Lost in translation?!

Robotics with type-safe coordinate systems

Maximilian Schmidt, Rasmus Mecklenburg, Konrad Nölle

Who are we?

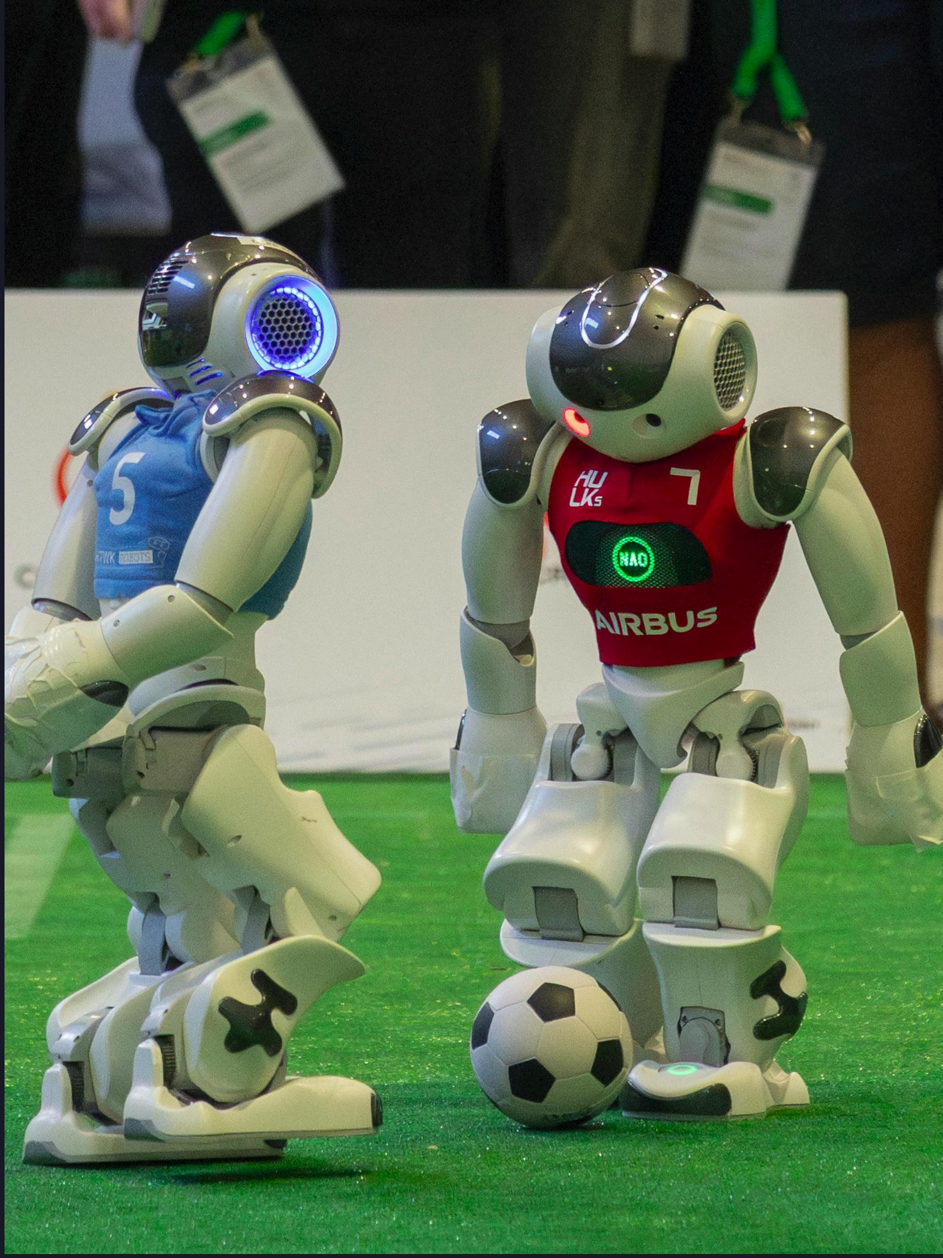
- Team from Hamburg (TUHH)
- Active in Robot Soccer since 2013
- Open Source Contribution to software and research papers

First team to seriously **integrate Rust** 🦀 in robot control

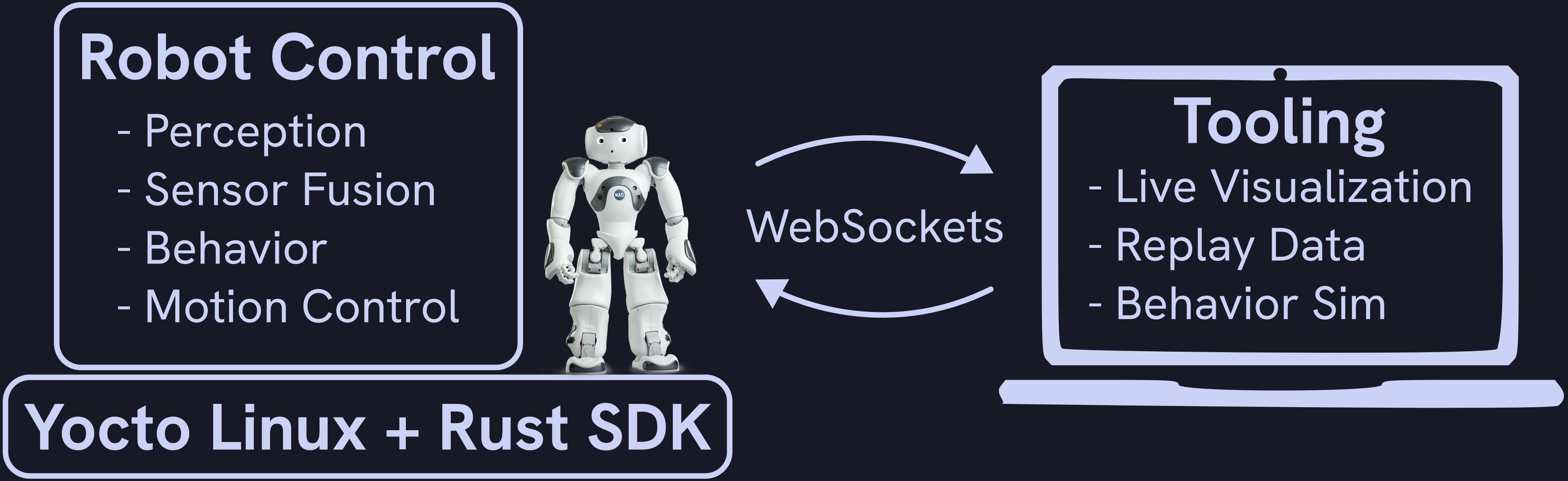
What is RoboCup?

- International competition
- autonomous soccer robots
- Standard Platform League (SPL):
  → all teams use identical robots

⟹ Real-time robotics under physical, strategic, and computational constraints



Where do we use Rust?



Robot Control

- Perception
- Sensor Fusion
- Behavior
- Motion Control

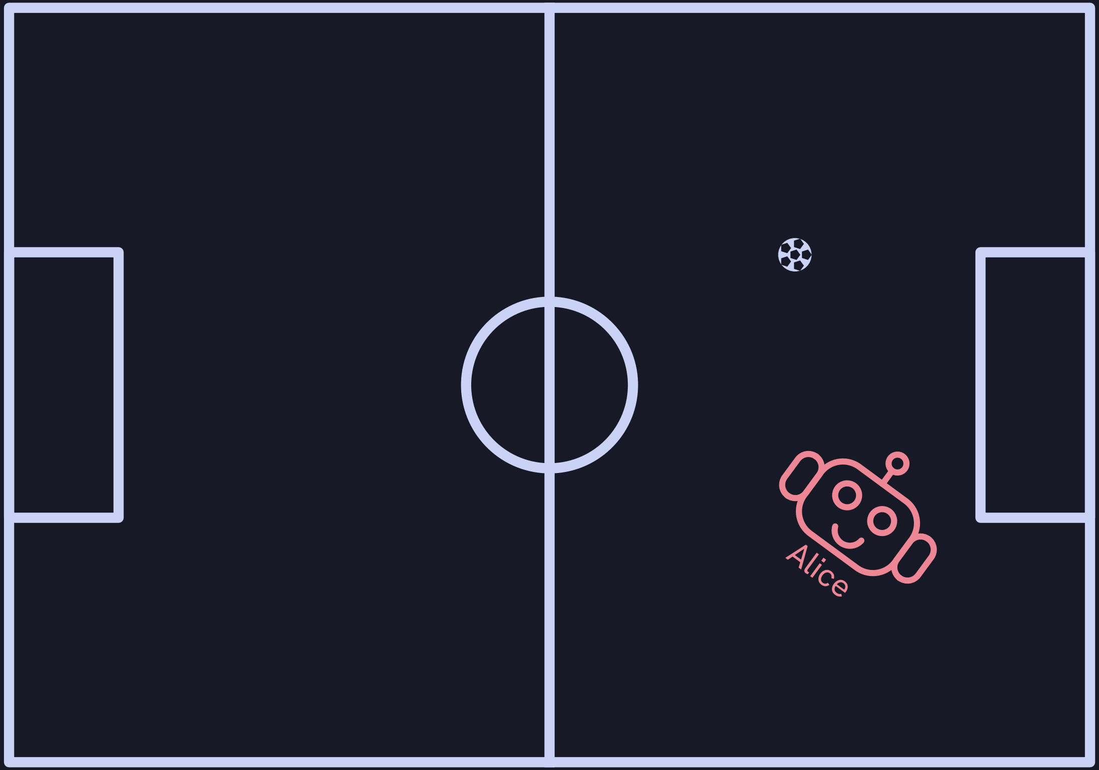
Yocto Linux + Rust SDK

WebSockets

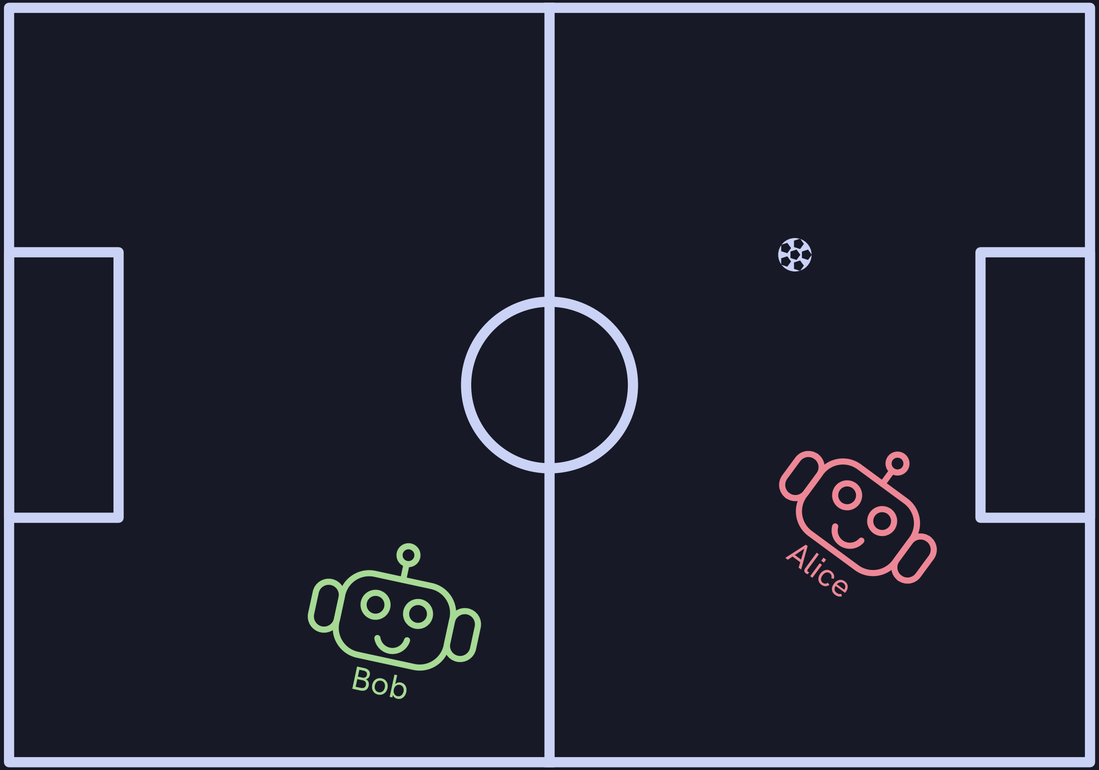
Tooling

- Live Visualization
- Replay Data
- Behavior Sim

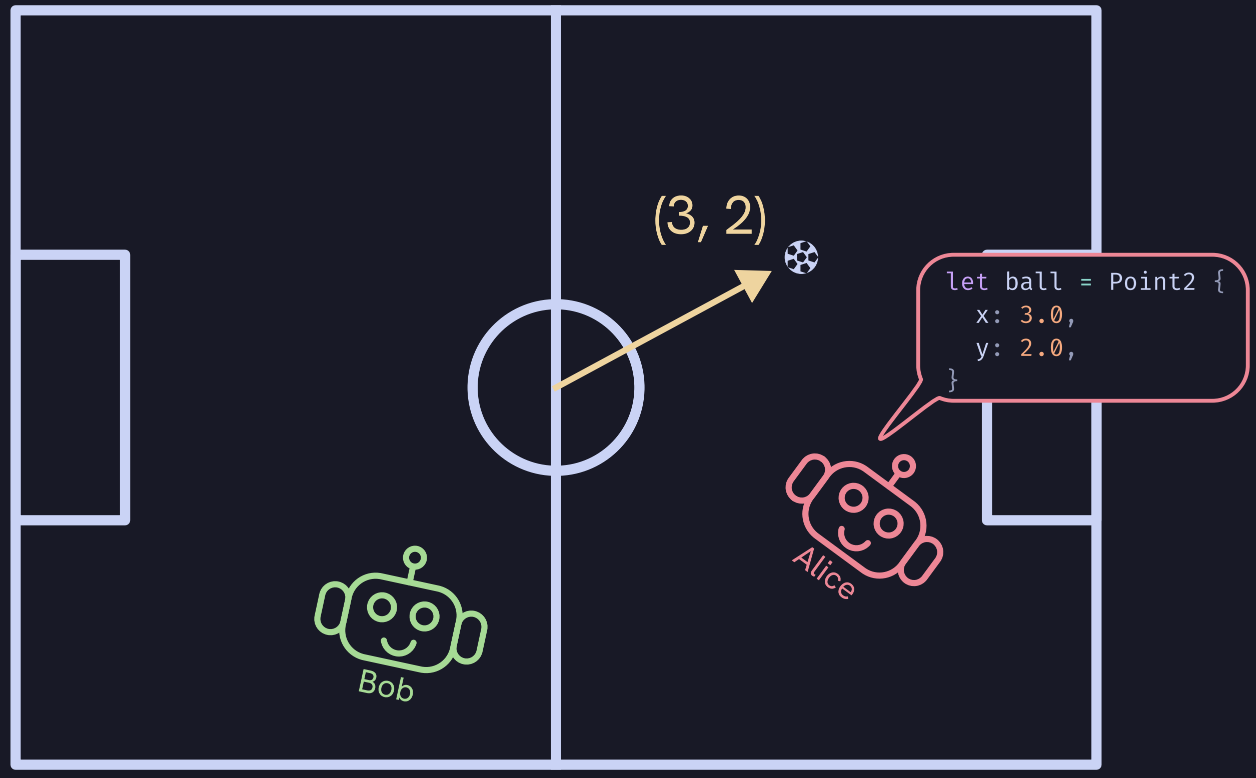
Meet Alice and Bob



Meet Alice and Bob

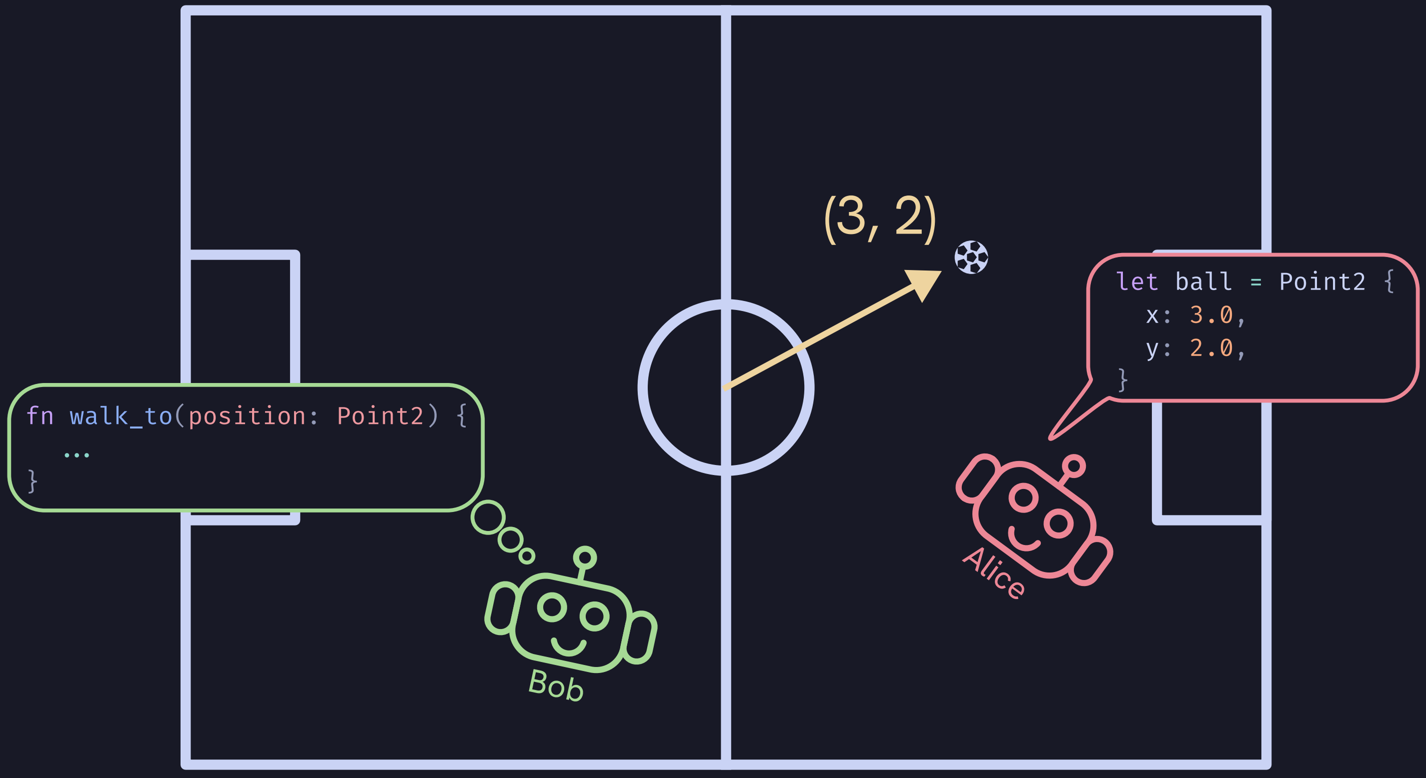


Meet Alice and Bob



```
let ball = Point2 {
  x: 3.0,
  y: 2.0,
}
```

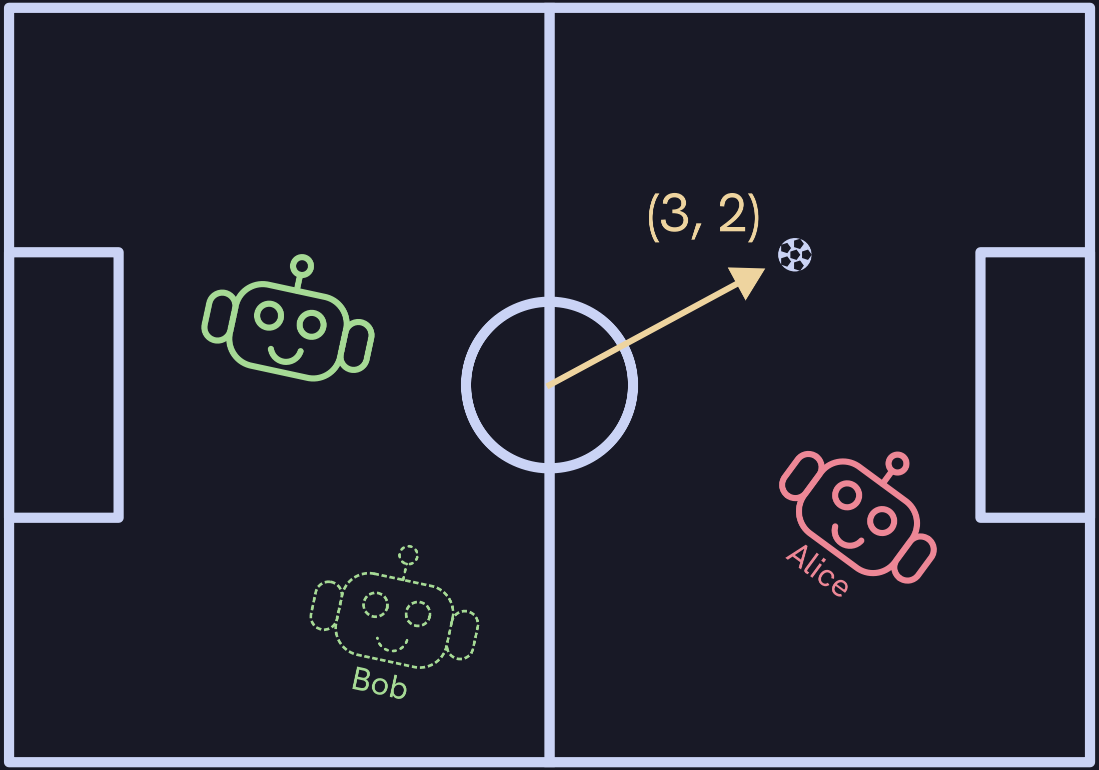
Meet Alice and Bob



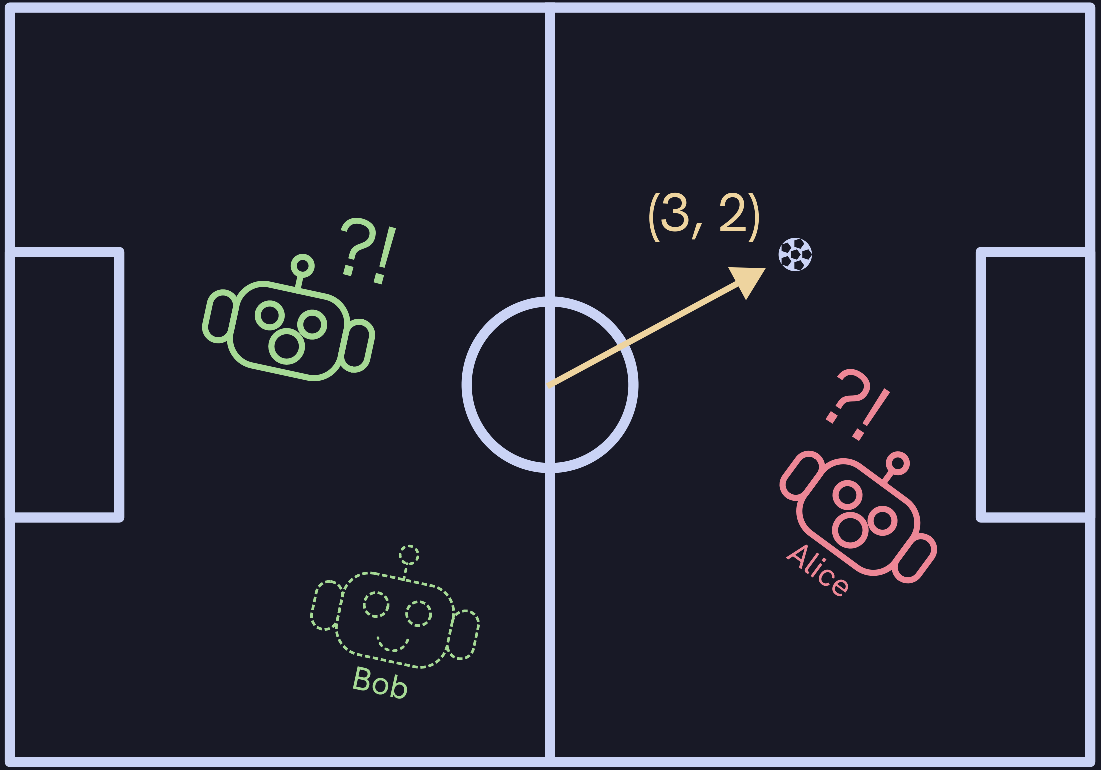
```
fn walk_to(position: Point2) {
   …
}
```

```
let ball = Point2 {
   x: 3.0,
   y: 2.0,
}
```

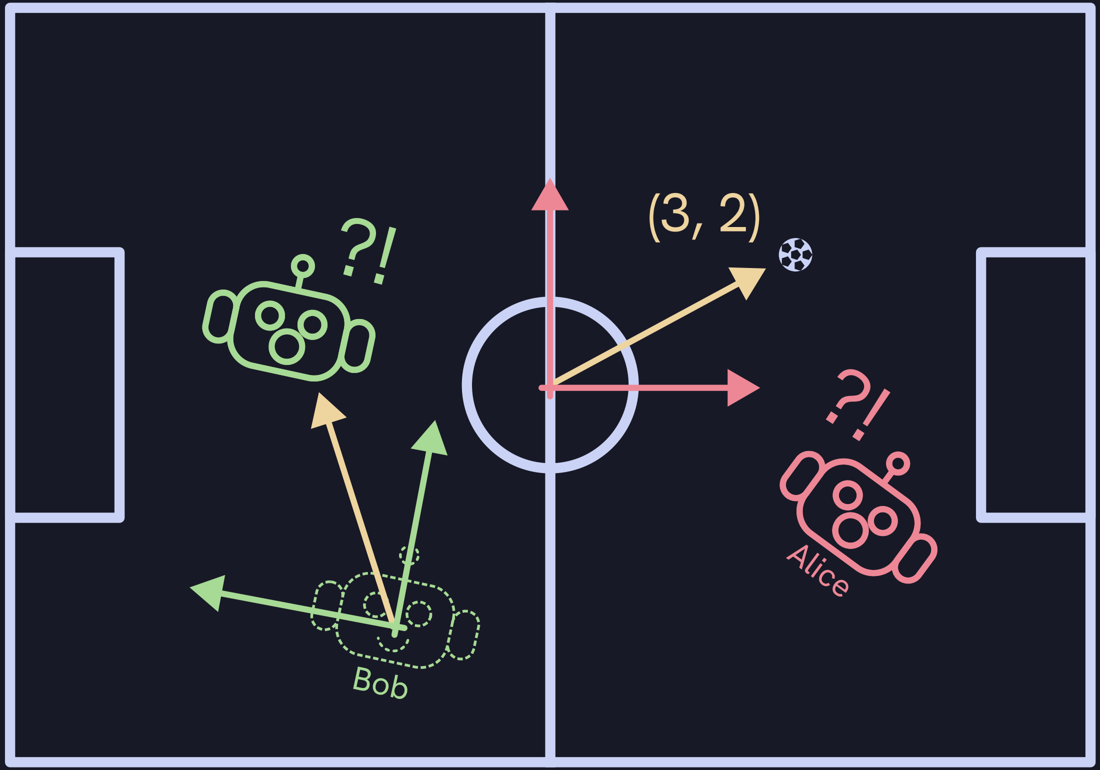
Meet Alice and Bob



Meet Alice and Bob



Meet Alice and Bob



Coordinates alone are not enough!

We’re missing the frame of reference

~~Requirements~~ Expectations

We need something that:

- □ prevents Alice & Bob bugs
- □ allows safe, easy frame conversion
- □ checked at compile time
- □ is zero cost
- □ self-documenting frames

How do we achieve all this?

How do we achieve all this?

```
struct Point2 {
  x: f32,
  y: f32,
}
```

How do we achieve all this?

```
struct Point2 {
  x: f32,
  y: f32,
  frame: Frame,
}

enum Frame {
  World,
  Robot,
  // and many more
}
```

How do we achieve all this?

```
struct Point2 {
  x: f32,
  y: f32,
  frame: Frame,
}

enum Frame {
  World,
  Robot,
  // and many more
}
```

This is not zero-cost...

How do we achieve all this?

```
struct WorldPoint2 {
  x: f32,
  y: f32,
}

struct RobotPoint2 {
  x: f32,
  y: f32,
}
```

How do we achieve all this?

```
struct WorldPoint2 {
  x: f32,
  y: f32,
}

struct RobotPoint2 {
  x: f32,
  y: f32,
}
```

This does not scale well...

How do we achieve all this?

```rust
struct Point2<Frame> {
  x: f32,
  y: f32,
}
```

How do we achieve all this?

```
struct Point2<Frame> {
  x: f32,
  y: f32,
}
```

And this does not (yet) compile

Enter Phantom Data

- A zero-sized marker type
- Carries compile-time information only
- Adds no runtime cost
- Used to “phantomly” associate a type parameter with data

The Solution: Make types generic over the frame

```
struct Point2 {
  x: f32,
  y: f32,
}
```

The Solution: Make types generic over the frame

⚠️ *ambiguous*

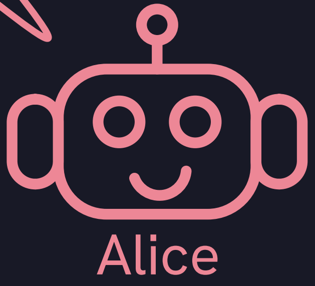
```
struct Point2 {
  x: f32,
  y: f32,
}
```

✅ *frame-safe*

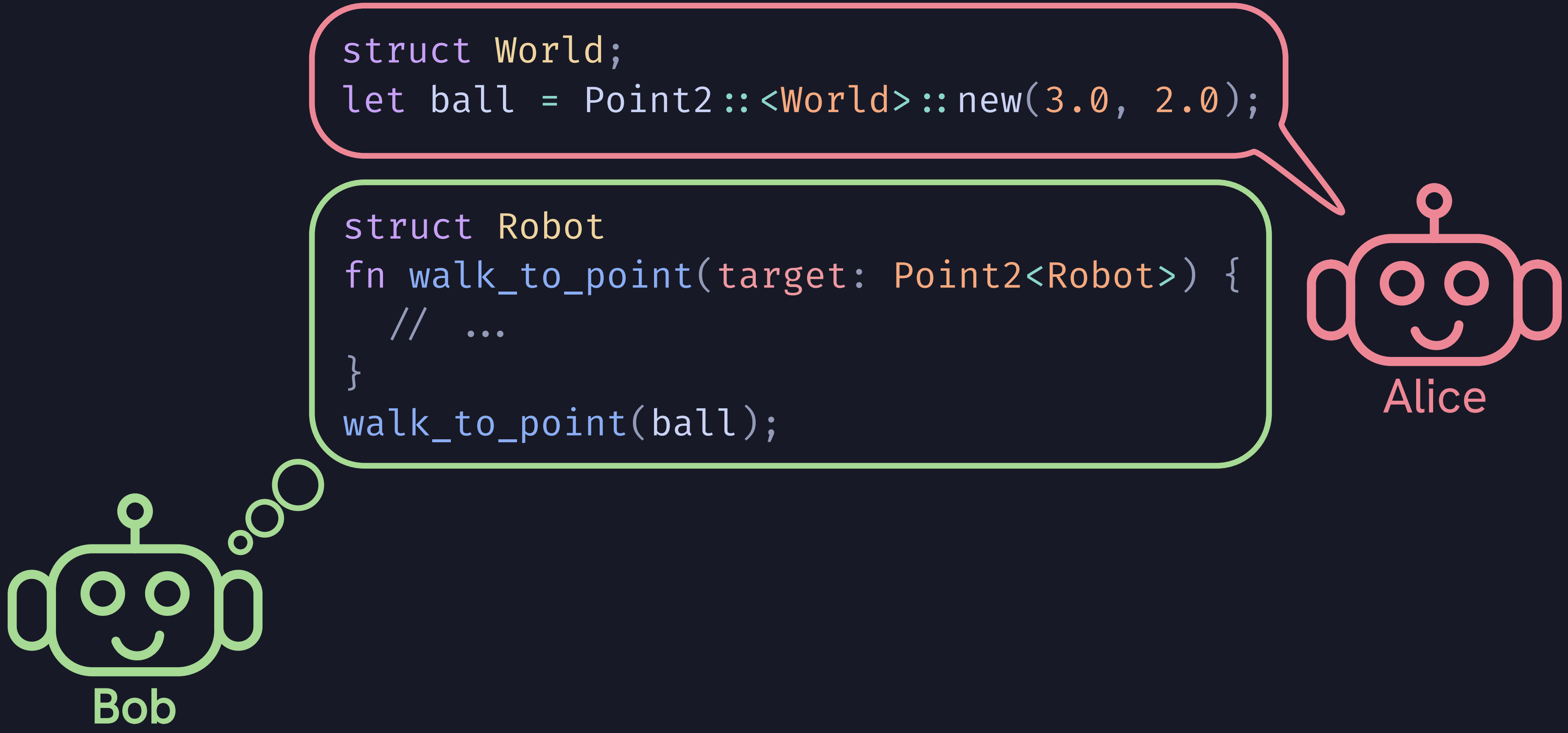
```
struct Point2<Frame> {
  x: f32,
  y: f32,
  frame: PhantomData<Frame>,
}
```

Now we can tag points with their reference frame!

```rust
struct World;
let ball = Point2::<World>::new(3.0, 2.0);
```



… piecing it together



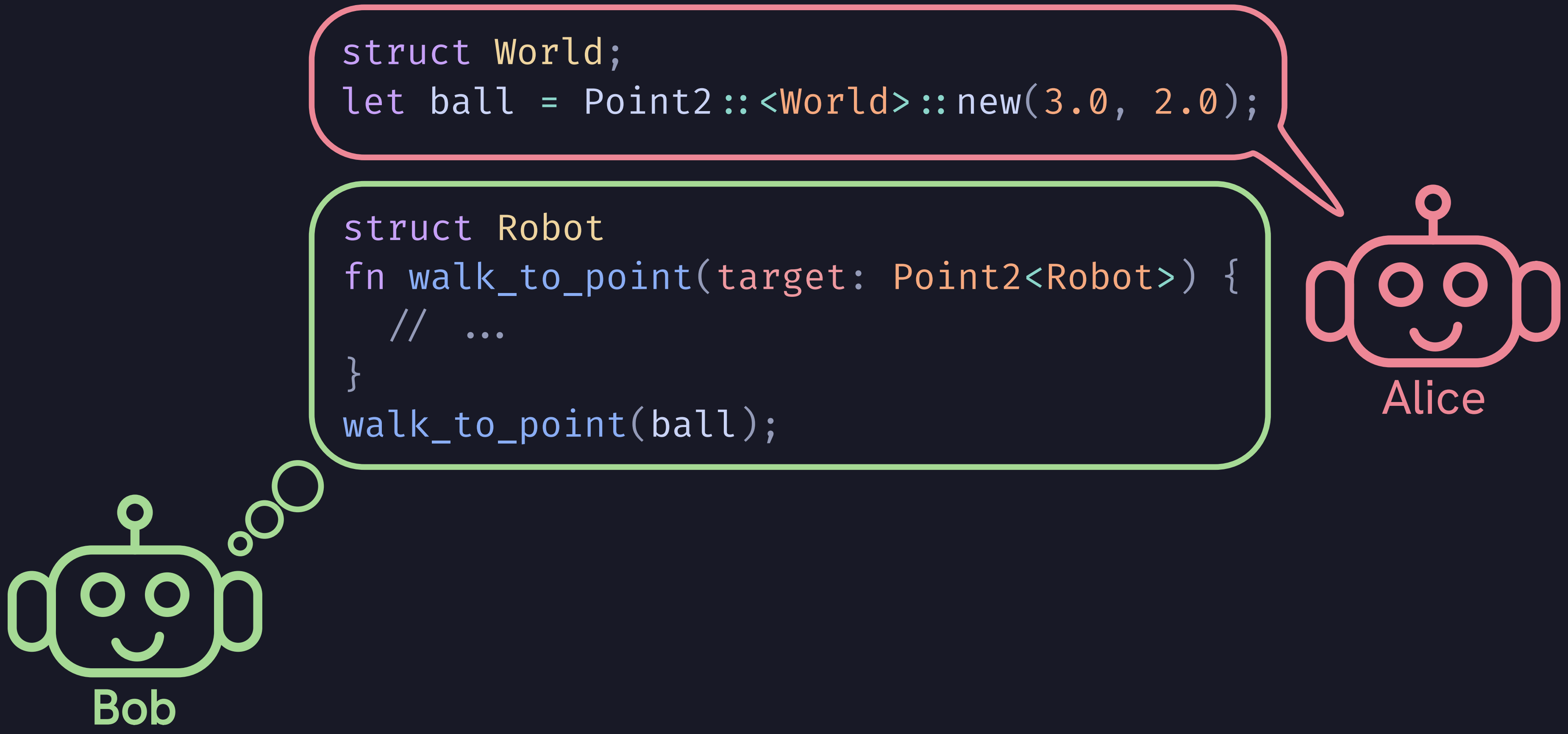
```
struct World;
let ball = Point2::<World>::new(3.0, 2.0);
```

```
struct Robot
fn walk_to_point(target: Point2<Robot>) {
  // ...
}
walk_to_point(ball);
```

And this does not compile!

```
error: expected `where`, `{`, `(`, or `;` after struct name, found
keyword `fn`
  --> examples/point_frame_mismatch.rs:22:1
   |
21 | struct Robot
22 | fn walk_to_point(target: Point2<Robot>) {
   | ^^ expected `where`, `{`, `(`, or `;` after struct name
```

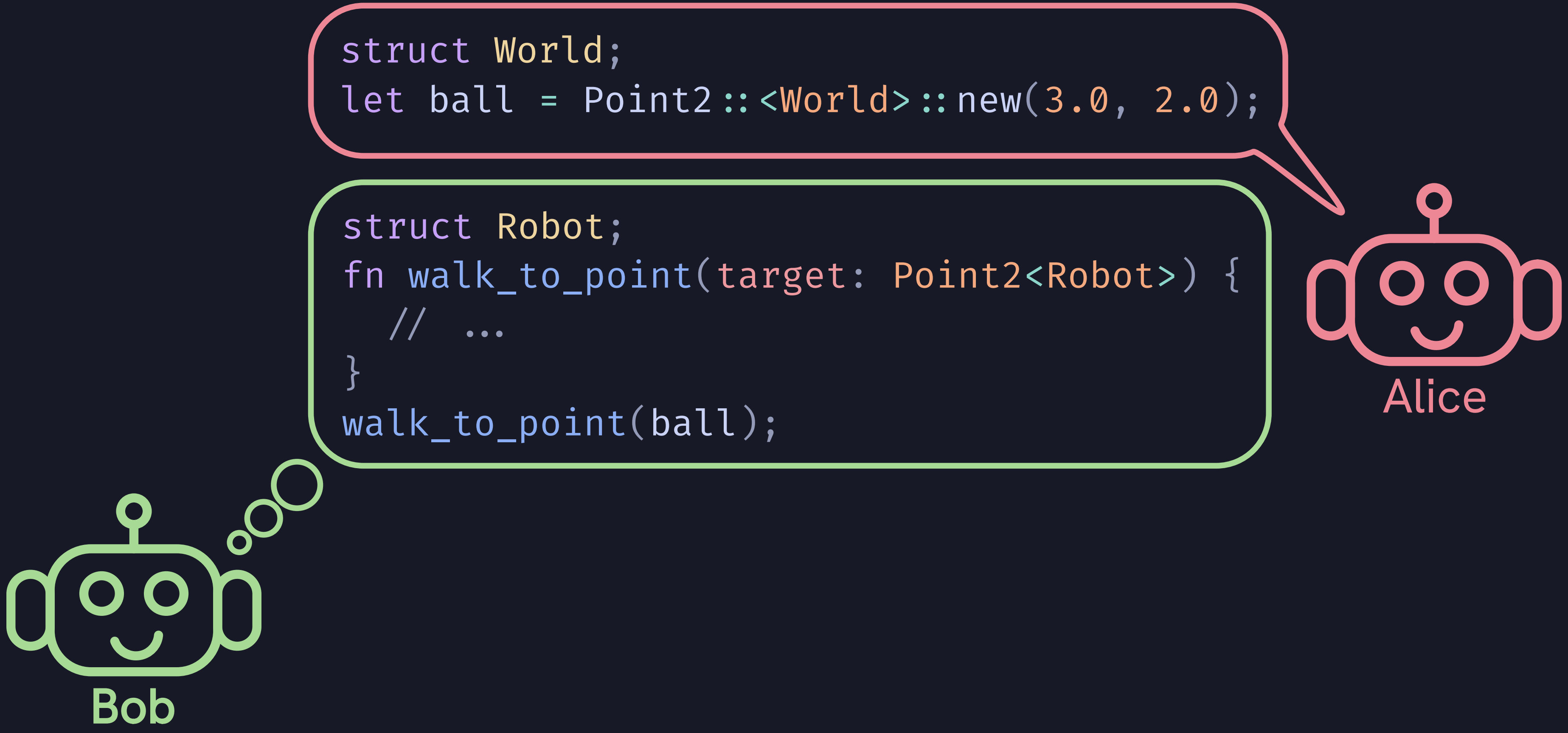
… piecing it together, take two



```rust
struct World;
let ball = Point2::<World>::new(3.0, 2.0);
```

```rust
struct Robot
fn walk_to_point(target: Point2<Robot>) {
  // …
}
walk_to_point(ball);
```

… piecing it together, take two

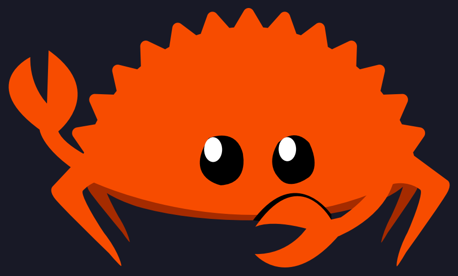


```
struct World;
let ball = Point2::<World>::new(3.0, 2.0);
```

```
struct Robot;
fn walk_to_point(target: Point2<Robot>) {
  // ...
}
walk_to_point(ball);
```

And this does not compile!

```
error[E0308]: mismatched types
  --> src/main.rs:28:19
   |
28 |     walk_to_point(ball);
   |     ------------- ^^^^
expected `Point2<Robot>`, found `Point2<World>`
   |     |
   |     arguments to this function are incorrect
   |
   = note: expected struct `Point2<Robot>`
              found struct `Point2<World>`
```



How do we transform a point?

How do we transform a point?

```
struct Isometry2 {
  x: f32,
  y: f32,
  angle: f32,
}
```

How do we transform a point?

⚠️ *ambiguous*

```
struct Isometry2 {
  x: f32,
  y: f32,
  angle: f32,
}
```

✅ *frame-safe*

```
struct Isometry2<From, To> {
  x: f32,
  y: f32,
  angle: f32,
  from: PhantomData<From>,
  to: PhantomData<To>,
}
```

How do we transform a point?

The magic happens in the `impl`:

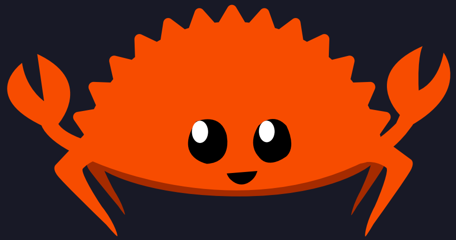
```rust
impl<From, To> Mul<Point2<From>> for Isometry2<From, To> {
  type Output = Point2<To>;

  fn mul(self, rhs: Point2<From>) -> Self::Output {
    // apply transformation
  }
}
```

Transform Alice’s ball

```rust
let ball: Point2<World> = ...;
let world_to_robot: Isometry2<World, Robot> = ...;

let ball_robot = world_to_robot * ball;
```

Transform Alice’s ball

```rust
let ball: Point2<World> = ...;
let world_to_robot: Isometry2<World, Robot> = ...;

let ball_robot = world_to_robot * ball;

fn walk_to_point(target: Point2<Robot>) {
  // ...
}
walk_to_point(ball_robot);
```



Wrong frames → compiler error

```
let ball: Point2<World> = ...;
let robot_to_world: Isometry2<Robot, World> = ...;

let ball_in_world = robot_to_world * ball;
```

Wrong frames → compiler error

```
error[E0308]: mismatched types
  --> examples/cannot_multiply.rs:28:42
   |
28 |     let ball_in_world = robot_to_world * ball;
   |                                          ^^^^
expected `Point2<Robot>`, found `Point2<World>`
   |
   = note: expected struct `Point2<Robot>`
              found struct `Point2<World>`
```

Linear Algebra is not only Points

Point

Linear Algebra is not only Points

Point

Vector

Linear Algebra is not only Points

Point

Vector

Plane

Linear Algebra is not only Points

Point

Vector

Plane

Hyperplane

Linear Algebra is not only Points

Point

Vector

Plane

Hyperplane

Translation

Linear Algebra is not only Points

Point

Vector

Plane

Hyperplane

Rotation

Translation

Linear Algebra is not only Points

Point

Vector

Plane

Hyperplane

Rotation

Translation

Reflection

Linear Algebra is not only Points

- Point
- Vector
- Plane
- Hyperplane

Rotation

Translation

Scaling

Reflection

Linear Algebra is not only Points

Point

Vector

Plane

Hyperplane

Rotation

Translation

Scaling

Shear

Reflection

Linear Algebra is not only Points

Point

Vector

Plane

Hyperplane

Rotation

Translation

Scaling

Projection

Shear

Reflection

Linear Algebra is not only Points

Point

Vector

Plane

Hyperplane

Rotation

Perspective

Translation

Projection

Scaling

Shear

Reflection

Linear Algebra is not only Points

Point

Vector

Plane

Hyperplane

Rotation

Perspective

Translation

Projection

Scaling

Shear

Reflection

Affine

Linear Algebra is not only Points

Point

Vector

Plane

Hyperplane

Rotation

Kinematic Chain

Perspective

Translation

Projection

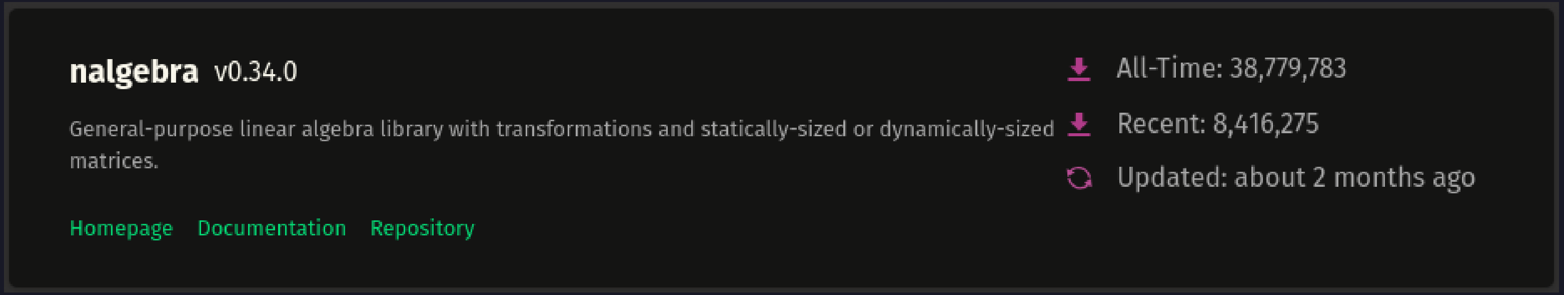
Scaling

Shear

Reflection

Affine

Let’s search on `crates.io`



nalgebra v0.34.0

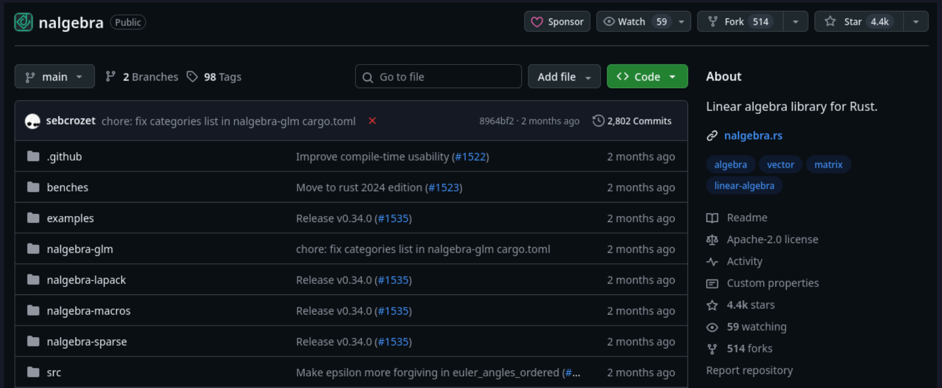
General-purpose linear algebra library with transformations and statically-sized or dynamically-sized matrices.

All-Time: 38,779,783

Recent: 8,416,275

Updated: about 2 months ago

`nalgebra` already has all these basic types



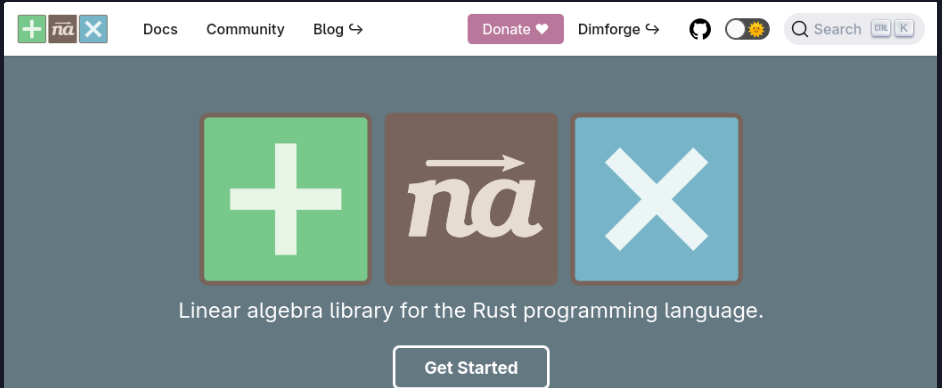
sebcrozet chore: fix categories list in nalgebra-glm cargo.toml 8964bf2 · 2 months ago 2,802 Commits

.github	Improve compile-time usability (#1522)	2 months ago
benches	Move to rust 2024 edition (#1523)	2 months ago
examples	Release v0.34.0 (#1535)	2 months ago
nalgebra-glm	chore: fix categories list in nalgebra-glm cargo.toml	2 months ago
nalgebra-lapack	Release v0.34.0 (#1535)	2 months ago
nalgebra-macros	Release v0.34.0 (#1535)	2 months ago
nalgebra-sparse	Release v0.34.0 (#1535)	2 months ago
src	Make epsilon more forgiving in euler_angles_ordered (#...	2 months ago

About

Linear algebra library for Rust.

nalgebra.rs



Linear algebra library for the Rust programming language.

Wrapping `nalgebra`

```rust
struct Framed<Frame, Inner> {
    frame: PhantomData<Frame>,
    pub inner: Inner,
}
```

Wrapping `nalgebra`

```rust
#[repr(transparent)]
struct Framed<Frame, Inner> {
    frame: PhantomData<Frame>,
    pub inner: Inner,
}
```

Wrapping `nalgebra`

```
#[repr(transparent)]
struct Framed<Frame, Inner> {
    frame: PhantomData<Frame>,
    pub inner: Inner,
}

type Point2<Frame, T> = Framed<Frame, nalgebra::Point2<T>>;
type Vector2<Frame, T> = Framed<Frame, nalgebra::Vector2<T>>;
// and more...
```

Using wrapped types is no different

```rust
use nalgebra::Point2;




fn walk_to_point(
  target: Point2<f32>,
) { /**/ }

fn robot_to_world(
  point: Point2<f32>,
) -> Point2<f32> { /**/ }
```

Using wrapped types is no different

⚠️ *ambiguous*

```rust
use nalgebra::Point2;




fn walk_to_point(
  target: Point2<f32>,
) { /**/ }

fn robot_to_world(
  point: Point2<f32>,
) -> Point2<f32> { /**/ }
```

✅ *frame-safe*

```rust
use linear_algebra::Point2;

struct World;
struct Robot;

fn walk_to_point(
  target: Point2<World, f32>
) { /**/ }

fn robot_to_world(
  point: Point2<Robot, f32>,
) -> Point2<World, f32> { /**/ }
```

… the same for transforms

```rust
#[repr(transparent)]
struct Transform<From, To, Inner> {
    from: PhantomData<From>,
    to: PhantomData<To>,
    pub inner: Inner,
}
```

… the same for transforms

```
#[repr(transparent)]
struct Transform<From, To, Inner> {
    from: PhantomData<From>,
    to: PhantomData<To>,
    pub inner: Inner,
}

type Isometry3<From, To, T> =
  Transform<From, To, nalgebra::Isometry3<T>>;
// and more ...
```

Defining Frames

```
$ head crates/coordinate_systems/src/lib.rs
```

```rust
/// 3D coordinate system centered on the robot.
///
/// Origin: hip of the robot
/// X axis pointing forward
/// Y axis pointing left
struct Robot;
/// 2D coordinate system centered on the robot.
///
/// Origin: center between feet, projected onto the ground.
/// X axis pointing forward
struct Ground;
```

Real-World Example

```
fn paint_target_feet(






) {
}
```

Real-World Example

```
fn paint_target_feet(
    painter: &TwixPainter<Ground>,






) {
}
```

Real-World Example

```
fn paint_target_feet(
    painter: &TwixPainter<Ground>,
    robot_to_walk: Isometry3<Robot, Walk>,
    robot_to_ground: Isometry3<Robot, Ground>,



) {
}
```

Real-World Example

```
fn paint_target_feet(
    painter: &TwixPainter<Ground>,
    robot_to_walk: Isometry3<Robot, Walk>,
    robot_to_ground: Isometry3<Robot, Ground>,
    support_sole: Pose3<Walk>,
    end_support_sole: Pose3<Walk>,
    end_swing_sole: Pose3<Walk>,
) {
}
```

Real-World Example

```
fn paint_target_feet(
    painter: &TwixPainter<Ground>,
    robot_to_walk: Isometry3<Robot, Walk>,
    robot_to_ground: Isometry3<Robot, Ground>,
    support_sole: Pose3<Walk>,
    end_support_sole: Pose3<Walk>,
    end_swing_sole: Pose3<Walk>,
) {
    let walk_to_robot = robot_to_walk.inverse();









}
```

Real-World Example

```
fn paint_target_feet(
    painter: &TwixPainter<Ground>,
    robot_to_walk: Isometry3<Robot, Walk>,
    robot_to_ground: Isometry3<Robot, Ground>,
    support_sole: Pose3<Walk>,
    end_support_sole: Pose3<Walk>,
    end_swing_sole: Pose3<Walk>,
) {
    let walk_to_robot = robot_to_walk.inverse();

    struct SupportSole;
    let upcoming_walk_to_support_sole =
        end_support_sole.as_transform::<SupportSole>().inverse();






}
```

Real-World Example

```rust
fn paint_target_feet(
    painter: &TwixPainter<Ground>,
    robot_to_walk: Isometry3<Robot, Walk>,
    robot_to_ground: Isometry3<Robot, Ground>,
    support_sole: Pose3<Walk>,
    end_support_sole: Pose3<Walk>,
    end_swing_sole: Pose3<Walk>,
) {
    let walk_to_robot = robot_to_walk.inverse();

    struct SupportSole;
    let upcoming_walk_to_support_sole =
        end_support_sole.as_transform::<SupportSole>().inverse();
    let target_swing_sole/*: Pose3<Robot> */ =
        support_sole.as_transform() * upcoming_walk_to_support_sole * end_swing_sole;




}
```

Real-World Example

```rust
fn paint_target_feet(
    painter: &TwixPainter<Ground>,
    robot_to_walk: Isometry3<Robot, Walk>,
    robot_to_ground: Isometry3<Robot, Ground>,
    support_sole: Pose3<Walk>,
    end_support_sole: Pose3<Walk>,
    end_swing_sole: Pose3<Walk>,
) {
    let walk_to_robot = robot_to_walk.inverse();

    struct SupportSole;
    let upcoming_walk_to_support_sole =
        end_support_sole.as_transform::<SupportSole>().inverse();
    let target_swing_sole/*: Pose3<Robot> */ =
        support_sole.as_transform() * upcoming_walk_to_support_sole * end_swing_sole;

    painter.paint_sole_polygon(
        robot_to_ground * walk_to_robot * target_swing_sole,
    );
}
```

Benefits and Conclusion

- □ prevents Alice & Bob bugs
- □ allows safe, easy frame conversion
- □ checked at compile time
- □ is zero cost
- □ self-documenting frames

Benefits and Conclusion

- ☑ prevents Alice & Bob bugs
- □ allows safe, easy frame conversion
- □ checked at compile time
- □ is zero cost
- □ self-documenting frames

Benefits and Conclusion

- ☑ prevents Alice & Bob bugs
- ☑ allows safe, easy frame conversion
- □ checked at compile time
- □ is zero cost
- □ self-documenting frames

Benefits and Conclusion

- ☑ prevents Alice & Bob bugs
- ☑ allows safe, easy frame conversion
- ☑ checked at compile time
- □ is zero cost
- □ self-documenting frames

Benefits and Conclusion

- ☑ prevents Alice & Bob bugs
- ☑ allows safe, easy frame conversion
- ☑ checked at compile time
- ☑ is zero cost
- □ self-documenting frames

Benefits and Conclusion

☑ prevents Alice & Bob bugs

☑ allows safe, easy frame conversion

☑ checked at compile time

☑ is zero cost

☑ self-documenting frames

hulks.de

Slides

@hulks_tuhh

hulks_tuhh

Made with typst and ❤