

Auto-Stack

Automatic Detection and Resizing of Repeating Structures in Voxel Builds

Nucleation resizer — design notes

June 15, 2026

Abstract

Auto-Stack takes a Minecraft build—a redstone circuit, a display, or a mechanical contraption such as a limestone flying machine—detects the repeating unit it is built from, and lets the user change how many times that unit repeats, producing a new build. A 4-bit adder becomes 8-bit; a 32×32 screen becomes 64×64 ; an N -module bore head becomes $2N$. The whole system reduces to one idea: a build is (locally) invariant under a lattice translation $\mathbf{v}$, and resizing means re-stamping its fundamental domain a different number of times along $\mathbf{v}$. This document describes how the period $\mathbf{v}$ is found, how the resize is performed for axis-aligned, diagonal, and two-dimensional lattices, how the repeating cell is made self-contained and re-stackable, and how non-block data (entities) is carried along.

Contents

1	The core idea	2
2	Two levels of abstraction	2
3	Finding the period	2
3.1	From the graph: the invariance test	2
3.2	From the voxels: self-overlap	3
4	Resizing: lattice-vector stamping	4
5	Two-dimensional builds: displays	4
6	Self-contained cells: support-aware extraction	6
7	Carrying entities	7
8	Implementation and the web tool	7
9	The library API	8
10	Scope and limitations	8
	Summary	8

1 The core idea

A build is a finite set of placed blocks

$$G = \{ (\mathbf{p}, b) : \mathbf{p} \in \mathbb{Z}^3, b \in \mathcal{B} \},$$

mapping integer voxel positions to block states b (a namespaced id plus properties such as a comparator’s **facing** or a repeater’s **delay**). A region of the build is *periodic* with period $\mathbf{v} \in \mathbb{Z}^3$ if translating it by $\mathbf{v}$ maps it onto itself:

$$(\mathbf{p}, b) \in G \iff (\mathbf{p} + \mathbf{v}, b) \in G \quad (\text{away from the boundary}).$$

The period $\mathbf{v}$ need not be axis-aligned. A carry-chain adder whose bit-cells march diagonally has $\mathbf{v} = (2, -2, 0)$; a vertically stacked datapath has $\mathbf{v} = (0, 2, 0)$; a flat display is periodic along *two* independent vectors $\mathbf{v}_1, \mathbf{v}_2$. Resizing is then mechanical: keep the irregular boundary, and replace the periodic interior with however many copies of one period the user asks for. Everything else in the system exists to (a) discover $\mathbf{v}$ robustly, (b) cut the build into period-cells correctly, and (c) re-stitch the cells so the result is a valid, working build.

2 Two levels of abstraction

The hard part of finding $\mathbf{v}$ is that the right notion of “repeating unit” depends on what kind of build it is.

Computational redstone (adders, displays). The functional structure is a graph of redstone components. Two cells can be logically identical while their voxels differ—signal-boosting repeaters are inserted at intervals, decoder taps vary per row. Here the period must be found on the *compiled graph*, not the voxels.

Structural / mechanical (flying machines, farms). These are mostly slime, honey, pistons, and observers—blocks the redstone compiler barely models, so the graph is nearly empty and useless. But the build’s *voxels* are literally periodic: the bore head repeats as a block pattern. Here the period is found directly on the voxels.

The system routes between the two with a cheap test on the block palette. Let $\mathcal{C}$ be the set of “computational” blocks (wire, comparator, repeater, torches, lamp, lever, button). Define the compute fraction

$$\phi = \frac{|\{(\mathbf{p}, b) \in G : b \in \mathcal{C}\}|}{|G|}.$$

If $\phi < 0.35$ the build is treated as *structural* and the period is detected from voxels (§3.2); otherwise it is *computational* and the period is detected from the graph (§3.1). A flying machine has $\phi \approx 0$; an adder has $\phi \approx 0.95$.

3 Finding the period

3.1 From the graph: the invariance test

For computational builds we extract the *structural* compile graph—a pre-fold graph in which every component (including each redstone-wire segment) is its own node with a recorded voxel position $\text{pos}(n)$ and a kind $\text{kind}(n) \in \{\text{Wire, Comparator, Repeater, Torch, Lamp, } \dots\}$.

Structural labels. Each node is given a Weisfeiler–Lehman label that captures its local role. Starting from $L_0(n) = \text{kind}(n)$, we refine for a few rounds,

$$L_{r+1}(n) = \text{hash}\left(L_r(n), \{\{L_r(u) : u \rightarrow n\}\}, \{\{L_r(w) : n \rightarrow w\}\}\right),$$

where $\{\{\cdot\}\}$ is a sorted multiset over in- and out-neighbours. Nodes that play the same role in different bit-positions receive the same label.

Candidate direction. The lattice direction is the dominant displacement between equal-label nodes. We tally $\mathbf{p}(m) - \mathbf{p}(n)$ over all pairs m, n with $L(m) = L(n)$, take the most frequent nonzero vector, and reduce it by the gcd of its components to a primitive direction $\mathbf{d}$. This works for diagonal lattices: a displacement $(2, -2, 0)$ reduces to $\mathbf{d} = (1, -1, 0)$.

The invariance test. The dominant displacement is often a *sub-lattice* generator, not the true cell period. In a comparator-based adder the comparators repeat every 2 blocks, but a full bit-cell—comparators *and* the sparser output layer—repeats every 4. To find the true period we test invariance directly. For a candidate $\mathbf{v} = m\mathbf{d}$, bucket every node by its phase along $\mathbf{v}$,

$$k(n) = \left\lfloor \frac{(\mathbf{p}(n) - \mathbf{o}) \cdot \mathbf{v}}{\|\mathbf{v}\|^2} \right\rfloor, \quad (1)$$

and let H_k be the histogram of kinds in bucket k . Then

Definition 1. $\mathbf{v}$ is a period if there is a run of at least three consecutive interior buckets with identical histograms, $H_k = H_{k+1} = \dots$.

We take the smallest multiple $m = 1, 2, 3, \dots$ of $\mathbf{d}$ that passes. This rejects the comparator sub-lattice (its buckets alternate $H_k \neq H_{k+1}$) and returns the genuine bit-cell period. Two implementation details matter: the bucketing in (1) uses *floor* division so each bucket is a contiguous fundamental domain (rounding would split a cell across two buckets), and the histogram comparison uses the coarse *kind* label, not the full WL label, because boundary effects make every node’s WL label eventually unique.

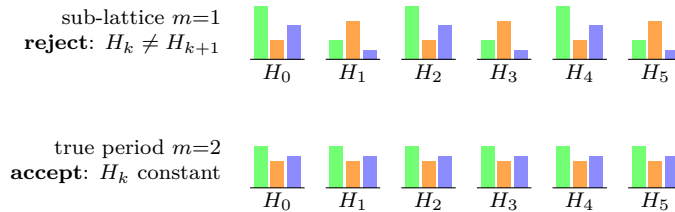


Figure 1: The invariance test. Each bar group is the kind-histogram H_k of phase bucket k . A candidate that is only a sub-lattice generator (top: the comparator step) produces histograms that alternate, so it is rejected; the smallest multiple whose interior histograms are constant (bottom: the full bit-cell) is the true period.

3.2 From the voxels: self-overlap

For structural builds there is no useful graph, so we find $\mathbf{v}$ directly. For each axis and each candidate period T along it, slide the build by $\mathbf{v} = T\hat{\mathbf{e}}_{\text{axis}}$ and measure how many overlapping voxels match:

$$\text{score}(\mathbf{v}) = |\{\mathbf{p} : (\mathbf{p}, b) \in G, (\mathbf{p} + \mathbf{v}, b) \in G, \text{ same id}\}|.$$

The true period *maximizes* the number of identical matches; a spurious short shift matches only a handful of blocks. We keep the highest-scoring $\mathbf{v}$ whose match ratio exceeds 90%, breaking ties toward the shorter period. On the 3×7 limestone bore this returns $\mathbf{v} = (6, 0, 0)$ at 97% self-overlap—the bore module—with no graph involved at all.

4 Resizing: lattice-vector stamping

Once $\mathbf{v}$ is known, the resize is a single mechanism that handles axis-aligned and diagonal periods uniformly. Axis alignment is merely the special case where $\mathbf{v}$ has one nonzero component.

Cells. Using the phase (1), assign every voxel to a cell k and record its *local* coordinate

$$\ell(\mathbf{p}) = \mathbf{p} - k(\mathbf{p}) \mathbf{v}.$$

Two voxels related by $+\mathbf{v}$ get the same local coordinate, so for a periodic region all interior cells share an identical local block set. Let C_k be cell k ’s map from local coordinate to block.

Phase alignment. The cell boundaries depend on where we place the phase origin. We sweep the phase offset over one period and pick the alignment that maximizes the longest run of byte-identical consecutive cells. That run is the periodic interior; the cells before it are the *head* (e.g. an adder’s input row, a machine’s engine) and the cells after it the *tail* (the output cap, the bore’s leading edge).

Reassembly. To resize the interior to N units, concatenate

$$\underbrace{C_0, \dots, C_{s-1}}_{\text{head}}, \underbrace{C_s, C_s, \dots, C_s}_{N \text{ copies of the unit}}, \underbrace{C_{s+r}, \dots}_{\text{tail}}$$

where s is the run start and r its original length, and stamp the i -th cell in this sequence at offset $i \mathbf{v}$:

$$G' = \{ (\ell + i \mathbf{v}, b) : (\ell, b) \in (i\text{-th cell}), i = 0, 1, 2, \dots \}. \quad (2)$$

Because the unit cell’s blocks are byte-identical and placed at consecutive multiples of $\mathbf{v}$, adjacent stamped cells connect exactly as the original interior did—the carry chain, the slime sheet, the bus all continue across the seam. For a diagonal $\mathbf{v}$ the tail simply slides diagonally; no special case is needed.

Proposition 1 (Faithfulness). *Resizing to the original unit count ($N = r$) reproduces the input exactly.*

This identity check is the system’s main correctness guarantee: on the diagonal adder, $N = 7$ reproduces all 634 blocks; on the 32^2 frame buffer, 7×7 reproduces all 24906.

5 Two-dimensional builds: displays

A flat display is periodic along two independent vectors $\mathbf{v}_1, \mathbf{v}_2$ (say the screen’s Y and Z axes), with the circuit depth X aperiodic. Detection runs the per-axis test on all three axes; if two axes are periodic we must still decide whether it is a genuine 2D grid or merely a 1D diagonal lattice seen in projection—a diagonal $\mathbf{v}$ shows a period on *both* of the axes it spans.

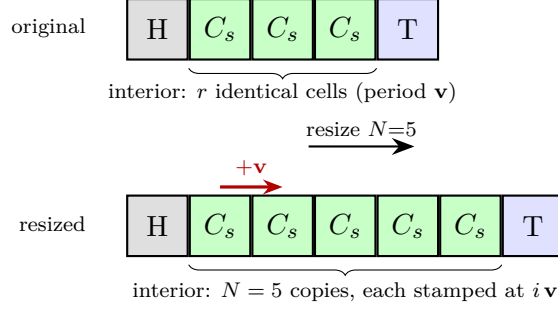


Figure 2: The resize is head + N copies of the unit cell + tail, each cell stamped at a successive multiple of the period $\mathbf{v}$. The head (e.g. an adder’s input row, a machine’s engine) and tail (output cap, leading edge) are kept verbatim; only the interior count changes.

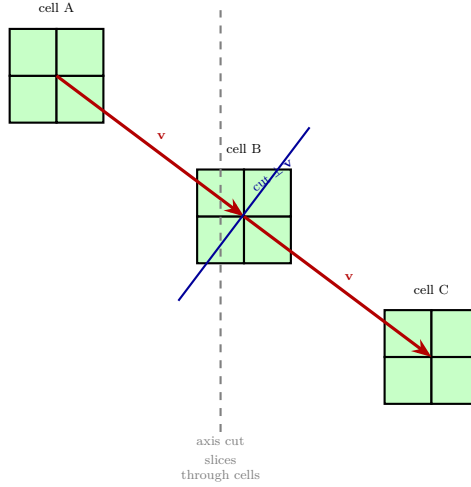


Figure 3: A diagonal carry-chain adder: bit-cells step by $\mathbf{v} = (2, -2, 0)$. Bucketing by phase along $\mathbf{v}$ (the blue cut, perpendicular to $\mathbf{v}$) separates the cells cleanly; an axis-aligned slab (gray) cuts diagonally through every cell and shears the carry chain. The same stamping that handles a vertical period $(0, 2, 0)$ handles this one—axis alignment is just the case where $\mathbf{v}$ has a single nonzero component.

The grid-fill guard. Bucket voxels into a 2D grid of cells (i, j) by phase along $\mathbf{v}_1$ and $\mathbf{v}_2$, and let the modal cell signature occur M times across an $I \times J$ grid. A true 2D grid fills a rectangle, so M is on the order of $I \cdot J$; a diagonal 1D lattice’s cells lie on a line $i + j = \text{const}$, so $M \approx \max(I, J)$. We declare 2D only when

$$M \geq 2 \max(I, J).$$

This correctly keeps the diagonal adder one-dimensional while admitting the frame buffer ($M = 180$ on a 19×18 grid).

Nine-slice tiling. Resizing a screen to $M_1 \times M_2$ interior tiles is the 2D analogue of head/interior/tail: the four *corners* stay fixed, the four *edges* (frame strips) scale 1D along the dimension they run, and the *interior* tiles in 2D. All nine regions follow one rule. Define a per-axis remap from a

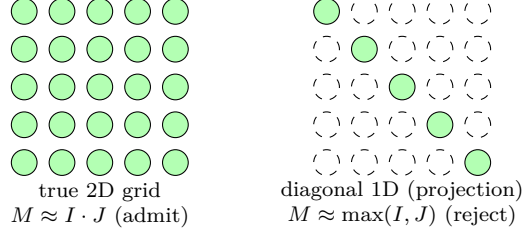


Figure 4: The grid-fill guard. Counting how many cells share the modal signature distinguishes a genuine 2D display (cells fill a rectangle) from a 1D diagonal lattice whose cells, projected onto two axes, lie on a line $i + j = \text{const}$. We declare 2D only when $M \geq 2 \max(I, J)$.

new index to the original cell index it copies,

$$\rho(i') = \begin{cases} i' & i' < s \quad (\text{head: identity}) \\ s & s \leq i' < s + M \quad (\text{interior: the unit}) \\ s + r + (i' - s - M) & \text{otherwise} \quad (\text{tail}). \end{cases}$$

Then a new cell (i', j') copies the original cell $(\rho_1(i'), \rho_2(j'))$, stamped at $i'\mathbf{v}_1 + j'\mathbf{v}_2$. Corners hit (head, head), edges hit (interior, boundary), and the interior hits (interior, interior)—one expression covers them all.

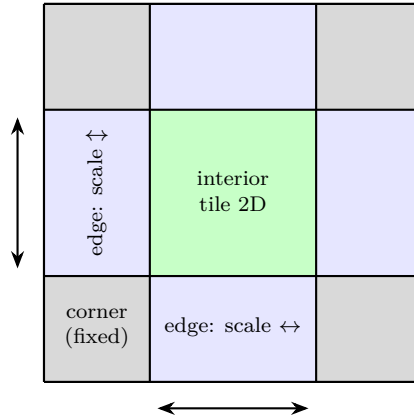


Figure 5: Nine-slice resize of a 2D display. The four corners are fixed; the top and bottom edges scale horizontally and the left/right edges vertically (1D); the interior tiles in both directions. The per-axis remap ρ_1, ρ_2 selects, for each new cell, the original cell to copy—one rule for all nine regions.

6 Self-contained cells: support-aware extraction

Resizing only needs the period and the byte-identical slabs. Extracting a single cell as a standalone, *functional* build is harder, because a graph node's voxel is not enough: a comparator rests on a block, a wall torch hangs off one, and a wire is supported from below. Cutting a cell on a raw coordinate window slices through these dependencies.

The fix is to grow the cell's voxel set by each component's *functional coupling*, which in redstone is directional, not a symmetric neighbourhood:

component	structural blocks pulled in
wire	support below; the horizontal neighbours it connects to
comparator / repeater	support below; the block it faces (output) and its back/side inputs
torch / wall torch	support or attachment block; the block it powers <i>above</i>
lamp	none (self-supporting)

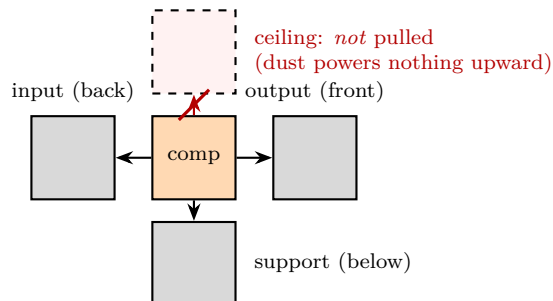


Figure 6: Functional coupling is directional. A comparator’s cell pulls in the support block below it and the blocks it faces (output) and reads from (input)—but *not* the block above. A symmetric 6-neighbour ball would wrongly drag in a ceiling.

Crucially this is *not* a 6-neighbour ball: dust does not power the block above it, so a naive ball wrongly drags in a “ceiling.” With directional coupling the extracted cell carries exactly the solids it depends on, all block states are preserved losslessly, and the cell tiles back onto the original 1:1 under $\pm\mathbf{v}$ shifts. Cross-cell blocks pulled in by the dilation are duplicated harmlessly: stacking places identical blocks at identical coordinates.

7 Carrying entities

Some builds place *entities*—a limestone bore drives instant block updates with minecarts riding detector rails. Entities are not blocks; they have floating-point positions and live outside the voxel grid. They tile with the same lattice: each entity at world position $\mathbf{p}_e$ is assigned to cell $k = \lfloor (\mathbf{p}_e - \mathbf{o}) \cdot \mathbf{v} / \|\mathbf{v}\|^2 \rfloor$, and in the reassembly (2) the entities of the cell copied into sequence slot i are re-emitted at $\mathbf{p}_e + (i - k)\mathbf{v}$. Boundary entities appear once; unit-cell entities are replicated per stamped module. On the bore, three minecarts at $x = 1.5, 7.5, 13.5$ become $N+1$ carts on the extended rails, spacing preserved.

8 Implementation and the web tool

The detector and resizer are a single Python module (`resize_lib.py`) built on the Nucleation library, which parses `.schem/.litematic` files, exposes the structural compile graph, and meshes builds to textured glTF. A dependency-free `http.server` front end lets the user drag in a file; the back end returns circuit properties, a textured GLB preview (rendered in the browser with `three.js`), and a resize control—one slider for 1D builds, two for 2D displays. Resizing returns a downloadable `.schem`.

A note on robustness. Real files are messy. The bore litematic labels its slime blocks `minecraft:slime`—the slime *mob*’s id—rather than `minecraft:slime_block`; with no blockstate for that id the mesher

falls back to the entity model and renders mobs. The tool normalizes such legacy/entity-colliding ids before meshing so the preview and the resized output both show the correct block.

9 The library API

The pure-voxel core of this system—region-coverage detection (§5) and lattice-vector resizing (§4)—is implemented in the Nucleation Rust crate as the `autostack` module, behind a feature flag of the same name, and exposed identically across the Python, WebAssembly, and C/FFI bindings (checked by an API-parity tool). Detection returns a ranked list of `Structure` descriptors—mode (1d/2d), period vector(s), coverage, region bounding box, unit-cell bounding box, and a label; resizing takes a structure (or its period vectors) plus the new cell count(s) and returns a new schematic.

operation	Rust core	Python / WASM / FFI
detect	<code>detect_structures(&s)</code>	<code>detect_structures()</code>
resize 1D	<code>resize_1d(&s, v, n)</code>	<code>autostack_resize_1d(vx,vy,vz,n)</code>
resize 2D	<code>resize_2d(&s, v1, v2, n1, n2)</code>	<code>autostack_resize_2d(...)</code>

The Rust implementation is validated byte-for-byte against the reference prototype on real builds: identical detected structures and identical resized block sets for the 1D adder and the 2D screen. The advanced capabilities (graph-based diagonal detection, near-periodic / booster resize, simulation-based verification, support-aware single-cell extraction) remain in the prototype and layer on behind the simulation feature.

10 Scope and limitations

- **Phase-projection vs. true interlock.** Bucketing by phase along $\mathbf{v}$ handles diagonal lattices and cells that snake in the plane perpendicular to $\mathbf{v}$. It does *not* handle cells that interlock *along* $\mathbf{v}$ itself; that would require assigning cell membership by following the graph’s translation map combinatorially rather than by geometric projection.
- **Near-periodic infrastructure.** Real displays are periodic at the graph level but not byte-identical at the voxel level: signal boosters are inserted by cumulative distance (with a heavier re-drive at the centre), and address decoders grow as a $\log N$ demux tree. Pure tiling reproduces the pixel array but not this infrastructure; resizing such a screen to a working larger one needs parametric *regeneration* of the boosters and decoder, which is circuit synthesis rather than tiling.
- **Functional verification.** The system guarantees the resized build tiles byte-identically; it does not (currently) simulate the result to confirm the carry propagates or the machine still flies. For clean lattices tiling implies correctness, but this is inferred, not proven.
- **Transparency.** The preview matches Nucleation’s native renderer (alpha-blended, double-sided, no depth write), which leaves cross-material translucent z-ordering (e.g. slime against water) imperfect without order-independent transparency.

Summary

The entire system is one equation applied at two abstraction levels. Find the translation $\mathbf{v}$ under which the build is invariant—on the compile graph for circuits (via an invariance test that rejects

sub-lattices), on the raw voxels for machines (via self-overlap). Cut the build into cells by phase along $\mathbf{v}$, keep the boundary, and re-stamp the interior cell N times at multiples of $\mathbf{v}$. The same mechanism resizes a vertically stacked adder, a diagonal carry-chain adder, a two-dimensional display, and a limestone flying machine, carrying support blocks and minecarts along with it.