

Using Golly’s Scripting Tools to Analyze Larger than Life’s Jitterbugs

Kellie Michele Evans¹[0000–0002–3258–9732]
and Brandon Ismalej¹[0009–0009–6170–1314]

California State University, Northridge, CA, 91330, USA
`kellie.m.evans@csun.edu`
`brandon.ismalej.671@my.csun.edu`

Abstract. Larger than Life is a family of cellular automata proposed by David Griffeath that generalizes John Conway’s celebrated Game of Life to large neighborhoods and update rules based on birth and survival intervals. Golly is an open source application for exploring the Game of Life and, more recently, Larger than Life. In this exploratory paper, we present new scripts for the Golly software that help identify and analyze the complex trajectories of certain viable patterns in Larger than Life, known as “jitterbugs,” including the Game of Life’s famous glider. We describe the scripts’ limitations and ideas about how to surpass them.

Keywords: Cellular automata · Game of Life · Larger than Life.

1 Introduction

Larger than Life (LtL), introduced by David Griffeath in the 1990s, generalizes John Conway’s Game of Life (Life) by using larger neighborhoods and birth and survival thresholds defined by four parameters [1]. Like Life, LtL can generate remarkably complex behavior from simple local rules, including persistent moving structures known as “jitterbugs,” the most famous example being Life’s glider [2]. These viable patterns can emerge from random initial configurations and, in some cases, can be engineered to perform computation through logic gates, memory, and control structures. Conway’s original interest in cellular automata was motivated by the search for a simple universal rule, and Life has been shown to be computationally universal [3], [4]. We conjecture that numerous “Life-like” LtL rules, including Bosco’s rule, are also universal [5]. In addition, LtL is mathematically significant because it provides a simple spatial and discrete analogue of nonlinear population dynamics models such as the Lotka–Volterra equations and the logistic map and it is interesting in its own right for its threshold-range scaling and numerous other challenging problems [6], [7].

Because LtL rules involve large neighborhoods and multiple threshold parameters, simulations are computationally intensive. Widespread exploration of LtL became practical only within the last decade through implementation in the remarkable Golly software [8]. Golly’s scripting capabilities now make detailed analysis of LtL dynamics possible. In what follows, we describe Lua scripts we

developed for Golly that identify and analyze jitterbugs and their nonlinear trajectories across the grid. We describe the scripts' limitations and ideas about how to surpass them.

2 Definitions

2.1 Larger than Life

An LtL rule is defined as follows. Each site of the *two-dimensional lattice* $\mathbf{Z}^2$ is in one of two *states*, *live* (1) or *dead* (0). This is the *initial configuration* of the system. The *neighborhood* $\mathcal{N}_\rho$ of a site consists of the $(2\rho + 1) \times (2\rho + 1)$ sites in the box surrounding and including it. That is, the neighborhood of the origin is $\mathcal{N}_\rho = \{y \in \mathbf{Z}^2 : \|y\|_\infty \leq \rho\}$ (ρ a natural number), so that its translate $\mathcal{N}_\rho^x = x + \mathcal{N}_\rho$ is the neighborhood of site $x \in \mathbf{Z}^2$. $\mathcal{N}_\rho$ is called the *generalized Moore* or “*range ρ ” box neighborhood. Each *time step*, all of the sites *update* (meaning change states or not) simultaneously according to the deterministic LtL *rule*, which in words is:*

Birth: A site that is dead at time t will become live at time $t + 1$ if and only if the number of live sites in its neighborhood at time t is in the closed interval $[\beta_1, \beta_2]$, $0 \leq \beta_1 \leq \beta_2$.

Survival: A site that is live at time t will remain live at time $t + 1$ if and only if the number of live sites in its neighborhood (itself included) at time t is in the closed interval $[\delta_1, \delta_2]$, $1 \leq \delta_1 \leq \delta_2$.

Death: A site that is dead at time t and does not become live at time $t + 1$ will remain dead at time $t + 1$. A site that is live at time t and does not remain live at time $t + 1$ will become dead at time $t + 1$.

Let $\mathcal{T}$ denote the LtL rule. That is, $\mathcal{T} : \{0, 1\}^{\mathbf{Z}^2} \mapsto \{0, 1\}^{\mathbf{Z}^2}$. Let $\xi_t(x) \in \{0, 1\}$ denote the state of the site $x \in \mathbf{Z}^2$ at time t and let ξ_t represent the state of all sites in $\mathbf{Z}^2$ at time t . As is customary, we will often think of the CA as a set-valued process, confounding ξ_t with $\{x : \xi_t(x) = 1\}$. The notation $\xi_t^A = \mathcal{T}^t(A) = A_t$ thus means that starting from configuration $\xi_0 = A$ we arrive at the set A_t of occupied sites after t iterations of rule $\mathcal{T}$. In range ρ , the LtL update rule is: $\mathcal{T}(A) = \{x \in A^c : \beta_1 \leq |\mathcal{N}_\rho^x \cap A| \leq \beta_2\} \cup \{x \in A : \delta_1 \leq |\mathcal{N}_\rho^x \cap A| \leq \delta_2\}$.

For each fixed range ρ , the LtL CA rules form a four-parameter family indexed by the endpoints β_1 and β_2 of the birth intervals and the endpoints δ_1 and δ_2 of the survival intervals. We denote each rule by the 5-tuple $(\rho, \beta_1, \beta_2, \delta_1, \delta_2)$. In this framework, Life has LtL parameters $(1, 3, 3, 3, 4)$. (A live site counts itself and so the survival interval is $[3, 4]$ rather than the more standard $[2, 3]$.)

2.2 Coherent Structures

Part of Life's intrigue is due to the numerous coherent structures that emerge when the rule updates starting from a random initial configuration. These local structures generalize to LtL rules with arbitrarily large ranges [7]. Let us define LtL's “bugs”. For the following, let $\mathcal{T}$ be a CA rule from the LtL family and let $A \subset \mathbf{Z}^2$ be a set of sites in state 1.

Definition 1. A bug is a finite configuration, Λ for which there exists a finite time, τ , and a nonzero displacement vector, $\mathbf{d} = (d_1, d_2)$, such that $\mathcal{T}^\tau(\Lambda) = \Lambda + \mathbf{d}$. The smallest such τ is a bug's period, mod translation, in the direction of $\mathbf{d}$. A bug has τ distinct phases: $\Lambda = \Lambda_0, \Lambda_1, \dots, \Lambda_{\tau-1}$.

LtL's bugs are generalizations of Life's famous spaceships. We consider bugs to be spaceships supported by rules whose neighborhoods are in ranges 2 and higher because the geometry of large range bugs is reminiscent of various insects; however, "bug" is essentially synonymous with spaceship [9].

Bugs are characterized according to their trajectories. Some bugs move horizontally or vertically; we call these orthogonal bugs and their displacement vectors $\mathbf{d} = (d_1, d_2)$ have exactly one component equal to 0. Others move along a diagonal and are thus called diagonal bugs; their displacement vectors satisfy $d_1 = \pm d_2$. Bugs whose trajectories are not horizontal, vertical, or diagonal are said to be disoriented; in the Life Lexicon, these are called "knightships" [9].

Example 1. Life's lightweight spaceship (LWSS) is its smallest orthogonal spaceship and was discovered by Conway in 1970. In LtL terminology, LWSS is an orthogonal bug with period $\tau = 4$, displacement vector $\mathbf{d} = (2, 0)$, and supporting rule (1,3,3,3,4) (Fig. 1).



Fig. 1. Life's LWSS at times $13k$, $k=0, 1, 2, \dots, 11$ (left to right).

Example 2. Life's glider is its most common spaceship and the first ever found, discovered by Richard Guy in 1970. In LtL terminology, the glider is a diagonal bug with period $\tau = 4$, displacement vector $\mathbf{d} = (1, 1)$, and supporting rule (1,3,3,3,4) (Fig. 2).



Fig. 2. Life's glider at times $13k$, $k = 0, 1, 2, 3, 4$ (lower left to upper right).

Example 3. Bosco's rule (5,34,45,34,58) supports numerous orthogonal, diagonal, and disoriented bugs. The most commonly occurring orthogonal bug has period $\tau = 6$ and displacement vector $\mathbf{d} = (5, 0)$. Its trajectory is shown in Fig. 3. The trajectory of one of Bosco's rule's diagonal bugs is shown in Fig. 4 and one of its disoriented bugs is shown in Fig. 5.



Fig. 3. Orthogonal bug supported by Bosco's rule at times $17k$, $k = 0, 1, 2, \dots, 11$.

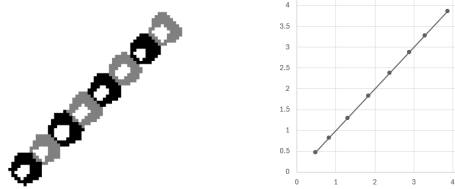


Fig. 4. Bosco's diagonal bug with $\tau = 16$ and $\mathbf{d} = (8, 8)$ at times $13k$, $k = 0, 1, 2, \dots, 15$ (left) and centroids at times 0 to 15 (right).

Most orthogonal, diagonal, and disoriented bugs have trajectories along a straight line. Some bugs follow trajectories that are obviously nonlinear, while others move in ways that are much harder to recognize visually. In these cases, scripting becomes especially useful for detecting and analyzing their trajectories.



Fig. 5. Bosco's disoriented bug with $\tau = 4$ and $\mathbf{d} = (2, 1)$ at times $13k$, $k = 0, 1, 2, \dots, 8$ (left) and centroids at times 0 to 7 (right).

The first bug we found that had an obviously nonlinear trajectory is supported by LtL rule $(4, 22, 31, 24, 38)$. The bug moves up and down, nearly vertically, only moving 4 cells to the right every 40 time steps. Due to the spatial overlap of its phases A_t for $t = 0, 1, 2, \dots, 40$, it is difficult to accurately represent its trajectory as a fixed image. The output from our script, `JitterFactorAvg`, provides a better representation of the bug's trajectory, shown to the right in Fig. 6. Let us define a jitterbug and its jitter factor and then our `JitterFactor` Lua scripts, which will allow us to compare jitter factors for various examples.

Definition 2. Let A be a bug, with supporting rule $\mathcal{T}$ and period τ . Let l be the line that passes through the centroid of A and the centroid of $\mathcal{T}^\tau(A)$. Line l is said to be the bug's trajectory line. A is a jitter bug if and only if there is at least one integer k , $0 < k < \tau$, such that line l does not pass through the centroid of $\mathcal{T}^k(A)$.

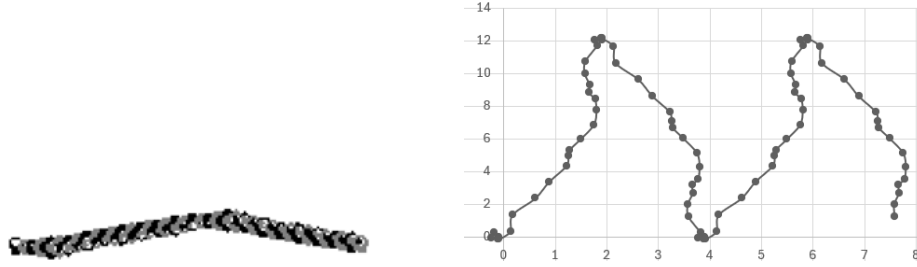


Fig. 6. Times $41k$, $k = 0, 1, 2, \dots, 40$ of the trajectory of orthogonal jitterbug with period $\tau = 40$ and $\mathbf{d} = (4, 0)$ supported by LtL rule $(4, 22, 31, 24, 38)$ (left) and centroids at times 0 to 80 (right).

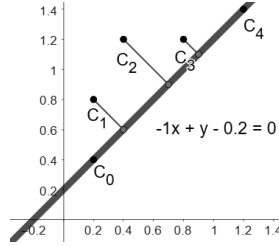


Fig. 7. The glider's trajectory line is shown in black and passes through the glider's centroids, C_0 and C_4 . Line l does not pass through the glider's other centroids, C_1, C_2, C_3 .

Example 4. Life's glider is a jitterbug. This can be seen in Figure 7. The glider's centroid at time t is denoted by C_t for $t = 0, 1, 2, 3, 4$. The glider's trajectory line, $l : y = x + 0.2$ is shown in black and passes through C_0 and C_4 . The glider's other centroids, C_1, C_2 , and C_3 do not lie on line l .

3 Jitter Factor Definition, Script, and Calculations

To quantify the deviation, or "jitter" a bug exhibits from its trajectory line, we define a scalar metric termed the *jitter factor*. The jitter factor measures how the centroids of a bug's $\tau - 1$ distinct phases deviate from its trajectory line.

The following computations are implemented via Lua scripts executed within Golly, which employs a statically embedded Lua interpreter (version 5.4.6) to run `.lua` scripts.

The centroid of a given pattern at time step t is defined as:

$$C_t = \left(\frac{1}{n_t} \sum_{i=1}^{n_t} x_i, \frac{1}{n_t} \sum_{i=1}^{n_t} y_i \right), \quad (1)$$

where n_t is the total number of live cells at time t , and (x_i, y_i) are the coordinates of each live cell.

For each phase of the given bug, a trajectory line is first established. This line is determined by the centroid at time $t = k$, denoted $C_k = (x_k, y_k)$, and the centroid at the end of one full period, denoted $C_{k+\tau} = (x_{k+\tau}, y_{k+\tau})$, for $k = 0, 1, 2, \dots, \tau - 1$. That is, the trajectory line, l_k is given by:

$$y - y_k = m(x - x_k), \text{ where } m = \frac{y_{k+\tau} - y_k}{x_{k+\tau} - x_k}, \text{ for } k = 0, 1, 2, \dots, \tau - 1 \quad (2)$$

If $x_{k+\tau} = x_k$, then the line is vertical and we measure the distance simply as horizontal displacement from x_k .

Given the centroid at each time step $C_t = (x_t, y_t)$, the perpendicular distance d_{kt} , from this point to the established trajectory line, l_k , is:

$$d_{kt} = \frac{|mx_t - y_t + b|}{\sqrt{m^2 + 1}}, \text{ where } b = y_k - mx_k \quad (3)$$

and in the case of a vertical trajectory line, where m is undefined, this reduces to:

$$d_{kt} = |x_t - x_k| \quad (4)$$

The jitter factor, J_k of phase k of the bug (Λ_k) over one period τ is then calculated as the mean absolute deviation from the trajectory line across each time step within that period:

$$J_k = \frac{1}{\tau - 1} \sum_{t=k+1}^{k+\tau-1} |d_{kt}|, \text{ for } k = 0, 1, 2, \dots, \tau - 1 \quad (5)$$

Our script computes the jitter factors as follows. For each k , $0 \leq k < \tau - 1$:

1. Initialize the pattern at time k ; this is phase k of the bug, denoted by Λ_k .
2. Compute the jitter factor J_k over the subsequent period $[k, k + \tau - 1]$.

This yields τ jitter factors $J_0, J_1, \dots, J_{\tau-1}$. The script then computes their arithmetic mean, the average jitter factor, J_{avg} , which is independent of the bug's phase:

$$J_{avg} = \frac{1}{\tau} \sum_{k=0}^{\tau-1} J_k \quad (6)$$

The resulting values, which are: the centroid coordinates at each time step, the trajectory line for Λ_0 , distances from the trajectory line, jitter factors for each phase of a bug, the average jitter factor, and the period, are systematically recorded into CSV files. This facilitates comprehensive subsequent analysis, and visualization.

The above is incorporated into the Lua script `JitterFactorAvg`. Another script, `JitterFactor`, outputs only the bug's period τ , displacement vector $\mathbf{d}$, and jitter factor for the current phase of a bug. `JitterFactor` is useful for initial

testing since, if there exists a k , $0 \leq k < \tau - 1$ such that $J_k \neq 0$, then $J_{avg} \neq 0$. That is, if $J_k \neq 0$, the bug is a jitterbug. JitterFactorAvg provides a CSV file of the bug’s centroids for the number of cycles, n , requested by the user. We also created, and used here, a Trail script to output the trajectory of a bug at phases selected by the user [10].

Example 5. The Trail script created the “ghost-trail” and JitterFactorAvg computed the centroids for the period 66 bug with $\mathbf{d} = (40, 0)$ shown in Fig. 8.

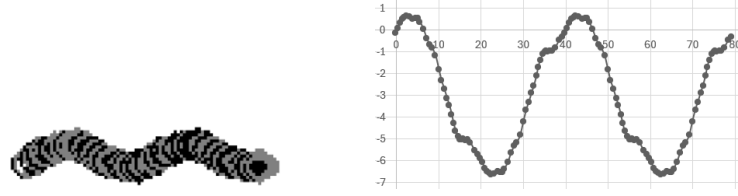


Fig. 8. Orthogonal jitterbug supported by LtL rule (5,34,44,34,58), showing “ghost-trail” trajectory for times 0 to 131 (left) and centroids (right).

Computing bugs’ jitter factors allows us to compare them quantitatively; we did this for the bugs discussed above (see Table 1). The range 4 bug shown in Fig. 6 was a record holder for nonlinearity until LtL was first implemented in Golly, when we immediately discovered “Jadyn’s jitterbug” (Fig. 9), which significantly surpasses the previous record.

Table 1. Bugs’ jitter factors, computed by the JitterFactorAvg script.

Bug’s: name, Fig.	$(\rho, \beta_1, \beta_2, \delta_1, \delta_2)$	J_{avg} (to 4 decimal places)
Life’s LWSS 1	(1, 3, 3, 3, 4)	0.5370
Life’s glider, 2	(1, 3, 3, 3, 4)	0.2357
Bosco’s orthogonal bug, 3	(5, 34, 45, 34, 58)	0
Bosco’s diagonal bug, 4	(5, 34, 45, 34, 58)	0
Bosco’s disoriented bug, 5	(5, 34, 45, 34, 58)	0.0646
Range 4 jitterbug, 6	(4, 22, 31, 24, 38)	4.7747
Range 5 jitterbug, 8	(5, 34, 44, 34, 58)	2.9170
Jadyn’s jitterbug, 9	(76,4565,5897,4565,6580)	≈ 6000

4 Conclusion

We described Lua scripts designed to identify and analyze LtL’s jitterbugs. Our JitterFactorAvg script performs well for small ranges, but crashes when processing large ones. A separate version of the script was developed for large ranges, but its performance is so slow, it took more than two days to compute the

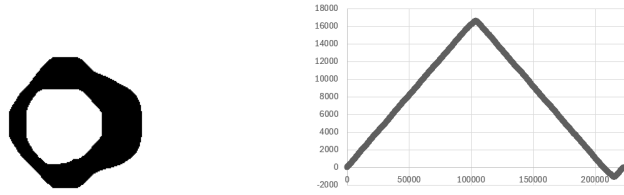


Fig. 9. Jady's jitterbug (left) is supported by LtL rule (76, 4565, 5897, 4565, 6580), has period 22286, and $\mathbf{d} = (222968, 0)$; its trajectory is also shown on the right.

centroids for Jady's jitterbug. We expect parallelizing computations to reduce runtime substantially.

Using Golly to study LtL has been transformative. It has enabled us to address questions that were previously inaccessible and many open questions remain, including: which LtL rules support jitterbugs? If a rule supports a bug, does it also always support a jitterbug? Life's gliders and LWSS are key ingredients in the proof that Life is universal. Are jitterbugs supported by other LtL rules similarly useful? Through scripting and automation, a standard laptop can function as a computational lab for generating and analyzing data, making this work broadly accessible. Beyond its scientific value, these methods provide a scalable foundation for studying more realistic and applied models with significant practical and societal relevance.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Griffeath, D.: *Self-organization of random cellular automata: Four snapshots*, in "Probability and Phase Transitions", 1994.
2. Evans, K.: *Larger than Life: digital creatures in a family of two-dimensional cellular automata*, 2001. Discrete Mathematics and Theoretical Computer Science, Volume AA, 2001, 177-192;
3. E. Berlekamp, J. Conway, and R. Guy.: *What is Life?*, in "Winning Ways for Your Mathematical Plays", 1982. Volume 2, Chapter 25, Academic Press, New York.
4. P. Bradbury, Life in Life, <https://www.youtube.com/watch?v=xP5-iLeKXE8>, accessed 2026/5/14
5. Evans, K.: Is Bosco's Rule universal? In "Lecture Notes in Computer Science," ed. Maurice Margenstern. 188-199. Springer-Verlag GmbH, Vol. 3354 (2005).
6. Evans, K.: "*Larger than Life: it's so nonlinear*", 1996. Ph.D. dissertation, University of Wisconsin - Madison.
7. Evans, K.: *Larger than Life: threshold-range scaling of Life's coherent structures*, Physica D: Nonlinear Phenomena, Vol. 183 (2003) 45-67.
8. Golly software, <https://golly.sourceforge.io/>, accessed 2026/05/06
9. Silver, S.: Life Lexicon Home Page, <https://conwaylife.com/ref/lexicon/lex.htm>
10. Ismalej, B., *LTL-GollyScripting*. GitHub repository. <https://github.com/Brandon-Ism/LTL-GollyScripting>, accessed 2026/05/21