

+ 22

Taking Static Type-Safety to the Next Level

Physical Units for Matrices

DANIEL WITHOPF



20
22



What's in it for you?

Physical units in vectors and matrices*

If it compiles, it works!

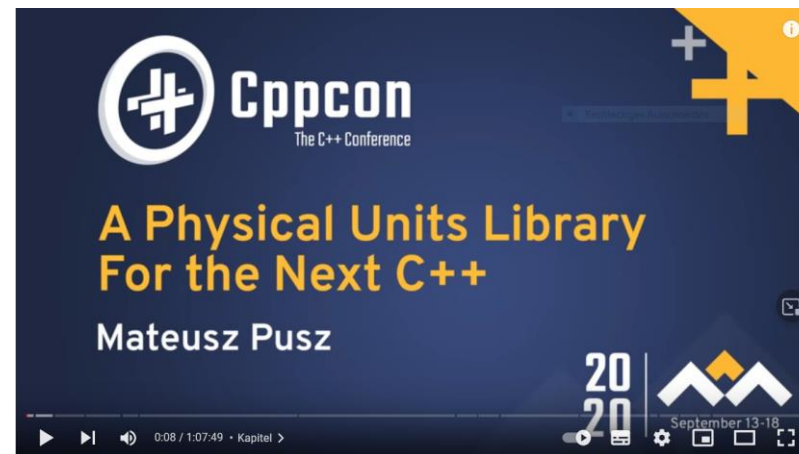
* Design principles

- Anticipate problematic usages and prevent them at compile-time
- Built-in static analyzer
- Enable users to make sweeping changes that work once they compile again
- Make user code as expressive as possible

Terminology

Physical units:

- Strong C++ types behaving as you would expect:
 $[m] + [m]$, $[m] + [s]$, $[m] * [s]$
- More info
 - <https://github.com/mpusz/units>
 - <http://wg21.link/P1935>



<https://www.youtube.com/watch?v=7dExYGSOJzo>

Linear Algebra	Math	TypeSafeMatrix
Vector	Point	VectorTag
	(Displacement) vector	DeltaVectorTag
Matrix	Matrix	...

Combining linear algebra with physical units

- Physical units: <https://github.com/mpusz/units> (<http://wg21.link/P1935>)
- Linear algebra: Eigen or <http://wg21.link/P1385>

```
si::length<si::metre> u = 3 * m;  
  
fs_vector<double, 3> v = { 1, 2, 3 };
```

- How can we combine these two libraries?

```
fs_vector<si::length<si::metre>, 3> v = { 1 * m, 2 * m, 3 * m };  
fs_vector<si::length<si::metre>, 3> u = { 3 * m, 2 * m, 1 * m };  
std::cout << "v + u = " << v + u << "\n"; // [4m, 4m, 4m]
```

A slightly more complex example

- Let's try to track a vehicle in 1 dimension
 - distance [m]
 - velocity [m/s]
 - acceleration [m/s²]

```
fs_vector<???, 3> v = { 1 * m, 2 * m / s, 3 * m / (s*s) };
```

- Which C++ data type can we use to represent this?
- `std::tuple` / `std::variant` to the rescue?

Unit structure in a matrix

- A 4x4 matrix can have up to 16 different physical unit types

$$\begin{array}{c|cccc}
 \text{Cov} & 0 & 1 & 2 & 3 \\
 \hline
 0 & u_{00} & u_{01} & u_{02} & u_{03} \\
 1 & u_{10} & u_{11} & u_{12} & u_{13} \\
 2 & u_{20} & u_{21} & u_{22} & u_{23} \\
 3 & u_{30} & u_{31} & u_{32} & u_{33}
 \end{array}$$

- Fortunately, the structure is simpler than this

$$\begin{array}{c|cccc}
 \text{Cov} & c_0 & c_1 & c_2 & c_3 \\
 \hline
 r_0 & r_0 c_0 & r_0 c_1 & r_0 c_2 & r_0 c_3 \\
 r_1 & r_1 c_0 & r_1 c_1 & r_1 c_2 & r_1 c_3 \\
 r_2 & r_2 c_0 & r_2 c_1 & r_2 c_2 & r_2 c_3 \\
 r_3 & r_3 c_0 & r_3 c_1 & r_3 c_2 & r_3 c_3
 \end{array}$$

Element access with classical linear algebra libraries

Can you tell what this line of code does?

```
vector_a[2] = vector_b[3];
```

- What do the vector entries describe?
- Is this an out-of-bounds access?

2nd try:

```
vector_a[VELOCITY_X] = vector_b[POSITION_X];
```

- Have the right index constants for the vector type been used?
- Is assigning a position to a velocity really intended?

3rd try:

```
vector_a[VELOCITY_X] = vector_b[VELOCITY_X];
```

- In which coordinate frame are vector_a and vector_b?

What we ideally want

Linear algebra library wishlist:

- Protection against out-of-bounds access (compile-time please!)
- Expressive (and enforced) names for vector / matrix entries
- Support for *non-uniform* physical units in vectors / matrices
- Physical units check for all matrix operations (compile-time)
- Coordinate frame annotation for vectors, matrices and transformations

Solution space

Named index structs

Identify each entry with a unique name: **DistanceX_C**

```
struct DX_C :  
  CartesianIdxType<VehicleFrontAxleCoords,  
    tsm::CartesianXAxis,  
    si::Metre>{};
```

← coordinate frame
← axis identifier
← si unit template

```
struct DY_SENSOR_C :  
  CartesianIdxType<SensorCoords,  
    tsm::CartesianYAxis,  
    si::Metre>{};
```

	Physical quantity	Axis	Coordinate frame
Distance	X	SENSOR	C
Distance	Y	SENSOR	C
Velocity	X	SENSOR	C
Velocity	Y	SENSOR	C
Acceler.	X	SENSOR	C
Acceler.	Y	SENSOR	C

```
struct VehicleFrontAxleCoords : public CoordinateSystem<si::Metre> {  
  using Moving = std::false_type; // is the frame moving wrt to the “fixed” earth frame?  
};
```

One type for almost everything

```
template<class Scalar, class RowIdxList, class ColIdxList, class MatrixTag>
class TypeSafeMatrix {
    ... // methods
private:
    Eigen::Matrix<Scalar, SizeOf<RowIdxList>::value, SizeOf<ColIdxList>::value> m_matrix;
};
```

```
template<class Scalar, class RowIdxList, class MatrixTag>
using TypeSafeVector = TypeSafeMatrix<Scalar,
                                     RowIdxList,
                                     tsm::TypeList<tsm::NoIdxType>,
                                     MatrixTag>;
```

```
template<class Scalar>
using PosVec3InVehicleFrame<Scalar> =
    tsm::TypeSafeVector<Scalar, tsm::TypeList<DX_C, DY_C, DZ_C>, tsm::VectorTag>;
```

Creating a vector and access to elements

```
PosVec3InVehicleFrame<double> position_vehicle{
    other_position.entry<DX_C>(),           // copy 1 entry from other vector (***)
    tsm::wrapCoeffSi<DY_C>(si::Metre<double>{1.}), // unit argument (**)
    tsm::wrapCoeff<DZ_C>(3.);               // plain scalar argument (*)
}
```

```
// full check, only the same index from the same vector type can be assigned
position_vehicle.assignEntry(other_position.entry<DX_C>()); // (***)
```

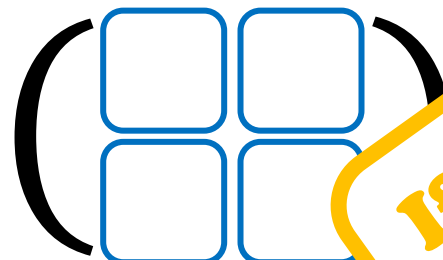
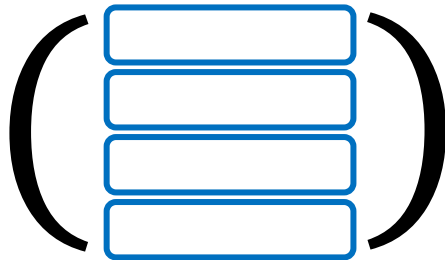
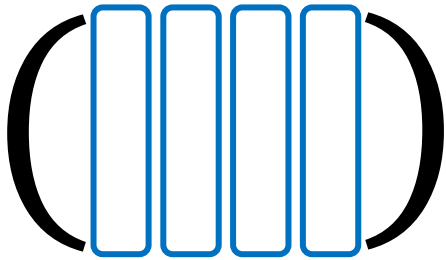
```
// unit check, allows assigning a y position from a different frame
position_sensor.coeffSiRef<DX_SENSOR_C>() = position_vehicle.coeffSi<DY_C>(); // (**)
```

```
// no checks except for scalar type, allows assigning a velocity to a position
position_vehicle.at<DX_C>() = velocity_sensor.at<VY_SENSOR_C>(); // (*)
```

More ways to create a vector / matrix

```
DeltaPosVec3InSensorFrame<double> delta_position_sensor{  
    si::Second<double>{2.0} * other_vel_sensor.head<2>(), // other 2d vector in same frame  
    other_delta_position.entry<DZ_SENSOR_C>()};
```

```
tsm::TypeSafeMatrix<double, tsm::TypeList<DX_C, VX_C>, tsm::TypeList<DX_C, VX_C>,  
    tsm::CovarianceMatrixTag> covariance{  
    tsm::wrapCoeffSi<DX_C, DX_C>{...}, tsm::wrapCoeffSi<DX_C, VX_C>{...},  
    tsm::wrapCoeffSi<VX_C, DX_C>{...}, tsm::wrapCoeffSi<VX_C, VX_C>{...}  
};
```



**If it compiles,
it works!**

Matrix element access

Declare a type-safe (jacobian) matrix

```
tsm::TypeSafeMatrix<double,  
    tsm::TypeList<MEAS_DR, // radial dist[m]  
                  MEAS_VR, // radial vel[m/s]  
                  MEAS_ANGLE>, // angle [rad]  
    tsm::TypeList<DX_C,    // distance x [m]  
                  DY_C,    // distance y [m]  
                  DZ_C>,   // distance z [m]  
    tsm::JacobianMatrixTag> jacobian{...};
```

Jacobian matrix	DX_C [m]	DY_C [m]	DZ_C [m]
MEAS_DR [m]			
MEAS_VR [m/s]			
MEAS_ANGLE [rad]			

Matrix element access

Which physical unit should be returned?

```
??? value = jacobian.coeffSi<MEAS_VR, DY_C>();
```

```
struct JacobianMatrixTag {  
    using RowExp = integral_constant<int, 1>;  
    using ColExp = integral_constant<int, -1>;  
};
```

Jacobian matrix	DX_C [m]	DY_C [m]	DZ_C [m]
MEAS_DR [m]			
MEAS_VR [m/s]		???	
MEAS_ANGLE [rad]			

Column exponent = -1

Row exponent = 1

Jacobian matrix	DX_C [m]	DY_C [m]	DZ_C [m]
[m]			
[m/s]		[m/s]^RowExp * [m]^ColExp	
[rad]			

Taxonomy of vector and matrix types

	Matrix / vector type	Row exponent	Column exponent	Nr of columns
Matrix types {	Covariance matrix	1	1	
	Jacobian matrix	1	-1	
	Information matrix	-1	-1	
Vector types {	Position vector (VectorTag)	1	0	1
	Displacement vector (DeltaVectorTag)	1	0	1
	Information vector	-1	0	1
Vector collections {	Position vector collection	1	0	>1
	Displacement vector collection	1	0	>1
	Information vector collection	-1	0	>1

*Now we have unit-safe
element access and out-of-bounds
protection*

Can we achieve even more???

Is unit-safety enough?

Let's try being unit-safe for all operations


Should this operation be allowed?

$$\begin{pmatrix} DX_C[m] \\ DY_C[m] \end{pmatrix} + \begin{pmatrix} VX_C[m/s] \\ VY_C[m/s] \end{pmatrix}$$

Should this be allowed?

$$\begin{pmatrix} [m] \\ [m] \end{pmatrix} + \Delta \begin{pmatrix} [m] \\ [m] \end{pmatrix}$$

Really?

 **Index types do not match!**

$$\begin{pmatrix} DX_C \\ DY_C \end{pmatrix} + \Delta \begin{pmatrix} DY_C \\ DZ_C \end{pmatrix}$$

Unit-safety is not enough,
we need index-type safety!



Matrix multiplication

```
cov_sensor = jacobian * cov_vehicle * jacobian.transpose();
```

1

$$\begin{matrix} \text{Jac} & \begin{matrix} & \begin{matrix} DX_C & DY_C & VX_C & VY_C \end{matrix} \\ \begin{matrix} DX_{SEN} \\ DX_{SEN} \\ VX_{SEN} \\ VY_{SEN} \end{matrix} & \begin{pmatrix} 1.0 & 0 & 0 & 0 \\ 0 & 1.0 & 0 & 0 \\ 0 & 0 & 1.0 & 0 \\ 0 & 0 & 0 & 1.0 \end{pmatrix} \end{matrix} \begin{matrix} \begin{matrix} -1 \\ 1 \end{matrix} \\ * \end{matrix} \begin{matrix} Cov & \begin{matrix} & \begin{matrix} DX_C & DY_C & VX_C & VY_C \end{matrix} \\ \begin{matrix} DX_C \\ DY_C \\ VX_C \\ VY_C \end{matrix} & \begin{pmatrix} 3.1 & 0 & 0 & 0 \\ 0 & 2.4 & 0 & 0 \\ 0 & 0 & 8.5 & 0 \\ 0 & 0 & 0 & 6.4 \end{pmatrix} \end{matrix} \begin{matrix} \begin{matrix} -1 \\ 1 \end{matrix} \\ * \end{matrix} \begin{matrix} Jac^T & \begin{matrix} & \begin{matrix} DX_{SEN} & DY_S & VX_S & VY_{SEN} \end{matrix} \\ \begin{matrix} DX_C \\ DY_C \\ VX_C \\ VY_C \end{matrix} & \begin{pmatrix} 1.0 & 0 & 0 & 0 \\ 0 & 1.0 & 0 & 0 \\ 0 & 0 & 1.0 & 0 \\ 0 & 0 & 0 & 1.0 \end{pmatrix} \end{matrix} \begin{matrix} \begin{matrix} 1 \\ 1 \end{matrix} \end{matrix}$$

What operator+ looks like

```
template<typename OtherLeaf_T>
auto operator+(const MatrixBase<OtherLeaf_T>& other) const
-> detail::MatrixExpression<detail::Promotion2dAddition<Leaf_T, OtherLeaf_T>,
                           decltype(this->underlying() + other.underlying())> {...}
```

```
template<typename Expr1, typename Expr2>
struct Promotion2dAddition
: RequiresIdenticalScalarType<Expr1, Expr2>,           // inject Scalar type
  RequiresIdenticalRowIndices<Expr1, Expr2>,           // inject RowIdxList type
  RequiresIdenticalColIndices<Expr1, Expr2>,           // inject ColIdxList type
  RequiresIdenticalRowUnitExponent<Expr1, Expr2>,
  RequiresIdenticalColUnitExponent<Expr1, Expr2>,
  RequiresMatrixTagsAdditionCompatible<typename Expr1::MatrixTag, // inject MatrixTag type
                                       typename Expr2::MatrixTag> {};
```

```
template<typename Expr1, typename Expr2>
struct RequiresIdenticalRowIndices {
    static_assert(detail::TypeIdentityChecker<typename Expr1::RowIdxList,
                                              typename Expr2::RowIdxList>::value,
                  "Row index types are not equal as required");
    using RowIdxList = typename Expr1::RowIdxList;
};
```

Compiler error messages

```
auto res = tsm::PosVector3InVehicleFrame{} + tsm::DeltaPosVector2InVehicleFrame{};
```

```
../type_safe_matrix/typelist_operations.h: In instantiation of 'class
detail::TypeIdentityChecker<TypeList<DX_C, DY_C, DZ_C>, TypeList<DX_C, DY_C> >':
../type_safe_matrix/promotion_precondition_checks.h:216:35:   required from 'class
detail::RequiresIdenticalRowIndices<TypeSafeMatrix<double, TypeList<DX_C, DY_C, DZ_C>,
TypeList<NoIdx>, VectorTag>, TypeSafeMatrix<double, TypeList<DX_C, DY_C>, TypeList<NoIdx>,
DeltaVectorTag> >'
../type_safe_matrix/typed_matrix_promotions.h:197:7:   required from 'class
detail::Promotion2dAddition<TypeSafeMatrix<double, TypeList<DX_C, DY_C, DZ_C>,
TypeList<NoIdx>, VectorTag>, TypeSafeMatrix<double, TypeList<DX_C, DY_C>, TypeList<NoIdx>,
DeltaVectorTag> >'
.....
../type_safe_matrix/test/usage_examples.cpp:505:67:   required from here
../type_safe_matrix/typelist_operations.h:66:3: error: static assertion failed: actual type
(1st template arg of TypeIdentityChecker) does not match desired type (2nd arg);
```

Compiler error message with C++20 concepts

```
auto res = tsm::PosVector3InSensorFrame{} + tsm::DeltaPosVector2InSensorFrame{};
```

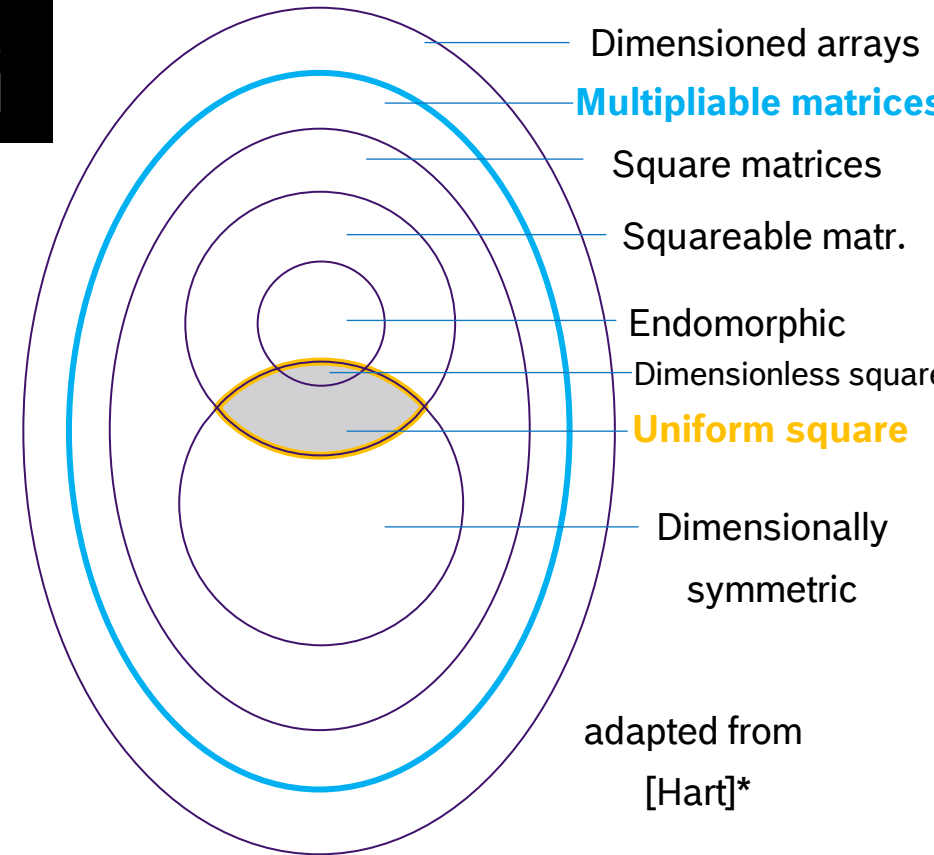
error: no match for 'operator+' (operand types are 'Vector3' {aka 'TypeSafeMatrix<double, TypeList<DX_C, DY_C, DZ_C>, TypeList<NoIdxType>, VectorTag>'} and 'Vector2' {aka 'TypeSafeMatrix<double, TypeList<DX_C, DY_C>, TypeList<NoIdxType>, VectorTag>'})

```
44 |     auto res = Vector3{} + Vector2{};
    |                  ~~~~~^~~~~~
    |                  |           |
    |                  |           TypeSafeMatrix<[...],TypeList<DX_C, DY_C>,[...],[...]>
    |                  |           TypeSafeMatrix<[...],TypeList<DX_C, DY_C, DZ_C>,[...],[...]>
33 |     TypeSafeMatrix operator+(const OtherT& other) requires Addable<TypeSafeMatrix,
    |     OtherT>                  ^~~~~~
<source>:33:19: note: constraints not satisfied
required for the satisfaction of 'Addable<TypeSafeMatrix<ScalarT, RowIdxListT, ColIdxListT,
MatrixTagT>, OtherT>'
note: nested requirement 'is_same_v<typename T1::RowIdxListType, typename
T2::RowIdxListType>' is not satisfied
```

The big picture

Uniform matrices vs. *Index-oriented design (TypeSafeMatrix)*

```
fs_vector<si::length<si::metre>, 3> v = {1*m, 2*m, 3*m};  
fs_vector<si::length<si::metre>, 3> u = {3*m, 2*m, 1*m};  
std::cout << "v + u = " << v + u << "\n"; // [4m,4m,4m]
```



*George W. Hart: *Multidimensional Analysis*, Springer

Uniform matrices vs. *Index-oriented design (TypeSafeMatrix)*

What can be added / multiplied?

Unit as value type

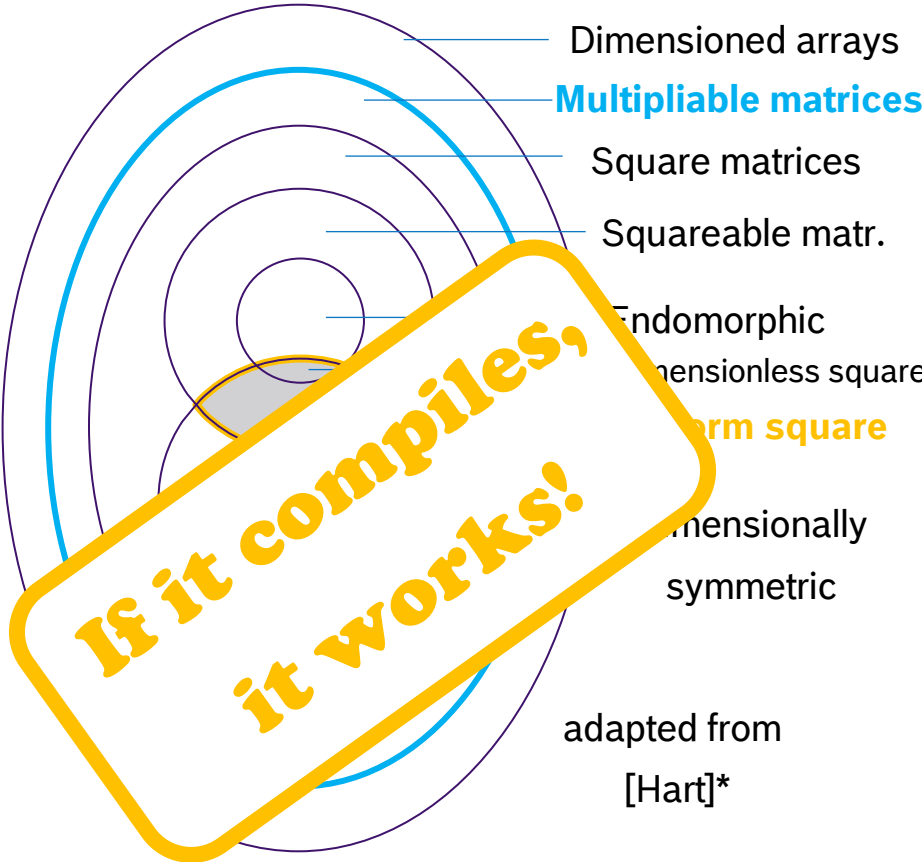
a + b	$\begin{pmatrix} DX \\ DY \end{pmatrix}$	$\begin{pmatrix} DY \\ DZ \end{pmatrix}$	$\Delta \begin{pmatrix} DX \\ DY \end{pmatrix}$	$\begin{pmatrix} VX \\ VY \end{pmatrix}$
$\begin{pmatrix} DX & DY \end{pmatrix}^T$	yes	yes	yes	no
$\begin{pmatrix} DY & DZ \end{pmatrix}^T$		yes	yes	no
$\Delta \begin{pmatrix} DX & DY \end{pmatrix}^T$			yes	no
$\begin{pmatrix} VX & VY \end{pmatrix}^T$				yes

A * B	jac	jac^T	cov	vector
jac	yes	yes	yes	yes
jac^T	yes	yes	yes	yes
cov	yes	yes	yes	yes
vector	yes	yes	yes	yes

TypeSafeMatrix

a + b	$\begin{pmatrix} DX \\ DY \end{pmatrix}$	$\begin{pmatrix} DY \\ DZ \end{pmatrix}$	$\Delta \begin{pmatrix} DX \\ DY \end{pmatrix}$	$\begin{pmatrix} VX \\ VY \end{pmatrix}$
$\begin{pmatrix} DX & DY \end{pmatrix}^T$	no	no	yes	no
$\begin{pmatrix} DY & DZ \end{pmatrix}^T$		no	no	no
$\Delta \begin{pmatrix} DX & DY \end{pmatrix}^T$			yes	no
$\begin{pmatrix} VX & VY \end{pmatrix}^T$				yes

A * B	jac	jac^T	cov	vector
jac	maybe	no	maybe	maybe
jac^T	no	maybe	no	no
cov	no	maybe	no	no
vector	no	no	no	no



adapted from
[Hart]*

*George W. Hart: *Multidimensional Analysis*, Springer

Transformation and rotation matrices

How to define an isometry (a transformation consisting of rotation and translation):

```
tsm::Isometry<double, tsm::VehicleFrontAxleCoords, tsm::OdomCoords, 3> vehicle_T_odom{...};
```

Scalar
type

Destination
frame

Source
frame

Dimension

Source
frame

This enables the compiler to check for correctness

```
// transform vector from odom to vehicle  
vehicle_position = vehicle_T_odom * odom_position;
```

Valid or not?

~~vehicle_velocity = vehicle_T_odom * odom_velocity;~~ **Velocities should not be translated => Compile error**

```
vehicle_velocity = vehicle_T_odom.linear() * odom_velocity;
```

```
vehicle_delta_vector = vehicle_T_odom.linear() * odom_delta_vector;
```

If it compiles, it works!

What are the most common bugs when using a C++ linear algebra library?

Expression templates and auto variables

```
// method returning by value  
Eigen::Vector3d calculateOffset() {...}  
Eigen::Vector3d vectorA = ...;
```

```
double result = (vectorA + calculateOffset()).norm();
```

```
auto sum = vectorA + calculateOffset();  
double result = sum.norm();
```

```
const auto& sum = vectorA + calculateOffset();  
double result = sum.norm();
```

How to prevent usage of MatrixExpression

```
template<class T>
void staticAssertIfLvalueMatrixExpression() {
    static_assert(!IsTsmMatrixExpression<std::decay_t<T>> || std::is_rvalue_reference<T>::value);
}

template<class Leaf>
class MatrixBase {
    template<class Other>
    auto operator+(Other&& other) const& -> detail::MatrixExpression<...> {
        detail::staticAssertIfMatrixExpression<Leaf>();
        detail::staticAssertIfLvalueMatrixExpression<decltype(other)>();
        return {underlying() + std::forward<Other>(other).underlying()};
    }
    template<class Other>
    auto operator+(Other&& other) const&& -> detail::MatrixExpression<...> {
        detail::staticAssertIfLvalueMatrixExpression<decltype(other)>();
        return {std::move(*this).underlying() + std::forward<Other>(other).underlying()};
    }
};
```

**If it compiles,
it works!**

What we have achieved

- Full support for physical units in vectors / matrices
 - Expressive (and enforced) names for vector / matrix entries
 - Protection against out-of-bounds access (compile-time)
- Proper taxonomy / domain model for linear algebra types
- Compatibility check of index structs and matrix types for linear algebra operations
- Built-in coordinate frame tagging
- Prevention of dangling reference bugs (and aliasing)
- **If it compiles, it works!**

Thank you for listening, looking forward to
your questions!

We at Bosch (Germany) <https://lnkd.in/d9w6pBVh>
are **hiring**