The Journal of Open Source Software

Elementary Cellular Automata as Multiplicative Automata

Daniel W. McKinley [1]

1 Independent Researcher, USA

DOI: 10.xxxxxx/draft

Software

- Review
- Repository
- Archive

Editor:

Submitted: 08 July 2024
Published: unpublished

Summary

Elementary cellular automata (ECA) are a set of simple binary programs in the form of truth tables called Wolfram codes that produce complex output when done repeatedly in parallel, and quaternions are frequently used to represent 3D space and its rotations in computer graphics. Both are well-studied subjects, this Java library puts them together in a new way. This project changes classical additive cellular automata into multiplicative automata (Wolfram, 2002, p. 886) via permutations, hypercomplex numbers, and pointer arrays. Valid solutions extend the binary ECA to complex numbers, produce a vector field, make an algebraic polynomial, and generate some very interesting fractals.

Statement of Need

Very loosely analogous to De Morgan's law in Boolean algebra, the main algorithm produces several multiplicative versions of any given standard additive binary Wolfram code up to 32 bits and is written to support user supplied complex 1-D input at row 0 with choice of type of multiplication tables and partial product tables among other parameters. It produces an algebraic polynomial and complex vector field output for any given Wolfram code, and the hypercomplex 5-factor identity solution allows for the complex extension of any binary cellular automata. The Cayley-Dickson and Fano construction libraries may be of value to the open source community as well.

There are other cellular automata implementations, Mathematica (Inc., n.d.), CellPyLib (Antunes, 2021), a JOSS Python project from three years ago, and others. This is not designed to replace those awesome general purpose utilities, it's focused on the set of Wolfram code operations. The GUI is designed to show enough to conclude that the math works and give a rough idea of aggregate behavior over parameters and the algorithm code is designed to be able to split off and be plugged in somewhere else. There are useful things you can build on it directly or indirectly, like making Bloch spheres out layers of complex number output, Fourier analysis, 2D automata, a complex version of the prime number automata (Wolfram, 2002, p. 640), Gray code and full group theory morphisms of multiplication tables and paths, and making N-D ellipses out of the paths through the multiplication tables, that are clear directions to go in but subject to a different set of decisions like application-specific tech debt and potential translation to C++ or Python and out of scope of this paper.

The Journal of Open Source Software

Functions

Hypercomplex unit vector implementation

	Negative sign bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Complex	2's place	1	i	-1	-i												
Quaternions	4's place	1	i	j	k	-1	-i	-j	-k								
Octonions	8's place	1	e1	e2	e3	e4	e5	e6	e7	-1	-e1	-e2	-e3	-e4	-e5	-e6	-e7

The Cayley-Dickson (CD) and Fano support classes are discussed in greater detail in the readme and the documentation, they along with the Galois class provide sets of multiplication tables to be compared with cellular automata. The CD multiplication implementation permutes the steps of splitting and recombining hypercomplex numbers to increase the scope of the CD equation, $(a, b)x(c, d) = (ac - d * b, da + bc*)$, where * is the conjugate. It verifies itself by producing the vector product of two symmetric groups of its degree operating on the up/down recursion factoradics when interacting with other CD multiplications and is the permutations of the unit vector bit layers excluding the negative sign. The Fano library octonions produce a triplet that is a linear match to the CD octonions as triplets$\{0\}$ when the up and down recursion factoradics are equal, and produce the triplet set of John Baez's Fano plane as triplets$\{10\}$. (Baez, 2001).

The main algorithm uses a set of permutations operating on cellular automata input, each permutation permuting the neighborhood, becoming a factor, with four kinds of multiplications. The multiplication tables are input as 2D but used as N-D, where N=numFactors.

	Multiplications A	Multiplications B	Multiplications C	Multiplications D
Type	Hypercomplex or finite, brute-force of all the permutations of that number of factors	Cartesian product summed by a hypercomplex or finite partial product table	Complex product	Permutation composition
Size	Wolfram code length = L	Size of neighborhood, log2(L)	Size of neighborhood, log2(L)	Size of neighborhood, log2(L)
Function	Validates permutation group, reproducing the Wolfram code as a pointer array	Applies a valid solution to a user given complex neighborhood	Like B, but does the normalization before the multiplication	Orders the cell's neighborhood vector from (B), post multiplication, pre normalization
Scope	Entire Wolfram code, every possible binary neighborhood	Single given input neighborhood	Single given input neighborhood	Single given input neighborhood
Produces	Set of permutations that changes the additive automata to multiplicative, with the given multiplication table	Polynomial	Output visually similar to B	Vector
Data type	Binary	Complex	Complex	Discrete permutation
Base 2 sum of neighborhood	Construction of factors, pre multiplication	Normalization, post multiplication	Construction of factors, pre multiplication	n/a
N-th root in normalization	n/a	N = size of neighborhood	N = number of factors	n/a

Multiplications A, additive to multiplicative
r = specific Wolfram code
n = binary neighborhood = 1*columnZero* + 2columnOne+ 4*columnTwo…2^(column)*columnCol, points to its value in r
h = hypercomplex unit vector from binary
H = inverse of h, binary value from hypercomplex unit vector
p = a permutation of the neighborhood
using hypercomplex multiplication, a valid permutation set produces:
WolframCode(r, n) = WolframCode(r, H(h(p(n)) * h(p(n)) * h(p(n)) … numFactors), though n may or may not equal H(…)
WolframCode(r, H(h(p(n)) * h(p(n)) * h(p(n)) … numFactors)) is a pointer array that always points to an equal value (0,1) within WolframCode(r, _)
each h(p(n)) in a valid solution is a factor template in the multiplication table for all values of

its axis

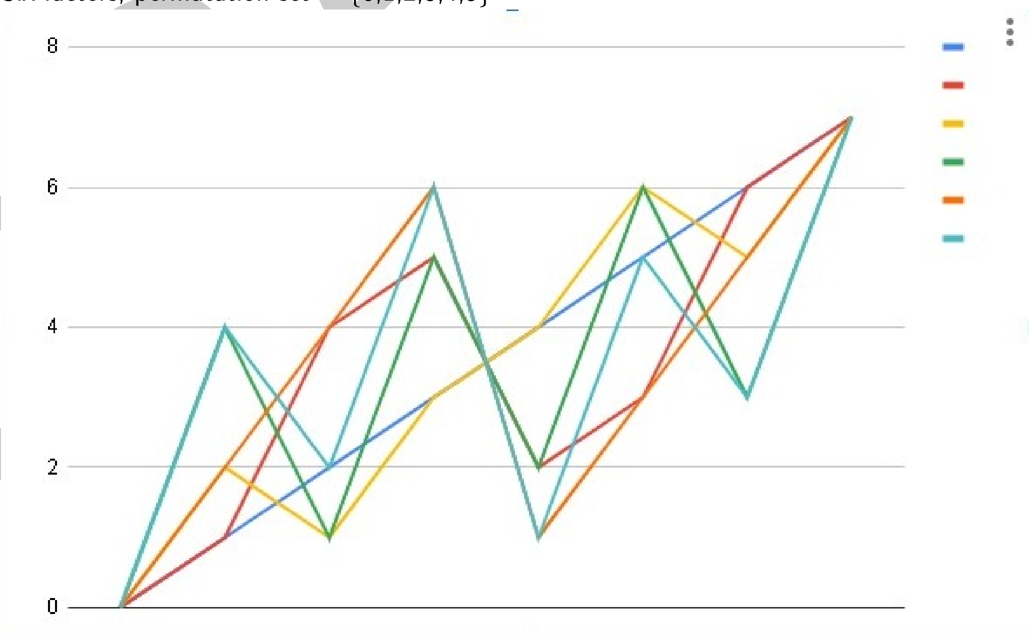
The first set of multiplications, column A, brute forces all possible sets of permutations on all possible binary neighborhoods of the Wolfram code. A permutation in the set rearranges the columns of the input neighborhood, these become a set of factors. A valid set of permutations is one that, for all possible input neighborhoods, the set of constructed factors using the permuted neighborhoods always multiplies out to a value that points to an equal value within the Wolfram code. The set of multiplication results is a pointer array that reproduces the original Wolfram code for every possible binary neighborhood.

Identity solutions of 5 factors using all zero permutations exist for Wolfram codes up to 32 bits in this library using hypercomplex numbers, XOR and Galois addition and more bits require more factors in increments of four. Galois multiplication takes a mix of numbers of factors to get the identity multiplication result array according to their, there is a function in the GaloisField class that provides it. The factors constructed are a loose diagonal through the multidimensional multiplication table, starting at the origin and ending at the opposite corner while zig-zagging. The path lengths of each factor and the result are included in ValidSolution results.

Permutations of 3 bit neighborhoods

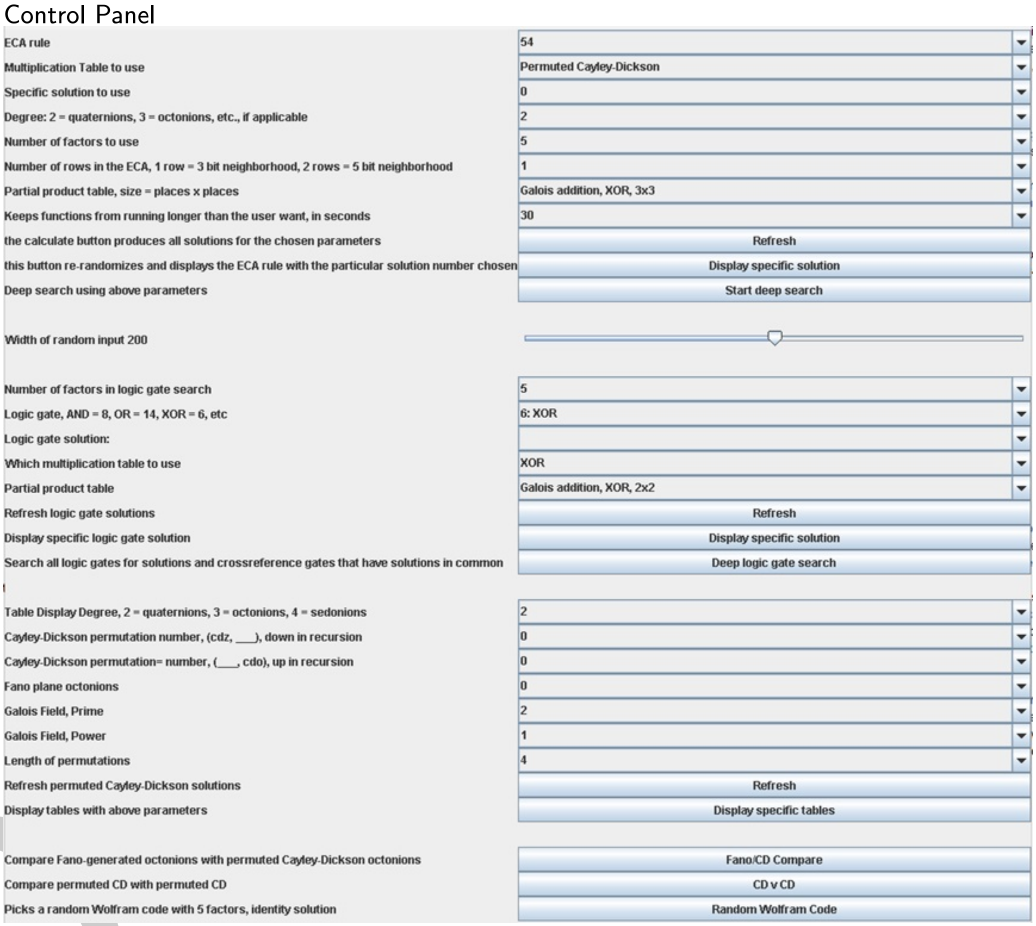
```
Permutation: 0, [0, 1, 2, 3, 4, 5, 6, 7]
Permutation: 1, [0, 1, 4, 5, 2, 3, 6, 7]
Permutation: 2, [0, 2, 1, 3, 4, 6, 5, 7]
Permutation: 3, [0, 4, 1, 5, 2, 6, 3, 7]
Permutation: 4, [0, 2, 4, 6, 1, 3, 5, 7]
Permutation: 5, [0, 4, 2, 6, 1, 5, 3, 7]
```

Flattened path through a six dimensional multiplication table
Six factors, permutation set = {0,1,2,3,4,5}



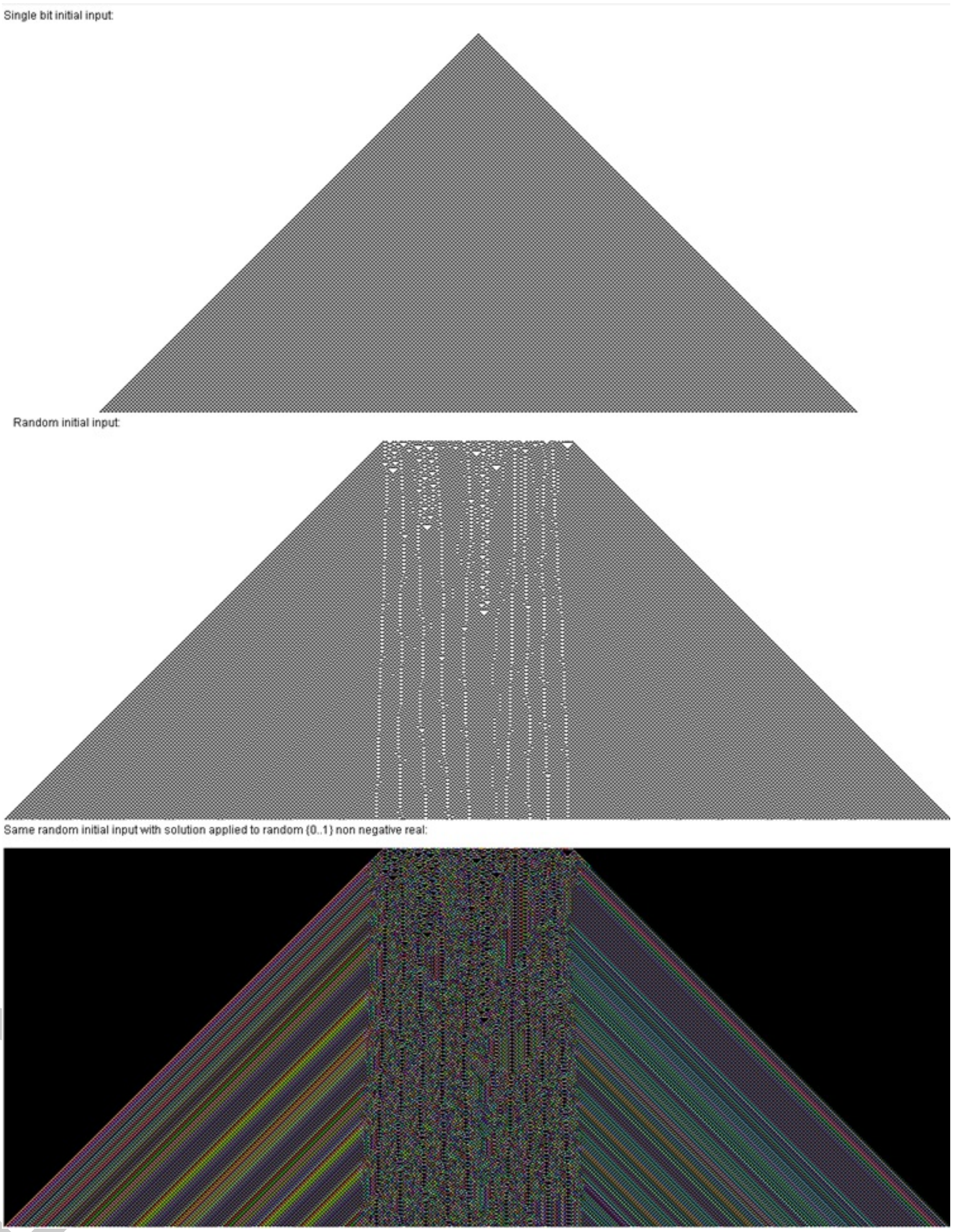
Multiplications B and C apply a valid solution from the first set of multiplications to any given individual neighborhood with binary, non-negative real, and complex values. Multiplication B is the Cartesian product of the permuted neighborhoods, using a closed partial product table to generate a polynomial. Multiplication C does the binary sum of complex neighborhood, then multiplies as complex. Both B and C take the n-th root of the result, with n = numColumns and n = numFactors, respectively. Multiplications B and C both include a binary weighted sum of the neighborhood, same as the construction of the factors from A, though B and C use complex. B, as part of the normalization and C as the construction. Multiplication C is the permutation composition product. B, just before the normalization is a neighborhood of multiplication results, with each column of it being a unit vector coefficient. This multiplication result neighborhood is permuted by the inverse of the permutation composition product to properly order the output vector. There are a couple of normalization parameters and a hybrid multiplicative-additive output option that are discussed more in the documentation.

Control Panel



ECA 54, binary and non negative real

The Journal of Open Source Software

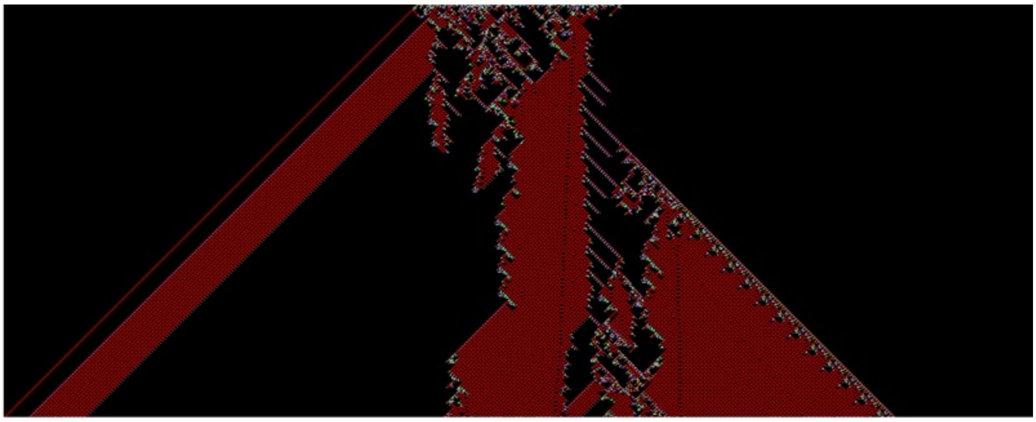


ECA 54, solution parameters, including polynomial

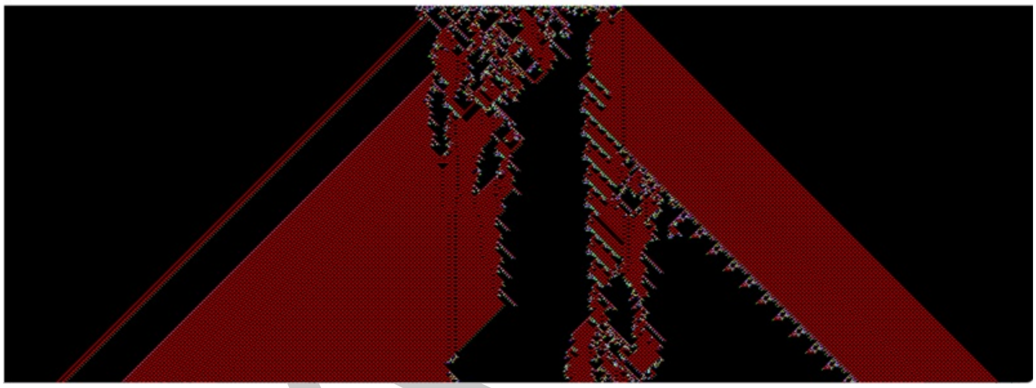
The Journal of Open Source Software

```
ValidSolution
Wolfram code: [0, 1, 1, 0, 1, 1, 0, 0]
Permutation: 0   Permuted Axis: [0, 1, 2, 3, 4, 5, 6, 7]
                 times
Permutation: 0   Permuted Axis: [0, 1, 2, 3, 4, 5, 6, 7]
                 times
Permutation: 0   Permuted Axis: [0, 1, 2, 3, 4, 5, 6, 7]
                 times
Permutation: 0   Permuted Axis: [0, 1, 2, 3, 4, 5, 6, 7]
                 times
Permutation: 0   Permuted Axis: [0, 1, 2, 3, 4, 5, 6, 7]
                 times
-----------------
Equals:                           [0, 1, 2, 3, 4, 5, 6, 7]
Apply Wolfram code to multiplication result
Equals:                           [0, 1, 1, 0, 1, 1, 0, 0]
Original Wolfram code:            [0, 1, 1, 0, 1, 1, 0, 0]

Permutation composition product: 0, inverse: 0

Multiplication table type: 0
2D multiplication table used:
[0, 1, 2, 3, 4, 5, 6, 7]
[1, 4, 7, 2, 5, 0, 3, 6]
[2, 3, 4, 5, 6, 7, 0, 1]
[3, 6, 1, 4, 7, 2, 5, 0]
[4, 5, 6, 7, 0, 1, 2, 3]
[5, 0, 3, 6, 1, 4, 7, 2]
[6, 7, 0, 1, 2, 3, 4, 5]
[7, 2, 5, 0, 3, 6, 1, 4]

numFactors: 5 numBits: 3

1*((a^5)*(b^0)*(c^0)) + 20*((a^3)*(b^1)*(c^1)) + 10*((a^2)*(b^3)*(c^0)) + 10*((a^2)*(b^0)*(c^3)) + 30*((a^1)*(b^2)*(c^2)) +
5*((a^0)*(b^4)*(c^1)) + 5*((a^0)*(b^1)*(c^4))

5*((a^4)*(b^1)*(c^0)) + 10*((a^3)*(b^0)*(c^2)) + 30*((a^2)*(b^2)*(c^1)) + 5*((a^1)*(b^4)*(c^0)) + 20*((a^1)*(b^1)*(c^3)) +
10*((a^0)*(b^3)*(c^2)) + 1*((a^0)*(b^0)*(c^5))

5*((a^4)*(b^0)*(c^1)) + 10*((a^3)*(b^2)*(c^0)) + 30*((a^2)*(b^1)*(c^2)) + 20*((a^1)*(b^3)*(c^1)) + 5*((a^1)*(b^0)*(c^4)) +
1*((a^0)*(b^5)*(c^0)) + 10*((a^0)*(b^2)*(c^3))
```

ECA 54, solution output, complex



Complex part



References

Antunes, L. M. (2021). CellPyLib: A python library for working with cellular automata. *Journal of Open Source Software*, *6*(67), 3608. https://doi.org/10.21105/joss.03608

Baez, J. (2001). *The octonions* (10.1090/S0273-0979-01-00934-X). Bulletin of the American Mathematical Society.

Inc., W. R. (n.d.). *Mathematica, version 14.0*. https://www.wolfram.com/mathematica

Wolfram, S. (2002). *A new kind of science*. Wolfram Media. ISBN: 1579550088