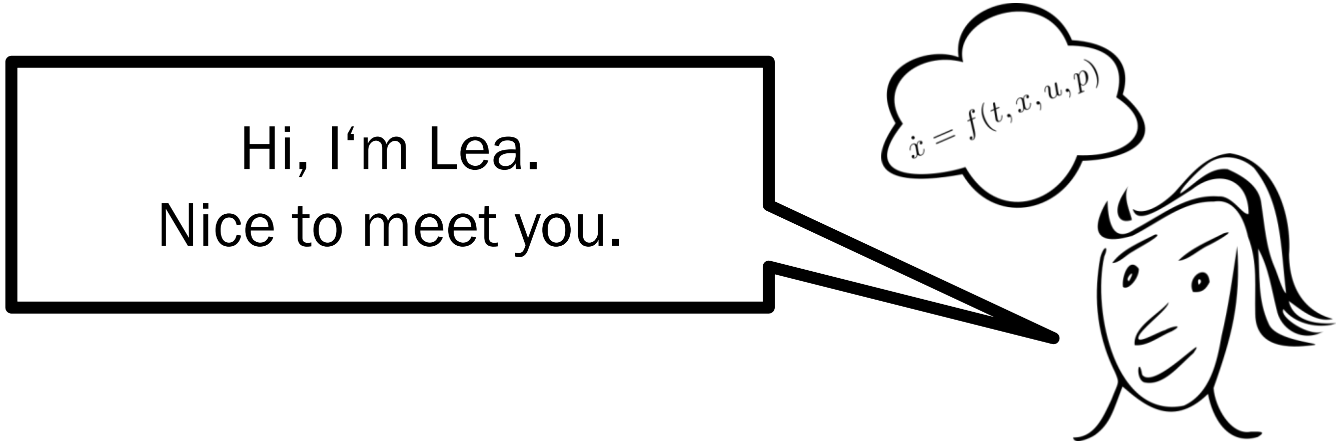
SIMPLIFY AND SECURE EQUATION SYSTEMS WITH TYPE-DRIVEN DEVELOPMENT

Arne Berger


Hi, I'm Lea.
Nice to meet you.
$\dot{x} = f(t, x, u, p)$

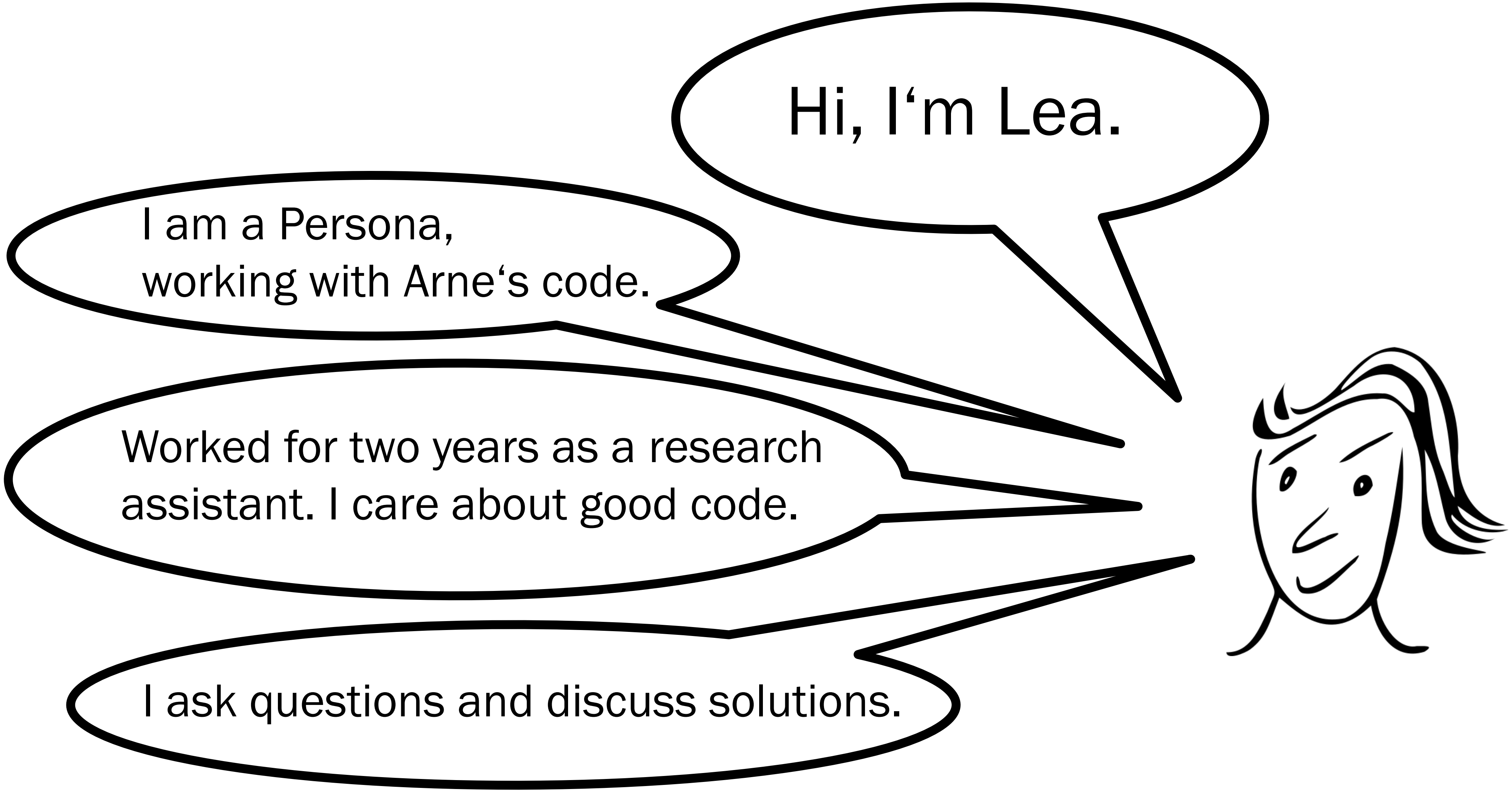

Hohe Düne
30 min
K 43
© openstreetmap.org contributers

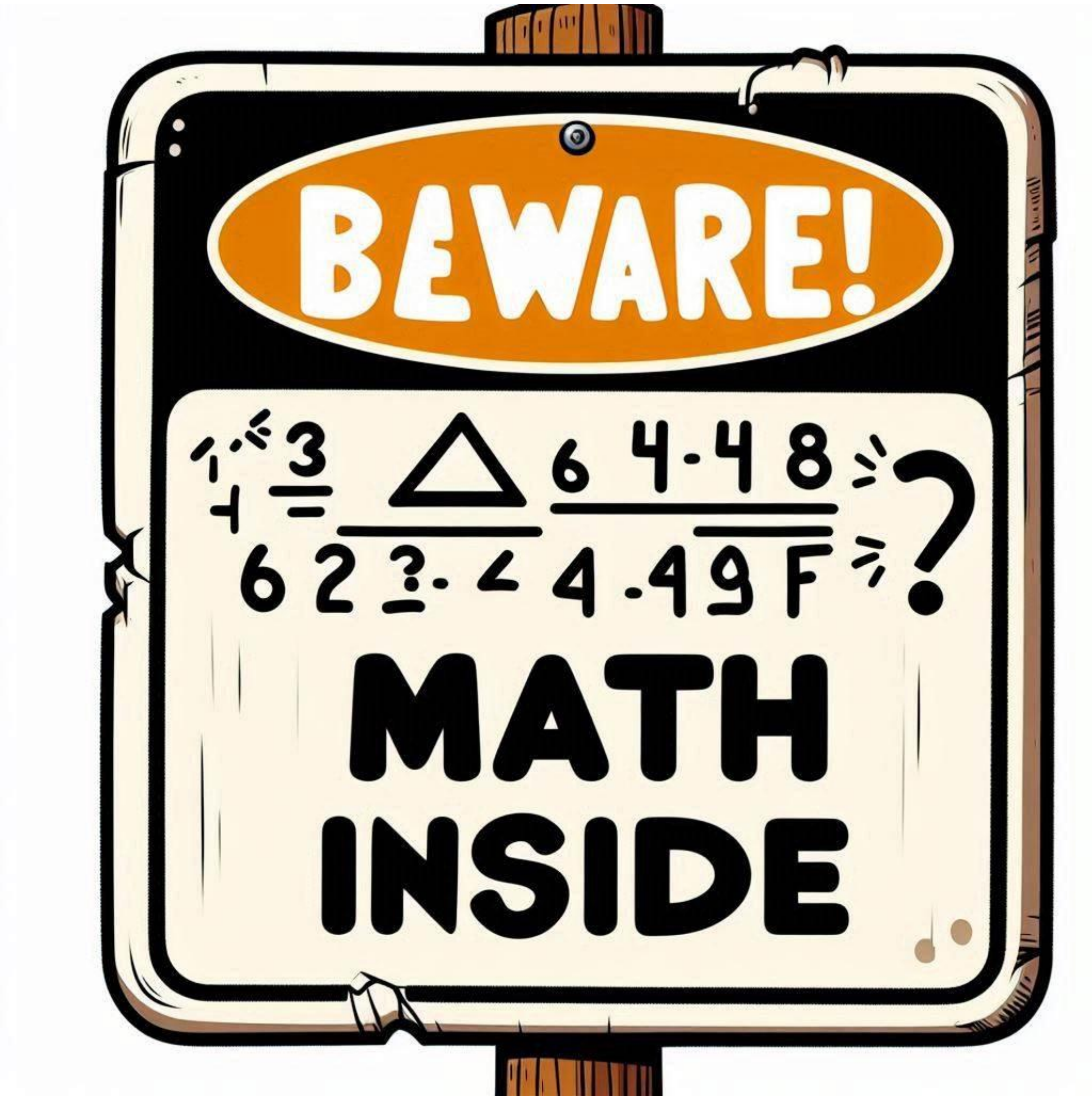
ABOUT THE AUTHOR

Hi, I‘m Arne.

Worked six years as a research assistant in optimal control of ships, cars and electrical grids.

WHO IS LEA?


Hi, I‘m Lea.
I am a Persona, working with Arne‘s code.
Worked for two years as a research assistant. I care about good code.
I ask questions and discuss solutions.

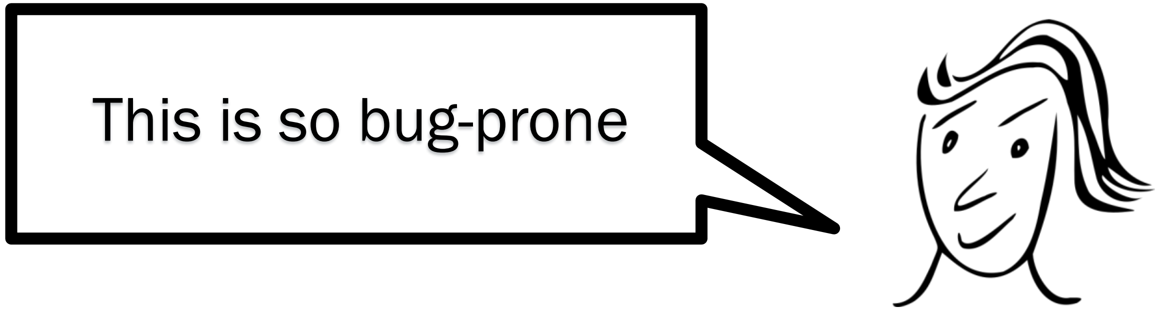

BEWARE!
MATH
INSIDE

The Problem

General equation:
$\dot{x}(t) = f(x(t), u(t))$

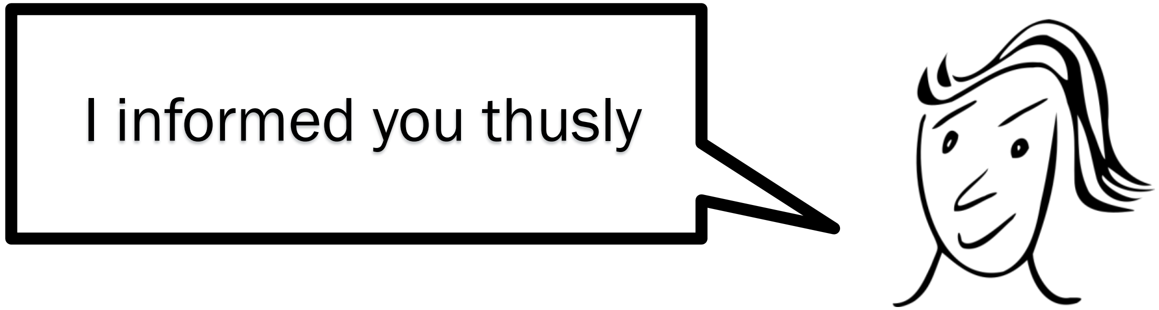
$$\begin{pmatrix} \dot{r_x}(t) \\ \dot{r_y}(t) \\ \dot{\psi}(t) \\ \dot{v}(t) \\ \dot{\omega}(t) \end{pmatrix} = \begin{pmatrix} v(t) \cdot \cos\big(\psi(t)\big) \\ v(t) \cdot \sin\big(\psi(t)\big) \\ \omega(t) \\ p_1 cos\big(\phi(t)\big) \cdot \frac{E(t)}{100} \\ p_2\, sin\big(\phi(t)\big) \cdot \frac{E(t)}{100} \end{pmatrix} \begin{array}{l} Position\,X \\ Position\,Y \\ Yaw \\ Velocity \\ Rate\ of\ Turn \end{array}$$

The Problem

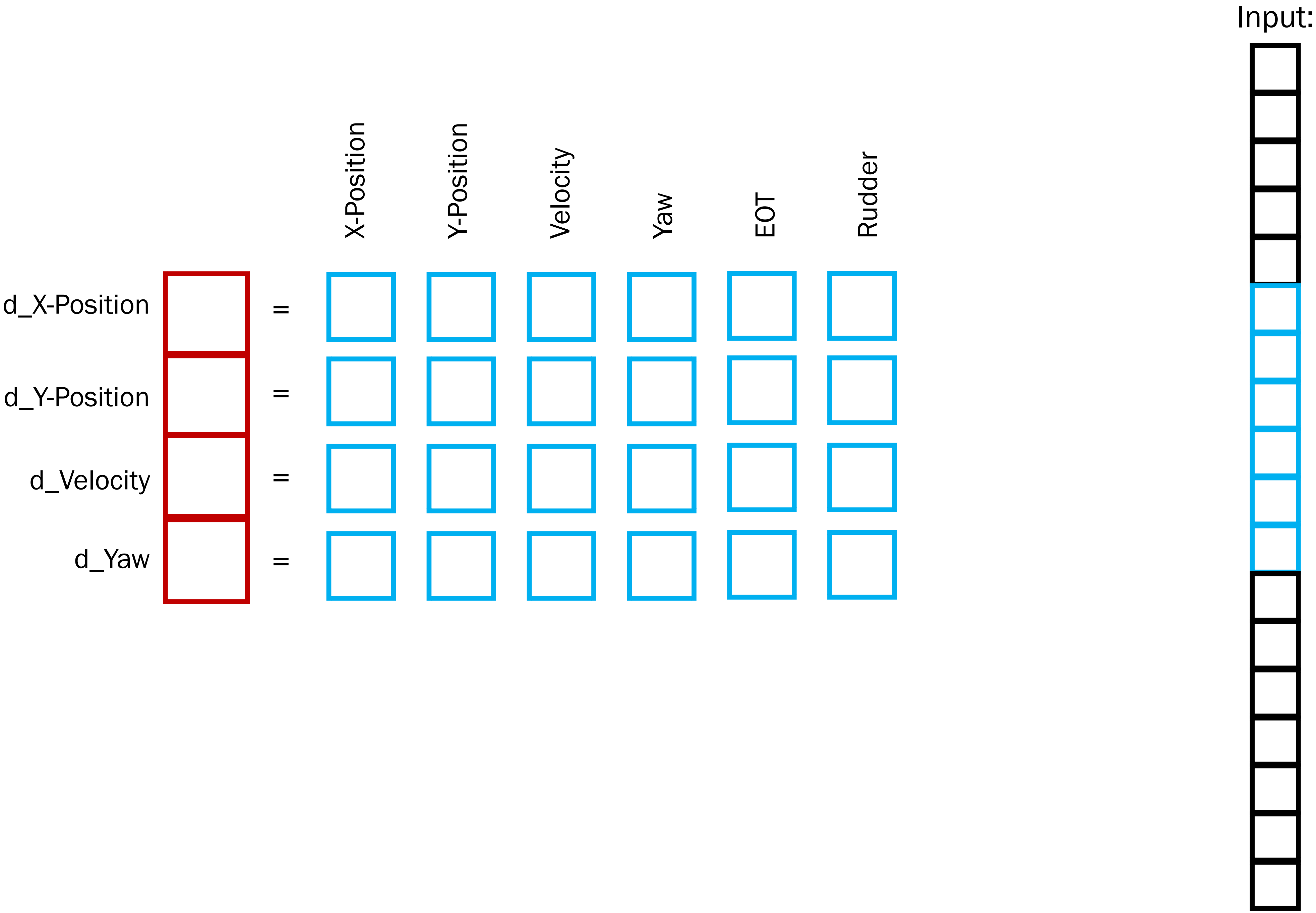


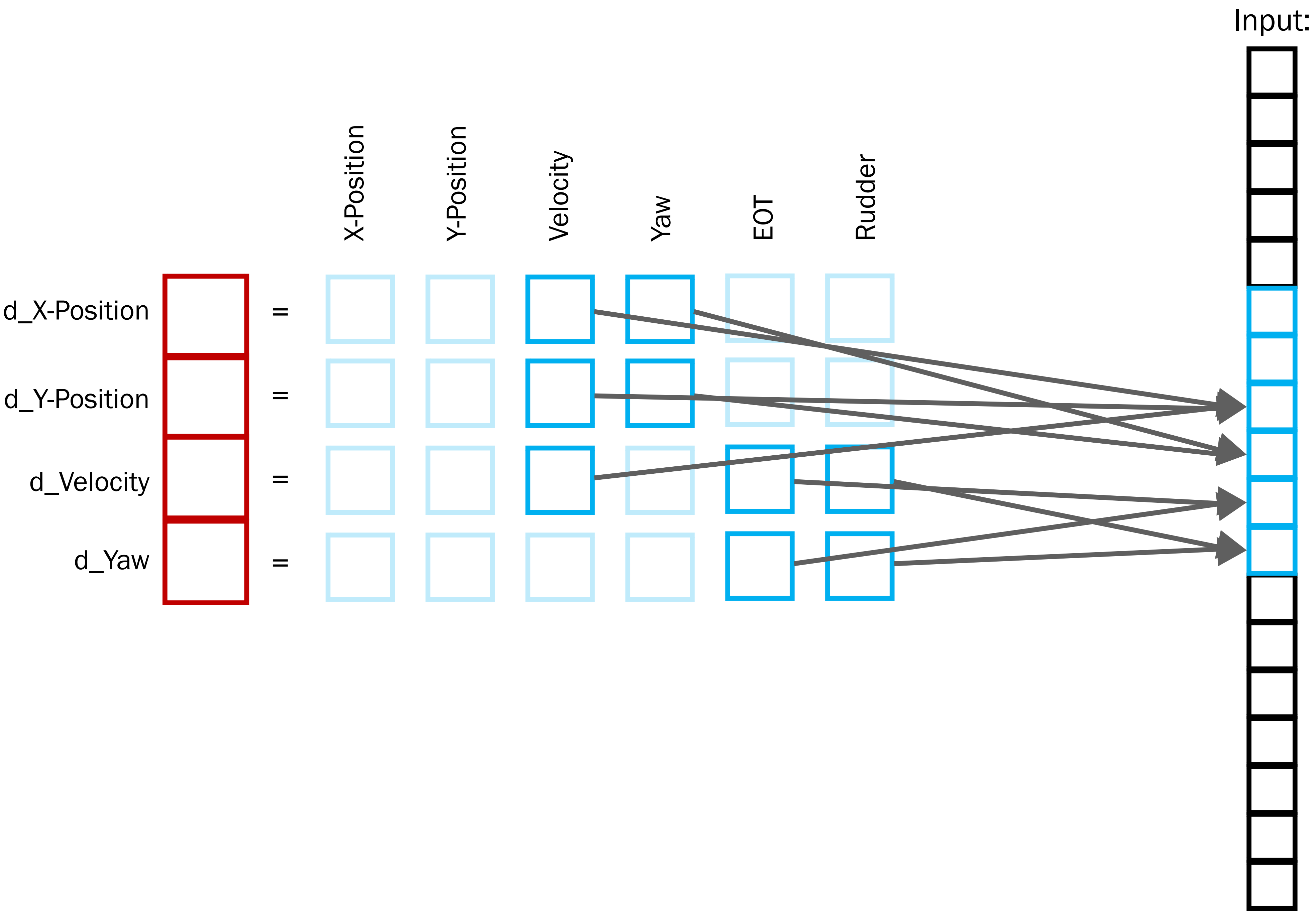

```
void Deneb::ode(double* dotX, const double* xAtT, const double* uAtT) const
{
    const double velocity = xAtT[2];
    const double yaw = xAtT[3];
    const double rot = xAtT[4];
    const double eot = uAtT[0];
    const double rudder_angle = uAtT[1];
    dotX[0] = velocity * cos(yaw);
    dotX[1] = velocity * sin(yaw);
    dotX[2] = rot;
    dotX[3] = p1 * cos(rudder_angle) * (eot / 100.0);
    dotX[4] = p2 * sin(rudder_angle) * (eot / 100.0);
}
```

The Problem

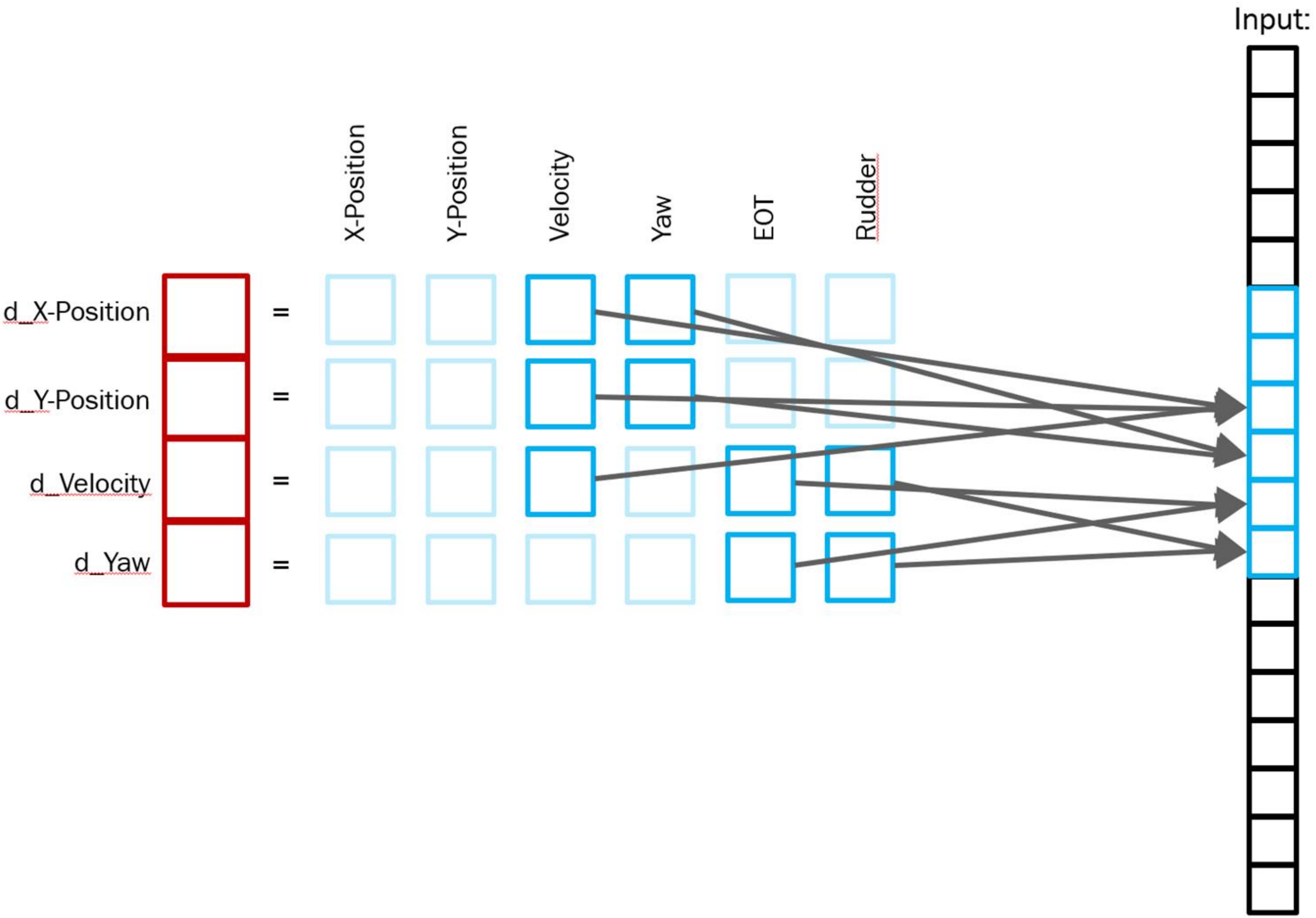


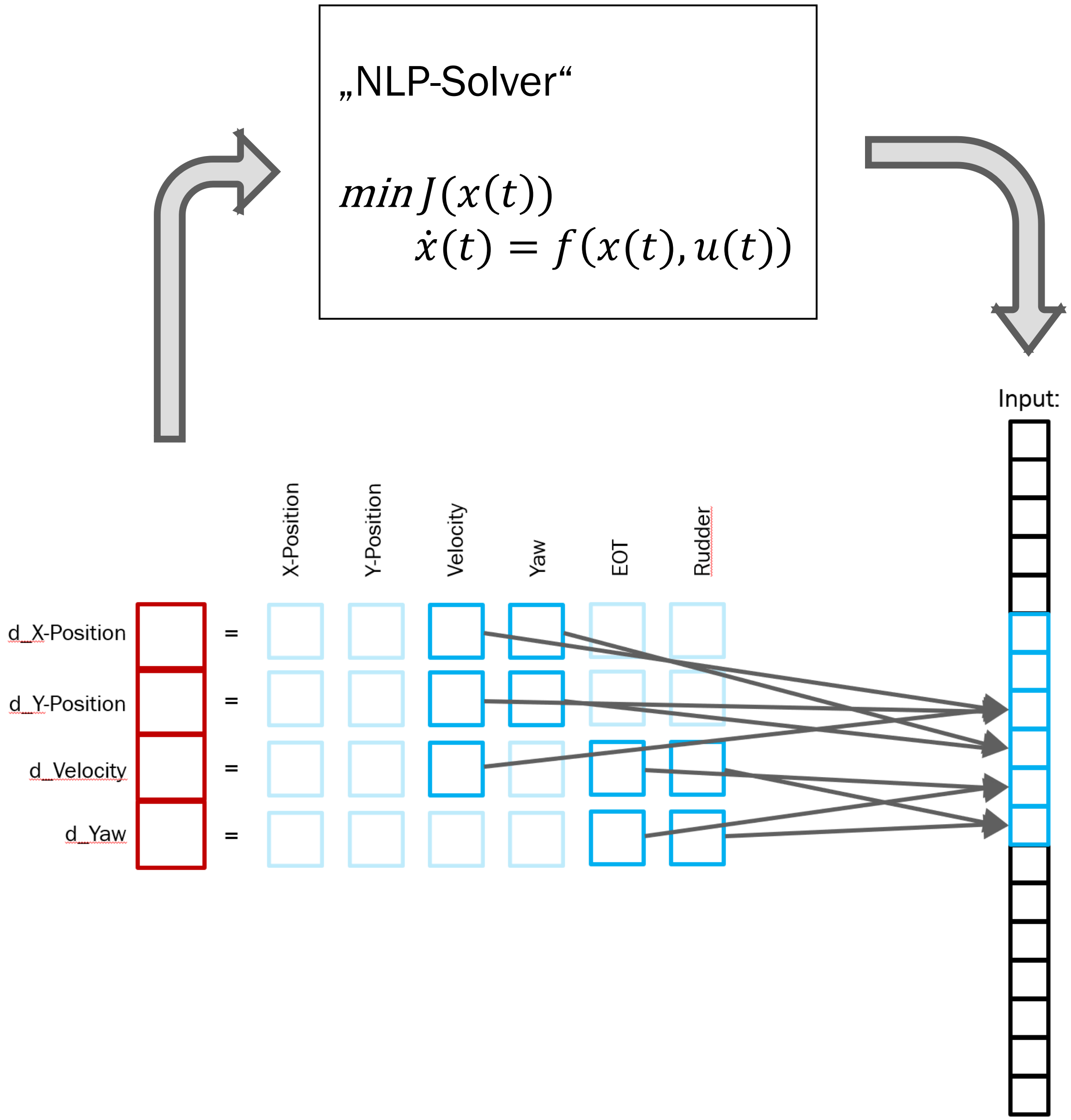

```
void Deneb::ode(double* dotX, const double* xAtT, const double* uAtT) const
{
    const double velocity = xAtT[2];
    const double yaw = xAtT[3];
    const double rot = xAtT[4];
    const double eot = uAtT[0];
    const double rudder_angle = uAtT[1];
    dotX[0] = velocity * cos(yaw);
    dotX[1] = velocity * sin(yaw);
    dotX[2] = rot;
    dotX[3] = p1 * cos(rudder_angle) * (eot / 100.0);
    dotX[4] = p2 * sin(rudder_angle) * (eot / 100.0);
}
```


Input:
X-Position
Y-Position
Velocity
Yaw
EOT
Rudder
d_X-Position
d_Y-Position
d_Velocity
d_Yaw
=
=
=
=



Input:
X-Position
Y-Position
Velocity
Yaw
EOT
Rudder
d_X-Position
=
d_Y-Position
=
d_Velocity
=
d_Yaw
=


Input:
X-Position
Y-Position
Velocity
Yaw
EOT
Rudder
d_X-Position
d_Y-Position
d_Velocity
d_Yaw
=
=
=
=


„NLP-Solver“
$min\,J(x(t))$
$\dot{x}(t) = f(x(t), u(t))$
Input:
X-Position
Y-Position
Velocity
Yaw
EOT
Rudder
d_X-Position
d_Y-Position
d_Velocity
d_Yaw
=
=
=
=

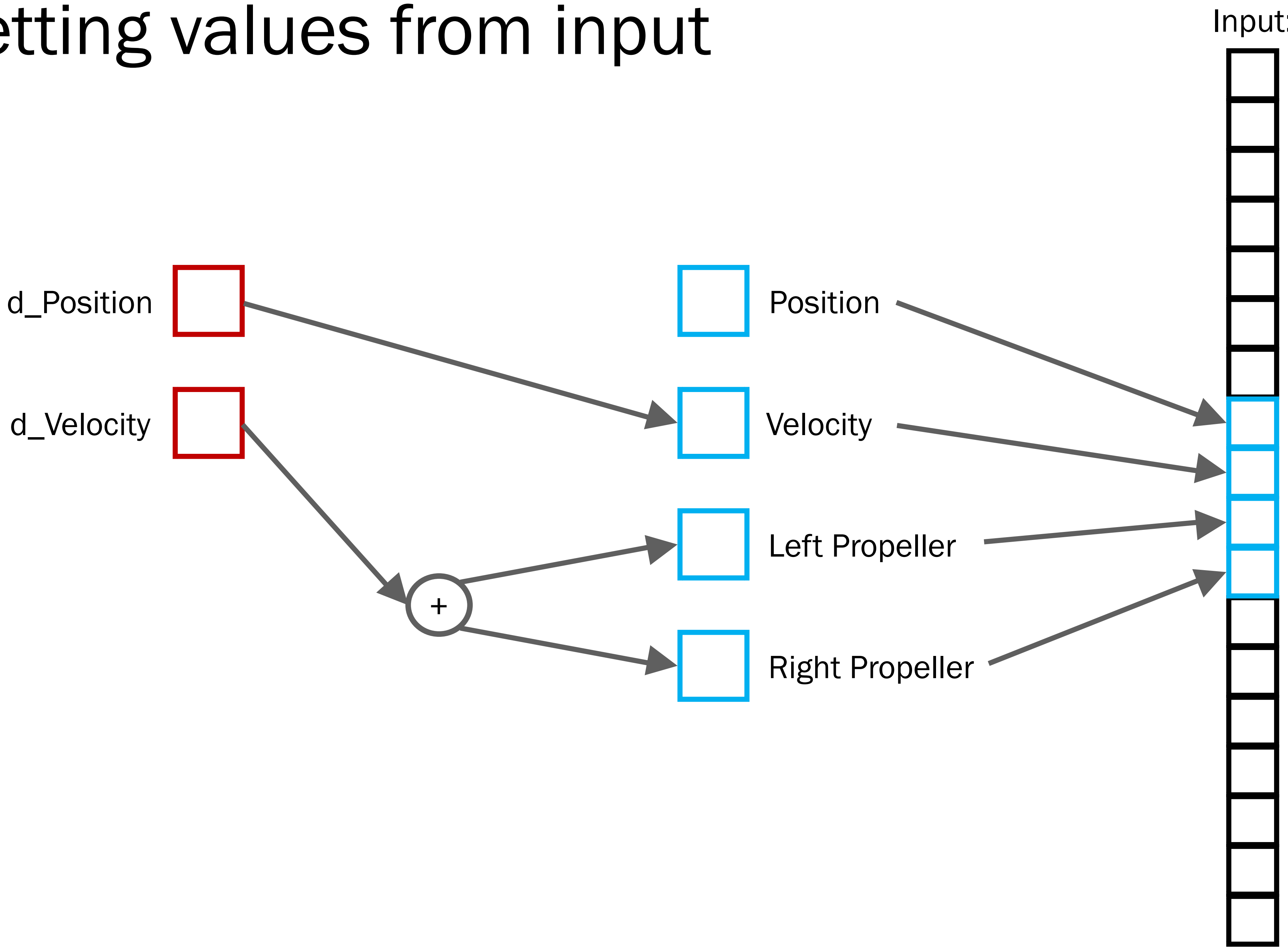
Goals

- Prevent bugs through compiler checks
- Intuitively usable by mathematicians
- Maintainable

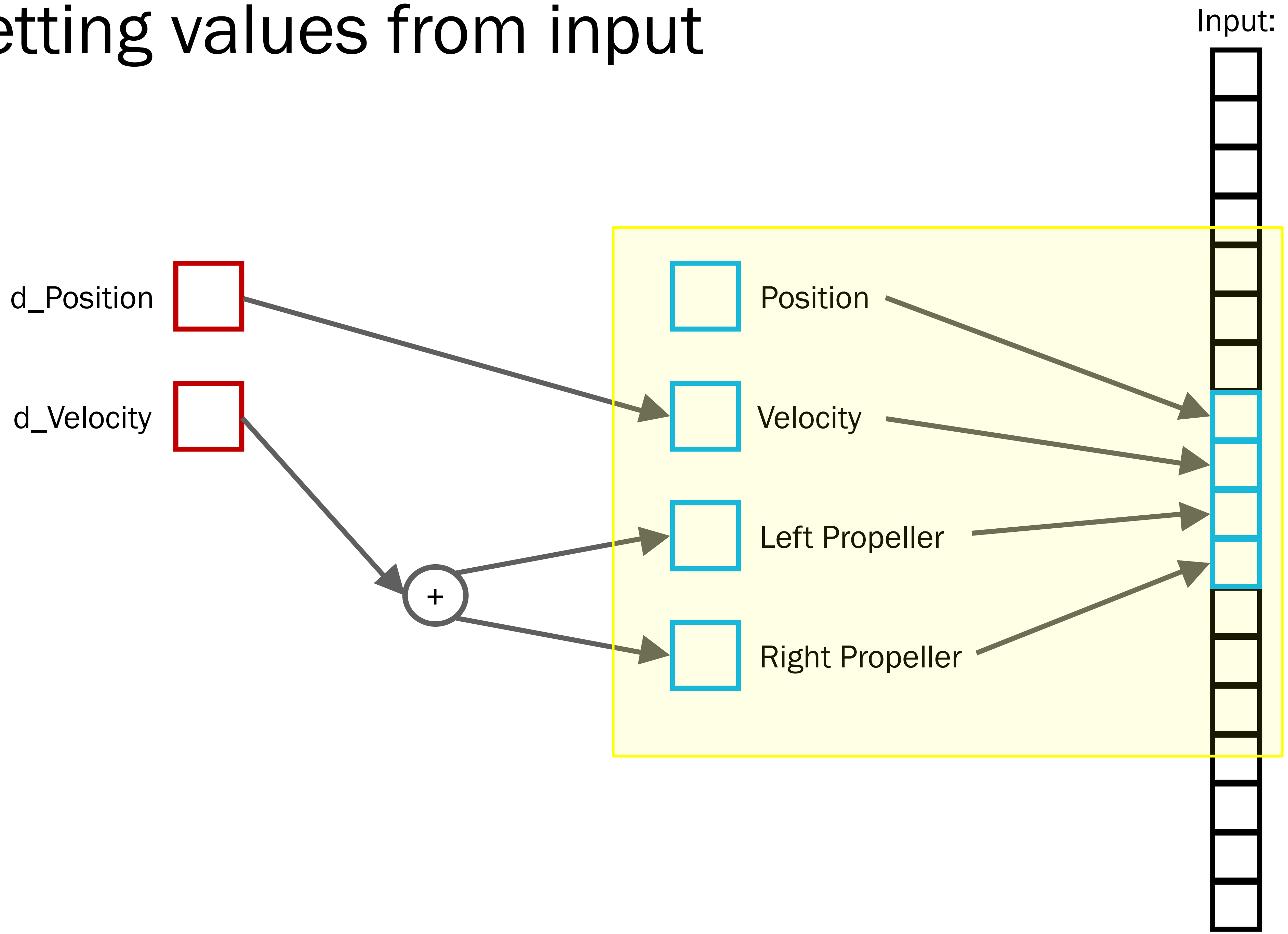
Let‘s start simple

$$\begin{pmatrix} \dot{r_x}(t) \\ \dot{v}(t) \end{pmatrix} = \begin{pmatrix} v(t) \\ a_{left}(t) + a_{right}(t) \end{pmatrix}$$

Getting values from input



Getting values from input



Indices for Types

```cpp
template <typename Tag>
struct Quantity{};

using Position = Quantity<struct Position_>;
using Velocity = Quantity<struct Velocity_>;
using AccPropellerLeft = Quantity<struct AccPropellerLeft_>;
using AccPropellerRight = Quantity<struct AccPropellerRight_>;
```

Indices for Types

```cpp
template <typename Tag>
struct Quantity{};

using Position = Quantity<struct Position_>;
using Velocity = Quantity<struct Velocity_>;
using AccPropellerLeft = Quantity<struct AccPropellerLeft_>;
using AccPropellerRight = Quantity<struct AccPropellerRight_>;

std::tuple<Position, Velocity> states;
std::tuple<AccPropellerLeft, AccPropellerRight> controls;
```

Indices for Types

```
template <typename Tag>
struct Quantity{};

using Position = Quantity<struct Position_>;
using Velocity = Quantity<struct Velocity_>;
using AccPropellerLeft = Quantity<struct AccPropellerLeft_>;
using AccPropellerRight = Quantity<struct AccPropellerRight_>;

std::tuple<Position, Velocity> states;
std::tuple<AccPropellerLeft, AccPropellerRight> controls;

get<0>(states); // yes
get<Position>(states); // yes
get_idx<Position>(states); // no
```

Indices for Types

```cpp
template <class T, typename... Ts>
struct Index<T, std::tuple<Ts...>> {

    static constexpr std::size_t index = []() {
        constexpr std::array<bool, sizeof...(Ts)> a{{std::is_same_v<T, Ts>...}};

        const auto it = std::find(a.begin(), a.end(), true);

        if (it == a.end()) {
            throw std::runtime_error("Not present");
        }

        return std::distance(a.begin(), it);
    }();
};
```

Indices for Types

Thank you „Jarod42“

stackoverflow.com

```
template <class T, typename... Ts>
struct Index<T, std::tuple<Ts...>> {

    static constexpr std::size_t index = []() {
        constexpr std::array<bool, sizeof...(Ts)> a{{std::is_same_v<T, Ts>...}};

        const auto it = std::find(a.begin(), a.end(), true);

        if (it == a.end()) {
            throw std::runtime_error("Not present");
        }

        return std::distance(a.begin(), it);
    }();
};
```

Indices for Types

```
template<class T, tuple_like Tuple>
static constexpr auto get_idx()
{
    return Index<T, Tuple>::index;
}
```

System of States

```
template<class T, tuple_like Tuple>
static constexpr auto get_idx()
{
    return Index<T, Tuple>::index;
}


using Test = std::tuple<int,double,float>;
constexpr auto idx_double = get_idx<double, Test>(); // 1
```

Getting values from input

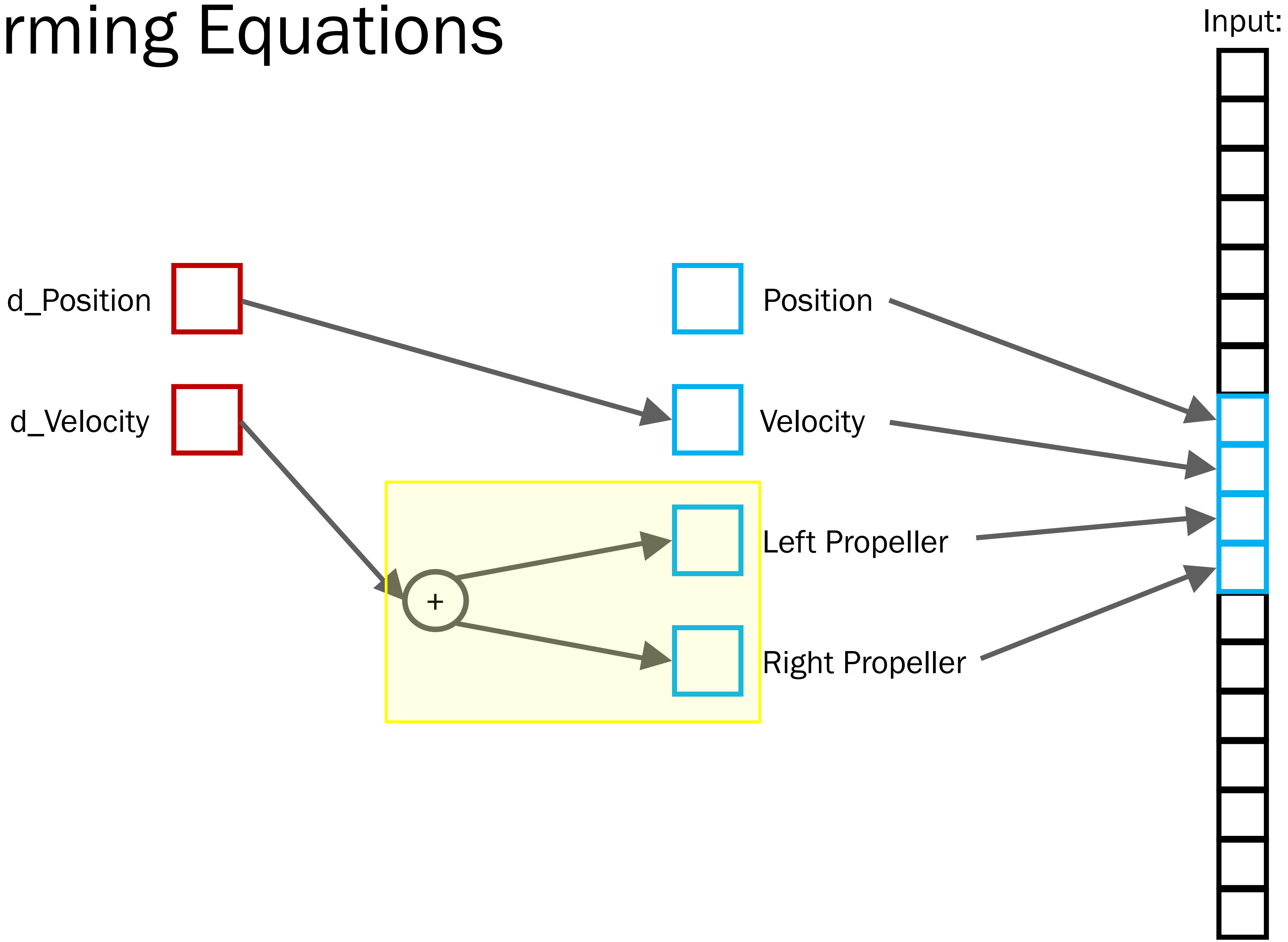
```
template <typename Tag>
struct Quantity {

    template <class Quantities, std::size_t N>
    constexpr static double evaluate(std::span<const double, N> arr) {
        return arr[get_idx<Quantity, Quantities>()];
    }

};
```

Getting values from input

Replacing `x[0]` with … that?

```
template <typename Tag>
struct Quantity {

    template <class Quantities, std::size_t N>
    constexpr static double evaluate(std::span<const double, N> arr) {
        return arr[get_idx<Quantity, Quantities>()];
    }

};

using Position = Quantity<class Position_>;
using Velocity = Quantity<class Velocity_>;
using ShipStates = std::tuple<Position, Velocity>;

constexpr std::array in{1.0, 2.0};
constexpr auto value = Velocity::evaluate<ShipStates>(std::span(in)); // 2.0
```

Forming Equations



Forming Equations

```
template <typename Lhs, typename Rhs>
struct Add
{
    template <class Quantities, std::size_t N>
    static constexpr double evaluate(std::span<const double, N> arr)
    {
        return Lhs::template evaluate<Quantities>(arr) +
               Rhs::template evaluate<Quantities>(arr);
    }
};
```

Forming Equations

```cpp
template <typename Lhs, typename Rhs>
struct Add
{
    template <class Quantities, std::size_t N>
    static constexpr double evaluate(std::span<const double, N> arr)
    {
        return Lhs::template evaluate<Quantities>(arr) +
               Rhs::template evaluate<Quantities>(arr);
    }
};

template <typename Lhs, typename Rhs>
constexpr auto operator+(Lhs /*lhs*/, Rhs /*rhs*/)
{
    return Add<Lhs, Rhs>{};
}
```

Constraining Operators


```
template <typename Lhs, typename Rhs>
struct Add
{
    template <class Quantities, std::size_t N>
    static constexpr double evaluate(std::span<const double, N> arr)
    {
        return Lhs::template evaluate<Quantities>(arr) +
               Rhs::template evaluate<Quantities>(arr);
    }
};

template <typename Lhs, typename Rhs>
constexpr auto operator+(Lhs /*lhs*/, Rhs /*rhs*/)
{
    return Add<Lhs, Rhs>{};
}
```

Constraining Operators

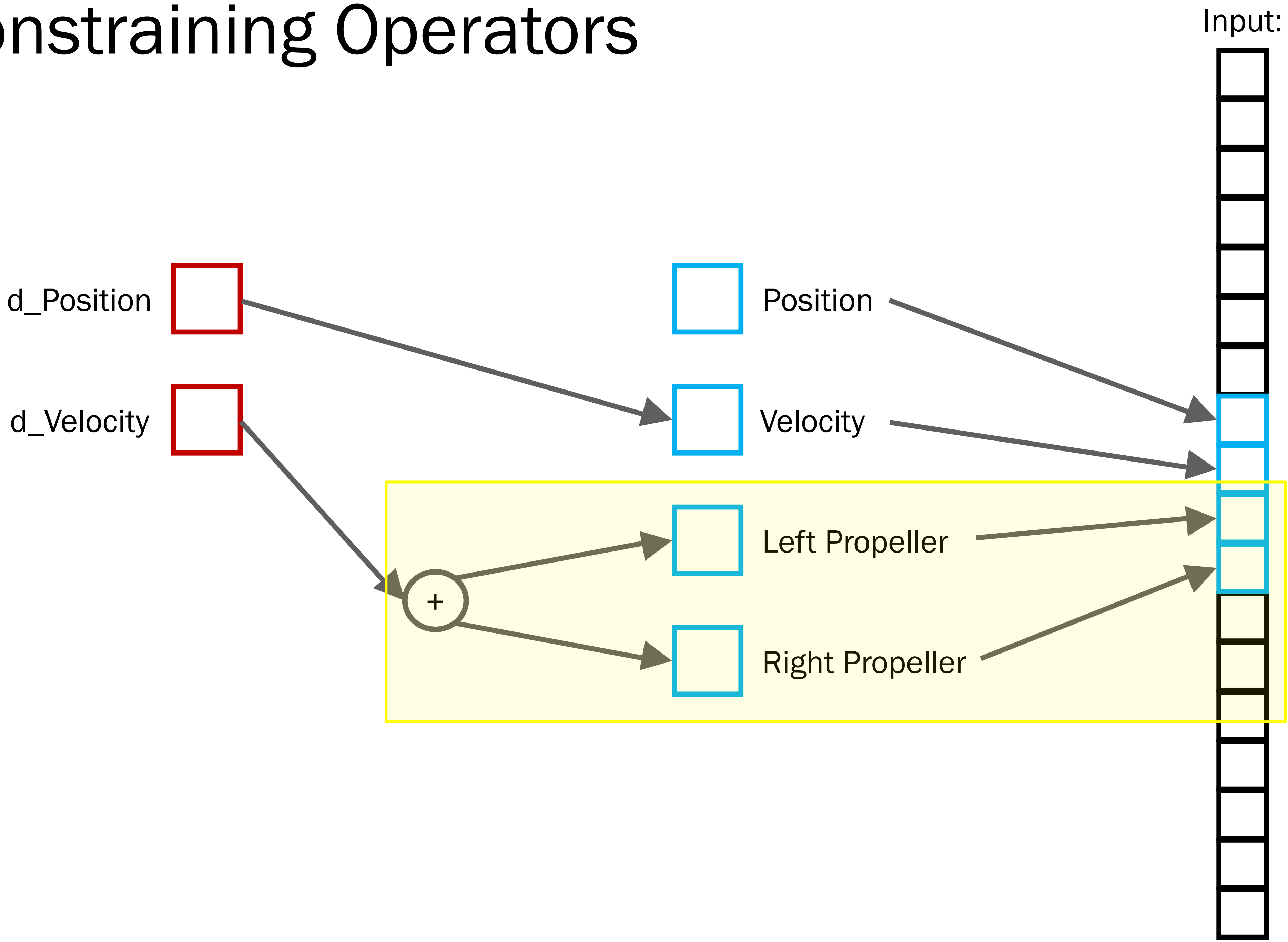
```
template <typename Expression, class Quantities, std::size_t N>
concept SystemExpression = requires(Expression expr, std::span<const double, N> in)
{
    { expr.template evaluate<Quantities, N>(in) } -> std::same_as<double>;
};
```

Constraining Operators

```
template <typename Expression, class Quantities, std::size_t N>
concept SystemExpression = requires(Expression expr, std::span<const double, N> in)
{
    { expr.template evaluate<Quantities, N>(in) } -> std::same_as<double>;
};
```

```
template <SystemExpression<???> Lhs, SystemExpression<???> Rhs>
constexpr auto operator+(Lhs /*lhs*/,
                         Rhs /*rhs*/)
{
    return Add<Lhs, Rhs>{};
}
```

Constraining Operators



Constraining Operators

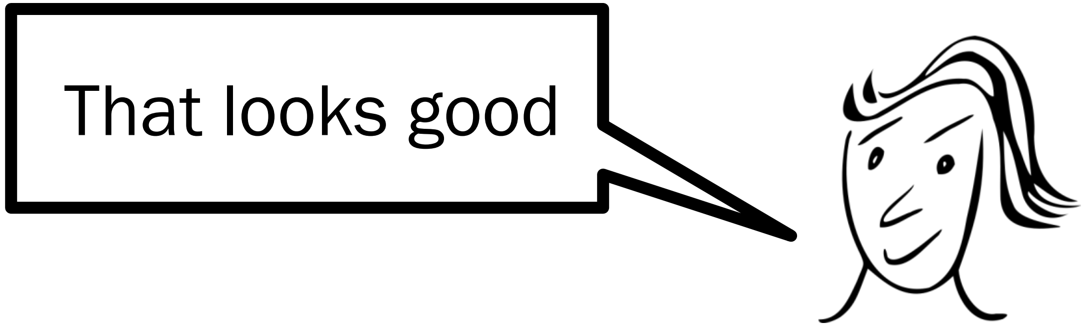
```
template <typename Tag>
struct Quantity {
    using DependsOn = std::tuple<Quantity>;

    template <class Quantities, std::size_t N>
    constexpr static double evaluate(std::span<const double, N> arr);
};

template <typename Lhs, typename Rhs>
struct Add
{
    using DependsOn = to_unique_tuple_t<
        tuple_cat_t<typename Lhs::DependsOn, typename Rhs::DependsOn>>;

    template <class Quantities, std::size_t N>
    static constexpr double evaluate(std::span<const double, N> arr);
};
```

Constraining Operators

```
template <typename Expression,
          typename Quantities = typename Expression::DependsOn,
          std::size_t N = std::tuple_size_v<typename Expression::DependsOn>>
concept SystemExpression = requires(Expression expr, std::span<const double, N> in)
{
    { expr.template evaluate<Quantities, N>(in) } -> std::same_as<double>;
};



template <SystemExpression Lhs, SystemExpression Rhs>
constexpr auto operator+(Lhs /*lhs*/,
                         Rhs /*rhs*/)
{
    return Add<Lhs, Rhs>{};
}
```

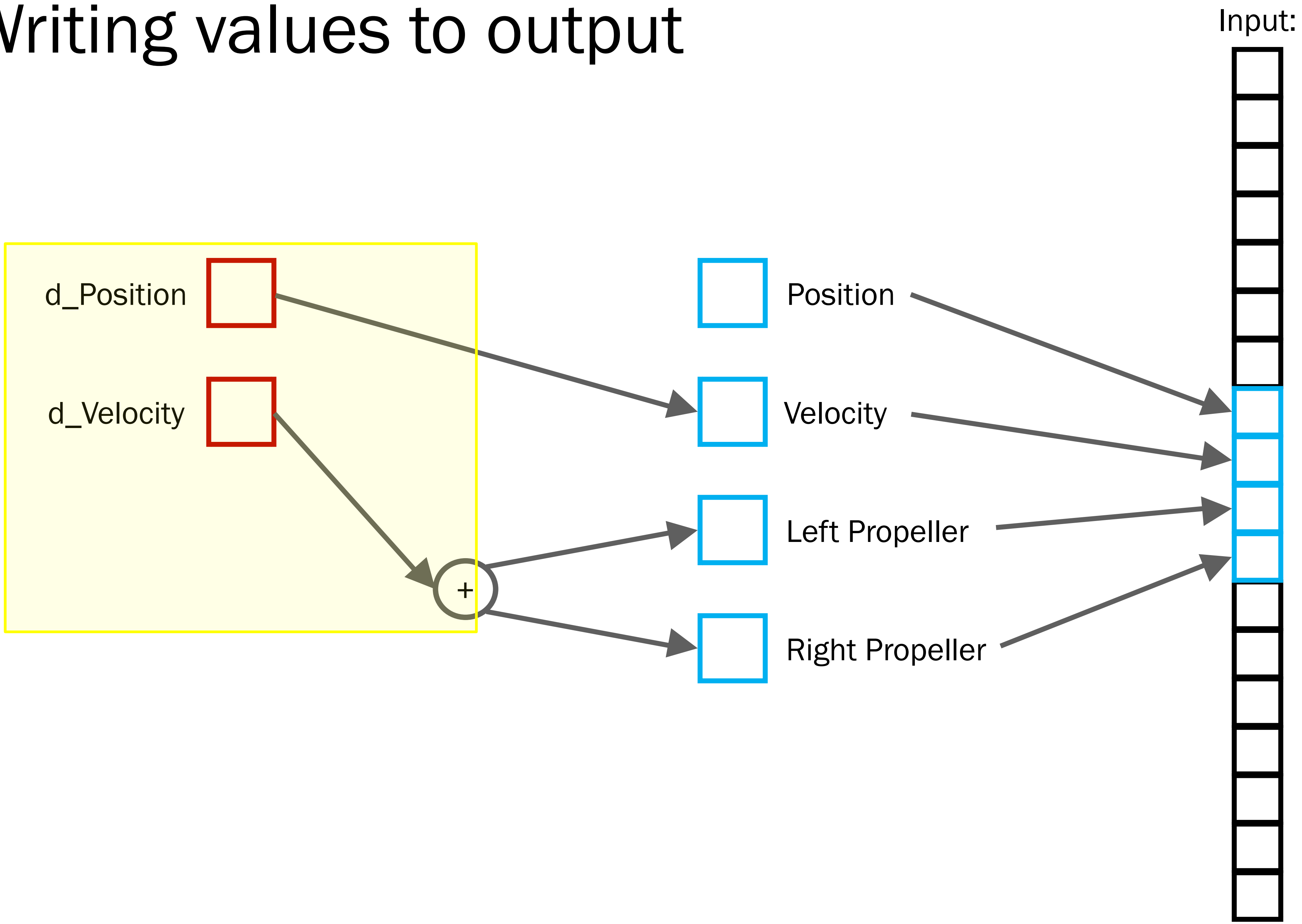
There is still this `evaluate` construct

```cpp
using Position = Quantity<class Position_>;
using Velocity = Quantity<class Velocity_>;
using AccPropLeft = Quantity<class AccLeft_>;
using AccPropRight = Quantity<class AccRight_>;
using ShipSystem = std::tuple<Position, Velocity, AccPropLeft, AccPropRight>;

int main() {
    auto dot_position = Velocity{};
    auto dot_velocity = AccPropLeft{} + AccPropRight{};

    std::array in{1.0, 2.0, 5.0, 5.0};
    auto dot_position_value = dot_position.evaluate<ShipSystem>(std::span(in)); // 2.0
    auto dot_velocity_value = dot_velocity.evaluate<ShipSystem>(std::span(in)); // 10.0
}
```




```
using Position = Quantity<class Position_>;
using Velocity = Quantity<class Velocity_>;
using AccPropLeft = Quantity<class AccLeft_>;
using AccPropRight = Quantity<class AccRight_>;
using ShipSystem = std::tuple<Position, Velocity, AccPropLeft, AccPropRight>;

int main() {
    auto dot_position = Velocity{};
    auto dot_velocity = AccPropLeft{} + AccPropRight{};

    std::array in{1.0, 2.0, 5.0, 5.0};
    auto dot_position_value = dot_position.evaluate<ShipSystem>(std::span(in)); // 2.0
    auto dot_velocity_value = dot_velocity.evaluate<ShipSystem>(std::span(in)); // 10.0
}
```

Writing values to output



Defining Derivatives

```
template <typename Operand_, typename Expression_>
struct Derivative
{
    using Expression = Expression_;
    using Operand = Operand_;
    using DependsOn = to_unique_tuple_t<typename Expression::DependsOn>;

    template <class Quantities, std::size_t N>
    static constexpr double evaluate(std::span<const double, N> arr)
    {
        return Expression::template evaluate<Quantities>(arr);
    }
};
```

Defining Derivatives

```cpp
template <typename Operand_, typename Expression_>
struct Derivative
{
    using Expression = Expression_;
    using Operand = Operand_;
    using DependsOn = to_unique_tuple_t<typename Expression::DependsOn>;

    template <class Quantities, std::size_t N>
    static constexpr double evaluate(std::span<const double, N> arr)
    {
        return Expression::template evaluate<Quantities>(arr);
    }
};
template <typename Operand, typename Expression>
constexpr auto dot(Expression /*expression*/)
{
    return Derivative<Operand, Expression>{};
}
```

Defining Derivatives

```
using Position = Quantity<class Position_>;
using Velocity = Quantity<class Velocity_>;
using AccPropLeft = Quantity<class AccLeft_>;
using AccPropRight = Quantity<class AccRight_>;
using ShipSystem = std::tuple<Position, Velocity, AccPropLeft, AccPropRight>;

int main() {
 auto dot_position = Velocity{};
 auto dot_velocity = AccPropLeft{} + AccPropRight{};

 std::array in{1.0, 2.0, 5.0, 5.0};
 std::array out{0.0, 0.0};
 out[0] = dot_position.evaluate<ShipSystem>(std::span(in));
 out[1] = dot_velocity.evaluate<ShipSystem>(std::span(in));
 // out = [2.0 , 10.0]
}
```

Defining Derivatives

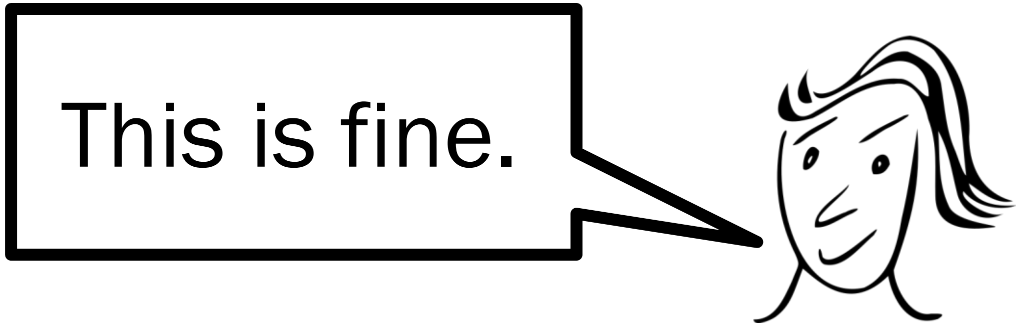
```
using Position = Quantity<class Position_>;
using Velocity = Quantity<class Velocity_>;
using AccPropLeft = Quantity<class AccLeft_>;
using AccPropRight = Quantity<class AccRight_>;
using ShipSystem = std::tuple<Position, Velocity, AccPropLeft, AccPropRight>;

int main() {
 auto dot_position = dot<Position>(Velocity{});
 auto dot_velocity = dot<Velocity>(AccPropLeft{} + AccPropRight{});

 std::array in{1.0, 2.0, 5.0, 5.0};
 std::array out{0.0, 0.0};
 out[get_idx<Position, ShipSystem>()]=dot_position.evaluate<ShipSystem>(std::span(in));
 out[get_idx<Velocity, ShipSystem>()]=dot_velocity.evaluate<ShipSystem>(std::span(in));
 // out = [2.0 , 10.0]
}
```

Defining Derivatives

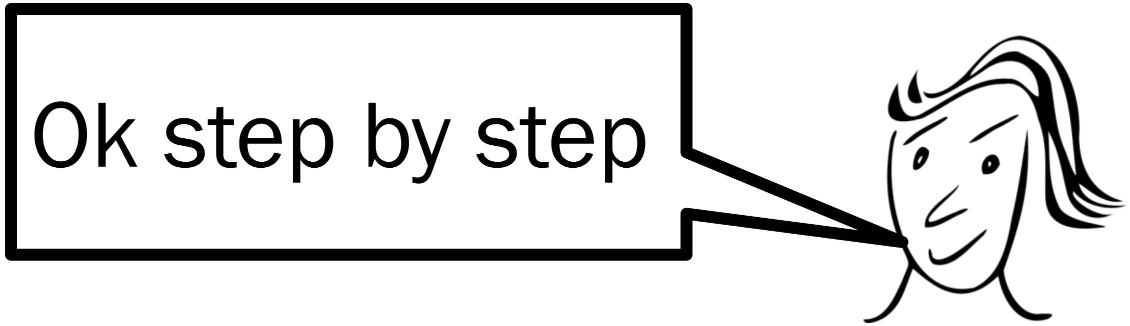
```cpp
using Position = Quantity<class Position_>;
using Velocity = Quantity<class Velocity_>;
using AccPropLeft = Quantity<class AccLeft_>;
using AccPropRight = Quantity<class AccRight_>;
using ShipSystem = std::tuple<Position, Velocity, AccPropLeft, AccPropRight>;

int main() {
 auto dot_position = dot<Position>(Velocity{});
 auto dot_velocity = dot<Velocity>(AccPropLeft{} + AccPropRight{});

 std::array in{1.0, 2.0, 5.0, 5.0};
 std::array out{0.0, 0.0};
 out[get_idx<decltype(dot_position)::Operand, ShipSystem>()] = /* ... */;
 out[get_idx<decltype(dot_velocity)::Operand, ShipSystem>()] = /* ... */;
 // out = [2.0 , 10.0]
}
```

Defining Derivatives

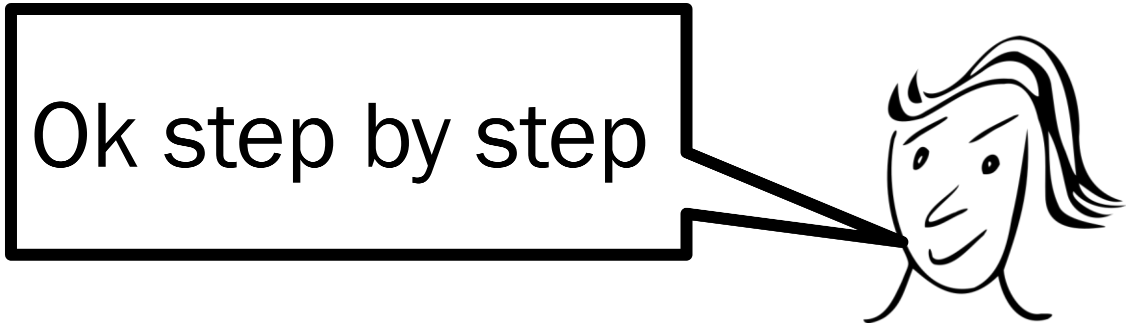
```
using Position = Quantity<class Position_>;
using Velocity = Quantity<class Velocity_>;
using AccPropLeft = Quantity<class AccLeft_>;
using AccPropRight = Quantity<class AccRight_>;

struct ShipMotion {
    constexpr static auto make_dot() {

        constexpr auto dot_position = dot<Position>(Velocity{});
        constexpr auto dot_velocity = dot<Velocity>(AccPropLeft{} + AccPropRight{});

        return std::make_tuple(dot_position, dot_velocity);
    }
};
```



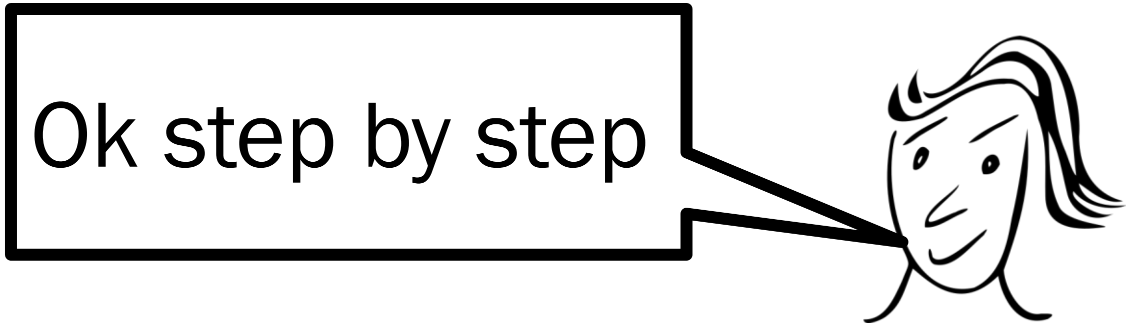

```
template <typename States, typename Controls, typename DerivativeSystem>
struct StateSpaceSystem
{
    constexpr static auto stateSize = std::tuple_size_v<States>;
    constexpr static auto controlSize = std::tuple_size_v<Controls>;

    constexpr static void evaluate(
        std::span<const double, stateSize + controlSize> statesIn,
        std::span<double, stateSize> derivativesOut)
    {
        tuple_for_each(
            DerivativeSystem::make_dot(),
            [statesIn, derivativesOut]<typename Derivative>(Derivative derivative) {
                constexpr auto outIdx = get_idx<typename Derivative::Operand, States>();
                derivativesOut[outIdx] = derivative.template evaluate<tuple_cat_t<States, Controls>>(statesIn);
            }
        );
    }
};
```



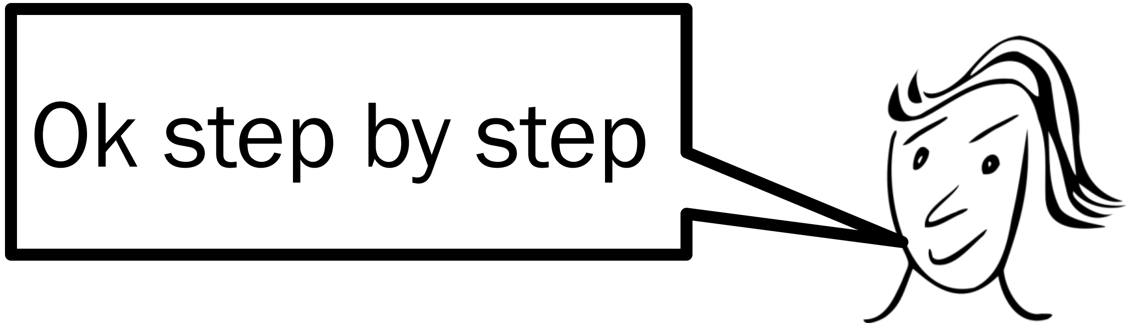

```
template <typename States, typename Controls, typename DerivativeSystem>
struct StateSpaceSystem
{
    constexpr static auto stateSize = std::tuple_size_v<States>;
    constexpr static auto controlSize = std::tuple_size_v<Controls>;

    constexpr static void evaluate(
        std::span<const double, stateSize + controlSize> statesIn,
        std::span<double, stateSize> derivativesOut)
    {
        tuple_for_each(
            DerivativeSystem::make_dot(),
            [statesIn, derivativesOut]<typename Derivative>(Derivative derivative) {
                constexpr auto outIdx = get_idx<typename Derivative::Operand, States>();
                derivativesOut[outIdx] = derivative.template evaluate<tuple_cat_t<States, Controls>>(statesIn);
            }
        );
    }
};
```




```
template <typename States, typename Controls, typename DerivativeSystem>
struct StateSpaceSystem
{
    constexpr static auto stateSize = std::tuple_size_v<States>;
    constexpr static auto controlSize = std::tuple_size_v<Controls>;

    constexpr static void evaluate(
        std::span<const double, stateSize + controlSize> statesIn,
        std::span<double, stateSize> derivativesOut)
    {
        tuple_for_each(
            DerivativeSystem::make_dot(),
            [statesIn, derivativesOut]<typename Derivative>(Derivative derivative) {
                constexpr auto outIdx = get_idx<typename Derivative::Operand, States>();
                derivativesOut[outIdx] = derivative.template evaluate<tuple_cat_t<States, Controls>>(statesIn);
            }
        );
    }
};
```




```
template <typename States, typename Controls, typename DerivativeSystem>
struct StateSpaceSystem
{
    constexpr static auto stateSize = std::tuple_size_v<States>;
    constexpr static auto controlSize = std::tuple_size_v<Controls>;

    constexpr static void evaluate(
        std::span<const double, stateSize + controlSize> statesIn,
        std::span<double, stateSize> derivativesOut)
    {
        tuple_for_each(
            DerivativeSystem::make_dot(),
            [statesIn, derivativesOut]<typename Derivative>(Derivative derivative) {
                constexpr auto outIdx = get_idx<typename Derivative::Operand, States>();
                derivativesOut[outIdx] = derivative.template evaluate<tuple_cat_t<States, Controls>>(statesIn);
            }
        );
    }
};
```




```
template <typename States, typename Controls, typename DerivativeSystem>
struct StateSpaceSystem
{
    constexpr static auto stateSize = std::tuple_size_v<States>;
    constexpr static auto controlSize = std::tuple_size_v<Controls>;

    constexpr static void evaluate(
        std::span<const double, stateSize + controlSize> statesIn,
        std::span<double, stateSize> derivativesOut)
    {
        tuple_for_each(
            DerivativeSystem::make_dot(),
            [statesIn, derivativesOut]<typename Derivative>(Derivative derivative) {
                constexpr auto outIdx = get_idx<typename Derivative::Operand, States>();
                derivativesOut[outIdx] = derivative.template evaluate<tuple_cat_t<States, Controls>>(statesIn);
            }
        );
    }
};
```

```cpp
using Position = Quantity<class Position_>;
using Velocity = Quantity<class Velocity_>;
using ShipStates = std::tuple<Position, Velocity>;
using AccPropLeft = Quantity<class AccLeft_>;
using AccPropRight = Quantity<class AccRight_>;
using ShipControls = std::tuple<AccPropLeft, AccPropRight>;

struct ShipMotion {
    constexpr static auto make_dot() {
        constexpr auto dot_position = dot<Position>(Velocity{});
        constexpr auto dot_velocity = dot<Velocity>(AccPropLeft{} + AccPropRight{});
        return std::make_tuple(dot_velocity, dot_position);
    }
};

int main() {
    std::array in{1.0, 2.0, 5.0, 5.0};
    std::array out{0.0, 0.0};
    StateSpaceSystem<ShipStates, ShipControls, ShipMotion>::evaluate(in, out);
    // out = [2.0 , 10.0]
}
```

What could possibly (still) go wrong?

$$\dot{x} = \frac{a}{b+c}$$

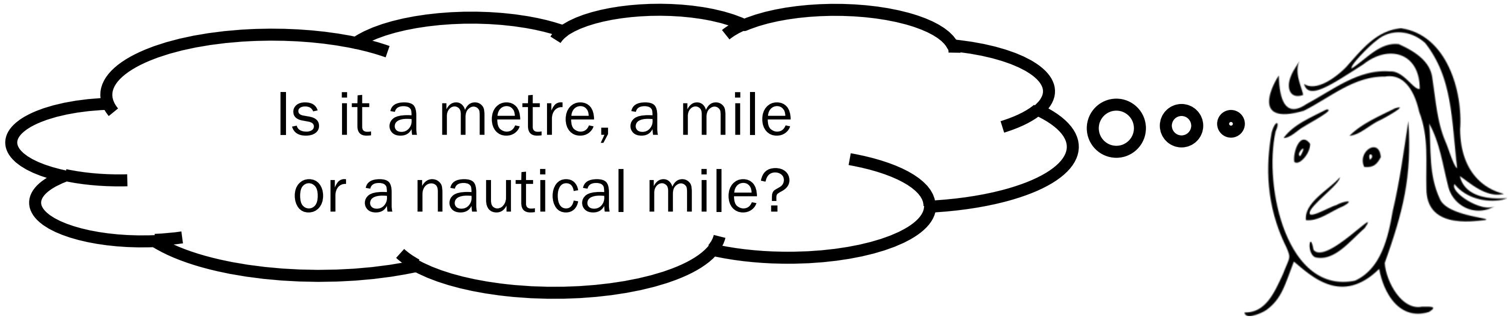
```
double dot_x = a / b + c;
```

The compiler needs to know the units:

https://github.com/mpusz/mp-units

by Mateusz Pusz et al. (v0.8.0)

```
template <typename Tag>
struct Quantity {

    using DependsOn = std::tuple<Quantity>;

    template <class Quantities, std::size_t N>
    constexpr static double evaluate(std::span<const double, N> arr);
};

using Position = Quantity<class Position_>;
```




```
template <typename Tag, typename Unit_>
struct Quantity {
    using Unit = Unit_;
    using DependsOn = std::tuple<Quantity>;

    template <class Quantities, std::size_t N>
    constexpr static double evaluate(std::span<const double, N> arr);
};

using Position = Quantity<class Position_, isq::si::length<isq::si::metre>>;
```

Units of Equations

```cpp
template <typename Lhs, typename Rhs>
struct Add
{

    using DependsOn = to_unique_tuple_t</*...*/>;

    template <class Quantities, std::size_t N>
    static constexpr double evaluate(std::span<const double, N> arr);
};
```

Units of Equations

```
template <typename Lhs, typename Rhs>
    requires std::is_same_v<typename Lhs::Unit, typename Rhs::Unit>
struct Add
{
    using Unit = typename Rhs::Unit;
    using DependsOn = to_unique_tuple_t</*...*/>;

    template <class Quantities, std::size_t N>
    static constexpr double evaluate(std::span<const double, N> arr);
};
```

Units of Derivatives

```cpp
template <typename Operand_, typename Expression_>
struct Derivative
{

    using Expression = Expression_;
    using Operand = Operand_;
    using DependsOn = to_unique_tuple_t<typename Expression::DependsOn>;

    template <class Quantities, std::size_t N>
    static constexpr double evaluate(std::span<const double, N> arr);
};
```

Units of Derivatives

```
template <typename Operand_, typename Expression_>
struct Derivative
{
    using Unit = derivative_in_time_t<typename Operand::Unit>;
    using Expression = Expression_;
    using Operand = Operand_;
    using DependsOn = to_unique_tuple_t<typename Expression::DependsOn>;

    template <class Quantities, std::size_t N>
    static constexpr double evaluate(std::span<const double, N> arr);
};

template<class Unit>
using derivative_in_time_t = decltype(std::declval<Unit>() / (time<second>{}));
```

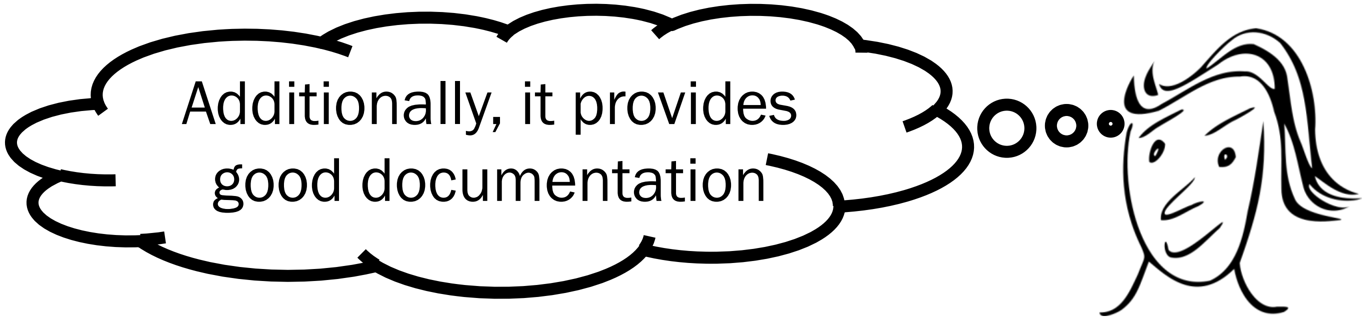
Units of Derivatives

```
template <typename Operand_, TimeDerivativeOf<Operand_> Expression_>
struct Derivative
{
    using Unit = derivative_in_time_t<typename Operand::Unit>;
    using Expression = Expression_;
    using Operand = Operand_;
    using DependsOn = to_unique_tuple_t<typename Expression::DependsOn>;

    template <class Quantities, std::size_t N>
    static constexpr double evaluate(std::span<const double, N> arr);
};

template<class Unit>
using derivative_in_time_t = decltype(std::declval<Unit>() / (time<second>{}));

template<typename Expression, typename Operand>
concept TimeDerivativeOf =
std::is_same_v<derivative_in_time_t<typename Operand::Unit>, typename Expression::Unit>;
```

Units in Action

```
using Position = Quantity<class Position_>;
using Velocity = Quantity<class Velocity_>;
using AccPropLeft = Quantity<class AccLeft_>;
using AccPropRight = Quantity<class AccRight_>;

struct ShipMotion {
    constexpr static auto make_dot() {

        constexpr auto dot_position = dot<Position>(Velocity{});
        constexpr auto dot_velocity = dot<Velocity>(AccPropLeft{} + AccPropRight{});

        return std::make_tuple(dot_position, dot_velocity);
    }
};
```

Units in Action




```
using Position = Quantity<class Position_, length<metre>>;
using Velocity = Quantity<class Velocity_, speed<metre_per_second>>;
using AccPropLeft = Quantity<class AccLeft_, acceleration<metre_per_second_sq>>;
using AccPropRight = Quantity<class AccRight_, acceleration<metre_per_second_sq>>;

struct ShipMotion {
    constexpr static auto make_dot() {

        constexpr auto dot_position = dot<Position>(Velocity{});
        constexpr auto dot_velocity = dot<Velocity>(AccPropLeft{} + AccPropRight{});

        return std::make_tuple(dot_position, dot_velocity);
    }
};
```

Result so far – where we started

```
void Deneb::ode(double* dotX, const double* xAtT, const double* uAtT) const
{
    const double velocity = xAtT[3];
    const double yaw = xAtT[2];
    const double rot = xAtT[4];
    const double eot = uAtT[0];
    const double rudder_angle = uAtT[1];
    dotX[0] = velocity * cos(yaw);
    dotX[1] = velocity * sin(yaw);
    dotX[2] = rot;
    dotX[3] = p1 * cos(rudder_angle) * (eot / 100.0);
    dotX[4] = p2 * sin(rudder_angle) * (eot / 100.0);
}
```

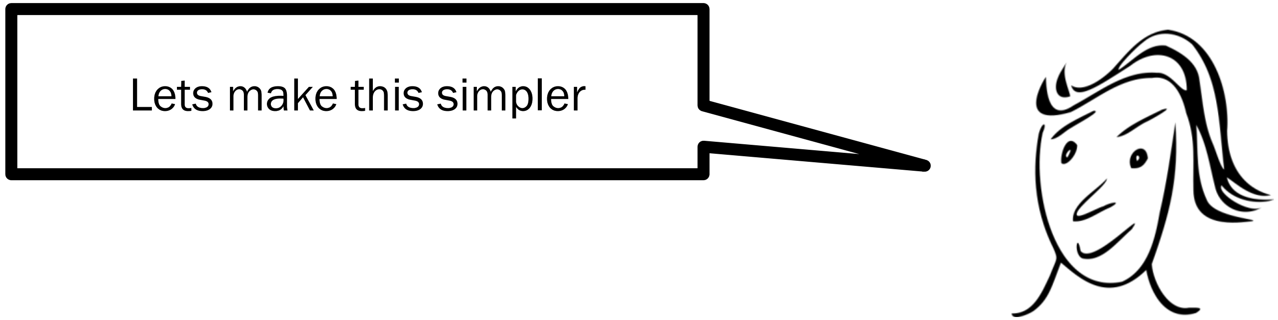
Result so far – where we are

```
struct DenebMotion
{
 constexpr static auto make_dot()
 {
  auto dot_pos_x0 = dot<PositionX0>(Velocity{} * cos(Yaw{}));
  auto dot_pos_x1 = dot<PositionX1>(Velocity{} * sin(Yaw{}));
  auto dot_yaw = dot<Yaw>(RateOfTurn{});
  auto dot_velocity = dot<Velocity>(p1{} * cos(PropellerAngle{}) * (EOT{} / hundred{}));
  auto dot_rot = dot<RateOfTurn>(p2{} * sin(PropellerAngle{}) * (EOT{} / hundred{}));

  return std::make_tuple(dot_pos_x0, dot_pos_x1, dot_yaw, dot_velocity, dot_rot);
 }
};
```

Result so far

```
using PositionX0 = Quantity<class PositionX0_, length<metre>>;
using PositionX1 = Quantity<class PositionX1_, length<metre>>;
using Velocity = Quantity<class Velocity_, speed<metre_per_second>>;
using Yaw = Quantity<class Yaw_, angle<radian, double>>;
using RateOfTurn = Quantity<class RateOfTurn_, angular_velocity<radian_per_second>>;
using PropellerAngle = Quantity<class PropellerAngle_, angle<radian, double>>;
using EOT = Quantity<class EOT_, dimensionless<one>>;

using p1 = ScalarValue<std::ratio<1, 2>, acceleration<metre_per_second_sq>>;
using p2 = ScalarValue<std::ratio<1, 3>, angular_acceleration<radian_per_second_sq>>;
using hundred = ScalarValue<std::ratio<100>, dimensionless<one>>;

struct DenebMotion
{
 constexpr static auto make_dot()
 {
  auto dot_pos_x0 = dot<PositionX0>(Velocity{} * cos(Yaw{}));
  auto dot_pos_x1 = dot<PositionX1>(Velocity{} * sin(Yaw{}));
  auto dot_yaw = dot<Yaw>(RateOfTurn{});
  auto dot_velocity = dot<Velocity>(p1{} * cos(PropellerAngle{}) * (EOT{} / hundred{}));
  auto dot_rot = dot<RateOfTurn>(p2{} * sin(PropellerAngle{}) * (EOT{} / hundred{}));

  return std::make_tuple(dot_pos_x0, dot_pos_x1, dot_yaw, dot_velocity, dot_rot);
 }
};
using DenebStates = std::tuple<PositionX0, PositionX1, Velocity, Yaw, RateOfTurn>;
using DenebControls = std::tuple<PropellerAngle, EOT>;
using DenebSystem = StateSpaceSystem<DenebStates, DenebControls, DenebMotion>;
```

Result so far




```
using PositionX0 = Quantity<class PositionX0_, length<metre>>;
using PositionX1 = Quantity<class PositionX1_, length<metre>>;
using Velocity = Quantity<class Velocity_, speed<metre_per_second>>;
using Yaw = Quantity<class Yaw_, angle<radian, double>>;
using RateOfTurn = Quantity<class RateOfTurn_, angular_velocity<radian_per_second>>;
using PropellerAngle = Quantity<class PropellerAngle_, angle<radian, double>>;
using EOT = Quantity<class EOT_, dimensionless<one>>;

using p1 = ScalarValue<std::ratio<1, 2>, acceleration<metre_per_second_sq>>;
using p2 = ScalarValue<std::ratio<1, 3>, angular_acceleration<radian_per_second_sq>>;
using hundred = ScalarValue<std::ratio<100>, dimensionless<one>>;

struct DenebMotion
{
 constexpr static auto make_dot()
 {
  auto dot_pos_x0 = dot<PositionX0>(Velocity{} * cos(Yaw{}));
  auto dot_pos_x1 = dot<PositionX1>(Velocity{} * sin(Yaw{}));
  auto dot_yaw = dot<Yaw>(RateOfTurn{});
  auto dot_velocity = dot<Velocity>(p1{} * cos(PropellerAngle{}) * (EOT{} / hundred{}));
  auto dot_rot = dot<RateOfTurn>(p2{} * sin(PropellerAngle{}) * (EOT{} / hundred{}));

  return std::make_tuple(dot_pos_x0, dot_pos_x1, dot_yaw, dot_velocity, dot_rot);
 }
};
using DenebStates = std::tuple<PositionX0, PositionX1, Velocity, Yaw, RateOfTurn>;
using DenebControls = std::tuple<PropellerAngle, EOT>;
using DenebSystem = StateSpaceSystem<DenebStates, DenebControls, DenebMotion>;
```

Defining the StateSpaceSystem

```
using PositionX0 = Quantity<class PositionX0_, length<metre>>;
using PositionX1 = Quantity<class PositionX1_, length<metre>>;
using Velocity = Quantity<class Velocity_, speed<metre_per_second>>;
using Yaw = Quantity<class Yaw_, angle<radian, double>>;
using RateOfTurn = Quantity<class RateOfTurn_, angular_velocity<radian_per_second>>;
using PropellerAngle = Quantity<class PropellerAngle_, angle<radian, double>>;
using EOT = Quantity<class EOT_, dimensionless<one>>;

using p1 = ScalarValue<std::ratio<1, 2>, acceleration<metre_per_second_sq>>;
using p2 = ScalarValue<std::ratio<1, 3>, angular_acceleration<radian_per_second_sq>>;
using hundred = ScalarValue<std::ratio<100>, dimensionless<one>>;

struct DenebMotion
{
 constexpr static auto make_dot()
 {
  auto dot_pos_x0 = dot<PositionX0>(Velocity{} * cos(Yaw{}));
  auto dot_pos_x1 = dot<PositionX1>(Velocity{} * sin(Yaw{}));
  auto dot_yaw = dot<Yaw>(RateOfTurn{});
  auto dot_velocity = dot<Velocity>(p1{} * cos(PropellerAngle{}) * (EOT{} / hundred{}));
  auto dot_rot = dot<RateOfTurn>(p2{} * sin(PropellerAngle{}) * (EOT{} / hundred{}));

  return std::make_tuple(dot_pos_x0, dot_pos_x1, dot_yaw, dot_velocity, dot_rot);
 }
};
using DenebStates = std::tuple<PositionX0, PositionX1, Velocity, Yaw, RateOfTurn>;
using DenebControls = std::tuple<PropellerAngle, EOT>;
using DenebSystem = StateSpaceSystem<DenebStates, DenebControls, DenebMotion>;
```

Defining the StateSpaceSystem

```
using DenebStates = std::tuple<PositionX0, PositionX1, Velocity, Yaw, RateOfTurn>;
using DenebControls = std::tuple<PropellerAngle, EOT>;
using DenebSystem = StateSpaceSystem<DenebStates, DenebControls, DenebMotion>;
```

Defining the StateSpaceSystem

```
using DenebStates = std::tuple<PositionX0, PositionX1, Velocity, Yaw, RateOfTurn>;
using DenebControls = std::tuple<PropellerAngle, EOT>;
using DenebSystem = StateSpaceSystem<DenebStates, DenebControls, DenebMotion>;

struct DenebMotion
{
 constexpr static auto make_dot()
 {
  auto dot_pos_x0 =   dot<PositionX0>(Velocity{} * cos(Yaw{}));
  auto dot_pos_x1 =   dot<PositionX1>(Velocity{} * sin(Yaw{}));
  auto dot_yaw =      dot<Yaw>       (RateOfTurn{});
  auto dot_velocity = dot<Velocity>  (p1{}*cos(PropellerAngle{}) * (EOT{} / hundred{}));
  auto dot_rot =      dot<RateOfTurn>(p2{}*sin(PropellerAngle{}) * (EOT{} / hundred{}));

  return std::make_tuple(dot_pos_x0, dot_pos_x1, dot_yaw, dot_velocity, dot_rot);
 }
};
```

Defining the StateSpaceSystem

```
using DenebStates = std::tuple<PositionX0, PositionX1, Velocity, Yaw, RateOfTurn>;
using DenebControls = std::tuple<PropellerAngle, EOT>;
using DenebSystem = StateSpaceSystem<DenebStates, DenebControls, DenebMotion>;

struct DenebMotion
{
 constexpr static auto make_dot()
 {
  auto dot_pos_x0 =   dot<PositionX0>(Velocity{} * cos(Yaw{}));
  auto dot_pos_x1 =   dot<PositionX1>(Velocity{} * sin(Yaw{}));
  auto dot_yaw =      dot<Yaw>       (RateOfTurn{});
  auto dot_velocity = dot<Velocity>  (p1{}*cos(PropellerAngle{}) * (EOT{} / hundred{}));
  auto dot_rot =      dot<RateOfTurn>(p2{}*sin(PropellerAngle{}) * (EOT{} / hundred{}));

  return std::make_tuple(dot_pos_x0, dot_pos_x1, dot_yaw, dot_velocity, dot_rot);
 }
};
```

Defining the StateSpaceSystem

```
using DenebStates = std::tuple<PositionX0, PositionX1, Velocity, Yaw, RateOfTurn>;
using DenebControls = std::tuple<PropellerAngle, EOT>;
using DenebSystem = StateSpaceSystem<DenebStates, DenebControls, DenebMotion>;

struct DenebMotion
{
 constexpr static auto make_dot()
 {
  auto dot_pos_x0 =   dot<PositionX0>(Velocity{} * cos(Yaw{}));
  auto dot_pos_x1 =   dot<PositionX1>(Velocity{} * sin(Yaw{}));
  auto dot_yaw =      dot<Yaw>       (RateOfTurn{});
  auto dot_velocity = dot<Velocity>  (p1{}*cos(PropellerAngle{}) * (EOT{} / hundred{}));
  auto dot_rot =      dot<RateOfTurn>(p2{}*sin(PropellerAngle{}) * (EOT{} / hundred{}));

  return std::make_tuple(dot_pos_x0, dot_pos_x1, dot_yaw, dot_velocity, dot_rot);
 }
};
```

Derivatives - Recap

```
template <typename Operand_, TimeDerivativeOf<Operand_> Expression_>
struct Derivative
{
    using Unit = derivative_in_time_t<typename Operand::Unit>;
    using Expression = Expression_;
    using Operand = Operand_;
    using DependsOn = to_unique_tuple_t<typename Expression::DependsOn>;

    template <class Quantities, std::size_t N>
    static constexpr double evaluate(std::span<const double, N> arr);
};
```

Derivatives - Recap

```cpp
template <typename Operand_, TimeDerivativeOf<Operand_> Expression_>
struct Derivative
{
    using Unit = derivative_in_time_t<typename Operand::Unit>;
    using Expression = Expression_;
    using Operand = Operand_;
    using DependsOn = to_unique_tuple_t<typename Expression::DependsOn>;

    template <class Quantities, std::size_t N>
    static constexpr double evaluate(std::span<const double, N> arr);
};
```

Computing States

```
template <typename... DerivEquation>
constexpr auto get_states_defined_by(std::tuple<DerivEquation...> /*drvs*/)
{
    return std::tuple<typename DerivEquation::Operand...>{};
}

template <typename DerivativeSystem>
using system_states_t = decltype(get_states_defined_by(DerivativeSystem::make_dot()));
```

Computing All Dependants

```
template <typename... DerivEquation>
constexpr auto get_state_dependencies_defined_by(std::tuple<DerivEquation...>/*drvs*/)
{
    using DependsOn = to_unique_tuple_t<
                        tuple_cat_t<typename DerivEquation::DependsOn...>
                      >;
    return DependsOn{};
}
```

Computing Controls

```
template <typename... DerivEquation>
constexpr auto get_controls_defined_by(std::tuple<DerivEquation...> drvs)
{
    using States = decltype(get_states_defined_by(drvs));
    using Dependencies = decltype(get_state_dependencies_defined_by(drvs));
    return distinct_tuple_of<Dependencies, States>{};
}

template <typename DerivativeSystem>
using system_controls_t=decltype(get_controls_defined_by(DerivativeSystem::make_dot()));
```

Computing Controls

```
template <typename... DerivEquation>
constexpr auto get_controls_defined_by(std::tuple<DerivEquation...> drvs)
{
    using States = decltype(get_states_defined_by(drvs));
    using Dependencies = decltype(get_state_dependencies_defined_by(drvs));
    return distinct_tuple_of<Dependencies, States>{};
}

template <typename DerivativeSystem>
using system_controls_t=decltype(get_controls_defined_by(DerivativeSystem::make_dot()));
```

Computing Controls

```
template <typename... DerivEquation>
constexpr auto get_controls_defined_by(std::tuple<DerivEquation...> drvs)
{
    using States = decltype(get_states_defined_by(drvs));
    using Dependencies = decltype(get_state_dependencies_defined_by(drvs));
    return distinct_tuple_of<Dependencies, States>{};
}

template <typename DerivativeSystem>
using system_controls_t=decltype(get_controls_defined_by(DerivativeSystem::make_dot()));
```

Computing the StateSpaceSystem

```
template <typename DerivativeSystem>
using StateSpaceSystemOf =
    StateSpaceSystem<
        system_states_t<DerivativeSystem>,
        system_controls_t<DerivativeSystem>,
        DerivativeSystem>;
```

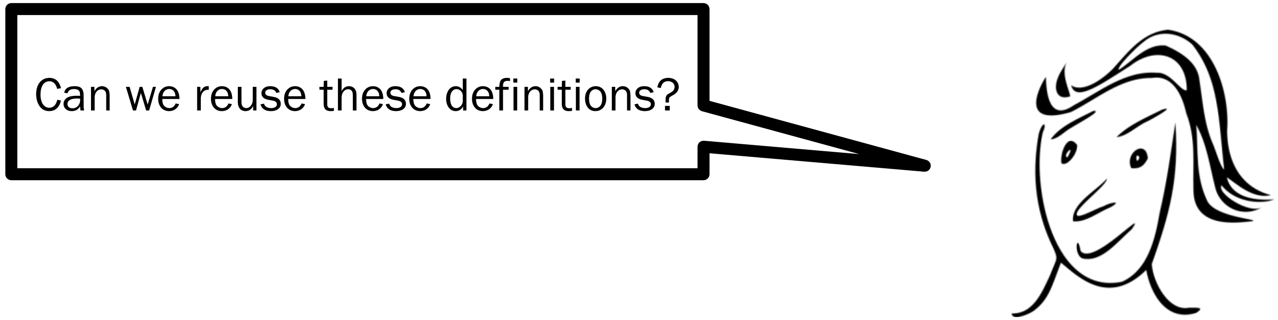
Result so far

```
using PositionX0 = Quantity<class PositionX0_, length<metre>>;
using PositionX1 = Quantity<class PositionX1_, length<metre>>;
using Velocity = Quantity<class Velocity_, speed<metre_per_second>>;
using Yaw = Quantity<class Yaw_, angle<radian, double>>;
using RateOfTurn = Quantity<class RateOfTurn_, angular_velocity<radian_per_second>>;
using PropellerAngle = Quantity<class PropellerAngle_, angle<radian, double>>;
using EOT = Quantity<class EOT_, dimensionless<one>>;

using p1 = ScalarValue<std::ratio<1, 2>, acceleration<metre_per_second_sq>>;
using p2 = ScalarValue<std::ratio<1, 3>, angular_acceleration<radian_per_second_sq>>;
using hundred = ScalarValue<std::ratio<100>, dimensionless<one>>;

struct DenebMotion
{
 constexpr static auto make_dot()
 {
  auto dot_pos_x0 = dot<PositionX0>(Velocity{} * cos(Yaw{}));
  auto dot_pos_x1 = dot<PositionX1>(Velocity{} * sin(Yaw{}));
  auto dot_yaw = dot<Yaw>(RateOfTurn{});
  auto dot_velocity = dot<Velocity>(p1{} * cos(PropellerAngle{}) * (EOT{} / hundred{}));
  auto dot_rot = dot<RateOfTurn>(p2{} * sin(PropellerAngle{}) * (EOT{} / hundred{}));

  return std::make_tuple(dot_pos_x0, dot_pos_x1, dot_yaw, dot_velocity, dot_rot);
 }
};
using DenebStates = std::tuple<PositionX0, PositionX1, Velocity, Yaw, RateOfTurn>;
using DenebControls = std::tuple<PropellerAngle, EOT>;
using DenebSystem = StateSpaceSystem<DenebStates, DenebControls, DenebMotion>;
```

Result so far

```
using PositionX0 = Quantity<class PositionX0_, length<metre>>;
using PositionX1 = Quantity<class PositionX1_, length<metre>>;
using Velocity = Quantity<class Velocity_, speed<metre_per_second>>;
using Yaw = Quantity<class Yaw_, angle<radian, double>>;
using RateOfTurn = Quantity<class RateOfTurn_, angular_velocity<radian_per_second>>;
using PropellerAngle = Quantity<class PropellerAngle_, angle<radian, double>>;
using EOT = Quantity<class EOT_, dimensionless<one>>;

using p1 = ScalarValue<std::ratio<1, 2>, acceleration<metre_per_second_sq>>;
using p2 = ScalarValue<std::ratio<1, 3>, angular_acceleration<radian_per_second_sq>>;
using hundred = ScalarValue<std::ratio<100>, dimensionless<one>>;

struct DenebMotion
{
 constexpr static auto make_dot()
 {
  auto dot_pos_x0 = dot<PositionX0>(Velocity{} * cos(Yaw{}));
  auto dot_pos_x1 = dot<PositionX1>(Velocity{} * sin(Yaw{}));
  auto dot_yaw = dot<Yaw>(RateOfTurn{});
  auto dot_velocity = dot<Velocity>(p1{} * cos(PropellerAngle{}) * (EOT{} / hundred{}));
  auto dot_rot = dot<RateOfTurn>(p2{} * sin(PropellerAngle{}) * (EOT{} / hundred{}));

  return std::make_tuple(dot_pos_x0, dot_pos_x1, dot_yaw, dot_velocity, dot_rot);
 }
};
using DenebSystem = StateSpaceSystemOf<DenebMotion>;
```

Result so far

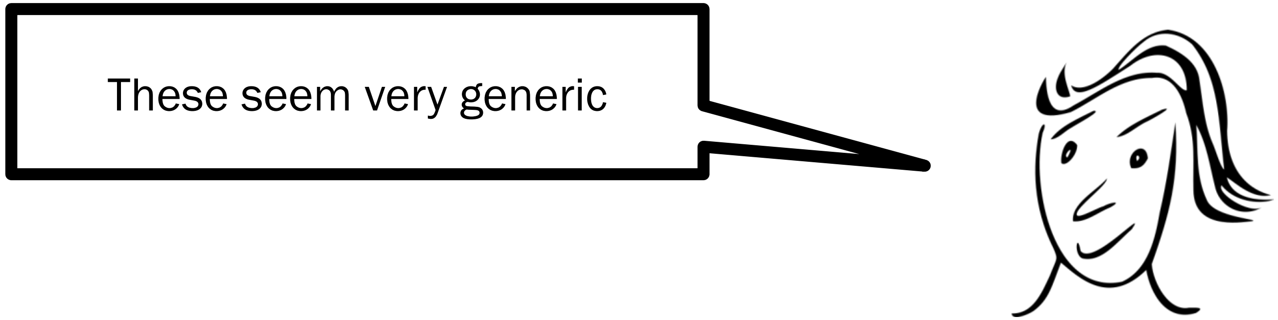



```
using PositionX0 = Quantity<class PositionX0_, length<metre>>;
using PositionX1 = Quantity<class PositionX1_, length<metre>>;
using Velocity = Quantity<class Velocity_, speed<metre_per_second>>;
using Yaw = Quantity<class Yaw_, angle<radian, double>>;
using RateOfTurn = Quantity<class RateOfTurn_, angular_velocity<radian_per_second>>;
using PropellerAngle = Quantity<class PropellerAngle_, angle<radian, double>>;
using EOT = Quantity<class EOT_, dimensionless<one>>;

using p1 = ScalarValue<std::ratio<1, 2>, acceleration<metre_per_second_sq>>;
using p2 = ScalarValue<std::ratio<1, 3>, angular_acceleration<radian_per_second_sq>>;
using hundred = ScalarValue<std::ratio<100>, dimensionless<one>>;

struct DenebMotion
{
 constexpr static auto make_dot()
 {
  auto dot_pos_x0 = dot<PositionX0>(Velocity{} * cos(Yaw{}));
  auto dot_pos_x1 = dot<PositionX1>(Velocity{} * sin(Yaw{}));
  auto dot_yaw = dot<Yaw>(RateOfTurn{});
  auto dot_velocity = dot<Velocity>(p1{} * cos(PropellerAngle{}) * (EOT{} / hundred{}));
  auto dot_rot = dot<RateOfTurn>(p2{} * sin(PropellerAngle{}) * (EOT{} / hundred{}));

  return std::make_tuple(dot_pos_x0, dot_pos_x1, dot_yaw, dot_velocity, dot_rot);
 }
};
using DenebSystem = StateSpaceSystemOf<DenebMotion>;
```

Result so far

```
struct DenebMotion
{
 constexpr static auto make_dot()
 {
  auto dot_pos_x0 = dot<PositionX0>(Velocity{} * cos(Yaw{}));
  auto dot_pos_x1 = dot<PositionX1>(Velocity{} * sin(Yaw{}));
  auto dot_yaw = dot<Yaw>(RateOfTurn{});
  auto dot_velocity = dot<Velocity>(p1{} * cos(PropellerAngle{}) * (EOT{} / hundred{}));
  auto dot_rot = dot<RateOfTurn>(p2{} * sin(PropellerAngle{}) * (EOT{} / hundred{}));

  return std::make_tuple(dot_pos_x0, dot_pos_x1, dot_yaw, dot_velocity, dot_rot);
 }
};
using DenebSystem = StateSpaceSystemOf<DenebMotion>;
```

Result so far




```
struct DenebMotion
{
 constexpr static auto make_dot()
 {
  auto dot_pos_x0 = dot<PositionX0>(Velocity{} * cos(Yaw{}));
  auto dot_pos_x1 = dot<PositionX1>(Velocity{} * sin(Yaw{}));
  auto dot_yaw = dot<Yaw>(RateOfTurn{});
  auto dot_velocity = dot<Velocity>(p1{} * cos(PropellerAngle{}) * (EOT{} / hundred{}));
  auto dot_rot = dot<RateOfTurn>(p2{} * sin(PropellerAngle{}) * (EOT{} / hundred{}));

  return std::make_tuple(dot_pos_x0, dot_pos_x1, dot_yaw, dot_velocity, dot_rot);
 }
};
using DenebSystem = StateSpaceSystemOf<DenebMotion>;
```

Combining Systems

```
struct Motion2D {
 constexpr static auto make_dot() {
  auto dot_pos_x0 = dot<PositionX0>(Velocity{} * cos(Yaw{}));
  auto dot_pos_x1 = dot<PositionX1>(Velocity{} * sin(Yaw{}));
  auto dot_yaw = dot<Yaw>(RateOfTurn{});
  return std::make_tuple(dot_pos_x0, dot_pos_x1, dot_yaw);
 }
};
struct TurnablePropeller {
 constexpr static auto make_dot() {
  auto dot_velocity = dot<Velocity>(p1{} * cos(PropellerAngle{}) * (EOT{} / hundred{}));
  auto dot_rot = dot<RateOfTurn>(p2{} * sin(PropellerAngle{}) * (EOT{} / hundred{}));
  return std::make_tuple(dot_velocity, dot_rot);
 }
};
using DenebMotion = Combine<Motion2D, TurnablePropeller>;
```

Combining Systems

```
struct Motion2D {
 constexpr static auto make_dot() {
  auto dot_pos_x0 = dot<PositionX0>(Velocity{} * cos(Yaw{}));
  auto dot_pos_x1 = dot<PositionX1>(Velocity{} * sin(Yaw{}));
  auto dot_yaw = dot<Yaw>(RateOfTurn{});
  return std::make_tuple(dot_pos_x0, dot_pos_x1, dot_yaw);
 }
};
struct TurnablePropeller {
 constexpr static auto make_dot() {
  auto dot_velocity = dot<Velocity>(p1{} * cos(PropellerAngle{}) * (EOT{} / hundred{}));
  auto dot_rot = dot<RateOfTurn>(p2{} * sin(PropellerAngle{}) * (EOT{} / hundred{}));
  return std::make_tuple(dot_velocity, dot_rot);
 }
};
using DenebMotion = Combine<Motion2D, TurnablePropeller>;
```

Combining Systems

```
struct Motion2D {
 constexpr static auto make_dot() {
  auto dot_pos_x0 = dot<PositionX0>(Velocity{} * cos(Yaw{}));
  auto dot_pos_x1 = dot<PositionX1>(Velocity{} * sin(Yaw{}));
  auto dot_yaw = dot<Yaw>(RateOfTurn{});
  return std::make_tuple(dot_pos_x0, dot_pos_x1, dot_yaw);
 }
};
struct TurnablePropeller {
 constexpr static auto make_dot() {
  auto dot_velocity = dot<Velocity>(p1{} * cos(PropellerAngle{}) * (EOT{} / hundred{}));
  auto dot_rot = dot<RateOfTurn>(p2{} * sin(PropellerAngle{}) * (EOT{} / hundred{}));
  return std::make_tuple(dot_velocity, dot_rot);
 }
};
using DenebMotion = Combine<Motion2D, TurnablePropeller>;
```

Combining Systems

```
struct Motion2D {
 constexpr static auto make_dot() {
  auto dot_pos_x0 = dot<PositionX0>(Velocity{} * cos(Yaw{}));
  auto dot_pos_x1 = dot<PositionX1>(Velocity{} * sin(Yaw{}));
  auto dot_yaw = dot<Yaw>(RateOfTurn{});
  return std::make_tuple(dot_pos_x0, dot_pos_x1, dot_yaw);
 }
};
struct TurnablePropeller {
 constexpr static auto make_dot() {
  auto dot_velocity = dot<Velocity>(p1{} * cos(PropellerAngle{}) * (EOT{} / hundred{}));
  auto dot_rot = dot<RateOfTurn>(p2{} * sin(PropellerAngle{}) * (EOT{} / hundred{}));
  return std::make_tuple(dot_velocity, dot_rot);
 }
};
using DenebMotion = Combine<Motion2D, TurnablePropeller>;
```

Combining Systems

```
struct TurnablePropeller {
 constexpr static auto make_dot() {
   auto dot_velocity = dot<Velocity>(p1{} * cos(PropellerAngle{}) * (EOT{} / hundred{}));
   auto dot_rot = dot<RateOfTurn>(p2{} * sin(PropellerAngle{}) * (EOT{} / hundred{}));
   return std::make_tuple(dot_velocity, dot_rot);
 }
};
struct StaticPropeller {
 constexpr static auto make_dot() {
   auto dot_velocity = dot<Velocity>(p3{} * (EOT_STATIC{} / hundred{}));
   return std::make_tuple(dot_velocity);
 }
};
```

Combining Systems

```
struct TurnablePropeller {
 constexpr static auto make_dot() {
   auto dot_velocity = dot<Velocity>(p1{} * cos(PropellerAngle{}) * (EOT{} / hundred{}));
   auto dot_rot = dot<RateOfTurn>(p2{} * sin(PropellerAngle{}) * (EOT{} / hundred{}));
   return std::make_tuple(dot_velocity, dot_rot);
 }
};
struct StaticPropeller {
 constexpr static auto make_dot() {
   auto dot_velocity = dot<Velocity>(p3{} * (EOT_STATIC{} / hundred{}));
   return std::make_tuple(dot_velocity);
 }
};
```

Combining Systems

```
struct TurnablePropeller {
 constexpr static auto make_dot() {
   auto dot_velocity = dot<Velocity>(p1{} * cos(PropellerAngle{}) * (EOT{} / hundred{}));
   auto dot_rot = dot<RateOfTurn>(p2{} * sin(PropellerAngle{}) * (EOT{} / hundred{}));
   return std::make_tuple(dot_velocity, dot_rot);
 }
};
struct StaticPropeller {
 constexpr static auto make_dot() {
   auto dot_velocity = dot<Velocity>(p3{} * (EOT_STATIC{} / hundred{}));
   return std::make_tuple(dot_velocity);
 }
};
```

Combining Systems - TODO

```
struct TurnablePropeller {
 constexpr static auto make_dot() {
   auto dot_velocity = dot<Velocity>(p1{} * cos(PropellerAngle{}) * (EOT{} / hundred{}));
   auto dot_rot = dot<RateOfTurn>(p2{} * sin(PropellerAngle{}) * (EOT{} / hundred{}));
}};
struct StaticPropeller {
 constexpr static auto make_dot() {
   auto dot_velocity = dot<Velocity>(p3{} * (EOT_STATIC{} / hundred{}));
}};
```

Combining Systems - TODO

```
struct TurnablePropeller {
 constexpr static auto make_dot() {
   auto dot_velocity = dot<Velocity>(p1{} * cos(PropellerAngle{}) * (EOT{} / hundred{}));
   auto dot_rot = dot<RateOfTurn>(p2{} * sin(PropellerAngle{}) * (EOT{} / hundred{}));
}};
struct StaticPropeller {
 constexpr static auto make_dot() {
   auto dot_velocity = dot<Velocity>(p3{} * (EOT_STATIC{} / hundred{}));
}};
```

Combining Systems - TODO

```
struct TurnablePropeller {
 constexpr static auto make_dot() {
   auto dot_velocity = dot<Velocity>(p1{} * cos(PropellerAngle{}) * (EOT{} / hundred{}));
   auto dot_rot = dot<RateOfTurn>(p2{} * sin(PropellerAngle{}) * (EOT{} / hundred{}));
}};
struct StaticPropeller {
 constexpr static auto make_dot() {
   auto dot_velocity = dot<Velocity>(p3{} * (EOT_STATIC{} / hundred{}));
}};
```

Combining Systems - TODO

```
struct TurnablePropeller {
 constexpr static auto make_dot() {
   auto dot_velocity = dot<Velocity>(p1{} * cos(PropellerAngle{}) * (EOT{} / hundred{}));
   auto dot_rot = dot<RateOfTurn>(p2{} * sin(PropellerAngle{}) * (EOT{} / hundred{}));
}};
struct StaticPropeller {
 constexpr static auto make_dot() {
   auto dot_velocity = dot<Velocity>(p3{} * (EOT_STATIC{} / hundred{}));
}};

auto dot_velocity = dot<Velocity>(
                          p1{} * cos(PropellerAngle{}) * (EOT{} / hundred{})
                        + p3{} * (EOT_STATIC{} / hundred{}));
```

Combining Systems - TODO

```
struct TurnablePropeller {
 constexpr static auto make_dot() {
   auto dot_velocity = dot<Velocity>(p1{} * cos(PropellerAngle{}) * (EOT{} / hundred{}));
   auto dot_rot = dot<RateOfTurn>(p2{} * sin(PropellerAngle{}) * (EOT{} / hundred{}));
}};
struct StaticPropeller {
 constexpr static auto make_dot() {
   auto dot_velocity = dot<Velocity>(p3{} * (EOT_STATIC{} / hundred{}));
}};

auto dot_velocity = dot<Velocity>(
                          p1{} * cos(PropellerAngle{}) * (EOT{} / hundred{})
                        + p3{} * (EOT_STATIC{} / hundred{}));

auto dot_rot = dot<RateOfTurn>(
    p2{} * sin(PropellerAngle{}) * (EOT{} / hundred{}));
```

Combining Systems - States

```
template <typename... DerivativeSystem>
using CombinedOperands =
        to_unique_tuple_t<tuple_cat_t<states_defined_by_t<DerivativeSystem>...>>;
```

Combining Systems - Operands

```
template <typename... DerivativeSystem>
using CombinedOperands =
        to_unique_tuple_t<tuple_cat_t<states_defined_by_t<DerivativeSystem>...>>;
```

Combining Systems - Expressions

```
template<std::size_t idx, typename Tuple>
using at_idx_t = std::remove_cvref_t<decltype(std::get<idx>(std::declval<Tuple>()))>;

template <typename Operand, typename... Derivative>
struct ExpressionOf<Operand, std::tuple<Derivative...>>{
  using Derivs = std::tuple<Derivative...>;
  using Operands = std::tuple<typename Derivative::Operand...>;

  template <typename Op = Operand> requires Contains<Op, Operands>::value
  constexpr static auto value() {
    return std::tuple<typename at_idx_t<get_idx<Op, Operands>(), Derivs>::Expression>{};
  }

  template <typename Op = Operand>
  constexpr static std::tuple<> value(){ return {}; }
};
```

Combining Systems - Expressions

```
template<std::size_t idx, typename Tuple>
using at_idx_t = std::remove_cvref_t<decltype(std::get<idx>(std::declval<Tuple>()))>;

template <typename Operand, typename... Derivative>
struct ExpressionOf<Operand, std::tuple<Derivative...>>{
  using Derivs = std::tuple<Derivative...>;
  using Operands = std::tuple<typename Derivative::Operand...>;

  template <typename Op = Operand> requires Contains<Op, Operands>::value
  constexpr static auto value() {
    return std::tuple<typename at_idx_t<get_idx<Op, Operands>(), Derivs>::Expression>{};
  }

  template <typename Op = Operand>
  constexpr static std::tuple<> value(){ return {}; }
};
```

Combining Systems - Expressions

```
template<std::size_t idx, typename Tuple>
using at_idx_t = std::remove_cvref_t<decltype(std::get<idx>(std::declval<Tuple>()))>;

template <typename Operand, typename... Derivative>
struct ExpressionOf<Operand, std::tuple<Derivative...>>{
  using Derivs = std::tuple<Derivative...>;
  using Operands = std::tuple<typename Derivative::Operand...>;

  template <typename Op = Operand> requires Contains<Op, Operands>::value
  constexpr static auto value() {
    return std::tuple<typename at_idx_t<get_idx<Op, Operands>(), Derivs>::Expression>{};
  }

  template <typename Op = Operand>
  constexpr static std::tuple<> value(){ return {}; }
};
```

Combining Systems - Expressions

```
template<typename Operand, tuple_like Derivatives>
constexpr auto expression_of_v = ExpressionOf<Operand, Derivatives>::value();

template <typename Operand, typename... DerivativeSystem>
constexpr auto combine_expressions_for()
{
  return combine_expressions(
    std::tuple_cat(expression_of_v<Operand, decltype(DerivativeSystem::make_dot())>...));
}

template <typename... Expression>
constexpr auto combine_expressions(std::tuple<Expression...> expr)
{
    return (std::get<Expression>(expr) + ...);
}
```

Combining Systems - Expressions

```
template<typename Operand, tuple_like Derivatives>
constexpr auto expression_of_v = ExpressionOf<Operand, Derivatives>::value();

template <typename Operand, typename... DerivativeSystem>
constexpr auto combine_expressions_for()
{
  return combine_expressions(
    std::tuple_cat(expression_of_v<Operand, decltype(DerivativeSystem::make_dot())>...));
}

template <typename... Expression>
constexpr auto combine_expressions(std::tuple<Expression...> expr)
{
    return (std::get<Expression>(expr) + ...);
}
```

Combining Systems - Expressions

```
template<typename Operand, tuple_like Derivatives>
constexpr auto expression_of_v = ExpressionOf<Operand, Derivatives>::value();

template <typename Operand, typename... DerivativeSystem>
constexpr auto combine_expressions_for()
{
  return combine_expressions(
    std::tuple_cat(expression_of_v<Operand, decltype(DerivativeSystem::make_dot())>...));
}

template <typename... Expression>
constexpr auto combine_expressions(std::tuple<Expression...> expr)
{
    return (std::get<Expression>(expr) + ...);
}
```

Combining Systems - Derivatives

```
template <typename... DerivativeSystem, typename... Operand>
constexpr auto combine_derivates_for(std::tuple<Operand...>)
{
    return std::make_tuple(
        dot<Operand>(combine_expressions_for<Operand, DerivativeSystem...>())...
      );
}
```

Combining Systems

```
template <typename... DerivativeSystem>
struct Combine
{
 using CombinedOperands =
         to_unique_tuple_t<tuple_cat_t<states_defined_by_t< DerivativeSystem >...>>;

 constexpr static auto make_dot()
 {
   return combine_derivates_for<DerivativeSystem...>(CombinedOperands{});
 }
};
```

Combining Systems

```
struct Motion2D {
 constexpr static auto make_dot() {/* ... */}
};
struct TurnablePropeller {
 constexpr static auto make_dot() {/* ... */}
};
```

```
using DenebMotion = Combine<Motion2D, TurnablePropeller>;
using DenebSystem = StateSpaceSystemOf<DenebMotion>;
```

Combining Systems

```
struct Motion2D {
 constexpr static auto make_dot() {/* ... */}
};
struct TurnablePropeller {
 constexpr static auto make_dot() {/* ... */}
};
struct StaticPropeller {
 constexpr static auto make_dot() {/* ... */}
};

using DenebMotion = Combine<Motion2D, TurnablePropeller, StaticPropeller>;
using DenebSystem = StateSpaceSystemOf<DenebMotion>;
```

Goals achieved?

- Prevent bugs through compiler checks
- Intuitively usable by mathematicians
- Maintainable

https://godbolt.org/z/PfqqWxPfT
(without units)

https://github.com/aber-code/codys