ARGoS Wind / Air Resistance — Developer Guide

Roi Sela

October 2025

Repository: project link

Contents

1 Repository Overview **2**

2 Architecture **2**
2.1 Core Controller: `CAirResistance` 2

3 Configuration Interface (recap) **3**

4 Algorithmic Details (with Equations) **3**
4.1 Impulse Model (first definition of $\mathbf{w}_{\text{eff}}$ + numeric example) 3
4.2 Wake Blocking Reduction 4

5 Pseudocode: Core Functions **5**
5.1 `Init(t_node)` 5
5.2 `EnsurePhysicsHandle()` 6
5.3 `HandleAerodynamicsPreStep()` 6
5.4 `ComputeEffectiveWind()` 7
5.5 `IsBlockedByRAB(out r)` 7
5.6 `DriveImpulse(v_cm_s)` 8
5.7 `ApplyWindImpulse()` 8
5.8 `HandleAerodynamicsPostStep()` 8
5.9 `GetYawRadians()` 8

6 Tuning and Debugging **9**

Audience & Scope

This guide targets developers who will extend, maintain, or embed the project's wind / air-resistance behavior in ARGoS 3 (dynamics2d / Chipmunk engine). It covers architecture, the impulse pipeline, aerodynamic blocking via RAB, configuration schema, and extension points. Language-agnostic pseudocode is provided for core functions.

1 Repository Overview

Path	Purpose
`controllers/air_resistance`	Base controller (`CAirResistance`): wind + drive impulse pipeline, RAB-based aerodynamic blocking.
`controllers/wind_aware`	Example derived controller (behavior details are documented in the user manual).
`loop_functions/wind_loop_functions`	Logic-only loop functions to read wind from XML and expose it.
`loop_functions/wind_loop_functions(Qt)`	Qt user functions that draw the wind vector arrow in the world view.
`examples/`	Minimal runnable ARGoS configs (documented in the user manual).

2 Architecture

2.1 Core Controller: `CAirResistance`

Lifecycle and Extensibility. `CAirResistance` derives from `CCI_Controller`. Lifecycle methods are virtual and designed for subclassing.

Important protected hooks (what they do).

- `EnsurePhysicsHandle()` — Idempotently cache the Chipmunk body (pointer, mass, space, COM) and derive the robot's effective radius in meters from the AABB. Safe to call every tick.
- `GetYawRadians()` — Return yaw ψ (world frame) from the positioning sensor; used for the forward unit vector $\hat{\mathbf{f}}(\psi)$.
- `HandleAerodynamicsPreStep()` — Reset $\mathbf{J}_{\text{sum}}$; add the wind impulse using the current effective wind $\mathbf{w}_{\text{eff}}$; broadcast the RAB radius (mm). No post-step scheduling here.
- `HandleAerodynamicsPostStep()` — Register a Chipmunk post-step callback to apply $\mathbf{J}_{\text{sum}}$ after collisions.
- `ComputeEffectiveWind()` — Compute and return $\mathbf{w}_{\text{eff}} = (1 - r)\mathbf{w}$ where r is the wake reduction computed from neighbors (see Sec. 4.2).
- `IsBlockedByRAB(Real& r)` — From RAB neighbors, compute $r \in [0, 1]$ via geometric overlap of wakes (details in Sec. 4.2).
- `DriveImpulse(Real v_cm_s)` — Add forward impulse $\frac{m}{100} v\,\hat{\mathbf{f}}(\psi)$ (cm/s $\rightarrow$ m/s factor 1/100).
- `ApplyWindImpulse()` — Add $\frac{m}{100}\,\mathbf{w}_{\text{eff}}$ to the per-tick accumulator.

Impulse Pipeline (per tick).

1. **Pre-step**: Reset the impulse accumulator; compute and add the wind impulse (possibly reduced by blocking); broadcast radius via RAB.
2. **Drive**: Add a forward drive impulse along current yaw using the desired base speed.
3. **Post-step**: Schedule a single Chipmunk *post-step* callback to apply the summed impulse after collisions.

Physics Integration. We downcast the ARGoS `dynamics2d` model to either `CDynamics2D SingleBodyObjectModel` (e-puck2) or `CDynamics2DMultiBodyObjectModel` (foot-bot). For multi-body robots, body index 0 is treated as the chassis. AABB is used to derive a reasonable self-radius R (meters), clamped to sane limits.

3 Configuration Interface (recap)

Under `<configuration>` declare:

```
<air_resistance angle_deg="0" magnitude="15.0"/>
```

`angle_deg` is the global wind direction (degrees, world frame); `magnitude` is wind speed in cm/s. Each controller instance accepts:

```
<params velocity="15.0"/>
```

4 Algorithmic Details (with Equations)

4.1 Impulse Model (first definition of $\mathbf{w}_{\text{eff}}$ + numeric example)

Per tick we accumulate world-frame impulses:

$$\mathbf{J}_{\text{wind}} = \frac{m}{100} \underbrace{\mathbf{w}_{\text{eff}}}_{\text{effective wind in cm/s}}, \tag{1}$$

$$\mathbf{J}_{\text{drive}} = \frac{m}{100}\, v_d\, \hat{\mathbf{f}}(\psi), \tag{2}$$

$$\mathbf{J}_{\text{sum}} = \mathbf{J}_{\text{wind}} + \mathbf{J}_{\text{drive}}. \tag{3}$$

Definition. Let $\mathbf{w}$ be the global wind and $r \in [0,1]$ the reduction due to upwind neighbors. The effective wind is

$$\boxed{\mathbf{w}_{\text{eff}} = (1-r)\,\mathbf{w}} \quad \text{(with } r \text{ computed in Sec. 4.2).}$$

Here m is body mass (Chipmunk), v_d the desired forward speed (cm/s), and $\hat{\mathbf{f}}(\psi)$ the unit forward vector given yaw ψ. A single post-step callback applies $\mathbf{J}_{\text{sum}}$ at the COM after collisions.

Numeric example (with picture). Let $m = 0.08\,\text{kg}$, $v_d = 10\,\text{cm/s}$, and $\mathbf{w} = (20\,\text{cm/s}, 0)$. If $r = 0.4 \Rightarrow \mathbf{w}_{\text{eff}} = (12\,\text{cm/s}, 0)$. With yaw $\psi = 60°$, $\hat{\mathbf{f}}(\psi) = (\cos 60°, \sin 60°) = (0.5, 0.866)$.

$$\mathbf{J}_{\text{wind}} = \tfrac{0.08}{100}(12,0) = (0.0096, 0), \quad \mathbf{J}_{\text{drive}} = \tfrac{0.08}{100} \cdot 10\,(0.5, 0.866) = (0.0040, 0.00693),$$

so $\boxed{\mathbf{J}_{\text{sum}} \approx (0.0136,\ 0.00693)\ \text{kg m/s}}$.

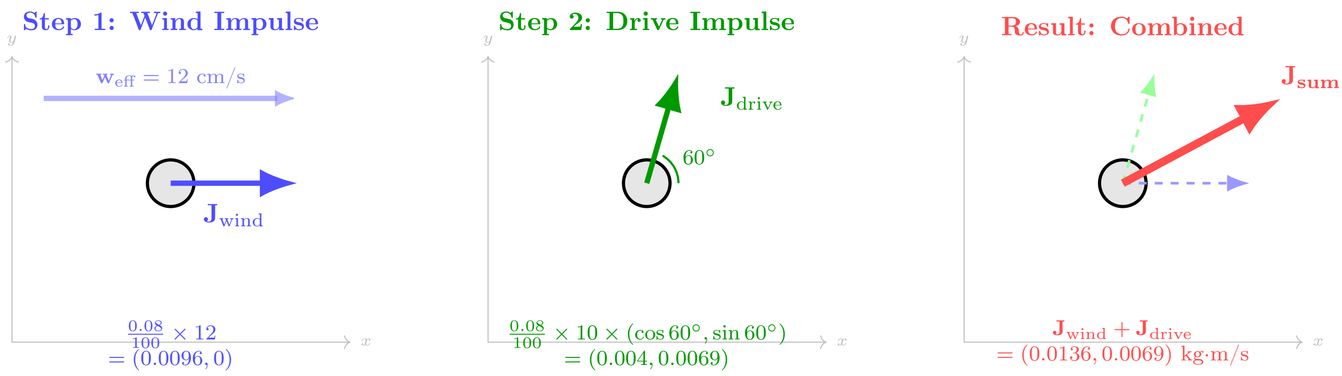

Figure 1: Impulse composition example with $m = 0.08$ kg, $v_d = 10$ cm/s, $\mathbf{w}_{\text{eff}} = 12$ cm/s, and $\psi = 60°$. **Left:** Wind pushes horizontally. **Middle:** Robot drives at 60°. **Right:** Both impulses combine (vector addition) to produce the actual motion impulse applied to the robot.

4.2 Wake Blocking Reduction

Problem we are solving. We need the shielding effect from an upwind blocker to fade out *smoothly* with downwind distance. If we used a hard cutoff (on/off at some distance), tiny sensor noise or sub-tick motion would make the effect flicker: one tick "blocked", next tick "not blocked". That produces visible jerks and unstable dynamics. A smooth ramp avoids this.

For each upwind neighbor i, compute an overlap score

$$r_i = \exp\left(-\tfrac{1}{2}\left(\tfrac{\ell_\perp}{\sigma_\perp}\right)^2\right) \cdot \text{smoothstep}\big(0, L;\, \ell_\parallel - g\big), \tag{4}$$

and take the boosted, clamped maximum

$$\boxed{r = \text{clamp}\left(0, 1,\ \gamma \cdot \max_{i \in \mathcal{N}} r_i\right)}, \qquad \mathbf{w}_{\text{eff}} = (1 - r)\mathbf{w}.$$

Here $\mathcal{N}$ is the set of upwind neighbors; $\max_i$ means the *maximum over neighbors.*

What is smoothstep **(and why use it)?** smoothstep$(a, b; x)$ maps x from $[a, b]$ to a smooth ramp in $[0, 1]$:

$$\text{smoothstep}(a, b; x) = \begin{cases} 0 & x \le a, \\ 3t^2 - 2t^3 & a < x < b, \quad t = \dfrac{x - a}{b - a}, \\ 1 & x \ge b. \end{cases}$$

Its slope is zero at both ends, so the blocking grows and dies out gently instead of snapping on and off. In our case we apply it to $\ell_\parallel - g$ over the interval $[0, L]$, which means: once the follower is a bit downwind of the blocker (past the upwind gate g), the reduction ramps in smoothly and reaches full effect by distance L.

Parameter meanings (visualized in Fig. 2 below).

- $\ell_\parallel$: along-wind separation, i.e., the projection of the blocker→follower vector onto the wind direction; it is the *downwind distance from the blocker to the follower* (positive when the follower is downwind).
- $\ell_\perp$: lateral offset from the wind centerline between the two robots.

- R: robot radius in meters, derived from the AABB of the embodied entity (see Sec. 3); used as the base length scale.
- $\sigma_\perp$ = `lateral_reach_radii` $\cdot R$: lateral "width" of the wake (controls the Gaussian term).
- L = `shadow_length_radii` $\cdot R$: downwind "length" of the wake (the smoothstep interval).
- g = `upwind_gate_radii` $\cdot R$: small upwind gap to ignore near side-by-side cases before the ramp begins.
- γ = `gamma_boost`: optional emphasis of mid-range overlaps before clamping to $[0, 1]$.

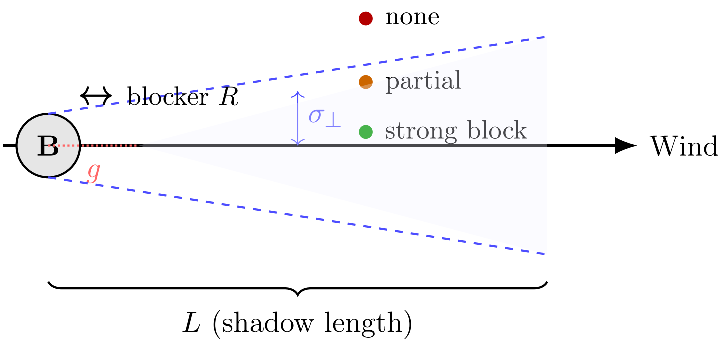

Figure 2: Blocking geometry. Adjusting $\sigma_\perp$ (lateral reach), L (shadow length), g (upwind gate), and γ (boost) reshapes the wake and the final reduction r.

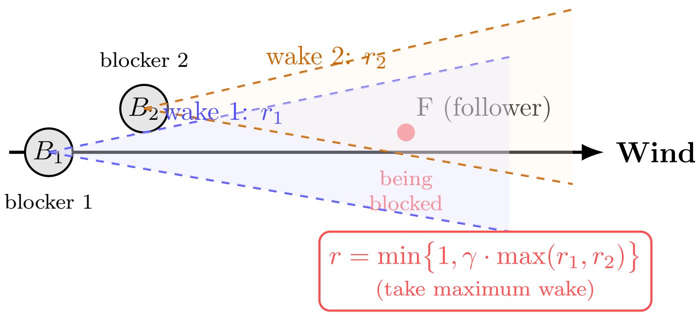

Figure 3: Two upwind blockers (B_1 and B_2) create overlapping wakes. The follower robot F experiences blocking from both. Each blocker produces r_i via (4). The final reduction r is the boosted/clamped *maximum* over all neighbor wakes—meaning F gets blocked by whichever wake is strongest at its position.

Two-blocker illustration (max over neighbors).

5 Pseudocode: Core Functions

Variables mirror the C++ but remain language-agnostic. Distances in meters unless noted; speeds in cm/s. Comments use only ASCII to avoid listing-encoding issues.

5.1 Init(t_node)

```
function Init(node)
  // devices
  rab_act <- GetActuator("range_and_bearing")
```

```
  rab_sen <- GetSensor ("range_and_bearing")
  pos_sen <- GetSensor ("positioning")
  wheels <- GetActuator("differential_steering")

  // controller params
  v_desired_cm_s <- ReadParam(node, "velocity", default=10.0)

  // wind from <configuration><air_resistance .../>
  (angle_deg, magnitude_cm_s) <- ReadGlobalWind()
  wind_cm_s <- (magnitude_cm_s * cos(rad(angle_deg)),
               magnitude_cm_s * sin(rad(angle_deg)))
end
```

5.2 EnsurePhysicsHandle()

```
function EnsurePhysicsHandle()
  if body_ready: return

  // fetch entity + embodied component
  entity <- GetSelfEntity()
  embodied <- entity.GetComponent("body")

  // derive self radius from AABB (meters), clamp to sane range
  bb <- embodied.GetBoundingBox()
  width <- bb.MaxCorner.x - bb.MinCorner.x
  depth <- bb.MaxCorner.y - bb.MinCorner.y
  self_radius_m <- 0.5 * max(width, depth)
  self_radius_m <- Clamp(self_radius_m, 0.01, 0.50)

  // cache dyn2d body + mass etc.
  model <- GetDynamics2DModelForRobot()
  if model is SingleBody: body <- model.GetBody()
  else: body <- model.GetBody(index=0) // chassis
  mass_kg <- Chipmunk.GetMass(body)

  body_ready <- true
end
```

5.3 HandleAerodynamicsPreStep()

```
function HandleAerodynamicsPreStep()
  EnsurePhysicsHandle()

  // reset per-tick accumulator
  J_sum <- (0, 0)

  // wind contribution using effective wind (Sec. 4.1, 4.2)
  ApplyWindImpulse()

  // RAB broadcast: radius in millimeters (byte 0 only)
  if rab_act != null:
      r_mm <- Clamp(round(self_radius_m * 1000), 0, 255)
      rab_act.SetData(0, r_mm)
end
```

5.4 ComputeEffectiveWind()

```
function ComputeEffectiveWind()
  if Norm(wind_cm_s) < 1e-9: return wind_cm_s
  r <- 0.0
  if IsBlockedByRAB(out r): // r computed by wake model (see Sec. 4.2)
      return max(0, 1 - r) * wind_cm_s
  else:
      return wind_cm_s
end
```

5.5 IsBlockedByRAB(out r)

```
function IsBlockedByRAB(out r)
  r <- 0.0
  if pos_sen == null or rab_sen == null: return false
  if Norm(wind_cm_s) < 1e-9: return false

  // Tunables (geometry in "blocker radii")
  lateral_reach_radii <- 3.0 // controls lateral Gaussian width
  shadow_length_radii <- 4.0 // how far wake reaches downwind
  gamma_boost <- 2.0 // 1.0 disables boost
  upwind_gate_radii <- 0.5 // minimum along-wind gap

  // World-frame unit wind direction
  wind_dir_hat <- Normalize(wind_cm_s)

  // My yaw (world), used to rotate RAB bearings
  yaw_world <- GetYawRadians()

  any <- false
  for msg in rab_sen.GetReadings():

      // range in cm -> meters
      range_m <- 0.01 * msg.Range
      if range_m <= 1e-6: continue

      // blocker radius from advertised byte[0] (mm->m) else fallback
      blocker_r_m <- self_radius_m
      if Size(msg.Data) >= 1:
          adv_m <- 0.001 * msg.Data[0]
          if 0.005 < adv_m and adv_m < 0.20: blocker_r_m <- adv_m

      // bearing to neighbor in world frame
      bearing_world <- yaw_world + msg.HorizontalBearing

      // vector ME->OTHER and OTHER->ME (meters, world)
      me_to_other <- (range_m * cos(bearing_world),
                      range_m * sin(bearing_world))
      other_to_me <- -me_to_other

      // along-wind component (positive means neighbor is upwind)
      along_m <- Dot(other_to_me, wind_dir_hat)

      // upwind gate: require some min gap to avoid side-by-side cases
      gate_m <- max(1e-6, upwind_gate_radii * blocker_r_m)
      if along_m <= gate_m: continue
```

```
    // lateral offset from wind centerline
    lateral_vec <- other_to_me - along_m * wind_dir_hat
    lateral_m <- Norm(lateral_vec)

    // wake widths and fades (Sec. 4.2)
    sigma <- lateral_reach_radii * blocker_r_m
    L <- shadow_length_radii * blocker_r_m

    lateral_cov <- exp(-0.5 * (lateral_m / max(1e-6, sigma))^2)
    fade <- smoothstep(0, L, along_m - gate_m)
    red_i <- lateral_cov * fade

    // optional non-linear emphasis
    red_i <- 1 - (1 - red_i) ^ gamma_boost

    r <- max(r, red_i) // max over neighbors
    any <- true

  r <- min(1.0, r)
  return any and (r > 1e-6)
end
```

5.6 DriveImpulse(v_cm_s)

```
function DriveImpulse(v_cm_s)
  yaw <- GetYawRadians()
  fwd <- (cos(yaw), sin(yaw)) // unit vector, world frame
  J_sum += (mass_kg/100.0) * v_cm_s * fwd
end
```

5.7 ApplyWindImpulse()

```
function ApplyWindImpulse()
  w_eff_cm_s <- ComputeEffectiveWind()
  if Norm(w_eff_cm_s) < 1e-9: return
  J_sum += (mass_kg/100.0) * w_eff_cm_s
end
```

5.8 HandleAerodynamicsPostStep()

```
function HandleAerodynamicsPostStep()
  space <- Chipmunk.Space(body)
  Chipmunk.AddPostStepCallback(space, lambda:
    Chipmunk.ApplyImpulseAtWorldPoint(body, J_sum, position=COM))
end
```

5.9 GetYawRadians()

```
function GetYawRadians()
  q <- pos_sen.GetReading().Orientation()
  return q.Yaw()
end
```

6 Tuning and Debugging

Key tunables (typical defaults)

- `lateral_reach_radii` widens or narrows the Gaussian cross-section. Larger $\Rightarrow$ stronger blocking even with lateral offset.
- `shadow_length_radii` lengthens the downwind extent. Larger $\Rightarrow$ blocking persists farther downwind.
- `upwind_gate_radii` ignores near side-by-side neighbors (requires some minimum along-wind separation).
- `gamma_boost` emphasizes mid-range overlaps ($r \leftarrow \min(1, \gamma \cdot r)$).

Debug tips

- Print intermediate terms inside the blocking loop: $\ell_{\parallel}$, $\ell_{\perp}$, r_i, and the final r.
- Verify the on-screen wind arrow matches your XML.
- If a robot seems unaffected, check: advertised radius byte (mm) $\rightarrow$ meters, AABB-derived R, and that neighbors are truly upwind (positive $\ell_{\parallel}$).

Appendix: Minimal C++ touchpoints (orientation only)

Post-step application (Chipmunk)

```
cpSpaceAddPostStepCallback(space, ApplyAccumPostStep, m_ptBody, payload);
```

RAB radius broadcast (mm)

```
UInt8 r_mm = (UInt8) std::min(255.0, std::round(m_fSelfRadiusM * 1000.0));
m_pcRABAct->SetData(0, r_mm);
```

References

References

[1] D. H. Stolfi and G. Danoy, "Design and analysis of an E-Puck2 robot plug-in for the ARGoS simulator," *Robotics and Autonomous Systems*, vol. 164, p. 104412, 2023. doi: 10.1016/j.robot.2023.104412.