Short version: nothing is “wrong” with your biology, the issue is your numerics. Forward Euler on a logistic-like ODE **does not** guarantee positivity. If your time step is too big relative to the effective growth rate and carrying capacity, the scheme can overshoot and drive a state negative—even if the underlying continuous model can’t.

Let’s unpack that in your specific code.

---

1. Your update equation *can* go negative

Take one of your state updates, e.g. for leaves:

```
N[t+1] = N[t] + dt * (aN_hat * N[t] * (1 - N[t]/max(kN_hat, 1e-9)))
```

This is Forward Euler applied to the logistic ODE

$$
\frac{dN}{dt} = a_{N,\text{hat}}, N \left(1 - \frac{N}{k_{N,\text{hat}}}\right).
$$

In continuous time, with $N(0) \ge 0$, the solution stays ≥ 0 forever.

In discrete time with Euler, you’re doing:

$$
N_{t+1} = N_t + \Delta t , a_{N,\text{hat}}, N_t \left(1 - \frac{N_t}{k_{N,\text{hat}}}\right).
$$

Factor out N_t:

$$
N_{t+1} = N_t \Big[1 + \Delta t , a_{N,\text{hat}} \left(1 - \frac{N_t}{k_{N,\text{hat}}}\right)\Big].
$$

So:

- If $N_t < k_{N,\text{hat}}$, the term in parentheses is positive, so you grow.
- If $N_t > k_{N,\text{hat}}$, that term is negative, which is how the model “pushes” you back toward the carrying capacity.

But if the negative part is **too large in magnitude**, the bracket can become negative:

$$
1 + \Delta t , a_{N,\text{hat}} \left(1 - \frac{N_t}{k_{N,\text{hat}}}\right) < 0
$$

which implies

$$
\Delta t , a_{N,\text{hat}} \left(\frac{N_t}{k_{N,\text{hat}}} - 1\right) > 1.
$$

When that inequality holds, you get $N_{t+1} < 0$.

So even without any explicit decay term, **Euler stepping can overshoot below zero** once a state gets above its effective carrying capacity and the time step is too large for the local dynamics.

---

2. Why it hits leaves / spikelets especially

The part that makes this more likely in your code is the time-varying effective parameters:

```
# Nutrient factors (bounded, nonnegative)
nuW = get_nutrient_factor(...)
nuF = get_nutrient_factor(...)
nuT = get_nutrient_factor(...)
nuR = get_nutrient_factor(...)

# Growth rates
aN_hat = aN        # simpler here, but aN might already be big

# Carrying capacities
kN_hat = kN * (nuT * nuR)**(1/2)
```

Things to notice:

- If **nuT** or **nuR** get very small (e.g., poor temperature / radiation), then $(\nu_T \nu_R)^{1/2}$ can be tiny → **kN_hat becomes very small**.
- If N_t hasn't dropped correspondingly, the ratio N_t / kN_{hat} can become huge.
- Then the logistic term $1 - N_t / kN_{hat}$ is a large negative number, and the Euler step happily takes a huge negative step.

Same logic applies to flower spikelets if their (a) and (k) get into ugly combinations via the nutrient factors.

You already avoid divide-by-zero with:

```
max(kN_hat, 1e-9)
```

but that only prevents NaNs; it does **not** prevent "massive negative derivative times dt".

So the root cause is:

Effective carrying capacity gets very small or state gets too large, and the Euler step is too big to stay in [0, k].

---

3. This is a known Euler problem, not you reinventing chaos (on purpose)

Forward Euler applied to stiff or strongly nonlinear systems is notorious for:

- breaking invariants (like non-negativity),
- overshooting,
- and even producing chaotic behavior in otherwise tame continuous systems.

Your model is basically a **time-varying logistic map** embedded in a larger system. Logistic maps are famous for doing weird stuff when parameters are large; Euler is turning your smooth ODE into something closer to that discrete logistic map in regimes with big effective growth.

---

4. Ways to fix or mitigate it

Here are practical knobs you can turn:

a) Reduce `dt` or substep the integration

Easiest conceptual fix:

- Make `dt` smaller so that (\Delta t , a_{\text{eff}} (N_t/k_{\text{eff}} - 1)) stays well below 1 even in the worst case.

If you don't want to change global `dt`, you can substep inside the loop when conditions are harsh:

```
substeps = 4  # or adapt based on aN_hat, kN_hat, etc.
sub_dt = dt / substeps
for _ in range(substeps):
    N[t+1] = N[t] + sub_dt * (aN_hat * N[t] * (1 - N[t]/max(kN_hat, 1e-9)))
    N[t+1] = max(N[t+1], 0.0)  # see (c)
    N[t] = N[t+1]
```

You'd do similarly for h, A, c, P.

b) Tame the nutrient scaling

If your `get_nutrient_factor` can produce values very close to 0 or very large, the resulting `k_hat` and `a_hat` can explode or collapse.

Common trick: clamp nutrient factors to reasonable ranges, e.g.:

```
nuW = np.clip(get_nutrient_factor(...), 0.1, 2.0)
```

You can also clamp the effective carrying capacities:

```
kN_hat = np.clip(kN_hat, kN_min, kN_max)
```

where `kN_min` is some biologically reasonable minimum (so you never "compress" the carrying capacity to nearly zero).

c) Enforce non-negativity explicitly

Not elegant mathematically, but very common in ecological / biological simulations:

```
h[t+1] = max(h[t+1], 0.0)
A[t+1] = max(A[t+1], 0.0)
N[t+1] = max(N[t+1], 0.0)
c[t+1] = max(c[t+1], 0.0)
P[t+1] = max(P[t+1], 0.0)
```

Or even:

```
if N[t+1] < 0:
    N[t+1] = 0
```

Physically, negative leaves don't exist; biologically you might reinterpret this as "plant lost almost all leaves, state floored at 0."

For counts (leaves, spikelets), another fancy option is to model them in log-space or with a softplus:

- Evolve a variable (z), and define (N = \exp(z)) or (N = \text{softplus}(z)), which is always positive.
- This requires rewriting the dynamics, though, so it's more invasive.

d) Use a more stable integrator

If performance permits, you can switch from hand-coded Euler to something like Runge–Kutta (RK4) or a semi-implicit method.

For example, you could implement a simple RK4 step for each variable. It still won't *guarantee* positivity, but it's much more stable for the same `dt`.

Since you're calling this inside a GA cost function, you'd need to balance:

- number of individuals × number of time steps × integrator complexity.

But even a low-overhead improvement like Heun's method (a 2-stage Runge–Kutta) can help a lot.

---

5. How to debug *where* it happens

A quick practical check:

```
for t in range(total_time_steps - 1):
    ...
    N_pred = N[t] + dt * (aN_hat * N[t] * (1 - N[t]/max(kN_hat, 1e-9)))
    if N_pred < 0:
        print("Negative N step at t=", t)
        print("N[t]=", N[t], "aN_hat=", aN_hat, "kN_hat=", kN_hat, "dt=", dt)
    N[t+1] = max(N_pred, 0.0)
```

My bet: you'll catch cases where `kN_hat` is extremely small or `aN_hat * dt` effectively huge.

So the punchline:

- The model equations themselves are fine: logistic growth can go down but not below zero in continuous time.
- The **combination of Forward Euler + time-varying tiny carrying capacities / large growth rates + finite dt** lets the discrete approximation overshoot below zero.
- Fix it by shrinking `dt` or substepping, clamping extremes in nutrient scaling and capacities, and optionally enforcing non-negativity.

You've basically built a little nonlinear dynamical system zoo; what you're seeing is it misbehaving in the discrete-time cage, not in the continuous-time wild.

For speed, I don't want to decrease dt. Would using RK4 instead of Forward Euler help?

Short answer: RK4 will *probably* reduce how often things blow up or go negative for the same `dt`, but it **does not guarantee** positivity. It's a nicer Euler, not a magic physics cop.

So if your constraint is "I really don't want to shrink `dt`", I'd consider three options, in roughly this order:

1. RK4: what it helps and what it doesn't

Right now a single state does:

```
N[t+1] = N[t] + dt * f(N[t])
# where f(N) = aN_hat * N * (1 - N/kN_hat)
```

Euler is first-order and has a small stability region. RK4 replaces that by:

```
k1 = f(N_t)
k2 = f(N_t + 0.5*dt*k1)
k3 = f(N_t + 0.5*dt*k2)
k4 = f(N_t + dt*k3)
N_{t+1} = N_t + dt*(k1 + 2*k2 + 2*k3 + k4)/6
```

With your code structure, note something nice:

- All the **nutrient factors** and **effective parameters** `a*_hat`, `k*_hat` depend only on `t` and the cumulative drivers (`WC`, `RC`, etc.), **not** on the plant states.
- That means inside each time step, the RHS is just (f(h, A, N, c, P)) with *fixed* `a*_hat`, `k*_hat`.

So you can:

1. Compute nutrient factors + `a*_hat`, `k*_hat` **once per step**.
2. Use those in RK4 stages for all 5 state variables.

Cost-wise, that's ~4x more multiplications/adds for the states, but **no extra nutrient-factor calls**. For a GA inner loop this might be acceptable.

This will:

- Reduce overshoot,
- Give you better accuracy for the same `dt`,
- Shrink the region where you get wild behavior.

But: in extreme cases (tiny `k_hat`, huge `a_hat`, large state), RK4 can still drive you negative. Explicit RK methods do not enforce invariants like "state ≥ 0".

So: **yes, RK4 helps stability**, **no, it doesn't solve the positivity problem in a guaranteed way**.

2. A very nice alternative: exact logistic step (same cost ballpark)

Because each variable follows a logistic-like ODE with *constant* `a_hat` and `k_hat` over the step, you can actually use the **closed-form solution** for each step instead of Euler.

For:

[
\frac{dy}{dt} = a,y\left(1 - \frac{y}{K}\right)
]

with constant (a,K > 0) over the step, the exact step from (y_n) to (y_{n+1}) over `dt` is:

[
y_{n+1} = \frac{K}{1 + \left(\frac{K}{y_n} - 1\right)e^{-a,dt}}.
]

This has two big perks:

- If (y_n > 0, a \ge 0, K > 0), then **(y_{n+1} > 0)** automatically.
- It's *very* stable; you don't need tiny `dt` for the logistic part.

You could implement a helper like:

```
def logistic_step(y, a, K, dt, eps=1e-12):
    K = max(K, eps)
    y = max(y, eps)
    exp_term = np.exp(-a * dt)
    return K / (1.0 + (K / y - 1.0) * exp_term)
```

Then inside your loop:

```
h[t+1] = logistic_step(h[t], ah_hat, kh_hat, dt)
A[t+1] = logistic_step(A[t], aA_hat, kA_hat, dt)
N[t+1] = logistic_step(N[t], aN_hat, kN_hat, dt)
c[t+1] = logistic_step(c[t], ac_hat, kc_hat, dt)
P[t+1] = logistic_step(P[t], aP_hat, kP_hat, dt)
```

This is still per-step work on the same order as Euler + a few `exp` calls, and far cheaper than cutting `dt` by a factor of, say, 4–10.

If any of your `a_hat` can be negative, you'd want to think a bit about what "logistic" means biologically there, but for your use-case they look like growth rates.

---

3. Cheap guardrails if you stay with Euler (or RK4)

Regardless of method, you can add safety rails with negligible cost:

- Clamp nutrient factors / capacities so `k*_hat` never gets absurdly tiny.
- Enforce non-negativity after stepping:

```
h[t+1] = max(h[t+1], 0.0)
A[t+1] = max(A[t+1], 0.0)
N[t+1] = max(N[t+1], 0.0)
c[t+1] = max(c[t+1], 0.0)
P[t+1] = max(P[t+1], 0.0)
```

For a GA objective, this kind of clipping is often totally acceptable: a schedule that tries to drive plants into physically impossible regimes just gets penalized.

TL;DR for design choice

- **RK4**: better stability and accuracy, same `dt`, ~4× state-update cost, still not positivity-guaranteed.
- **Exact logistic step**: very stable, naturally positivity-preserving (under mild assumptions), a bit more math but similar cost per step.
- **Clamping + Euler**: simplest hack, numerically crude but often absolutely fine for GA.

If your priority stack is “speed > elegance > ironclad positivity”, I’d do:

1. Switch each state update to the exact logistic step;
2. Add a floor at 0 as a final guard;
3. Optionally clamp extreme nutrient factors/carrying capacities.

That way your simulated plants don’t mysteriously develop negative flowers, even when your GA explores weird corners of design space.