PAL Guide

This guide starts with a working model, then explains PAL's components, behavior, constraints, and specialized operations. Begin with the component overview and work through the tutorial in order. For current public signatures and method descriptions, use the API Reference. For quick syntax, use the Cheatsheet. For implementation structure and design rationale, see PAL Architecture and Design.

Installation

Install PAL from PyPI:

```
python -m pip install PythonAutomataLibrary
```

Visualization dependencies are included in the standard installation. To copy the bundled example programs into an editable folder, run:

```
python -m PythonAutomataLibrary.ExampleInstaller
```

This creates `PAL-Examples/` in your current directory. To select a different parent directory, use `python -m PythonAutomataLibrary.ExampleInstaller /path/to/directory` (or a quoted Windows path). The directory must already exist, and an existing `PAL-Examples/` folder will never be overwritten. The examples and documentation PDFs are bundled with the installation; the generated folder includes `Documentation/` with the PDF guides, and generating it requires no network connection.

Then import it with `import PythonAutomataLibrary as pal`. Prebuilt platform wheels are intended to include the native C core, so users do not need to compile it separately. For source builds and release details, see Distribution.

Choose the right PAL structure

Need	PAL structure
One typed value per lattice site	`Grid`
Individual agents with identity/properties	`AgentGrid`
Integer counts without individual identity	`PopGrid`
Continuous diffusing/advecting field	`PDEgrid`
Reusable integer query result	`IList`
Binomial/multinomial draws	`Multinomial`

You can combine these components: for example, cells in an `AgentGrid` can consume a nutrient represented by a `PDEgrid`. Some components accumulate changes until `Update()` applies them together; the timing of those updates matters.

Build a cell model, one feature at a time

The tutorial develops **one simple model** of cells on a two-dimensional lattice. We place and draw cells, move them into empty neighboring sites, compile the movement rule, add a boundary-fed nutrient field, and finally represent cells as population counts. The biological premise stays the same so each stage can focus on a new PAL feature.

Each numbered stage has a matching **complete, runnable program** in `Examples/Tutorial/`, named `01_PlaceCells.py` through `07_PopulationCounts.py`. After installing PAL, run `python -m PythonAutomataLibrary.ExampleInstaller` (or `python -m PythonAutomataLibrary.ExampleInstaller /path/to/parent`) to copy them into `PAL-Examples/Tutorial/`. For example, run `python PAL-Examples/Tutorial/01_PlaceCells.py` from the destination parent directory. The numbered files accumulate the changes described below, so you can run any stage without manually assembling snippets. Blocks marked **complete program** run independently when saved as Python files; other snippets modify the preceding program. The detailed API sections remain available for complete behavior and less common features.

PAL begins in safe mode, which checks operations for common errors. The tutorial uses this default. Later, the performance section explains how to select fast mode and compile frequently executed code.

1. Place cells on a lattice

An `AgentGrid` stores identifiable cells and their positions. We choose a nonstackable 30 × 30 lattice: at most one cell can occupy each site. A `Grid`, by contrast, stores one typed value per site and does not create individual cell identities.

Complete program — initial cells:

```
import PythonAutomataLibrary as pal

SIZE = 30

def Setup(cells: pal.AgentGrid):
    for x, y in ((14, 14), (15, 14), (14, 15), (15, 15)):
        cells.NewAgentSQ(x, y)

def main():
    cells = pal.NewAgentGrid((SIZE, SIZE), isStackable=False)
    Setup(cells)
    print("Initial cells:", len(cells.All()))

if __name__ == "__main__":
    main()
```

`NewAgentSQ(x, y)` places a cell at lattice coordinates `(x, y)`. An agent has an identity (its handle) as well as a position. The lattice dimensions are not the number of cells: most sites are initially empty.

2. Draw the cells immediately

`StartPixWindow` creates a pixel grid and a window. `pal.AwaitWindows()` waits until windows are ready before the program continues. Each lattice site becomes one pixel, enlarged by `scale`. We draw the background black and occupied sites red.

Complete program — place and display cells:

```
import time
import PythonAutomataLibrary as pal

SIZE = 30
BLACK = pal.RGB(0, 0, 0)
RED = pal.RGB(255, 0, 0)

def Draw(cells: pal.AgentGrid, pix: pal.Pix):
    pix[:] = BLACK
    for agent in cells.All():
        pix[cells.I(agent)] = RED

def Setup(cells: pal.AgentGrid):
    for x, y in ((14, 14), (15, 14), (14, 15), (15, 15)):
        cells.NewAgentSQ(x, y)

def main():
    cells = pal.NewAgentGrid((SIZE, SIZE), isStackable=False)
    Setup(cells)
    pix, window = pal.StartPixWindow(SIZE, SIZE, scale=12, title="Cells")
    Draw(cells, pix)
    window.Update()
    pal.AwaitWindows()
    while window.IsOpen():
        time.sleep(0.05)
    window.Close()

if __name__ == "__main__":
    main()
```

You should see four red cells near the center. If you change their initial coordinates, the drawing changes accordingly. `window.Update()` publishes the pixels you wrote; drawing into `pix` alone does not refresh the display. Later we will call `Draw` after every timestep.

3. Find neighbors and empty sites

In this model, cells move only into empty neighboring sites. `pal.MooreHood(2, excludeCenter=True)` describes the eight relative offsets around a 2D site. `cells.Hood(..., x, y)` converts those offsets to valid lattice coordinates. `XSQ` and `YSQ` give an agent's lattice site.

An `IList` is PAL's reusable list of integers. We use it to collect empty neighboring sites. Each site has coordinates `(x, y)`, but `IList` stores a single integer per site: `cells.ToI(x, y)` converts coordinates to that site's index. Later, `cells.ItoX(i)` and `cells.ItoY(i)` convert the index back to coordinates. Add the following code to the program from stage 2, immediately after the color constants:

```
MOORE = pal.MooreHood(2, excludeCenter=True)

def EmptyNeighbors(cells: pal.AgentGrid, agent: int, empty: pal.IList):
    empty.Clear()
    for x, y in cells.Hood(MOORE, cells.XSQ(agent), cells.YSQ(agent)):
        if cells.counts[x, y] == 0:
            empty.Append(cells.ToI(x, y))
    return empty
```

Create `empty = pal.NewIList()` once in `main`, immediately after `Setup(cells)`. To inspect the result before adding movement, insert this line immediately afterward:

```
print(EmptyNeighbors(cells, cells.All()[0], empty).All())
```

`cells.All()[0]` selects one of the initial agents, `EmptyNeighbors` fills the reusable list with empty neighboring site indices, and `.All()` returns those indices for printing. The printed values are **site indices, not agent handles**; `ItoX` and `ItoY` recover their coordinates. At a nonperiodic boundary, `Hood` omits offsets outside the grid.

4. Move cells to empty neighboring sites

Each cell takes a random walk: at every timestep, it chooses one available empty neighboring site and moves there. No cells are created or removed. We iterate a shuffled snapshot of existing agents; moves update occupancy immediately, so later cells see the new positions.

Continue with the program modified in stage 3, retaining `MOORE`, `EmptyNeighbors`, and the reusable `empty` list. Remove the optional diagnostic `print` if you added it. Define the following `Step` function alongside `Draw` and `Setup`:

```
def Step(cells: pal.AgentGrid, empty: pal.IList):
    for agent in cells.All(shuffle=True):
        EmptyNeighbors(cells, agent, empty)
        if len(empty):
            site = empty.Random()
            cells.MoveSQ(agent, cells.ItoX(site), cells.ItoY(site))
```

Replace the stationary display loop with:

```
    pal.AwaitWindows()
    while window.IsOpen():
        Step(cells, empty)
        Draw(cells, pix)
        window.Update()
        time.sleep(0.05)
    window.Close()
```

The cells wander without overlapping. A cell with no empty neighbors stays put. Selecting uniformly among *available* sites means the probability of each direction depends on occupancy.

5. Compile the timestep

The movement rule runs every timestep, making it a good candidate for compilation. `@pal.njit(cache=True)` compiles the function for faster repeated execution; the first call still takes time to compile, and compatible later runs can reuse the cached result. Because the neighborhood results already use `IList`, no change in data representation is needed. Add the decorator to `Step` and annotate its PAL arguments:

```
@pal.njit(cache=True)
def Step(cells: pal.AgentGrid, empty: pal.IList):
    for agent in cells.All(shuffle=True):
```

```
        EmptyNeighbors(cells, agent, empty)
        if len(empty):
            site = empty.Random()
            cells.MoveSQ(agent, cells.ItoX(site), cells.ItoY(site))
```

Replace the earlier `EmptyNeighbors` definition with this compiled version as well:

```
@pal.njit(cache=True)
def EmptyNeighbors(cells: pal.AgentGrid, agent: int, empty: pal.IList):
    empty.Clear()
    for x, y in cells.Hood(MOORE, cells.XSQ(agent), cells.YSQ(agent), unroll=True):
        if cells.counts[x, y] == 0:
            empty.Append(cells.ToI(x, y))
    return empty
```

Both functions now run as compiled code. `main`, window creation, and rendering can remain in ordinary Python. In the compiled `EmptyNeighbors`, `unroll=True` expands the small, fixed eight-neighbor loop, which can reduce loop overhead. It has no unrolling benefit in the ordinary Python version from stage 3, and larger neighborhoods or complex loop bodies can increase compilation time and code size. The simulation rule and data representation are unchanged. `cache=True` can reuse compiled specializations in later runs; the performance section explains the tradeoffs.

6. Add a diffusing nutrient

The cells now share a continuous nutrient field, represented by a `PDEgrid` on the same lattice. Nutrient **enters from the four boundaries**, diffuses inward, and is **consumed wherever a cell is present**. This behavior will remain unchanged when we replace individual agents with population counts in the next stage.

Add this alongside the cell-grid construction in `main`:

```
nutrient = pal.NewPDEgrid((SIZE, SIZE))
nutrient.SetTimeSpaceStep(1.0, 1.0, 1.0)
nutrient[:] = 1.0
```

The initial concentration is 1 everywhere. Each timestep, cells remove nutrient locally and the boundaries maintain an exterior concentration of 1:

```
def StepNutrient(cells: pal.AgentGrid, nutrient: pal.PDEgrid):
    for agent in cells.All():
        x, y = cells.XSQ(agent), cells.YSQ(agent)
        nutrient[x, y] = max(0.0, nutrient[x, y] - 0.2)
    nutrient.Diffusion(0.1, xMinBC=1.0, xMaxBC=1.0, yMinBC=1.0, yMaxBC=1.0)
    nutrient.Update()
```

Draw the nutrient concentration with a grayscale `ColorScale`. Define the colors alongside the other constants, and add the drawing function alongside `Draw`:

```
NUTRIENT_COLORS = (BLACK, pal.RGB(255, 255, 255))

def DrawNutrient(nutrient: pal.PDEgrid, pix: pal.Pix):
```

```
    for i in range(len(nutrient)):
        pix[i] = pal.ColorScale(NUTRIENT_COLORS, max(0.0, min(1.0, nutrient[i])))
```

In `main`, create a second pixel window alongside the cell window:

```
nutrientPix, nutrientWindow = pal.StartPixWindow(SIZE, SIZE, scale=12, title="Nutrient")
```

Keep the movement step from stage 5. Construct `nutrient` once alongside `cells`, then update and draw both windows in the simulation loop:

```
Step(cells, empty)                # move cells first
StepNutrient(cells, nutrient)     # consume and diffuse nutrient
Draw(cells, pix)                 # display the updated cells
window.Update()
DrawNutrient(nutrient, nutrientPix)
nutrientWindow.Update()
```

`StepNutrient` can remain an ordinary Python function for this first version; compiling the cell movement step does not require compiling every other function. The boundary arguments specify fixed exterior concentrations: unlike the default no-flux boundaries, these permit nutrient to diffuse into the modeled region when cells deplete it. `Diffusion` queues the transport changes and `Update()` commits them. The values are illustrative, not biologically calibrated.

What you will see: two windows update together. The cell window shows moving red cells, while the nutrient window uses a black-to-white `ColorScale` gradient (0 = black, 1 = white). The `DrawNutrient` function above is also discussed in the PixWindow section. Close both windows when the simulation ends. See Boundary conditions for transport for boundary options.

7. Replace individual cells with population counts

Now relax the **one cell per site** constraint. With multiple occupancy allowed, cells no longer need to search for empty destinations. A `PopGrid` holds counts instead of individual agents, and a `Multinomial` distributes each site's cells among eight possible movement directions. This is a related random walk, **not an exact translation** of the agent model: movement is synchronous, multiple cells can share a site, and a cell at a boundary may stay because some directions lead outside the grid. There is no division or death: the total number of cells remains constant. The nutrient field from stage 6 is retained, including its boundary supply and cellular consumption.

This stage starts a separate, complete program rather than modifying the stage 6 program in place. Replace the individual-agent grid and its empty-neighbor movement rule with the count-based implementation below. You can keep the stage 6 version separately to compare their behavior.

Complete program — count-based random movement:

```
import time
import PythonAutomataLibrary as pal

SIZE = 30
BLACK = pal.RGB(0, 0, 0)
RED = pal.RGB(255, 0, 0)
NUTRIENT_COLORS = (BLACK, pal.RGB(255, 255, 255))

MOORE = pal.MooreHood(2, excludeCenter=True)
```

```
@pal.njit(cache=True)
def Step(cells: pal.PopGrid, multi: pal.Multinomial):
    for i in cells.All():
        multi.Setup(cells[i])
        x, y = cells.ItoX(i), cells.ItoY(i)
        for nx, ny in cells.Hood(MOORE, x, y, unroll=True):
            moved = multi.Sample(1.0 / 8)
            cells.Add(-moved, i)
            cells.Add(moved, nx, ny)
    cells.Update()

@pal.njit(cache=True)
def StepNutrient(cells: pal.PopGrid, nutrient: pal.PDEgrid):
    for i in range(len(cells)):
        nutrient[i] = max(0.0, nutrient[i] - 0.2 * cells[i])
    nutrient.Diffusion(0.1, xMinBC=1.0, xMaxBC=1.0, yMinBC=1.0, yMaxBC=1.0)
    nutrient.Update()

@pal.njit(cache=True)
def Draw(cells: pal.PopGrid, pix: pal.Pix):
    pix[:] = BLACK
    for i in range(len(cells)):
        if cells[i]:
            pix[i] = RED

def DrawNutrient(nutrient: pal.PDEgrid, pix: pal.Pix):
    for i in range(len(nutrient)):
        pix[i] = pal.ColorScale(NUTRIENT_COLORS, max(0.0, min(1.0, nutrient[i])))

def main():
    pal.Seed(1234)
    cells = pal.NewPopGrid((SIZE, SIZE))
    for x, y in ((14, 14), (15, 14), (14, 15), (15, 15)):
        cells[x, y] = 1
    initialPop = cells.GetPop()
    multi = pal.NewMultinomial()
    nutrient = pal.NewPDEgrid((SIZE, SIZE))
    nutrient.SetTimeSpaceStep(1.0, 1.0, 1.0)
    nutrient[:] = 1.0
    pix, window = pal.StartPixWindow(SIZE, SIZE, scale=12, title="Population random walk")
    nutrientPix, nutrientWindow = pal.StartPixWindow(SIZE, SIZE, scale=12, title="Nutrient")
    DrawNutrient(nutrient, nutrientPix)
    nutrientWindow.Update()
    Draw(cells, pix)
    window.Update()
    pal.AwaitWindows()
    while window.IsOpen() and nutrientWindow.IsOpen():
        Step(cells, multi)
        assert cells.GetPop() == initialPop  # movement conserves cell count
        StepNutrient(cells, nutrient)
        Draw(cells, pix)
        window.Update()
        DrawNutrient(nutrient, nutrientPix)
        nutrientWindow.Update()
        time.sleep(0.05)
    window.Close()
    nutrientWindow.Close()

if __name__ == "__main__":
    main()
```

Each of the eight Moore-neighborhood directions has probability `1/8`. At an interior site, all eight directions are valid, so every cell moves; at an edge or corner, some directions are unavailable, so cells may stay. `Setup(cells[i])` starts a multinomial allocation for all cells currently at site `i`; each `Sample(1.0 / 8)` assigns cells to one valid direction. At a nonwrapped boundary, invalid directions are skipped, and their unallocated probability mass corresponds to cells staying at their original site. For example, a corner site has only three valid Moore-neighborhood destinations: each receives probability `1/8`, while the remaining `5/8` is the probability of staying. This differs from the individual-agent model, which selects uniformly among available empty neighbors and therefore moves whenever at least one is available. Probabilities refer to the **original** site's population, not the remaining unassigned cells. Queued transfers are committed together with `Update()`, so arrivals do not move again in the same timestep. The assertion checks the model's population-conservation invariant after every movement step; if you later add cell birth or death, revise or remove that check.

The nutrient update now consumes 0.2 concentration units per cell at each site, so multiple occupancy increases local consumption. The same fixed-value boundaries replenish nutrient in both representations. The two versions implement the same basic random-walk idea, but not the same exclusion process: `AgentGrid` avoids occupied destinations and updates agents sequentially; `PopGrid` permits multiple occupancy and moves all counts synchronously. That distinction matters when comparing results. See Multinomial for more about sampling and solver reuse.

Where to go next

You have now constructed and visualized model state, queried neighborhoods, moved cells, compiled repeated work, evolved a nutrient field, and changed cell representations without changing that field. The remainder of this guide is organized by task. Start with Core use, Construction, and Shared spatial API for conventions common to the grids. For compilation and execution options, see Performance. Then consult the section for the component you need: Grid, AgentGrid, IList, PopGrid, Random numbers and Multinomial, or PDEgrid. The later sections cover visualization, saving PAL objects, and organizing larger models. You do not need to use every feature in the tutorial model.

Core use

These import, execution-mode, and compilation conventions apply throughout the guide. The tutorial uses PAL's default safe mode.

```
import PythonAutomataLibrary as pal
```

PAL starts in **safe mode**. Safe mode checks public operations and provides useful errors. To select fast mode:

```
pal.FastMode()
```

Call `FastMode()` before constructing any PAL model or window object. The first such construction locks the process-wide mode. Fast mode removes safety checks from performance-sensitive paths; it is intended for models already validated in safe mode.

Use `@pal.njit(cache=True)` for repeated model computations. This complete example creates an agent grid in Python, then counts its agents in a compiled function:

```
import PythonAutomataLibrary as pal

@pal.njit(cache=True)
def CountAgents(grid: pal.AgentGrid):
    count = 0
```

```
    for agent in grid.All():
        count += 1
    return count

grid = pal.NewAgentGrid((30, 30))
grid.NewAgentSQ(10, 12)
print(CountAgents(grid))  # 1
```

PAL objects must be constructed outside `@pal.njit` functions. Create grids, lists, samplers, and windows in ordinary Python code, then pass them into compiled functions for computation. Methods that create agents within an existing `AgentGrid`, such as `NewAgent`, can still be called inside compiled functions. `pal.njit` wraps Numba `njit` and additionally transforms PAL model code. Among other PAL-specific compilation behavior, annotated PAL arguments allow safe-mode errors in compiled model code to report the model source line.

Construction

Use the component overview to select the state or helper type you need. PAL constructs these objects with `New...` functions—for example, `pal.NewAgentGrid(...)`, `pal.NewPDEgrid(...)`, and `pal.NewIList()`. The API Reference lists constructor signatures; the sections below explain the arguments and behavior that matter when setting up a model.

Spatial grids use one to three dimensions. A **negative dimension enables wrapping on that axis**, while its absolute value gives the size. For example:

```
grid = pal.NewAgentGrid((100, 100))    # 100 x 100, no wrapping
grid = pal.NewAgentGrid((-100, 100))   # x wraps, y does not
grid = pal.NewPDEgrid((-100, -100))    # x and y wrap
```

`numAgentProps` controls how many floating-point properties each agent has. `isStackable=False` means at most one agent can occupy a lattice site; `True` allows multiple agents per site. `PopGrid.capacity` selects the smallest integer storage type that can hold the requested **per-site** count; it does not impose a strict population limit.

Constructor validation and limits

Each spatial dimension must be a nonzero integer within the supported int32 range (excluding the minimum signed value), and the product of absolute axis lengths must not exceed the int32 maximum. Fractional, nonfinite, zero, oversized, or malformed dimension inputs raise `ValueError` during construction. `AgentGrid` alone also accepts `()` for a zero-dimensional nonspatial population.

For validation, `numAgentProps` must be a nonnegative integer fitting AgentGrid's int32 metadata budget (which also reserves entries for spatial dimensions); fractional, nonfinite, negative, or oversized values raise `ValueError`. `isStackable` accepts booleans or `0` / `1`; other values raise `ValueError`.

`NewMultinomial(other)` accepts another PAL Multinomial in the **same safety mode** and copies its solver configuration, not its current sampling state. An unrelated object raises `TypeError`; mixing safe and fast samplers raises `ValueError`. Create a new sampler without `other` to use the default solver.

Shared spatial API

`Grid`, `AgentGrid`, `PopGrid`, and `PDEgrid` deliberately share spatial conventions. Geometry metadata (`xDim`, `yDim`, `zDim`, `nDims`, `wrapX`, `wrapY`, `wrapZ`) is read as attributes; geometry operations such as `ToI(...)` and `Box(...)` are methods.

Geometry

PAL identifies lattice sites either by their spatial coordinates, such as `(x, y)`, or by a single integer index. Coordinates are convenient for spatial rules; linear indices are useful for storing or sampling sites in integer collections. The conversion methods let you move between these representations. For example, on a 30 × 30 grid:

```
import numpy as np
import PythonAutomataLibrary as pal

grid = pal.NewGrid((30, 30), np.int32)
site = grid.ToI(10, 12)
print(grid.ItoX(site), grid.ItoY(site))  # 10 12
```

The integer `site` identifies the same lattice location as the coordinate pair `(10, 12)`; it is not an agent handle. The following members expose this geometry and conversion behavior:

Member	Meaning
`len(grid)`	Number of lattice sites.
`xDim`, `yDim`, `zDim`	Axis sizes. Missing axes are invalid in safe mode.
`nDims`	Number of spatial dimensions.
`wrapX`, `wrapY`, `wrapZ`	Whether each axis wraps.
`ToI(x, y=-1, z=-1)`	Convert coordinates to a linear lattice index.
`ItoX(i)`, `ItoY(i)`, `ItoZ(i)`	Recover coordinates from a linear index.
`InWrapX/Y/Z(value)`	Wrap an integer lattice coordinate on that axis, or return `-1` when outside a nonwrapped axis.

The same geometry names are used in 1D, 2D, and 3D. Supply only coordinates/axes that exist for the grid. `Grid`, `PopGrid`, and `PDEgrid` use `InWrapX/Y/Z` for lattice coordinates; `AgentGrid` distinguishes lattice `InWrapSQX/Y/Z` from continuous `InWrapX/Y/Z`.

Regions and neighborhoods

Use `Box` to visit sites in a rectangular region, and `Hood` to visit sites at specified offsets around a particular location. Both yield coordinates, so they can be used directly in spatial loops. The following are **reference signatures** showing the 1D, 2D, and 3D forms, not executable calls with defined variables:

```
grid.Box(x1, x2)
grid.Box(x1, x2, y1, y2)
grid.Box(x1, x2, y1, y2, z1, z2)

grid.Hood(hood, x)
grid.Hood(hood, x, y)
grid.Hood(hood, x, y, z)
```

For example, on a two-dimensional grid, the first loop visits a 3 × 3 region, while the second visits the neighboring sites around `(10, 10)` without including the center:

```
for x, y in grid.Box(9, 12, 9, 12):
    ...

neighbors = pal.MooreHood(2, excludeCenter=True)
for x, y in grid.Hood(neighbors, 10, 10):
    ...
```

`Box` iterates coordinates in the half-open rectangular region `[x1, x2)`, `[y1, y2)`, `[z1, z2)` for the dimensions supplied. It yields an integer in 1D, `(x, y)` in 2D, and `(x, y, z)` in 3D. Bounds must be int32 integers, and the number of bound pairs must match grid dimensionality. On wrapped axes, out-of-range coordinates are wrapped; on nonwrapped axes, they are skipped. A box wider than a wrapped axis can therefore yield repeated sites. `Hood` maps relative offsets around a lattice position to the same coordinate forms, preserving offset order and duplicates; wrapped coordinates are mapped into the grid and nonwrapped out-of-range coordinates are omitted. Offset tuple lengths must match grid dimensionality. `Box` and `Hood` are direct-iteration constructs in compiled code: top-level `break` or `continue` from their loop body is unsupported; use explicit `range` loops when that control flow is needed.

Built-in neighborhoods:

```
pal.MooreHood(dim, excludeCenter=False)
pal.VonNeumannHood(dim, excludeCenter=False)
pal.CircleHood(dim, rad, excludeCenter=False)
```

`MooreHood` contains offsets with each coordinate in `[-1, 1]`. `VonNeumannHood` contains the center and axis-adjacent offsets. `CircleHood` contains integer offsets whose Euclidean distance from the center is at most `rad`; `rad` may be noninteger, but must be finite and nonnegative. Set `excludeCenter=True` to omit the zero offset. All three constructors support dimensions 1–3 and reject other dimension counts with `ValueError`. Each returns a tuple of integer offset tuples; these describe relative positions, not site indices, until mapped through `grid.Hood(...)`.

Performance: `FastMode()`, `njit(cache=True)`, and `unroll=True`

PAL provides three complementary performance controls:

- **`pal.FastMode()`** selects PAL's fast implementation, reducing safety-check overhead during execution. Call it once, before constructing any PAL model or window objects; otherwise PAL remains in safe mode. Fast mode is not a substitute for validating model correctness, and its speed advantage depends on the workload.
- **`@pal.njit(cache=True)`** compiles a Python function to native code, avoiding Python interpreter overhead in repeated numerical loops. The first call for a particular argument-type specialization still incurs compilation cost. `cache=True` saves

compiled specializations to disk so later runs can reuse them when compatible; it does not make the first compilation faster, and some changes or environments require recompilation.

- `unroll=True` only affects performance inside functions compiled with `@pal.njit`; it provides no unrolling benefit in ordinary Python execution. It expands the loop body for each neighborhood offset, removing ordinary loop-control overhead and exposing individual operations to optimization. Use it only for **stencil operations over relatively small neighborhoods with simple loop bodies**: both the number of neighbors and the amount of code inside the loop increase generated code size and compilation time. For large neighborhoods or complex loop bodies, prefer ordinary iteration.

Returning PAL objects from compiled helpers: In the `EmptyNeighbors` tutorial, returning the modified `IList` produced Numba's `Cannot cache compiled function ... as it uses dynamic globals` warning on Windows, whereas modifying the same list in place without returning it allowed caching. Prefer passing PAL objects into compiled helpers and modifying them in place when a return value is unnecessary. This is an observed caching limitation for this example, not a blanket rule that every PAL-object return prevents caching; ordinary scalar return values remain useful. The warning concerns disk caching, not whether the function can be JIT-compiled.

Each option targets a different cost: **fast mode reduces runtime checks, JIT compilation reduces interpreter overhead, caching avoids compatible recompilation across runs, and unrolling reduces inner-loop overhead**. You can combine them, but benchmark representative workloads.

Example: unrolled Von Neumann neighborhood

For a small, fixed neighborhood, PAL can compile the neighbor operations directly rather than executing a conventional loop. Consider summing values in the four cardinal directions around `(x, y)`:

```
VON_NEUMANN = pal.VonNeumannHood(2, excludeCenter=True)

@pal.njit(cache=True)
def NeighborSum(grid: pal.Grid, x: int, y: int):
    total = 0.0
    for nx, ny in grid.Hood(VON_NEUMANN, x, y, unroll=True):
        total += grid[nx, ny]
    return total
```

Conceptually, on an interior site of a nonwrapping grid, this is equivalent to:

```
total = 0.0
total += grid[x - 1, y]
total += grid[x + 1, y]
total += grid[x, y - 1]
total += grid[x, y + 1]
```

The offset order does not affect this sum. The example illustrates four operations, not the generated machine code. At boundaries, `Hood` still handles valid coordinates and wrapping according to the grid's configuration.

Why this is fast: The neighborhood has a fixed number of offsets known at compilation time. With `unroll=True`, Numba can specialize each operation, eliminating the ordinary neighborhood-loop control and making the resulting code easier to optimize. This can improve performance when an agent-based model repeatedly visits the same small neighborhood, although the benefit depends on the workload. It preserves PAL's compact neighborhood syntax rather than requiring manually expanded code.

The option changes how the loop is compiled, not which neighbors are visited. For large neighborhoods or substantial loop bodies, the duplicated operations can outweigh the runtime benefit; compare against ordinary iteration before enabling it.

Safety contracts and mode selection

Safe and fast objects share the same modeling API; PAL starts in safe mode. To opt into fast mode, call `pal.FastMode()` **before constructing any PAL model or window object**. The first such construction locks the process-wide mode selection; a later `pal.FastMode()` raises `RuntimeError`. Run safe and fast comparisons in separate processes rather than trying to switch modes mid-run. Safe mode validates public operations and is the development/default mode; fast mode assumes the model obeys those contracts and removes performance-sensitive checks. Examples of safe-mode checks include lattice bounds and dimensionality, dead-agent and occupancy errors, PopGrid overflow, and PDE field/interface array shape or dimensionality. Invalid operations in fast mode are outside the supported contract rather than an alternative error-handling API.

Use safe mode until the model is correct, then benchmark fast mode. Do not rely on safe-mode exceptions as model control flow.

Grid

Use `Grid` for one typed value per lattice site, such as terrain type, occupancy flags, environmental measurements, or a precomputed spatial map. Unlike `AgentGrid`, it does not track individual agents; unlike `PDEgrid`, it has no transport solver.

```
import numpy as np
import PythonAutomataLibrary as pal

grid = pal.NewGrid((30, 30), np.float32)
grid[10, 12] = 2.5
```

The dimensions set the lattice size, and the dtype determines the value type at each site. For example, use `np.float32` for a floating-point field or an integer dtype for categories or counts. Supported dtypes are `bool`, signed/unsigned 8/16/32/64-bit integers, and 32/64-bit floats.

`Grid` supports both coordinate indexing (`grid[x, y]` on this 2D grid) and linear indexing (`grid[i]` , where `i` is a single site index). Slices select rectangular regions. Continuing the example above:

```
print(grid[10, 12])             # 2.5
site = grid.ToI(10, 12)
print(grid[site])               # 2.5, accessed by linear index
region = grid[2:5, 3:8]       # a 3-by-5 NumPy array
region[:] = 7.0                # modifies the copy, not grid
print(grid[10, 12])             # still 2.5
grid[2:5, 3:8] = 7.0            # writes directly to the grid
print(grid[3, 4])               # 7.0
print(grid[10, 12])             # still 2.5 (outside the region)
```

To modify the PAL grid, assign to `grid[x, y]` or directly to a grid slice such as `grid[:] = 0`. Three-dimensional grids additionally accept `grid[x, y, z]`. Slice reads return detached NumPy copies. Scalar and slice reads/writes work in Python and compiled PAL model code. In safe mode, invalid scalar indices and coordinates are rejected; fast mode assumes valid access. Safe mode validates bounds, dimensionality, and whether assigned values are representable by the Grid dtype: boolean grids accept only `0` / `1`; integer grids reject fractional, nonfinite, and out-of-range assignments; floating grids reject nonfinite or out-of-range assignments. Use a wider dtype when a calculation may exceed its current range. A detached slice copy is not a live view of the Grid.

AgentGrid

Use `AgentGrid` when individual agents retain their identities while moving, reproducing, dying, or changing properties. It manages agent handles and locations for models that track particular entities.

```
agents = pal.NewAgentGrid((30, 30), numAgentProps=2)
agent = agents.NewAgentSQ(10, 12)
agents[agent, 0] = 1.5
```

PAL returns an integer **handle** when it creates an agent. The handle continues to identify that agent after movement; it is not a lattice-site index. Here `numAgentProps=2` reserves two floating-point properties per agent, indexed as `0` and `1`. Agent properties are stored on the grid and accessed with `agents[agent, property]`.

Population and lifecycle

By default an `AgentGrid` is not stackable: at most one agent may occupy a lattice site. Set `isStackable=True` when multiple agents per site are required. Safe mode rejects creation or movement into an occupied site on a nonstackable grid without changing model state.

Operation	Meaning
`GetPop()`	Number of living agents.
`Alive(agent)`	Whether an agent handle is alive.
`NewAgentSQ(...)`	Create an agent at a lattice site.
`NewAgent(...)`	Create an agent at a continuous position.
`Dispose(agent)`	Remove an agent.
`All(shuffle=False)`	Snapshot of all living agents; optionally shuffled.

`All()` has **snapshot semantics**. The returned collection can be iterated while agents are created or disposed without changing the current iteration. `All(shuffle=True)` returns the same population in randomized order.

Working with `IList`. `AgentGrid` identifies agents by integer handles and lattice sites by integer indices, so `pal.NewIList()` is useful for collecting either kind of identifier during queries. For example, when choosing an empty neighboring site, clear a reusable `IList`, append each eligible site's index with `agents.ToI(x, y)`, then select one with `Random()` and convert it back with `agents.ItoX(i)` and `agents.ItoY(i)`. Do not mix site indices with agent handles. The empty-neighbor tutorial shows this pattern in code; the IList section explains its storage and iteration behavior.

The following is a *pattern*, not a standalone executable example: `ShouldDie` represents a death rule you define for your model.

```
for agent in agents.All():
    if ShouldDie(agent):
        agents.Dispose(agent)
```

Position

PAL distinguishes **lattice sites** (integer coordinates) from **continuous positions** (floating-point coordinates). Use lattice coordinates for site-based placement and continuous coordinates for off-lattice movement. On the two-dimensional `agents` grid created above, the lattice operations look like this:

```
print(agents.XSQ(agent), agents.YSQ(agent))  # 10 12
agents.MoveSQ(agent, 11, 12)
print(agents.I(agent))  # linear index of the new site
```

For continuous positions, use `agents.X(agent)`, `agents.Y(agent)`, and `agents.Move(agent, x, y)`. In three dimensions, the corresponding `ZSQ(agent)`, `Z(agent)`, and third coordinate arguments are available. The snippets here show the 2D forms; the methods also support the appropriate 1D forms.

`NewAgentSQ` and `MoveSQ` accept either a linear lattice index or lattice coordinates in every spatial dimension. SQ placement uses the center of the selected cell: after `NewAgentSQ(x, y)`, for example, `XSQ(agent)==x` and `YSQ(agent)==y`, while `X(agent)==x+0.5` and `Y(agent)==y+0.5`. Continuous `NewAgent` and `Move` instead keep the supplied spatial coordinates, with `XSQ` / `YSQ` / `ZSQ` identifying the containing cell.

Occupancy and queries

These queries inspect site occupancy without changing the population. The agent in the preceding example moved to `(11, 12)`, so query that site:

```
print(agents.counts[11, 12])  # 1
for agent in agents.AgentsAt(11, 12):
    print(agent)
```

The count gives the number of occupants; `AgentsAt` yields their handles. The corresponding query forms are `agents.LastAgent(x, y)`, `agents.AgentsAt(x, y)`, and `agents.counts[x, y]` on a 2D grid; supply the appropriate number of coordinates for other dimensions.

`LastAgent` returns the most recently stacked agent at a site, or `-1` if the site is empty. Iterate with `for agent in agents.AgentsAt(x, y):` in Python or inside `@pal.njit`. Do not create, dispose, or move agents during direct iteration, including moves within the same lattice site. Safe mode detects these changes and raises `RuntimeError`; fast mode omits that check. In Python, `AgentsAt(i)` also accepts a linear site index, including for multidimensional grids; `AgentsAt(x, y)` and `AgentsAt(x, y, z)` use coordinates. This overload is distinct from the continuous-position `AgentsInRadius` query. `counts` exposes lattice occupancy counts and supports linear-site, coordinate, and slice indexing (`agents.counts[i]`, `agents.counts[x, y]`, or `agents.counts[:, :]`). It is a **read-only query view**, not a second mutable population grid. Slice queries return detached arrays of occupancy counts. A nonspatial (`dimensions=()`) AgentGrid has no lattice counts; querying `len(agents.counts)` raises `ValueError`.

Radius iteration

`AgentsInRadius` searches around a **continuous position**, not just within one lattice site. Continuing from the earlier 2D example, the agent is centered at `(11.5, 12.5)` after moving to site `(11, 12)`. A radius-2 query around that point finds it:

```
for nearby, dx, dy, distSq in agents.AgentsInRadius(2.0, 11.5, 12.5):
    print(nearby, dx, dy, distSq)
```

The returned values are the agent handle, the x/y displacement from the query center, and the squared distance. For this centered query, the existing agent has zero displacement and squared distance. In one dimension, the iterator yields `(agent, dx)`; in three dimensions, it yields `(agent, dx, dy, dz, distSq)`. The general signature is `AgentsInRadius(rad, x[, y[, z]], exclude=None)`, where the number of coordinates matches the grid dimensionality.

The radius is inclusive: agents with distance exactly `rad` are included. Returned displacements point from the query center to the agent and use the shortest periodic displacement on wrapped axes; `distSq` is their squared Euclidean distance in 2D/3D. `exclude` omits a specified agent handle. As with Python-side `AgentsAt`, structural modification during iteration is unsupported, and the generation check is enabled only in safe mode.

Wrapping

Wrapping connects opposite lattice edges. As described in Construction, a negative dimension enables wrapping on that axis. For example, create a 30 × 30 agent grid whose x-axis wraps while its y-axis does not:

```
wrapped = pal.NewAgentGrid((-30, 30))
print(wrapped.InWrapSQX(-1))  # 29
print(wrapped.InWrapSQY(-1))  # -1 (outside the nonwrapped axis)
```

`InWrapSQX/Y/Z(value)` handles **integer lattice coordinates**; `InWrapX/Y/Z(value)` handles **continuous positions**. On a nonwrapped axis, an out-of-range coordinate returns `-1`. `DispWrapX/Y/Z(start, end)` returns the shortest signed displacement from `start` to `end` across a periodic boundary. The suffix (`X`, `Y`, or `Z`) selects an axis present in the grid; it is not necessary to supply all three axes for a 2D model.

IList

Use `IList` as a reusable collection of integer identifiers, such as site indices or agent handles, for iteration, shuffling, or random selection. Typical uses include neighborhood queries, candidate destinations, and subsets of agents. Unlike a mathematical set, it preserves insertion order and can contain repeated identifiers; `Append()` does not enforce uniqueness.

```
items = pal.NewIList()
```

For example, use an `IList` to save two lattice-site indices from a grid and revisit them later:

```
import PythonAutomataLibrary as pal

grid = pal.NewAgentGrid((30, 30))
sites = pal.NewIList()
sites.Append(grid.ToI(10, 12))
sites.Append(grid.ToI(11, 12))
for i in sites.Iter():
    print(grid.ItoX(i), grid.ItoY(i))
```

The list stores integer identifiers, **not copies of grid sites or agents**. You can also store agent handles, but do not confuse those with site indices: a handle identifies an individual agent, while a site index identifies a lattice location. Stored values must be nonnegative int32 integers. Use `Random()` to choose one stored identifier at random, or `Shuffle()` to randomize their order.

Operation	Meaning
`Append(i)`	Append a nonnegative int32 value; returns the IList for chaining.
`Clear()`	Remove all entries; returns the IList for chaining.
`Random()`	Return a random entry.
`Shuffle()`	Shuffle entries in place; returns the IList for chaining.
`All()`	Return a detached copy.
`Iter()`	Return a no-copy/live iterable view.
`len(items)`, `items[i]`	Length/index access.

Use `Iter()` for one-pass iteration when a copy is unnecessary. Unlike `Iter()`, `All()` makes a snapshot that remains independent of later list changes. For example, continuing from the `sites` list above:

```
saved = sites.All()  # copy the two indices
sites.Clear()        # empty the original IList
print(len(sites))    # 0
print(len(saved))    # 2
```

`Iter()` reads current contents without copying; it does not preserve a snapshot if the list changes. Use `All()` to retain identifiers across subsequent modifications.

PopGrid

Use `PopGrid` when per-site population counts matter but individual identities do not. It supports count-based dynamics and movement without representing each individual as a separate agent.

```
pop = pal.NewPopGrid((30, 30), capacity=100)
pop[10, 12] = 20
```

`PopGrid` stores integer population counts. Indexing and direct assignment modify the current population immediately.

```
print(pop[10, 12])  # read the count at a site
pop[10, 12] = 10   # replace that site's count
pop[:] = 0        # clear counts at all sites
```

These are separate operations; the last clears the population. The next example independently resets `(10, 12)` to 20.

Linear (`pop[:]`) and spatial (`pop[:, :]`) slice reads return detached `int64` NumPy arrays; slice assignment writes to the population. Population changes can instead be accumulated and applied together:

```
pop[10, 12] = 20     # establish the starting population
pop.Add(-3, 10, 12)   # queue removal of three individuals
pop.Add(3, 11, 12)    # queue their arrival at the next site
print(pop[10, 12])    # 20: pending changes are not yet visible
pop.Update()          # apply both changes together
print(pop[10, 12])    # 17
```

`Add` changes only the pending delta. Until `Update()`, the source stays at 20 and the destination is unchanged. Afterward, the source has 17 and the destination has gained 3. More generally, until `Update()`, `GetPop()`, indexing, and `All()` still reflect the current population. A single `Update()` commits the queued changes together, avoiding order-dependent reads during a synchronous timestep. Direct assignments during that timestep still change current values immediately.

Population movement with `Multinomial`. When individuals at a site can move in several directions, `pal.NewMultinomial()` provides a sampler for distributing that site's population among the possible outcomes. Call `Setup(n)` with the site's population, then `Sample(p)` for each outcome, and use `pop.Add()` to queue the resulting transfers. Finally, call `pop.Update()` to apply the transfers together. See the count-based random walk for a complete example.

Operation	Meaning
`GetPop()`	Total population over all sites.
`All()`	Copy of the linear indices of currently nonzero sites.
`Reset()`	Clear current populations and pending changes.

`capacity=None` selects signed int64 storage. An explicit `capacity` must be a nonnegative int64 integer; invalid, fractional, nonfinite, or out-of-range values raise `ValueError` during construction. PAL selects uint8, uint16, uint32, or int64 storage according to that value. **The requested capacity is not an enforced limit:** for example, `capacity=100` selects uint8, allowing site counts up to 255. Safe mode checks direct writes against the selected storage range and checks the final site counts and signed-int64 total population at `Update()`. `Add()` queues signed changes; pending deltas have their own integer range and are checked for overflow when queued. If `Update()` validation fails, current counts and pending deltas remain unchanged. Fast mode omits these safety checks.

Random numbers and Multinomial

Use PAL's random-number functions to select agents, destinations, or other stochastic outcomes. Use `Multinomial` to allocate counts among several outcomes without sampling each individual separately.

```
import PythonAutomataLibrary as pal

pal.Seed(1234)
u = pal.Random()     # uniform float in [0, 1)
i = pal.RandInt(10)  # integer from 0 through 9
```

This example seeds PAL's random-number generator before drawing two values. The seed fixes the random stream for reproducible runs; `Random()` and `RandInt()` each advance that stream.

`Seed` accepts integral seeds in the uint64 domain (`0` through `2**64 - 1`); invalid or nonfinite inputs raise `ValueError`. `RandInt(max)` requires a positive integer in the int64 domain and returns an integer in `[0, max)`; zero, negative, fractional,

nonfinite, or out-of-range bounds raise `ValueError`. `Random()` returns a uniform floating-point draw. PAL random functions share one seeded stream across Python and compiled PAL calls, so reseeding reproduces the same call sequence across that boundary. The stream is call-order dependent: adding or removing a random draw changes later results. This does not seed NumPy's separate RNG.

Multinomial allocations

A binomial draw counts successes among `n` trials. A multinomial allocation distributes those trials among mutually exclusive outcomes, such as moving left, right, or staying put. In the example below, each of 100 individuals independently has a 20% chance of moving left, a 30% chance of moving right, and a 50% chance of staying. The sampled counts vary, but their sum is always 100.

```
multi = pal.NewMultinomial()
multi.Setup(100)
left = multi.Sample(0.2)
right = multi.Sample(0.3)
stay = 100 - left - right  # unallocated individuals

independent = multi.Binomial(100, 0.2)  # separate binomial draw
```

`Setup(n)` initializes a chainable multinomial sequence with `n` trials and total probability mass 1. Each `Sample(p)` takes that category's probability in the original multinomial distribution, draws from the remaining trials using the corresponding conditional probability, then removes the sampled count and `p` from the remaining state. Thus `Sample(0.2)` followed by `Sample(0.3)` assigns original category probabilities of 0.2 and 0.3; the second call does not mean 30% of the remainder. In the example, the remaining probability is 0.5, and `stay` accounts for individuals not allocated to either sampled category. In safe mode cumulative requested probability cannot exceed 1. `Binomial(n, p)` performs an independent binomial draw and does not consume the current multinomial sequence. `NewMultinomial(other)` copies solver configuration but starts with fresh sampling state.

Reusing the binomial solver core

To make independent multinomial allocations while reusing a solver, pass one sampler to another constructor:

```
movement = pal.NewMultinomial()
division = pal.NewMultinomial(movement)

movement.Setup(100)
moved = movement.Sample(0.2)

division.Setup(50)
dividing = division.Sample(0.1)
```

`NewMultinomial(movement)` **shares the underlying reusable binomial solver core**, avoiding a separate solver setup, but the two `Multinomial` objects keep **independent remaining-count and remaining-probability state**. The second sampler does not inherit the first sampler's partially sampled `Setup` sequence. Call `Setup` separately for each allocation. Sharing the solver does not create separate random streams: both samplers use PAL's common RNG, so results still depend on call order.

PDEgrid

Use `PDEgrid` for continuous spatial fields that evolve through diffusion or advection, such as nutrients, oxygen, signaling molecules, or temperature. Unlike a plain `Grid`, it provides transport operations.

```
field = pal.NewPDEgrid((30, 30))
field[10, 12] = 1.0
```

`PDEgrid` stores a continuous `float32` scalar field. As with `PopGrid`, there are two ways to change it: **direct assignment** changes the current field immediately, while `Add` queues a change that becomes visible only after `Update()`. Continuing from the field created above:

```
print(field[10, 12])  # 1.0
field.Add(-0.25, 10, 12)
print(field[10, 12])  # still 1.0: the change is pending
field.Update()
print(field[10, 12])  # 0.75
```

When several processes act in one timestep, accumulate their changes and call `Update()` once to apply them together. Transport operations such as `Diffusion` and `Advection` also accumulate changes rather than modifying the current field immediately.

Slice reads return detached `float32` NumPy arrays: editing a previously read slice does not modify the field, whereas assigning directly to a field slice does. Both linear (`field[:]`) and spatial (`field[:, :]`) slices are supported. `Reset()` clears the current field and pending changes.

Space and time

The time step `dt` specifies how much simulated time a transport update represents, while `dx`, `dy`, and `dz` specify the physical spacing between lattice sites along each axis. These values determine how diffusion and advection rates translate into changes on the grid.

```
field.SetTimeSpaceStep(0.1, 1.0, 1.0)  # dt, dx, dy for a 2D field
print(field.Dt(), field.Dx(), field.Dy())
```

For this two-dimensional field, each transport step represents `0.1` units of simulated time, and adjacent lattice sites are `1.0` spatial unit apart in either direction. The same API supports `dz` and `Dz()` for three-dimensional fields.

Transport methods use these spacings. Choose consistent units: if time is measured in hours and distance in millimeters, diffusion rates have units of mm²/hour and advection velocities have units of mm/hour. For explicit diffusion, the size of the change depends on combinations such as `rate * dt / dx**2`; for advection, it depends on combinations such as `velocity * dt / dx`. This is why changing the timestep or grid spacing can change both the numerical result and whether a transport step is stable. `Dy()` and `Dz()` apply only when those dimensions exist; safe mode rejects missing-dimension access. `dt`, `dx`, `dy`, and `dz` must be finite and positive in safe mode.

Boundary conditions for transport

Boundary conditions govern transport across the modeled region's edges. PAL's Cartesian diffusion and advection methods share boundary argument names, but their physical effects differ:

Boundary choice	How to specify it	Meaning
No flux	Omit the boundary argument, or pass `None`	No material crosses that nonwrapped face
Fixed exterior concentration	Pass a scalar, such as `xMinBC=1.0`	Prescribe the concentration just outside that face
Periodic	Construct the grid with a negative dimension on that axis, such as `(-30, 30)`	The two opposite faces connect; do not pass explicit BC values on that axis

Each name specifies an **axis** (`x` , `y` , or `z`) and **side** (`Min` or `Max`). For example, `xMinBC` controls the left face and `yMaxBC` the top face of a 2D grid. Unspecified nonwrapped faces are closed by default.

For example, on a nonwrapped 2D nutrient field:

```
field = pal.NewPDEgrid((30, 30))
field.SetTimeSpaceStep(1.0, 1.0, 1.0)
field.Diffusion(0.1, xMinBC=1.0, xMaxBC=0.0)
field.Update()
```

This prescribes an exterior concentration of 1 on the left and 0 on the right, with no flux across the top and bottom. The transport operation accumulates changes; `Update()` makes them current. Start with a nonuniform field or run multiple steps to see the resulting spatial profile.

For **diffusion**, the exterior concentration determines the gradient and therefore the boundary flux. For **advection**, it supplies the concentration entering through an inflow face; on an outflow face the upwind calculation uses the interior concentration instead. Thus the same `xMinBC` argument has a different numerical role in the two processes.

For a spatially varying boundary concentration, pass a C-contiguous `float32` array containing one value per site along the face. Scalar values are simpler when the whole face has the same condition. The boundary arguments apply to Cartesian methods; radial diffusion instead uses `outerBC` for its outer radial face and symmetry at the center.

Diffusion

Diffusion spreads a quantity from higher to lower concentrations. To illustrate, place concentration 1 at the center of an otherwise empty field and diffuse it for one timestep:

```
field = pal.NewPDEgrid((30, 30))
field.SetTimeSpaceStep(0.1, 1.0, 1.0)
field[15, 15] = 1.0
field.Diffusion(0.1)
field.Update()
print(field[15, 15])  # less than 1.0: concentration spreads outward
```

The `0.1` passed to `SetTimeSpaceStep` is the timestep; the `0.1` passed to `Diffusion` is the diffusion rate. The first call to `Diffusion` queues changes; `Update()` applies them. With no boundary arguments, the nonwrapped edges have no flux.

Choose the diffusion method according to how its rate varies across space. The following are **reference signatures**, not a sequence of calls to run together:

```
field.Diffusion(rateConstant, ...)
field.DiffusionMask(rateConstant, mask)
field.DiffusionField(rateConstants, ...)
field.DiffusionInterfaces(rateConstantsX, rateConstantsY=None, rateConstantsZ=None, ...)
field.DiffusionADI(rateConstant, ...)
field.DiffusionRadialCircle(rateConstant, outerBC=None)
field.DiffusionRadialSphere(rateConstant, outerBC=None)
```

`Diffusion` uses one constant diffusion rate. `DiffusionField` uses spatially varying cell-centered rates; each interior face uses the harmonic mean of its two adjacent cell rates, so a zero rate on either side blocks diffusion across that face. `DiffusionInterfaces` supplies the positive-face interface rate for each cell and axis directly; the negative face uses the neighboring cell's stored positive-face rate. `DiffusionMask` treats mask values `>= 0` as active and values `< 0` as blocked; diffusion occurs only between adjacent active cells, so the magnitude of a nonnegative mask value is irrelevant. `DiffusionADI` uses Crank-Nicolson in 1D, Peaceman-Rachford ADI in 2D, and Douglas-Gunn directional solves in 3D. `DiffusionRadialCircle` and `DiffusionRadialSphere` use radial shell finite-volume discretizations on a nonwrapped 1D grid with at least two cells. Cell `i` spans radii `i*dx` to `(i+1)*dx`, so cell 0 is the central disk/sphere and symmetry gives zero flux through its inner face. `outerBC` optionally sets the exterior concentration at the outer face.

Cartesian diffusion methods accept optional `xMinBC`, `xMaxBC`, `yMinBC`, `yMaxBC`, `zMinBC`, and `zMaxBC` fixed boundary values. A boundary may be a scalar applied uniformly across the face or a C-contiguous `float32` array with one value per site on that face; for example, an x-face array has `yDim*zDim` values in 3D. On a nonwrapped face, `None` means no flux through that face; wrapped axes are periodic and do not accept explicit boundary conditions. Field/interface arrays must match the grid geometry and dimensionality. Safe mode validates boundary size/type/finite values, explicit-scheme stability, and other inputs before changing pending state.

Advection

Advection transports a quantity along a specified velocity field, such as a chemical carried by flowing fluid. Unlike diffusion, it does not depend on concentration gradients. For example, start with a concentration at the center of a 2D field and move it toward increasing x with a constant positive x-velocity:

```
field = pal.NewPDEgrid((30, 30))
field.SetTimeSpaceStep(0.1, 1.0, 1.0)
field[15, 15] = 1.0
field.Advection(1.0, 0.0)
field.Update()
print(field[15, 15], field[16, 15])  # source decreases; next site gains
```

The velocity arguments specify motion along x and y. Positive x-velocity moves concentration toward increasing x; the timestep and grid spacing determine the distance moved per update. Here the flow advances only a fraction of one lattice cell in the timestep, so the concentration is distributed between sites rather than shifted by a whole cell. As with diffusion, the transport call queues changes until `Update()`.

Choose the advection method according to how velocity is represented. These are **reference signatures**, not a sequence of operations:

```
field.Advection(vx, vy=0.0, vz=0.0, ...)
field.AdvectionField(xVels, yVels=None, zVels=None, ...)
field.AdvectionInterfaces(xVels, yVels=None, zVels=None, ...)
```

Advection uses a conservative first-order upwind discretization. `Advection` uses constant velocity components. `AdvectionField` uses cell-centered velocity fields and averages adjacent values at interior faces. `AdvectionInterfaces` supplies positive-face velocities directly; the negative face uses the neighboring cell's stored positive-face velocity. Supply velocity components only for dimensions that exist. The same scalar-or-face-array Cartesian boundary arguments used by diffusion are available: `None` closes a nonwrapped face to advective flux, while supplied values provide the exterior concentration used for inflow. Safe mode checks the CFL condition before accumulating changes.

Visualization

Use PAL's visualization components to inspect a running model. `PixWindow` displays lattice pixels; OpenGL windows display 2D or 3D geometry. Rendering remains separate from simulation computation.

PixWindow

Choose `PixWindow` for a two-dimensional lattice view where each site or pixel is assigned a color. It is well suited to displaying cell occupancy, population counts, and scalar fields without drawing individual geometric shapes.

```
pix, window = pal.StartPixWindow(30, 30, scale=12, title="PAL")
```

Return type: `StartPixWindow(...) -> tuple[pal.Pix, pal.PixWindow]`. The first returned object, `pix`, has type `pal.Pix`; the second, `window`, has type `pal.PixWindow`. Use these public types when annotating functions that accept either object, for example `def Draw(pix: pal.Pix): ...` or `def Refresh(window: pal.PixWindow): ...`.

This creates a 30 × 30 pixel buffer and a display window, with each model pixel enlarged for visibility. Set `headless=True` when generating frames without an on-screen window.

`pix` is a write-oriented pixel buffer, not a general-purpose numeric `Grid`: it supports assignment but does not expose a public pixel-read API. Assign integer RGB colors by coordinate or linear index. For example, to color one site red and then display it:

```
pix[10, 12] = pal.RGB(255, 0, 0)
window.Update()
```

You can also assign packed RGB integers such as `0xFF0000`, and use a linear site index (`pix[i]`) instead of coordinates.

To display the nutrient field introduced in tutorial stage 6, use a second pixel window and map concentrations from 0 (black) to 1 (white). This matches the runnable tutorial files:

```
NUTRIENT_COLORS = (pal.RGB(0, 0, 0), pal.RGB(255, 255, 255))

def DrawNutrient(nutrient: pal.PDEgrid, pix: pal.Pix):
```

```
    for i in range(len(nutrient)):
        pix[i] = pal.ColorScale(NUTRIENT_COLORS, max(0.0, min(1.0, nutrient[i])))

nutrientPix, nutrientWindow = pal.StartPixWindow(SIZE, SIZE, scale=12, title="Nutrient")
DrawNutrient(nutrient, nutrientPix)
nutrientWindow.Update()
```

Call `DrawNutrient(nutrient, nutrientPix)` and `nutrientWindow.Update()` after each nutrient update. The normalized concentration determines the grayscale intensity. Manage both windows together and close them when the simulation ends.

It also provides `xDim`, `yDim`, `ToI(x, y)`, `ItoX(i)`, `ItoY(i)`, and `len(pix)`.

The window lifecycle offers the following operations (shown as reference calls, not a sequence to execute verbatim):

```
window.Update()
window.Save(path, block=False)
window.StartGif(path, delay=100)
window.AddGifFrame(block=False)
window.StopGif()
window.IsOpen()
window.Close()
```

For example, after creating `pix` and `window` as above, record two differently colored frames:

```
window.StartGif("frames.gif")
pix[:] = pal.RGB(0, 0, 0)
pix[10, 12] = pal.RGB(255, 0, 0)
window.Update()
window.AddGifFrame()
pix[10, 12] = pal.RGB(0, 0, 255)
window.Update()
window.AddGifFrame()
window.StopGif()
```

`Update()` publishes the current pixel buffer to the renderer. `StartGif()` begins a GIF recording, each `AddGifFrame()` appends the currently rendered frame, and `StopGif()` finalizes the file. `block=True` on asynchronous output operations waits for that requested output to complete before returning; call `StopGif()` before relying on the GIF file as complete.

OpenGLWindow

Choose `OpenGLWindow` to draw circles, boxes, and lines, including in 3D. Unlike `PixWindow`, it renders geometric shapes rather than individual colored pixels.

```
draw, window = pal.StartOpenGLWindow(30, 30, width=800, height=800, title="PAL")
```

Return type: `StartOpenGLWindow(...) -> tuple[pal.OpenGLDraw, pal.OpenGLWindow]`. The first returned object, `draw`, has type `pal.OpenGLDraw`; the second, `window`, has type `pal.OpenGLWindow`. Use these public types in function signatures, for example `def Draw(draw: pal.OpenGLDraw): ...` or `def Refresh(window: pal.OpenGLWindow): ...`.

This opens an 800 × 800 window displaying a 30 × 30-unit 2D drawing area. Supply `zDim` for a three-dimensional scene; `headless=True` requests off-screen rendering.

To draw and display a red circle centered at `(15, 15)`:

```
draw.Circle(3.0, pal.RGB(255, 0, 0), 15.0, 15.0)
window.Update()
```

Other drawing operations include:

```
draw.Circle(rad, color, x, y, z=0.0)
draw.Box(xLen, color, x, y, z=0.0, yLen=None, zLen=None)
draw.BoxSQ(color, x, y, z=0.0)
draw.Line(width, color, x1, y1, x2, y2, z1=0.0, z2=0.0)
```

These are reference signatures with placeholder arguments, not a runnable example.

Window controls include `Borders(width, color)`, `Camera(x, y, z, yaw=None, pitch=None)`, `Background(color)`, `Clear()`, `Update()`, `IsOpen()`, `Save(path, block=False)`, `StartGif(path, delay=100)`, `AddGifFrame(block=False, timeout=30)`, `StopGif(timeout=30)`, and `Close()`. The GIF lifecycle is the same as for `PixWindow`: start recording, append rendered frames, then stop to finalize the file.

Colors are packed integer RGB values such as `0xFF0000`. `pal.ColorScale(colors, value)` interpolates RGB channels across a sequence of colors for normalized `value`; values outside `[0, 1]` clamp to the endpoint colors.

Set `headless=True` for off-screen image/GIF generation. OpenGL headless rendering requires a supported standalone backend such as EGL.

Use `pal.AwaitWindows()` when model execution should wait for PAL windows to finish their lifecycle.

Saving PAL objects

Use Python's `pickle` module to save and reload PAL objects for checkpoints or reproducible experiments. Load pickle files only from trusted sources.

Example: save and load model state

After creating `cells` and `nutrient` in the count-based tutorial, save them from ordinary Python code, not from an `@pal.njit` function:

```
import pickle

with open("model.pkl", "wb") as f:
    pickle.dump({"cells": cells, "nutrient": nutrient}, f)

with open("model.pkl", "rb") as f:
    state = pickle.load(f)
```

```
cells = state["cells"]
nutrient = state["nutrient"]
```

This restores the saved grids, including their current values. It does **not** restore the random-number generator's state or a display window. To resume a simulation, also save any needed timestep counters and other model state, and recreate external resources such as windows. A fixed seed can reproduce a run from its beginning when the initial state and sequence of random calls match, but it does not by itself resume the same stochastic trajectory halfway through.

Only load pickle files you trust: loading a malicious pickle can execute arbitrary Python code.

What a checkpoint preserves

PAL saves model state rather than process-local pointers. The saved contents depend on the object:

Object	Preserved state
`AgentGrid`	Geometry, occupancy, agents, and properties
`PopGrid`	Geometry, capacity, current counts, and pending deltas
`PDEgrid`	Geometry, time/space steps, current field, and pending deltas
`IList`	Stored integer identifiers
`Multinomial`	Current `Setup` / `Sample` remainder

Safe/fast identity is also restored. In particular, a `PopGrid` or `PDEgrid` can be saved before `Update()` without losing its queued changes.

Organizing larger models

Once a model uses several grids and parameter arrays, it can be useful to group them into a single object. This organization is optional.

Package model components with a named tuple

Passing every grid and parameter separately can make signatures unwieldy. Python's `typing.NamedTuple` groups related objects under named fields that Numba can access in compiled functions.

For example, package an agent grid, a nutrient field, and a NumPy array of rates. The three named-tuple fields hold references to those objects; the grids and array allocate their own storage. The named tuple does not need to reserve space for every cell or nutrient value:

```
from typing import NamedTuple
import numpy as np
import PythonAutomataLibrary as pal

class Model(NamedTuple):
    cells: pal.AgentGrid
    nutrient: pal.PDEgrid
    rates: np.ndarray
```

```
@pal.njit(cache=True)
def Consume(model: Model):
    for agent in model.cells.All():
        x, y = model.cells.XSQ(agent), model.cells.YSQ(agent)
        model.nutrient[x, y] -= model.rates[0]

@pal.njit(cache=True)
def Step(model: Model):
    Consume(model)
    model.rates[1] += 1.0

def main():
    cells = pal.NewAgentGrid((8, 8), isStackable=False)
    cells.NewAgentSQ(3, 3)
    nutrient = pal.NewPDEgrid((8, 8))
    nutrient[:] = 1.0
    model = Model(cells=cells, nutrient=nutrient, rates=np.array([0.1, 0.0]))
    Step(model)
    print(model.nutrient[3, 3], model.rates[1])

if __name__ == "__main__":
    main()
```

The `NamedTuple` class gives editors field names and type information for autocomplete, including `model.cells` and `model.nutrient`. Both `Step` and `Consume` receive the same named model, and changes to the contained grids and arrays persist. The named tuple itself is immutable: `model.cells = other_cells` is not allowed. For a changing scalar, use a mutable array as shown with `rates[1]`, or return an updated value.

Performance: Numba compiles named-field access, but passing a composite from Python into a compiled function can have conversion overhead. Prefer calling multiple compiled functions from a compiled timestep, as above, rather than repeatedly entering compiled code for tiny operations. Whether the composite is faster than separate arguments depends on the workload; benchmark performance-sensitive applications. See Numba's supported Python features.