is that we can see visually that certain Boolean functions have the property known as *linear separability*. This means a single hyperplane defined by an orientation and offset can be used to separate the regions where the output is true or false. This plane actually represents a so called "threshold function", or "majority decision function", where the orientation of the plane represents the "weights" of the input variables, and the offset is a threshold on the weighted sum of the inputs beyond which the output is true. In fact, 104 of the 256 ternary Boolean functions are linearly separable by so called "threshold functions"[50]. It is possible to capture 230 out of the 256 functions if we allow *two* thresholds, corresponding to a second parallel plane. This fact is illustrated later in Figure 17. The next section will go into more detail on the theory of these majority decision/threshold functions, their applications and prospects for a valid approach to building logic and intelligence into mechanical metamaterials.

3.5.3 Game of Life Rules as Bi-threshold

Additionally, the rules of Game of Life can also be seen as a weighted sum with two thresholds. If all of the states of the neighbourhood of a central cell are weighted with a value of 1, and the central cell's state is weighted 0.5, then any sum between 2.5 and 3.5 will result in a cell being alive the next generation. This representation is shown in Figure 14.

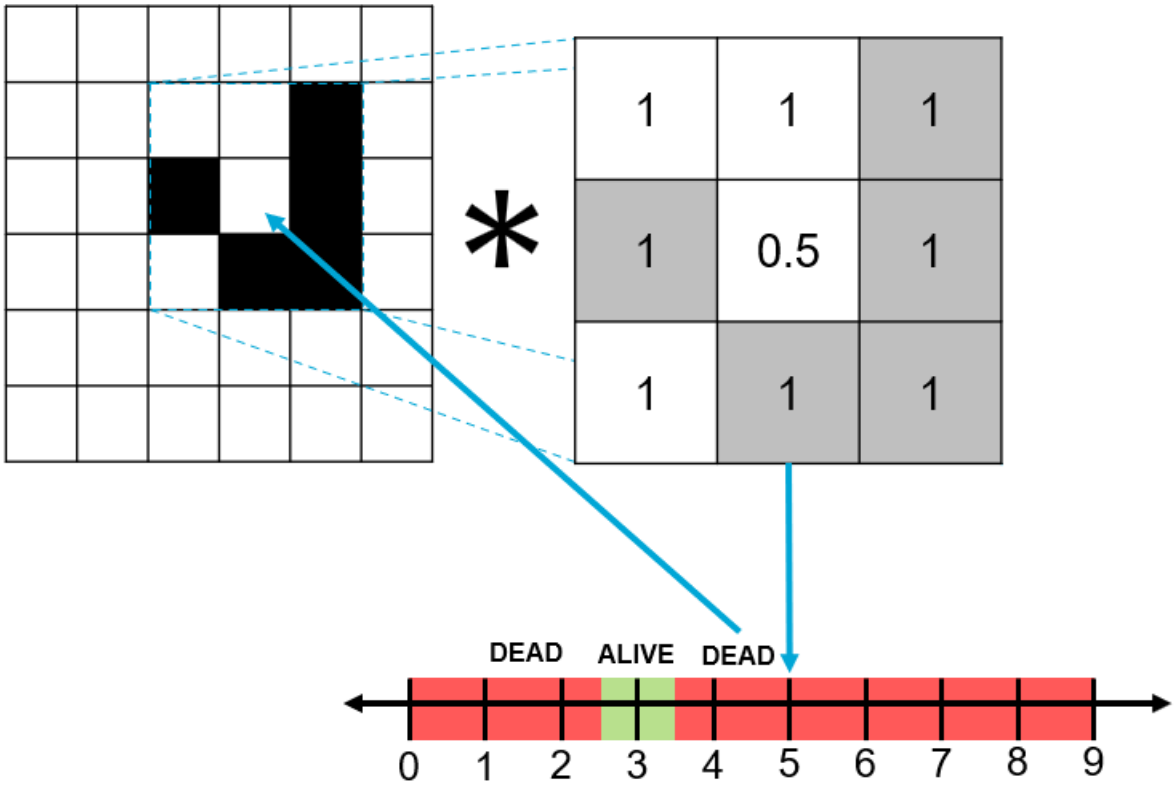


Figure 14: The game of Life can be represented as a bi-threshold system as shown

3.6 Threshold gates

3.6.1 Introduction

Threshold gates or elements are devices that can output either 0 or 1 depending on whether their weighted sum of inputs exceeds a certain threshold value. They are an alternative approach to conventional logic gates such as AND, OR and NOT gates. They are used in artificial neural networks, logic design and signal processing, though they remain a niche research area despite their utility. The theory of threshold elements was pioneered by Saburo Muroga, who published several papers on their properties, transformations and applications[51]. In this section, I will introduce the theory and discuss how it relates to the goal of developing intelligent metamaterials. I introduce the generalisation of threshold gates with multiple thresholds. As introduced in the previous section on One of the key components of this project is a bi-threshold gate that has two threshold values instead of one. A bi-threshold gate can cross two thresholds and has three regions in its input-output characteristic curve.

3.6.2 Threshold Logic

Threshold logic is a branch of logic that studies how to represent and manipulate Boolean functions using threshold elements. A Boolean function is any function that takes a finite number n of binary inputs (0 or 1) and produces a binary output. A threshold element or majority decision element is defined as an element whose output value is

$$1, \quad \text{if} \sum_{i=1}^{n} w_i x_i \geq T$$

$$0, \quad \text{if} \sum_{i=1}^{n} w_i x_i < T$$

where w_i is the weighting factor for the input x_i and T is the threshold value for that element. For our purposes, we are most interested in ternary (3-input) Boolean functions, in order to implement ECAs in metamaterials, particularly the ones with the most interesting and powerful properties such as Rule 110.
It has been shown that only 104 out of the 256 possible ternary Boolean functions can be represented by a threshold gate[50], the linearly separable functions. The linearity of these functions limits their potential for computation.

3.6.3 Multi- and Bi-threshold gates

Multi-threshold threshold elements are a generalisation of the threshold element, in which k thresholds $k = (1, 2, 3, ...)$ are used instead of one. Each threshold crossed alternates the output between 0 and 1. The benefit is that any arbitrary Boolean function can be realised with a single threshold element with a sufficiently large k[52]. Of course, a higher k implies a higher complexity of the element itself, so it is desirable to minimise the required number of thresholds for a given application. The special case where $k = 2$ are known as bi-threshold elements and have particular relevance to this project. As shown previously, almost all (230/256) ECAs can be implemented with bi-threshold logic, as can Conway's Game of Life. The emergent computational properties of these systems implies in my view that a system of cellularly networked bi-threshold elements could have sufficient complexity to imbue a metamaterial with information processing capabilities. As shown previously, these systems behave vastly differently depending on their initial states[47], so could also be "programmed" to an arbitrary functionality and also respond to external stimulus after manufacture. Multi-threshold systems have also been shown to have significant space improvements over traditional logic gates. The area of a given Boolean function of n bits has an exponential (unbounded) size $O(exp^n)$ when made with AND, OR and NOT gates. It is shown that bi-threshold elements reduce this requirement to $O(n^2)$ and multi-threshold elements reduce it further to $O(n)$[53, 54].

3.6.4 Application and implementation

Physical implementations of these elements, called Multi-Threshold Threshold Gates, has been accomplished through the negative differential resistance devices, which operate in a monostable to bistable regime. They essentially act as an electronic bi-stable element, allowing for the implementation of these thresholds using voltages and resistances[55, 56].
Theoretical uses for bi-threshold elements specifically are currently being explored, though there are challenges to their general utility. Bi-threshold activation functions for neural networks has been shown to decrease the

number of layers, though the difficulty in optimising the element has been a barrier. This is because the optimisation of a bi-threshold activation has been shown to be an NP-complete problem, meing no efficient algorithm has been found to compute it. Despite this, the utility has still motivated researchers to find applications and other approaches to apply them to solve problems[57, 58, 59].
Despite this difficulty in applying the theory at large scale in computer science, mechanisms researchers have serendipitously designed a schema for making multi-threshold gates in compliant mechanisms[60]. This approach uses a single bistable unit as the output, and sequentially adds more orthogonal buckled beams to add more stable points.

4 Research Plan

4.1 Research Questions

This thesis project aims to propose a novel approach for intelligent metamaterials, departing from the current design paradigm of derived from electronic computing. A compliant bi-threshold element will be developed as a building block for an elementary cellular automata, to demonstrate the feasibility of this approach. The goal is to embody Rule 110 ECA and determine a general design schema to enable the embodiment of any ECA.

4.2 System Concept

This project proposes a mechanism concept based on Merkle buckled beam logic[24], which is extended with multi-stable compliant mechanisms[60] to embody bi-threshold logic. The system's constitutive elements are designed to include the decision element, which has a nonlinear response that encodes the threshold function, the storage or encoding of the output of the decision element, the interconnection and self-connection of the elements to tessellate the structure into an elementary cellular automaton (ECA) structure, the parametric design of the threshold element weights, such as the coupling springs between units, and the update clocking mechanism, which supplies energy to update the metamaterial state based on the transition rule.

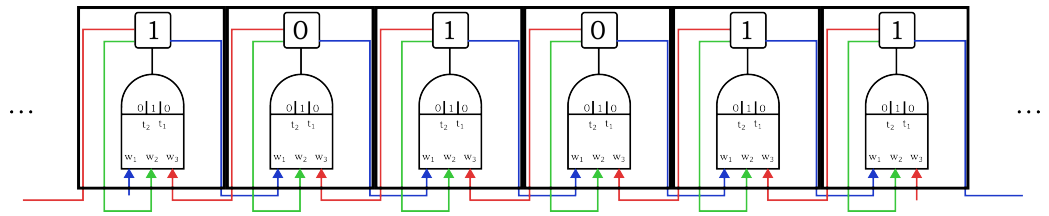


Figure 15: The proposed architecture of the metamaterial, where each unit cell has a bi-threshold decision element, a memory element and interconnections to its neighbours

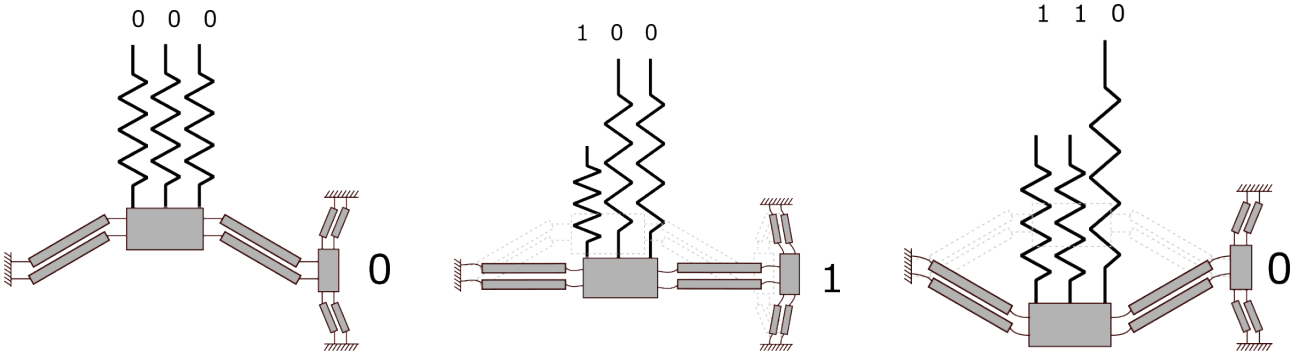


Figure 16: The proposed decision element (a) in the ground state (b) when the first threshold has been crossed and (c) in the saturated state.

4.3 Equivalence classes of 3 input Boolean functions

The primary aim of this project is to implement Rule 110 cellular automata, with the overarching goal of generalizing this approach to embody any of the 230 realizable ECAs. To achieve this, we aim to show that a subset of the ternary Boolean functions can be implemented, which all other functions are equivalent to. We define equivalent functions as those that can be obtained by permuting input variables, complementing one or more input variables, or complementing the output. Complementation refers to negation, where 1 becomes 0 and vice versa. It is shown in Figure 17 that all but two of these equivalence classes can be represented by single or bi-threshold gates. The

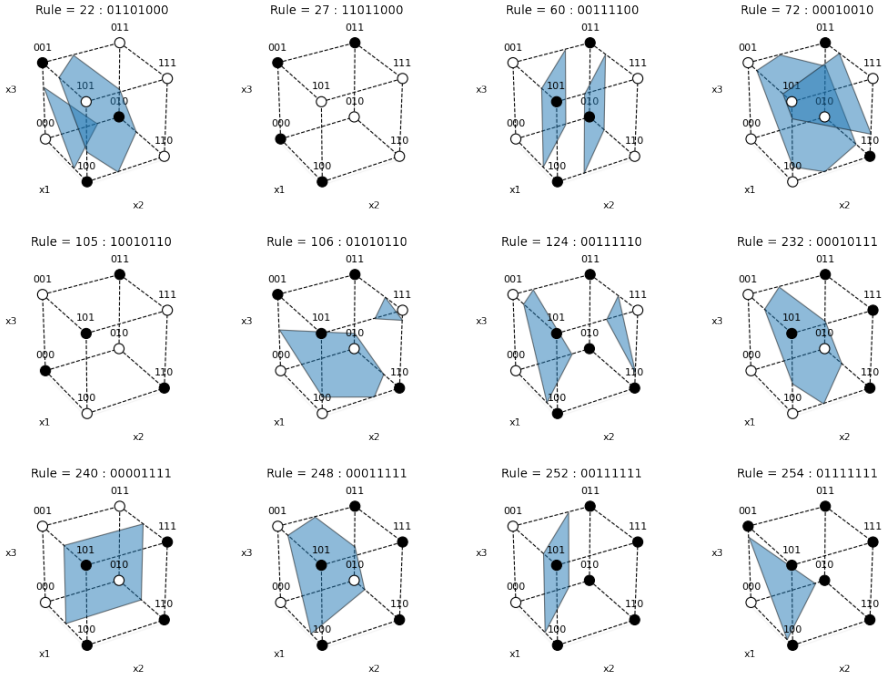


Figure 17: The 12 base functions of the equivalent classes, with the thresholds represented. Only the 26 functions associated rules 27 and 105 require three thresholds to represent.

4.4 Pseudo-Rigid body model of tristable element

As part of a feasibility study, a psuedo rigid body model was derived of the concept threshold gate in order to understand its mechanics. As can be seen in Figure 18, the PRBM consists of a crank slider mechanism with an additional compressive spring element to model the bistable beam element. The models geometry is parameterised by four lengths, H_1. L_1,H_2, L_2. The compliance of the mechanism is parameterised by three stiffnesses, two torsional k_θ, and k_ϕ, and one linear stiffness k_L. The mechanism takes the force F as input and the resultant state is the output displacements δ_1 and δ_2. The potential energy of the system is calculated by:

$$PE = k_\theta(\theta_0 - \theta)^2 + \frac{1}{2}k_\phi(\phi_0 - \phi)^2 + \frac{1}{2}k_L(L_0 - L)^2 \tag{1}$$

The input force F can be found by

$$F = \frac{dPE}{d\delta_1} \tag{2}$$

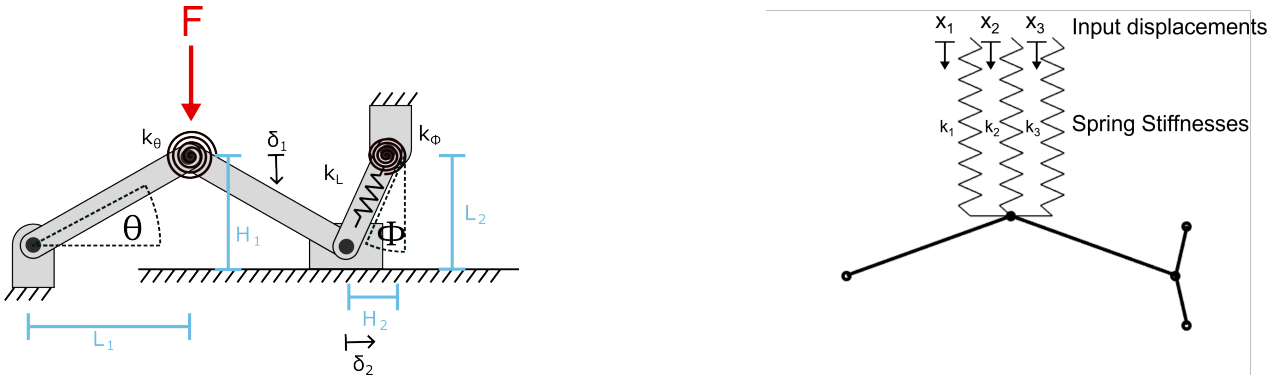
The nonlinear kinematics of the mechanism have been derived as follows:

$$\delta_2 = 2\left(\sqrt{L_1^2 + H_1^2 - (H_1 - \delta_1)^2} - L_1\right) \tag{3}$$

$$\theta = \tan^{-1}\left(\frac{H_1 - \delta_1}{L_1}\right) \tag{4}$$

$$\phi = \tan^{-1}\left(\frac{H_2 - \delta_2}{L_2}\right) \tag{5}$$

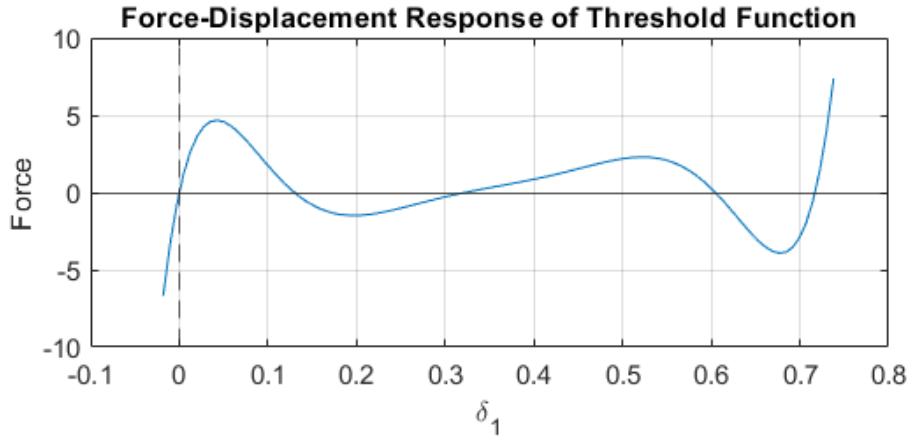
$$L = \sqrt{L_2^2 + (H_2 - \delta_2)^2} \tag{6}$$



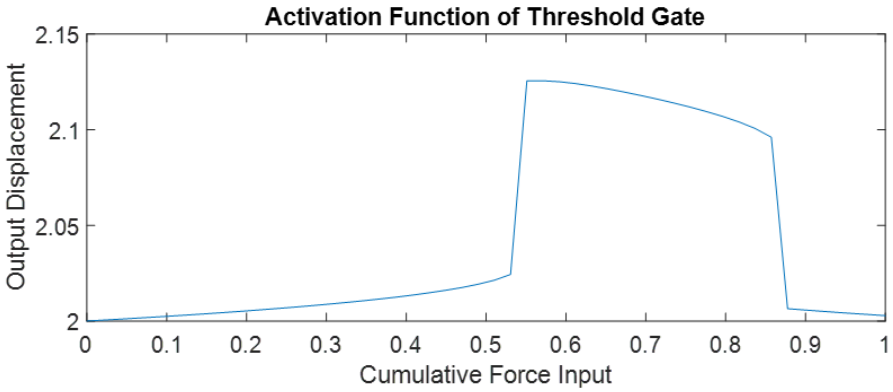
(a) (b)

Figure 18: The proposed decision element (a) psuedorigid body model parameterisation (b) Input displacement and stiffness parameterisation as constructed in MATLAB

The PRBM gives a useful analytical model to show the principle of operation of the mechanism. Preliminary results are shown in Figure 19. The tunable tri-stability of the mechanism agree with the results of Chen[60], partially validating the PRBM approach.



(a)



(b)

Figure 19: (a)Example force displacement plot generated by the PRBM. The three stable equilibria show disred multistability (b) Resultant output displacement over cumulative force input shows desired bi-threshold behaviour.

4.5 Research Plan

Based on the feasibility studies conducted so far, the plan for the next steps of the project is as follows:

1. Analysing the pseudo rigid body model to understand its properties and behaviour, and converting those insights to a detailed mechanical design.
2. Performing design and simulation steps to parameterize the properties of the unit cell in terms of its design parameters and the logical function we wish to embody.
3. Designing and developing the interconnection and tessellation systems needed to build a functional elementary cellular automaton for Rule 110.
4. Building and demonstrating the capabilities of the tessellated unit cell with Rule 110 ECA.
5. Quantifying approach performance and analysing the results.

4.6 Evaluation Criteria

The success of this project will be evaluated on several quantifiable metrics.

Threshold Design Freedom The degree of control we have over threshold values within the design space depends on the physical parameters of the design, which limits the values we can select for the thresholds. This limits the types of functions that can be created using this approach.

Interconnection Design Freedom The design freedom we have over the input weighting and interconnection system determines the complexity of the system that can be modelled. For example, in Figure 17, only weights of 1 and 2 were needed, but to account for negation, a NOT gate on some inputs will be required, and some other ratios of input weighting may be necessary. This is a design problem that must be solved to generalise this approach. The number of realisable systems (x/256) is a metric for the potential of this system.

Stability The robustness of the integrated system should be quantified by measuring the error rate during operation. The design should minimise errors, and ideally, the system should have a margin for manufacturing errors to ensure robustness.

References

[1] Tobias A Schaedler and William B Carter. "Architected Cellular Materials". In: (2016). DOI: 10.1146/annurev-matsci-070115-031624. URL: www.annualreviews.org.

[2] James Utama Surjadi et al. "Mechanical Metamaterials and Their Engineering Applications". In: *Advanced Engineering Materials* 21.3 (Mar. 2019). ISSN: 15272648. DOI: 10.1002/ADEM.201800864.

[3] Katia Bertoldi et al. *Flexible mechanical metamaterials*. Oct. 2017. DOI: 10.1038/natrevmats.2017.66.

[4] M. A. McEvoy and N. Correll. "Materials that couple sensing, actuation, computation, and communication". In: *Science* 347.6228 (Mar. 2015). ISSN: 10959203. DOI: 10.1126/SCIENCE.1261689.

[5] Katherine S. Riley et al. "Neuromorphic Metamaterials for Mechanosensing and Perceptual Associative Learning". In: *Advanced Intelligent Systems* 4.12 (Dec. 2022), p. 2200158. ISSN: 2640-4567. DOI: 10.1002/AISY.202200158. URL: https://onlinelibrary.wiley.com/doi/full/10.1002/aisy.202200158%20https://onlinelibrary.wiley.com/doi/abs/10.1002/aisy.202200158%20https://onlinelibrary.wiley.com/doi/10.1002/aisy.202200158.

[6] Yan Fang et al. "Pattern recognition with "materials that compute"". In: *Science Advances* 2.9 (Sept. 2016). ISSN: 23752548. DOI: 10.1126/SCIADV.1601114/SUPPL{_}FILE/1601114{_}SM.PDF. URL: https://www.science.org/doi/10.1126/sciadv.1601114.

[7] Rudolf M. Füchslin et al. "Morphological Computation and Morphological Control: Steps Toward a Formal Theory and Applications". In: *Artificial Life* 19.1 (Jan. 2013), pp. 9–34. ISSN: 1064-5462. DOI: 10.1162/ARTL{_}A{_}00079. URL: https://direct.mit.edu/artl/article/19/1/9/2747/Morphological-Computation-and-Morphological.

[8] *Citation Gecko*. URL: https://citationgecko.azurewebsites.net/.

[9] Rami Alfattani and Craig Lusk. "A lamina-emergent frustum using a bistable collapsible compliant mechanism". In: *Journal of Mechanical Design, Transactions of the ASME* 140.12 (Dec. 2018). ISSN: 10500472. DOI: 10.1115/1.4037621/366184. URL: https://asmedigitalcollection.asme.org/mechanicaldesign/article/140/12/125001/366184/A-Lamina-Emergent-Frustum-Using-a-Bistable.

[10] Melina K. Blees et al. "Graphene kirigami". In: *Nature 2015 524:7564* 524.7564 (July 2015), pp. 204–207. ISSN: 1476-4687. DOI: 10.1038/nature14588. URL: https://www.nature.com/articles/nature14588.

[11] Jesse L. Silverberg et al. "Using origami design principles to fold reprogrammable mechanical metamaterials". In: *Science* 345.6197 (2014). ISSN: 10959203. DOI: 10.1126/science.1252876.

[12] Z. Y. Wei et al. "Geometric mechanics of periodic pleated origami". In: *Physical Review Letters* 110.21 (May 2013). ISSN: 00319007. DOI: 10.1103/PHYSREVLETT.110.215501. URL: http://arxiv.org/abs/1211.6396%20http://dx.doi.org/10.1103/PhysRevLett.110.215501.

[13] Zuolin Liu et al. "Discriminative transition sequences of origami metamaterials for mechano-logic". In: (Jan. 2022). DOI: 10.48550/arxiv.2201.06221. URL: https://arxiv.org/abs/2201.06221v1.

[14] N S Saravana Jothi and A Hunt. "Active mechanical metamaterial with embedded piezoelectric actuation". In: 10 (2022), p. 91117. DOI: 10.1063/5.0101420. URL: https://doi.org/10.1063/5.0101420.

[15] Babak Haghpanah et al. "Programmable Elastic Metamaterials". In: *Advanced Engineering Materials* 18.4 (Apr. 2016), pp. 643–649. ISSN: 1527-2648. DOI: 10.1002/ADEM.201500295. URL: https://onlinelibrary.wiley.com/doi/full/10.1002/adem.201500295%20https://onlinelibrary.wiley.com/doi/abs/10.1002/adem.201500295%20https://onlinelibrary.wiley.com/doi/10.1002/adem.201500295.

[16] Ryan H. Lee, Erwin A. B. Mulder, and Jonathan B. Hopkins. "Mechanical neural networks: Architected materials that learn behaviors". In: *Science Robotics* 7.71 (Oct. 2022). ISSN: 2470-9476. DOI: 10.1126/SCIROBOTICS.ABQ7278. URL: https://www.science.org/doi/10.1126/scirobotics.abq7278.

[17] ML van and Martin van Hecke. "Profusion of transition pathways for interacting hysterons". In: *Physical Review E* 104.5 (2021), p. 54608. DOI: 10.1103/PhysRevE.104.054608. URL: https://hdl.handle.net/1887/3245513.

[18] Jiangnan Ding and Martin Van Hecke. "Sequential Snapping and Pathways in a Mechanical Metamaterial". In: ().

[19] Sicong Shan et al. "Multistable Architected Materials for Trapping Elastic Strain Energy". In: *Adv. Mater* (2015). DOI: 10.1002/adma.201501708. URL: www.advmat.de.

[20] Ahmad Rafsanjani et al. "Snapping Mechanical Metamaterials under Tension". In: *Advanced Materials* 27.39 (Oct. 2015), pp. 5931–5935. ISSN: 1521-4095. DOI: 10.1002/ADMA.201502809. URL: https://onlinelibrary.wiley.com/doi/full/10.1002/adma.201502809%20https://onlinelibrary.wiley.com/doi/abs/10.1002/adma.201502809%20https://onlinelibrary.wiley.com/doi/10.1002/adma.201502809.

[21] David Restrepo, Nilesh D. Mankame, and Pablo D. Zavattieri. "Phase transforming cellular materials". In: *Extreme Mechanics Letters* 4 (Sept. 2015), pp. 52–60. ISSN: 2352-4316. DOI: 10.1016/J.EML.2015.08.001.

[22] Hiromi Yasuda et al. *Mechanical computing*. Oct. 2021. DOI: 10.1038/s41586-021-03623-y.

[23] M L Roukes. "Mechanical Computation, Redux?" In: ().

[24] R. C. Merkle. "Two types of mechanical reversible logic". In: *Nanotechnology* 4.2 (1993), pp. 114–131. ISSN: 09574484. DOI: 10.1088/0957-4484/4/2/007.

[25] Ralph C. Merkle et al. "Mechanical Computing Systems Using Only Links and Rotary Joints". In: *Journal of Mechanisms and Robotics* 10.6 (Dec. 2018). ISSN: 1942-4302. DOI: 10.1115/1.4041209. URL: https://asmedigitalcollection.asme.org/mechanismsrobotics/article/10/6/061006/477620/Mechanical-Computing-Systems-Using-Only-Links-and.

[26] Jordan R. Raney et al. "Stable propagation of mechanical signals in soft media using stored elastic energy". In: *Proceedings of the National Academy of Sciences of the United States of America* 113.35 (Aug. 2016), pp. 9722–9727. ISSN: 10916490. DOI: 10.1073/PNAS.1604838113/SUPPL{_}FILE/PNAS.1604838113.SM07.WMV. URL: https://www.pnas.org/doi/abs/10.1073/pnas.1604838113.

[27] Michelle Berry et al. "Mechanical signaling cascades". In: ().

[28] Yuanping Song et al. "Additively manufacturable micro-mechanical logic gates". In: *Nature Communications 2019 10:1* 10.1 (Feb. 2019), pp. 1–6. ISSN: 2041-1723. DOI: 10.1038/s41467-019-08678-0. URL: https://www.nature.com/articles/s41467-019-08678-0.

[29] Mohammad Usman Waheed. "Functional mechanical metamaterials - development of programmable mechanical structures". In: (2022). DOI: 10.25560/95429. URL: http://spiral.imperial.ac.uk/handle/10044/1/95429.

[30] Youngseok Oh. "Synthesis of Multistable Equilibrium Compliant Mechanisms." PhD thesis. 2008. URL: http://deepblue.lib.umich.edu/handle/2027.42/61608.

[31] Yang Gao et al. "A Planar Single-Actuator Bi-Stable Switch Based on Latch-Lock Mechanism". In: *2019 20th International Conference on Solid-State Sensors, Actuators and Microsystems and Eurosensors XXXIII, TRANSDUCERS 2019 and EUROSENSORS XXXIII* (June 2019), pp. 705–708. DOI: 10.1109/TRANSDUCERS.2019.8808375.

[32] Bicheng Chen et al. "Magnetic force induced tristability for dielectric elastomer actuators". In: *Smart Materials and Structures* 26.10 (Sept. 2017), p. 105007. ISSN: 0964-1726. DOI: 10.1088/1361-665X/AA8282. URL: https://iopscience.iop.org/article/10.1088/1361-665X/aa8282%20https://iopscience.iop.org/article/10.1088/1361-665X/aa8282/meta.

[33] Yijie Jiang, Lucia M. Korpas, and Jordan R. Raney. "Bifurcation-based embodied logic and autonomous actuation". In: *Nature Communications 2019 10:1* 10.1 (Jan. 2019), pp. 1–10. ISSN: 2041-1723. DOI: 10.1038/s41467-018-08055-3. URL: https://www.nature.com/articles/s41467-018-08055-3.

[34] P. R. Kuppens et al. "Monolithic binary stiffness building blocks for mechanical digital machines". In: *Extreme Mechanics Letters* 42 (Jan. 2021), p. 101120. ISSN: 2352-4316. DOI: 10.1016/J.EML.2020.101120.

[35] Xiaomei Yao et al. "Using binary-stiffness beams within mechanical neural-network metamaterials to learn". In: *Smart Materials and Structures* 32.3 (Feb. 2023), p. 035015. ISSN: 0964-1726. DOI: 10.1088/1361-665X/ACB519. URL: https://iopscience.iop.org/article/10.1088/1361-665X/acb519%20https://iopscience.iop.org/article/10.1088/1361-665X/acb519/meta.

[36] Jin Qiu, Jeffrey H Lang, and Alexander H Slocum. "A Curved-Beam Bistable Mechanism". In: *JOURNAL OF MICROELECTROMECHANICAL SYSTEMS* 13.2 (2004). DOI: 10.1109/JMEMS.2004.825308.

[37] Hang Zhang et al. "Hierarchical mechanical metamaterials built with scalable tristable elements for ternary logic operation and amplitude modulation". In: *Science Advances* 7.9 (Feb. 2021). ISSN: 23752548. DOI: 10.1126/SCIADV.ABF1966/SUPPL{_}FILE/ABF1966{_}SM.PDF. URL: https://www.science.org/doi/10.1126/sciadv.abf1966.

[38] Lennard J Kwakernaak and Martin Van Hecke. "Counting and Sequential Information Processing in Mechanical Metamaterials". In: ().

[39] Norman Margolus. "Programmable matter: Concepts and realization". In: ().

[40] Andrew Ilachinski. *Cellular Automata*. WORLD SCIENTIFIC, July 2001. ISBN: 978-981-02-4623-5. DOI: 10.1142/4702. URL: https://books.google.com/books/about/Cellular_Automata.html?hl=nl&id=3Hx2lx_pEF8C.

[41] Stephen Wolfram. *A New Kind of Science*. Wolfram Media, 2002. URL: https://www.wolframscience.com/nks/.

[42] Melanie Mitchell. "Computation in Cellular Automata: A Selected Review". In: (1996).

[43] W. Daniel Hillis. "The connection machine: A computer architecture based on cellular automata". In: *Physica D: Nonlinear Phenomena* 10.1-2 (Jan. 1984), pp. 213–228. ISSN: 0167-2789. DOI: 10.1016/0167-2789(84)90263-X.

[44] Matthew Cook. "Universality in Elementary Cellular Automata". In: (1985).

[45] Stephen Wolfram. "Universality and complexity in cellular automata". In: *Physica D: Nonlinear Phenomena* 10.1-2 (Jan. 1984), pp. 1–35. ISSN: 0167-2789. DOI: 10.1016/0167-2789(84)90245-8.

[46] *Universal computer - LifeWiki*. URL: https://conwaylife.com/wiki/Universal_computer.

[47] R Ichard, K Squier, and Ken Steiglitz. "Programmable Parallel Arithmetic in Cellular Automata Using a Particle Model". In: *Complex Systems* 8 (1994), pp. 311–323.

[48] Fangyue Chen et al. "Implementation of arbitrary boolean functions via CNN". In: *Proceedings of the IEEE International Workshop on Cellular Neural Networks and their Applications* (2006). DOI: 10.1109/CNNA.2006.341641.

[49] Dafydd Ravenscroft and Luigi G. Occhipinti. "2D Material Memristor Devices for Neuromorphic Computing". In: *International Workshop on Cellular Nanoscale Networks and their Applications* 2021-September (2021). ISSN: 21650179. DOI: 10.1109/CNNA49188.2021.9610802.

[50] S. Muroga, I. Toda, and M. Kondo. "Majority Decision Functions of up to Six Variables". In: *Mathematics of Computation* 16.80 (Oct. 1962), p. 459. ISSN: 00255718. DOI: 10.2307/2003136.

[51] Saburo Muroga, Iwao Toda, and Satoru Takasu. "THEORY OF MAJORITY DECISION ELEMENTS". In: ().

[52] Donald R. Haring. "Multi-Threshold Threshold Elements". In: *IEEE Transactions on Electronic Computers* EC-15.1 (1966), pp. 45–65. ISSN: 03677508. DOI: 10.1109/PGEC.1966.264375.

[53] Vasken Bohossian and Jehoshua Bruck. "Multiple Threshold Neural Logic". In: *Advances in Neural Information Processing Systems* (1997).

[54] Ingo Wegener. "The complexity of the parity function in unbounded fan-in, unbounded depth circuits". In: *Theoretical Computer Science* 85 (1991), pp. 155–170.

[55] Maciej Nikodem. "Synthesis of multithreshold threshold gates based on negative differential resistance devices". In: *IET Circuits, Devices & Systems* 7.5 (Sept. 2013), pp. 232–242. ISSN: 1751-8598. DOI: 10.1049/IET-CDS.2012.0368. URL: https://onlinelibrary.wiley.com/doi/full/10.1049/iet-cds.2012.0368%20https://onlinelibrary.wiley.com/doi/abs/10.1049/iet-cds.2012.0368%20https://ietresearch.onlinelibrary.wiley.com/doi/10.1049/iet-cds.2012.0368.

[56] M J Avedillo et al. "Multi-threshold Threshold Logic Circuit Design Using Resonant Tunneling Devices". In: *Electronics Letters* 39.21 (2003), pp. 1502–1503.

[57] Vladyslav Kotsovsky and Anatoliy Batyuk. "Feed-forward Neural Network Classifiers with Bithreshold-like Activations". In: *International Scientific and Technical Conference on Computer Sciences and Information Technologies* 2022-November (2022), pp. 9–12. ISSN: 27663639. DOI: 10.1109/CSIT56902.2022.10000739.

[58] Vladyslav Kotsovsky, Anatoliy Batyuk, and Ivan Mykoriak. "The Computation Power and Capacity of Bithreshold Neurons". In: *International Scientific and Technical Conference on Computer Sciences and Information Technologies* 1 (Sept. 2020), pp. 28–31. ISSN: 27663639. DOI: 10.1109/CSIT49958.2020.9322014.

[59] Vinay Deolalikar. "A two-layer paradigm capable of forming arbitrary decision regions in input space". In: *IEEE Transactions on Neural Networks* 13.1 (Jan. 2002), pp. 15–21. ISSN: 10459227. DOI: 10.1109/72.977261.

[60] Guimin Chen, Yanjie Gou, and Aimei Zhang. "Synthesis of compliant multistable mechanisms through use of a single bistable mechanism". In: *Journal of Mechanical Design, Transactions of the ASME* 133.8 (Aug. 2011). ISSN: 10500472. DOI: 10.1115/1.4004543/478284. URL: https://asmedigitalcollection.asme.org/mechanicaldesign/article/133/8/081007/478284/Synthesis-of-Compliant-Multistable-Mechanisms.