About

This project is basically somewhat of an answer to the age-old question:

> *Could we treat every step of a classical-mechanics simulation as nothing more than a lambda-calculus reduction?*

In other words: if **all** updates are pure functions, can we assemble them with ordinary function composition and still end up with a useful, reasonably fast physics engine (emphasis on reasonably)?

I decided to try the abstract this stuff in Haskell because Haskell is already “λ-calculus with types”. Everything you’ll see in the codebase follows the same pattern:

```
step :: Δt -> world -> world
```

where `world` is an immutable value and `step` is a lambda that rewrites that value. The final result was an EDSL that I call "Phynjo".

Before pointing out the limits of Haskell, it's worth looking at its strengths which can be summed up with four words: determinism, referential transparency and explicit domains.

Basically, identical inputs will yield identical outputs and a failing test can be rerun with the same seeds to produce byte-for-byte equal traces. This would, in theory, make debugging and profiling reproducible. We can also replace the expression `step dt w` by its value anywhere in the code. Here, algebraic reasoning is more valid and we have property-based testing such as `totalEnergy (step dt w) = totalEnergy w + O(dt²)`. Most importantly, though, we can define a numeric rule that modifies a set of components like so:

```
    NumericRule
      { nrDomain = Set.fromList [a, b]
      , nrStep   = kickForce3D …
      }
```

When two rules scheduled in parallel refer to overlapping domains, the interpreter will raise an error instead of chaotically overwriting data. This explicit conflict detection provides three key concurrency advantages:

1. **Race Prevention:** By rejecting overlapping writes upfront, we avoid non-deterministic state corruption that would require complex locking in imperative systems
2. **Compositional Safety:** Parallel rules become verifiable at compile-time – if domains don't overlap, execution is automatically thread-safe without synchronization
3. **Determinism Preserved:** The pure functional core ensures parallel evaluation of non-conflicting rules always yields the same result as sequential evaluation

Now, even with these pros, it's obviously not going to compete in terms of performance with languages like C/C++. For instance, this EDSL would probably not be helpful for applications that require that sub-

millisecond latency; applications that'd need haptic feedback, for instance, would need an outer loop that samples the immutable state and performs the required I/O. The core remains pure and a thin shell handles scheduling here.

And even though GHC manages the garbage values for each tick producing a fresh world value well, the allocation rate will never quite be the same as in a C engine that updates memory in place.

So, please note that performance isn't the goal here (though it would be convenient): its the semantic modeling of problems in classical physics. While we will be going through certain benchmarks, speed was not the priority in this project.

Classical Mechanics Overview

To understand the codebase properly, its important to get some analytical background on the domain. It might be helpful to think of physical systems as state machines where we update positions and momenta through discrete time steps.

It's worth noting that classical mechanics is a big topic and to properly understand it, you'll need to move beyond this (heavily) summarized pieces of texts. (Introduction to Mechanics by Kleppner and Kolenkow is a great resource).

For the purposes of this EDSL, we'll be focusing on the following three topics:

Simple Linear Dynamics

We can describe physical systems in three mathematically equivalent ways: Newtonian, Lagrangian and Hamiltonian. The idea behind Newtonian physics is that force causes a change in velocity (`F=ma` , the famous consequence of the second law). The equivalent equations would look something like:

$$p' = p + \Delta t \cdot F$$

$$q' = q + \Delta t \cdot \frac{p'}{m}$$

The idea is that force changes momentum (p) and momentum changes position (q).

On the other hand, the idea behind Lagrangian is that systems evolve to minimize "action" (kinetic - potential energy) and Hamiltonian works behind the usual $TE = PE + KE$ for a closed system; it's equivalent equations would look something like:

$$\frac{dq}{dt} = -\frac{\partial H}{\partial p} \quad \frac{dq}{dt} = \frac{\partial H}{\partial q}$$

Here, $H = TE = KE + PE$. This principle appears implicitly in our implementation when we perform energy checks.

Rotational and Rigid Body Dynamics

We can't model a spinning top the same way we'd model a falling apple. Rotation is the point where linear dynamics alone doesn't suffice. To properly describe such motion, we introduce the concept of *rigid bodies*, objects that do not deform under the forces they experience. Unlike a particle or point mass, a rigid body has both mass and shape, which means its motion involves both translation (like the

apple falling) and rotation (like the spinning top). Modeling rigid bodies allows us to account for how forces and torques affect not just the center of mass, but the orientation of the entire object.

Basically, when a force hits a rigid body, two things happen simultaneously:

1. The whole object moves in the direction of the force (like a particle)
2. The object starts spinning around its center of mass

So, to model it computationally we'll have to keep track of where the body is (COM position), how its oriented (state of rotation), how fast its moving (linear momentum) and how fast its spinning (angular momentum). You should be able to find rigid bodies defined accordingly like so:

```
data RigidBody = RB
  { position :: Vec3        -- Center of mass location
  , orientation :: Quaternion -- Rotation state
  , linearMomentum :: Vec3  -- Mass × velocity
  , angularMomentum :: Vec3 -- Rotational "spin"
  }
```

We use **quaternions** to represent orientation because they avoid several pitfalls of other representations like Euler angles and rotation matrices. Euler angles suffer from *gimbal lock*, a phenomenon where you lose a degree of freedom in rotation, making them unreliable for tracking complex motion. Rotation matrices, while accurate, are overkill in size (9 values for 3 degrees of freedom, really?), and are harder to keep orthonormal under numerical integration. Quaternions, on the other hand, are compact (just 4 numbers), efficient to compute with, and inherently avoid gimbal lock.

Collisions

Ever tried hitting two things together?

Newton's laws govern smooth motion, but collisions happen faster than our timestep Δt. We resolve this through impulse-momentum theory:

- **Impulse** (J) = $\int F \cdot dt$ (force integrated over collision time)
- Changes momentum instantaneously: Δp = J

For a particle:

$$m\mathbf{v}' - m\mathbf{v} = \mathbf{J}$$

For rigid bodies, we add rotational effects:

$$\begin{aligned} m\Delta\mathbf{v}_{\text{COM}} &= \mathbf{J} \\ \mathbf{I}\Delta\boldsymbol{\omega} &= \mathbf{r} \times \mathbf{J} \end{aligned}$$

where $\mathbf{r}$ is the vector from center-of-mass to contact point.

Newton's restitution law quantifies energy remaining after collision:

$$\mathbf{v}_{\text{rel}}^{\text{after}} \cdot \mathbf{n} = -\epsilon(\mathbf{v}_{\text{rel}}^{\text{before}} \cdot \mathbf{n})$$

- ϵ = coefficient of restitution ($0 \leq \epsilon \leq 1$)

- $\mathbf{n}$ = collision normal vector
- $\mathbf{v}_{\text{rel}}$ = relative velocity at contact point

We determine impulse magnitude by momentum conservation, restitution law and geometric constraints

For two bodies A and B:

$$J = \frac{-(1+\epsilon)\mathbf{v}_{\text{rel}} \cdot \mathbf{n}}{\frac{1}{m_A} + \frac{1}{m_B} + \mathbf{n} \cdot (\mathbf{I}_A^{-1}(\mathbf{r}_A \times \mathbf{n}) \times \mathbf{r}_A + \mathbf{n} \cdot (\mathbf{I}_B^{-1}(\mathbf{r}_B \times \mathbf{n}) \times \mathbf{r}_B}$$

The denominator represents **effective mass** or the resistance to impulse considering both translation and rotation.

When surfaces slide during impact, Coulomb friction applies:

$$\mathbf{J}_t = -\mu \|\mathbf{J}_n\| \hat{\mathbf{t}}$$

- μ = friction coefficient
- $\mathbf{t}$ = tangential direction

It's worth noting that friction impulse can't cause reversal of sliding direction

This leads to piecewise solutions:

$$\mathbf{J}_t = \begin{cases} -\mu \|\mathbf{J}_n\| \hat{\mathbf{t}} & \text{if sliding} \\ \text{solution to } \mathbf{v}_t' = 0 & \text{if sticking} \end{cases}$$

When multiple collisions occur at once (e.g., Newton's cradle), we face a **linear complementarity problem** (LCP):

$$\begin{aligned} 0 \leq \mathbf{J}_n \perp \mathbf{A}\mathbf{J} + \mathbf{b} \geq 0 \\ \mathbf{J}_t \in \text{Friction cone} \end{aligned}$$

where:

- $\mathbf{A}$ = "mass matrix" coupling all contacts
- $\mathbf{b}$ = pre-impact velocities
- $\perp$ means $\mathbf{J}_n^T(\mathbf{A}\mathbf{J} + \mathbf{b}) = 0$ (either impulse or separation)

This matrix formulation gives us null interpenetration while respecting friction constraints.

For objects in sustained contact (like a book on table), we transition to **constraint-based dynamics**:

$$\begin{aligned} \phi(\mathbf{q}) \geq 0 \quad &(\text{non-penetration}) \\ \nabla\phi \cdot \mathbf{v} \geq 0 \quad &(\text{no inward velocity}) \\ \lambda \geq 0 \quad &(\text{contact force magnitude}) \end{aligned}$$

with the complementarity condition:

$$\lambda(\nabla\phi \cdot \mathbf{v}) = 0$$

Also, Collisions break Hamiltonian structure:

$$\Delta T = -\frac{1}{2}(1-\epsilon^2)\frac{m_1 m_2}{m_1+m_2}(\mathbf{v}_{\text{rel}}\cdot\mathbf{n})^2 + \text{friction losses}$$

Data Structures

`Components.hs`

A **component** is a label that identifies a physical entity: an atom, a rigid body, or whatever else.

The type is intentionally recursive.

```
data Component
  = AtomicC   String          -- single body, e.g. "ball-7"
  | Composite [Component]     -- ordered list of sub-components
  deriving (Eq, Ord, Show)
```

- `children` returns the immediate sub-list.
- `subcomponents` flattens the entire tree.
- `arity` counts direct children and is later used to validate joint definitions.

`EventWorkflow.hs`

`EventWorkflow` is one of the modules for the Boolean layer. Each `EventRule` is like a small chip: it consumes the current flag set, possibly flips a bit or two, and possibly emits an event. A useful simulation, however, needs many chips working together. I considered dropping straight into `Action` composition, but that loses the semantic difference between “do these two rules **now**” and “do one, then on the next tick do the other”. The workflow datatype keeps that distinction explicit.

```
data EventWorkflow
  = ERun EventRule
  | ESeq EventWorkflow EventWorkflow
  | EPar EventWorkflow EventWorkflow
```

`ERun` is the trivial wrapper; `ESeq` and `EPar` are the thing.

Sequential composition is just function composition over time. If I have a state `(tick, flags)` and I give it to `w1`, the result becomes the input to `w2`. The interpreter does exactly that and nothing more; it does not go inside the rules.

It's worth noting that parallel composition needs a safety check: two branches must not update the same flag. That requirement is enforced at run time by computing the union of `erDomain`s. If the intersection is non-empty the interpreter throws.

`toProcess` lowers the workflow to the generic `Process` defined earlier. The translation is structural. `ERun` becomes `PAct (ruleToAction r)`, `ESeq` becomes `PSeq`, and so on. That means the only interpreter I need to maintain is `applyProcessWorld / Phen`, and Boolean workflows piggy-back on it automatically.

So, as an example, suppose I want a contact sensor that, once it sees the flag `onGround`, waits one tick and then clears it. I write two rules:

```
setRule    = … -- sets onGround when penetration detected
clearRule  = … -- clears onGround unconditionally
```

Then

```
sensorWF = ESeq (ERun setRule) (ERun clearRule)
```

gives me exactly one-tick hysteresis. If I add a second, unrelated sensor I can combine them safely:

```
fullWF = EPar sensorWF otherSensorWF
```

because their domains are disjoint.

`EventRule.hs`

The Boolean part of the language needed a truly atomic unit of behaviour: “look at a handful of flags right now, decide what those flags should be on the next tick, possibly announce that something happened.” That atomic unit is `EventRule`.

```
data EventRule = EventRule
  { erDomain :: Set Component
  , erStep   :: Literal -> Literal
  , erEvents :: Int -> Literal -> Phenomenon
  }
```

`erDomain` is the whole key to composability. If I write a rule for a door latch (`latchOpen`, `latchClosed`) I can guarantee no other rule touches those two flags by declaring them here. Later, when I run two rules in parallel, the interpreter checks that the domains are disjoint before it merges their results. Without this field I would end up silently overwriting someone else’s flag.

`erStep` is completely pure: given only the slice of flags in the domain it produces the slice for the next tick. Because it is pure, QuickCheck and unit tests can nail the behaviour down with no IO and no global state. I deliberately excluded the global tick from `erStep`; the tick lives in `erEvents` because it matters only for time-stamping events.

`erEvents` is also pure. Many rules don’t emit anything, so they just return `emptyPhen`. When a rule does emit, it receives the same domain slice and the integer tick and can build any `Phenomenon` it wants, typically via the helper `epsilon component tick`.

Embedding a rule into the engine is handled by `ruleToAction`. The function literally splits the full world literal into two pieces (`domain` and `outside`), runs `erStep` on the first, pastes the result back together with the second, and increments the tick. That is all. The phenomenon function is forwarded directly. After the conversion an `EventRule` behaves like any other `Action`, which means the generic `Process` interpreter can execute it without knowing anything about domains or events.

In the grand scheme of things, `EventRule` is a data record of two lambdas and one set, but it is the very important pivot that lets Boolean logic plug into the exact same interpreter infrastructure as the

numeric rules. That uniformity pays off every time I add a new subsystem: I implement a pure function, declare its domain, and the rest of the engine will schedule it correctly.

`NumericRule.hs`

`NumericRule` is the atom of the continuous half of the DSL. The idea makes the bold assumption that every bit of classical mechanics, no matter how complicated, can be written as a **pure function** that takes a timestep `dt` and an `NState` and returns a new `NState`. Once that function is wrapped in a record together with the set of bodies it may touch, the interpreter can schedule it alongside any other rule.

```
data NumericRule = NumericRule
  { nrDomain :: Set Component
  , nrStep   :: Double -> NState -> NState
  }
```

`nrDomain` enforces locality. If I compose two rules in parallel and they try to update the same momentum entry, the merge step in `NumericWorkflow` will detect the overlap and abort. That single set keeps the rest of the engine honest.

`nrStep` is kept completely unconstrained. A rule may read both `q` and `p`, update one or both, allocate new maps, or reuse them. The only promise is purity: the same inputs yield the same outputs. That design made unit testing straightforward; I can build a tiny `NState`, call the function, and check an exact result.

Writing a rule is therefore just writing a lambda. For example, the drift rule is

```
nrStep = \dt st ->
  foldr
    (\(c,m) acc ->
       let q0 = lookupPos c acc
           p0 = lookupMom c acc
       in insertPos c (q0 + dt * p0 / m) acc)
    st
    masses
```

The helper

```
applyRule :: NumericRule -> Double -> NState -> NState
applyRule NumericRule{nrStep} = nrStep
```

exists mostly for readability in tests. It hides the record selector and keeps call-sites short.

`NumericWorkflow.hs`

The Boolean side had `EventWorkflow`; I needed the same idea for the numeric layer, a data structure that lets me put together a handful of `NumericRule`s into a single stepper without writing a bespoke interpreter for every integrator. That became `NumericWorkflow`.

```
data NumericWorkflow
  = WRun NumericRule
  | WSeq NumericWorkflow NumericWorkflow
  | WPar NumericWorkflow NumericWorkflow
```

`WRun` is the leaf: “apply this one rule”. `WSeq a b` means “run `a`, then feed the result to `b`”, so it advances the world by *two* sub-steps. `WPar a b` runs both branches on the *same* input state and merges the outputs in a single sub-step provided those branches touch disjoint bodies.

The interpreter is about forty lines. The sequential case is trivial recursion. The parallel case is slightly subtle: both branches read the same `NState`, so they must write to disjoint momentum/position maps or the merge would be ambiguous. I check that with

```
let da = workflowDomain a
    db = workflowDomain b
in  if Set.null (da `Set.intersection` db)
        then mergeStates …
        else error "NumericWorkflow: parallel overlap"
```

`mergeStates` unions the two `SLit` maps; because the domains are disjoint, `Map.union` is safe.

With that interpreter in place the higher-level integrator functions become simple AST builders. The leap-frog constructor returns

```
WSeq (WRun driftHalf) (WSeq (WRun kickFull) (WRun driftHalf))
```

and Forest–Ruth is just `foldr1 WSeq (map mkSlice coeffs)`.

Literals

`ScalarLiteral` is the simplest container in the engine: a map from `Component` to a `Double`.

```
newtype SLit = SL (Map Component Double)
```

The choice looks mundane, but three small conventions keep later code short and predictable.

- **Sparsity** Positions, momenta, or coefficients may be undefined for many components. A missing key represents a mathematical zero. The helper

```
    lookupSL :: Component -> SLit -> Double    -- defaults to 0
```

converts that convention into code.

- **In-place updates** `insertSL` returns a new map with one entry replaced or added.
- **Incremental adjustment** Many rules need “add this increment unless the key is absent; if absent, start from a default”. That pattern appears often enough to justify a helper:

```
    adjustSL :: (Double -> Double)  -- update function
             -> Double              -- default if key not present
             -> Component
             -> SLit
             -> SLit
```

Physical quantities carry dimensions. Encoding them at the type level prevents accidental additions such as “metres plus kilograms”. `ULiteral` adds a phantom type parameter `u`:

```
newtype ULiteral u = ULit (Map Component (Quantity u Double))
```

`u` is drawn from `Numeric.Units.Dimensional` (e.g. `DLength`, `DMass`).

- **Phantom type** `u` is not used at runtime, only by the compiler. Two `ULiteral`s with different `u` cannot be combined without an explicit conversion.
- **Quantity** `Quantity u Double` is a value tagged with its dimension. The operator `(*~)` attaches a unit; `(/~)` strips it.

```
    x = 2.0 *~ metre        :: Quantity DLength Double
    v = 3.0 *~ (metre/second)
```

These units disappear after compilation; execution works with raw `Double`s.

`NState`

An n-body point-mass system is fully described by the canonical pair

$$(\mathbf{q}, \mathbf{p}) \;=\; \bigl(q_1, \ldots, q_n,\; p_1, \ldots, p_n\bigr), \qquad p_i \;=\; m_i \dot{q}_i,$$

where $q_i \in \mathbb{R}$ is a one-dimensional position, $m_i > 0$ is the mass, and pip_i is the conjugate momentum.

The engine stores that pair in an *immutable* record:

```
data NState = NS
  { q :: SLit          -- positions  qi   [m]
  , p :: SLit          -- momenta    pi   [kg·m/s]
  }
```

Here `SLit` is the sparse map.

Mathematically $q : \mathcal{C} \to \mathbb{R}$ is a partial function from the set of components $\mathcal{C}$ to reals; entries not present are implicitly zero. In code that convention is encoded in

```
lookupPos :: Component -> NState -> Double   -- returns 0 if missing
lookupMom :: Component -> NState -> Double   -- idem
```

A *drift* update is the discrete version of $\dot{q}_i = \partial H / \partial p_i = p_i / m_i$:

$$q_i^{\text{new}} = q_i^{\text{old}} + \Delta t \, \frac{p_i^{\text{old}}}{m_i}.$$

Translated:

```
insertPos c (oldQ + dt * oldP / m) st
```

Momentum kicks follow Hamilton's second equation $\dot{p}_i = -\partial H / \partial q_i$.

$$p_i^{\text{new}} = p_i^{\text{old}} + \Delta t \, F_i(\mathbf{q})$$

with one line:

```
adjustSL (+ Δp) 0 c (p st)
```

The CAS

I'd like to point out that the goal here was not to build a full computer-algebra system but to get *just enough* structure to derive Euler–Lagrange equations automatically.

The datatype is a straightforward expression tree:

```
data Expr
  = Var   String           -- x
  | Const Rational         -- 3,  7/5
  | Add   Expr Expr        -- u + v
  | Sub   Expr Expr        -- u - v
  | Mul   Expr Expr        -- u · v
  | Div   Expr Expr        -- u / v
  | Pow   Expr Expr        -- u ^ v
  | Neg   Expr             -- -u
  | Sin   Expr | Cos Expr | Tan Expr
  | Exp   Expr | Log Expr
  deriving (Eq, Ord, Show)
```

Infix helpers

```
(.+.) = Add   ; (.-.) = Sub
(.*.) = Mul   ; (./.) = Div
(.^.) = Pow
neg   = Neg
```

They shorten test cases:

```
x, y :: Expr
x = Var "x";  y = Var "y"
expr = x .+. neg (y .^. Const 2)
```

Operator precedences mirror the pretty printer so round-tripping through `pretty . parse` is stable.

Differenciation

The engine must turn a symbolic Lagrangian $L(q, \dot{q})$ into its Euler–Lagrange residual $\frac{\mathrm{d}}{\mathrm{d}t}(\partial L/\partial \dot{q}_i) - \partial L/\partial q_i$.

I needed a reliable derivative, so I had `CAS.Differentiate` encode the textbook rules directly in Haskell pattern-matching and relies on a few helper functions to keep expression size under control.

```
differentiate :: String   -- variable v
              -> Expr     -- expression f
              -> Expr     -- ∂f/∂v, already simplified
differentiate v expr = S.simplify (D.differentiate v expr)
```

`D.differentiate` is the raw derivative; the final call to `S.simplify`.

Construct	Rule implemented	Code Chunk
Variable	$\partial x/\partial v = 1$ if names match, 0 otherwise	`diff (Var x)`
Constant	derivative is 0	`diff (Const _) = Const 0`
Sum	linearity	`diff (Add a b) = Add (diff a) (diff b)`
Product	$\partial(uv) = u'v + uv'$	`diff (Mul a b)` after *flattening*
Quotient	standard quotient rule	`diff (Div a b)`
Power u^n with rational n	$nu^{n-1}u'$	`diff (Pow a (Const n))`
Power u^v general	$u^v\left(v' \ln u + vu'/u\right)$	`diff (Pow a b)` fallback
$-u$	sign preserved	`diff (Neg a) = Neg (diff a)`
Elementary functions	chain rule	`diff (Sin a)`, `Cos`, `Tan`, `Exp`, `Log`

The naïve product rule would expand $u_1u_2u_3$ into a sum of three triple products.

To avoid exponential blow-up `flattenMul` rewrites the operand tree into a flat list before applying the rule, so only *two* `Mul` nodes are introduced:

```
diff (Mul a b) =
  Add (stripOne (Mul (diff a) b))
      (stripOne (Mul a (diff b)))
```

`stripOne` removes multiplicative factors equal to 11. For commutative combinations of constants the simplifier pass combines them into a single rational.

Now you might be wondering: "oh but it doesn't really simplify complex equations to human readable format, does it?", and you're correct. Just don't tell anyone else about it.

Building the Physics EDSL

Vectors

`Physics.Integrators.LeapfrogNR` has two modules that recur in every 3-D example:

1. a minimal algebra on cartesian triples `Vec3`;
2. an adaptive velocity–Verlet (leap-frog) integrator for Newtonian gravity.

Both parts are fully independent of the rest of the engine, so they can be tested in isolation and compared against analytical or high-precision solutions.

The type alias is fixed throughout the project:

```
type Vec3 = (Double, Double, Double)   -- (x, y, z)
```

Although a record or small array would be marginally faster, the tuple keeps pattern-matching concise and avoids an external dependency.

Function	Formula
`vadd u v`	$u + v$
`vsub u v`	$u - v$
`vscale k v`	$k\,v$
`vdot u v`	$u \cdot v$
`vnorm2 v`	$\|v\|^2$

All are implemented with direct tuple pattern matches; GHC unboxes the intermediate doubles in optimised builds.

A gravitational acceleration onto body *i* is written

```
accOne g massMap pos i
```

with

$$\mathbf{a}_i = \sum_{j \neq i} \frac{G\, m_j\, (\mathbf{r}_j - \mathbf{r}_i)}{\left(\|\mathbf{r}_j - \mathbf{r}_i\|^2 + \varepsilon^2\right)^{3/2}}, \qquad \varepsilon = \texttt{soft} = 10^{-3}\,\mathrm{m},$$

using *Plummer softening* to avoid the singularity at zero separation.

Forces

I have a force generator that takes the current state of the system and returns a translation-space force ($\mathbf{F}$) plus, for rigid bodies, a rotation-space torque ($\boldsymbol{\tau}$). The **Forces DSL** packages several predefined generators, lets them be added and scaled algebraically, and finally converts them into

- a **one-dimensional** `NumericRule` that updates momenta pip_i in an `NState`, or
- a **three-dimensional** `Force3D` callback consumed by the rigid-body integrator.

```
type ForceField = NState -> Component -> Vec3

data Force
  = Gravity  Double
  | Spring   Component Component Double Double
  | Drag     Double
  | Custom   ForceField
```

- **Gravity** Constant acceleration $a_y = -g$. One-dimensional scenes treat g as a scalar $F = -m_i g$.
- **Spring** Hooke law between bodies *i* and *j* with stiffness k and rest length ℓ_0.

$$F_i = -k\,(|q_i - q_j| - \ell_0)\ \operatorname{sgn}(q_i - q_j), \qquad F_j = -F_i.$$

- **Drag** Linear damping $F = -\gamma v$.
- **Custom** Any user-supplied pure function from state to vector.

```
forceNR
  :: Force                  -- algebraic description
  -> [(Component,Double)]   -- (body, mass) table
  -> NumericRule            -- kick rule
```

For each body c with mass m_c

$$p_c' \;=\; p_c + h\,F_c(q).$$

`nrDomain` is the set of all listed components, so the rule refuses to run in parallel with any other numeric rule that touches the same bodies.

For the Newtonian potential

$$V(q) = -\sum_{i<j} \frac{G m_i m_j}{|q_i - q_j|},$$

the force on body *i* is

$$F_i(q) = \sum_{j \neq i} \frac{G\,m_i m_j}{(q_j - q_i)^2}\,\operatorname{sgn}(q_j - q_i).$$

`gravNR` pre-builds

- an `Int -> Double` mass lookup,
- an `Int` list of indices,

so each evaluation visits every pair once, giving $O(n^2)$ cost without allocation. The small Plummer softening used in 3-D is not required in 1-D.

```
newtype Force3D =
  Force3D { runForce3D ::
```

```
                RigidState -> Component -> (Vec3, Vec3) }
```

`runForce3D state c` returns

- $\mathbf{F}_c$ – translation-space force (N),
- $\boldsymbol{\tau}_c$ – world-space torque (N·m).

Provided constructors:

Name	$\mathbf{F}$	$\boldsymbol{\tau}$
`gravity3D g masses`	$(0, -mg, 0)$	$\mathbf{0}$
`spring3D i j k l0`	$\pm k(\lvert\mathbf{r}_i - \mathbf{r}_j\rvert - \ell_0)\hat{\mathbf{r}}_{ij}$	$\mathbf{0}$
`drag3D γ`	$-\gamma\mathbf{v}$	$\mathbf{0}$

A user may wrap any other effect in

```
Force3D (\state c -> (f state c, tau state c))
```

`kickForce3D` transforms these outputs into updates of linear velocity and world-space angular velocity using the body-space inertia tensor.

Given a `Force3D` field $(\mathbf{F}_c, \boldsymbol{\tau}_c) = \texttt{field}\ st\ c$ in world coordinates, the rune performs

1. **Linear momentum**

$$\mathbf{v}_c' = \mathbf{v}_c + \frac{h}{m_c}\,\mathbf{F}_c.$$

2. **Angular momentum**
   Express torque and angular velocity in body coordinates:

$$\tau_B = R(Q)^{\mathsf{T}}\boldsymbol{\tau}_W, \qquad \boldsymbol{\omega}_B = R(Q)^{\mathsf{T}}\boldsymbol{\omega}_W.$$

```
Euler's rigid-body equation
```

$$\dot{\boldsymbol{\omega}}_B = I^{-1}\big(\tau_B - \boldsymbol{\omega}_B \times (I\boldsymbol{\omega}_B)\big)$$

   is integrated by a first-order step:

$$\boldsymbol{\omega}_B' = \boldsymbol{\omega}_B + h\,\dot{\boldsymbol{\omega}}_B.$$

   Finally convert back to world frame

$$\boldsymbol{\omega}_W' = R(Q)\boldsymbol{\omega}_B'$$

Quaternion Mathematics

`RigidBody` and `RigidState`

```
data RigidBody = RigidBody
  { rbIdent   :: Component
  , rbMass    :: Double                -- kg
  , rbInertia :: InertiaTensor         -- body frame
  , rbPos0    :: Vec3                  -- m
  , rbOri0    :: Quaternion            -- unit (w,x,y,z)
  , rbVel0    :: Vec3                  -- m s⁻¹
  , rbAngVel0 :: Vec3                  -- rad s⁻¹, world frame
  }
```

- `rbInertia` is a 3×3 symmetric tensor expressed in the **body** coordinate frame that coincides with the principal axes at $t = 0$.
- `rbOri0` must satisfy $w^2 + x^2 + y^2 + z^2 = 1$.
- Velocities are world-space vectors. Converting them to the body frame requires the orientation matrix.

```
data RigidState = RigidState
  { rsPos    :: Map Component Vec3
  , rsOri    :: Map Component Quaternion
  , rsVel    :: Map Component Vec3
  , rsAngVel :: Map Component Vec3     -- world frame
  }
```

Default values used by the lookup helpers are

$$\mathbf{0} = (0, 0, 0), \qquad Q_{\text{unit}} = (1, 0, 0, 0).$$

`insertRigid` writes one component into all four maps at once, which gives us the maps that remain keyed by the same component set.

Quaternion to Rotation Matrix

For a unit quaternion $Q = (w, x, y, z)$ the active rotation matrix that transforms **body vectors into world vectors** is

$$R(Q) = \begin{pmatrix} w^2 + x^2 - y^2 - z^2 & 2(xy - wz) & 2(xz + wy) \\ 2(xy + wz) & w^2 - x^2 + y^2 - z^2 & 2(yz - wx) \\ 2(xz - wy) & 2(yz + wx) & w^2 - x^2 - y^2 + z^2 \end{pmatrix}.$$

`quatToMatrix` implements these nine polynomials directly:

```
ww = w*w; xx = x*x; yy = y*y; zz = z*z
wx = w*x; wy = w*y; wz = w*z
xy = x*y; xz = x*z; yz = y*z
```

and returns three row vectors packed in the `InertiaTensor` alias

```
type InertiaTensor = (Vec3,Vec3,Vec3)
```

The transpose is needed in `driftRot` when world-space angular velocity $\boldsymbol{\omega}_W$ must be expressed in body coordinates:

$$\boldsymbol{\omega}_B = R(Q)^{\mathsf{T}}\,\boldsymbol{\omega}_W.$$

Let $\boldsymbol{\omega} = (\omega_x, \omega_y, \omega_z)$ be angular velocity in **body coordinates**. The quaternion derivative is

$$\dot{Q} = \tfrac{1}{2}\,\Omega(\boldsymbol{\omega})\,Q, \qquad \Omega(\omega) = \begin{pmatrix} 0 & -\omega_x & -\omega_y & -\omega_z \\ \omega_x & 0 & \omega_z & -\omega_y \\ \omega_y & -\omega_z & 0 & \omega_x \\ \omega_z & \omega_y & -\omega_x & 0 \end{pmatrix}.$$

`integrateQuat` applies a first-order Euler step

$$Q' = Q + \tfrac{1}{2}\,\Delta t\,\Omega(\omega)\,Q$$

and renormalises

$$Q_{\text{new}} = \frac{Q'}{\|Q'\|}.$$

The renormalisation corrects numerical drift of $\|Q\| - 1$ but breaks perfect symplecticity.
Energy growth measured later on is \mathcal $O(\Delta t^3)$, acceptable for the time steps used.

```
mw = -qx*ωx - qy*ωy - qz*ωz
mx =  qw*ωx + qy*ωz - qz*ωy
my =  qw*ωy - qx*ωz + qz*ωx
mz =  qw*ωz + qx*ωy - qy*ωx

half = 0.5 * dt
qw'  = qw + half * mw
qx'  = qx + half * mx
qy'  = qy + half * my
qz'  = qz + half * mz
norm = sqrt (qw'*qw' + qx'*qx' + qy'*qy' + qz'*qz')
```

`(qw'/norm, qx'/norm, qy'/norm, qz'/norm)` is returned.

A **rune** is the rigid–body counterpart of a `NumericRule` : a pure function that rewrites a `RigidState` inside its declared component set. `Physics.RigidBodyUtilities.Rigid3DNR` defines three runes:

Symbol in code	Math	Order	Symplectic
`driftTrans`	$(\mathbf{r}, \mathbf{p}) \mapsto (\mathbf{r} + \frac{h}{m}\mathbf{p}, \mathbf{p})$	second	Yes
`driftRot`	$Q \mapsto \text{integrateQuat}(h, \boldsymbol{\omega}_B, Q)$	first	Yes (rotation-only)
`kickForce3D`	Euler equation update for $\mathbf{p}, \boldsymbol{\omega}$	first	Yes (impulse)

Each rune advertises a `domainR :: Set Component` ; the interpreter will refuse to compose two runes in parallel when their domains overlap.

Basic Contact

Sphere on Ground Plane `contactGroundF`

Assume the plane is $y = 0$ with upward normal $\mathbf{n} = (0, 1, 0)$. For body c with centre $p = (x, y, z)$ and radius r

$$d = r - y.$$

Penetration occurs when $d > 0$. Current relative velocity at the contact point

$$\mathbf{v}_{\text{rel}} = \mathbf{v} + \boldsymbol{\omega} \times (0, -r, 0).$$

Normal component $v_n = \mathbf{v}_{\text{rel}} \cdot \mathbf{n}$.

Effective inverse mass

$$K^{-1} = \frac{1}{m} + (\,\mathbf{r} \times \mathbf{n}\,) \cdot \big(I^{-1}(\mathbf{r} \times \mathbf{n})\big), \qquad \mathbf{r} = (0, -r, 0).$$

Impulse magnitude

$$j_n = \begin{cases} -(1 + e)\, v_n / K^{-1}, & v_n < 0, \\ 0, & v_n \geq 0 \end{cases}$$

where $e = e(v_n)$ is a user-supplied restitution curve (`eFun`).

Tangential velocity $\mathbf{v}_t = \mathbf{v}_{\text{rel}} - v_n \mathbf{n}$. If $\|\mathbf{v}_t\| > 0$,

$$j_t^{\text{slide}} = -\frac{\|\mathbf{v}_t\|}{K_t^{-1}}, \qquad K_t^{-1} = \frac{1}{m} + (\mathbf{r} \times \mathbf{t}) \cdot I^{-1}(\mathbf{r} \times \mathbf{t}),$$

with direction $\mathbf{t} = \mathbf{v}_t / \|\mathbf{v}_t\|$.

Coulomb limit $|j_t| \leq \mu\, |j_n|$; the final impulse is

$$j_t = \text{clip}\big(j_t^{\text{slide}},\ -\mu |j_n|,\ \mu |j_n|\big),$$

where $\mu = \mu(\|\mathbf{v}_t\|)$ is another user curve (`μFun`).

$$\Delta\mathbf{v} = \frac{1}{m}\,(\,j_n \mathbf{n} + j_t \mathbf{t}\,), \Delta\boldsymbol{\omega} = I^{-1}\big(\mathbf{r} \times (\,j_n \mathbf{n} + j_t \mathbf{t}\,)\big).$$

The rune accumulates these Δ contributions for all bodies then vehicles `M.unionWith vadd` .

Depth d is reduced by

$$\Delta\mathbf{r} = \beta\, \max(d - \text{slop},\, 0)\, \mathbf{n},$$

with $\beta = 0.2$ and $\text{slop} = 1$ cm hard-coded. This term is added directly to `rsPos` .

Sphere–sphere solver `contactSpheresF`

Handles N identical or non-identical spheres. The algorithm is **Jacobi**: iterate over all unordered pairs, compute impulses as if other pairs were frozen, accumulate $\Delta\mathbf{v}$ and $\Delta\boldsymbol{\omega}$ in maps, and after one pass add the totals to the state. Repeat `it` times.

For bodies $1, 2$ with world centres $\mathbf{p}_1, \mathbf{p}_2$, radii r_1, r_2,

$$\mathbf{d} = \mathbf{p}_1 - \mathbf{p}_2, \qquad d = \|\mathbf{d}\|, \qquad \mathbf{n} = \mathbf{d}/d, \qquad \text{penetration} = r_1 + r_2 - d.$$

Contact exists when penetration > 0.

Relative velocity at the contact point

$$\mathbf{v}_{\text{rel}} = \mathbf{v}_1 + \boldsymbol{\omega}_1 \times (-r_1\mathbf{n}) - \mathbf{v}_2 - \boldsymbol{\omega}_2 \times (+r_2\mathbf{n}).$$

From here the impulse calculation copies the plane case, replacing $\mathbf{r}$ by $\pm r_{1,2}\mathbf{n}$ and using individual masses and inertia tensors.

The parameter `it` chooses Jacobi passes. Five to ten passes resolve most penetrations for up to 10^3 spheres. Higher iteration counts increase accuracy at $O(N^2)$ cost; convergence is linear, so doubling passes almost halves the residual overlap.

Baumgarte term splits the depth proportionally to inverse masses:

$$\begin{aligned}\Delta\mathbf{p}_1 &= \quad \beta d_{\text{corr}} \frac{m_2}{m_1 + m_2}\,\mathbf{n},\\ \Delta\mathbf{p}_2 &= -\beta d_{\text{corr}} \frac{m_1}{m_1 + m_2}\,\mathbf{n}.\end{aligned}$$

`contactSpheresF` accumulates these shifts in a map and adds the total once per iteration.

Pure velocity impulses remove worsening penetration but cannot *recover* depth already present at time t. Baumgarte correction augments the position by a small fraction of the depth at every step:

$$\mathbf{p} \leftarrow \mathbf{p} + \beta\,\max(d - \text{slop}, 0)\,\mathbf{n}, \qquad 0 < \beta < 1.$$

- $\beta = 0$ no correction, bodies can interpenetrate and stay stuck.
- $\beta \approx 0.1$–0.3 depth decays geometrically without visible jitter.
- $\beta \to 1$ aggressive correction, may introduce energy gain.

The constant d_{slop} prevents micro-oscillations when penetration is below a user-chosen tolerance (here 1 cm).

Collision Modeling

Bounding Volumes

Collision detection starts with a **broad phase** that prunes obviously non-intersecting object pairs. The filter is based on *bounding volumes*—simple shapes that enclose the physical geometry but are fast to test for overlap. `Physics.Collision.Types` defines three variants:

Abbreviation	Parameters	Shape
AABB	min corner, max corner	axis-aligned box
SphereBB	centre, radius	sphere
OBB	centre, half-sizes, quaternion rotation	oriented box

Each bounding volume implements containment and intersection tests; the broad phase combines them into sweep-and-prune or uniform-grid algorithms.

Shape records

```
type ComponentBB = (Component, BoundingVolume)

data BoundingVolume
  = BB_AABB   AABB
  | BB_Sphere SphereBB
  | BB_OBB    OBB
  deriving (Eq, Show)
```

Axis-aligned bounding box `AABB`

```
data AABB = AABB
  { aMin :: Vec3    -- (xmin,ymin,zmin)
  , aMax :: Vec3    -- (xmax,ymax,zmax)
  } deriving (Eq, Show)
```

All faces align with the world axes. Updating an AABB requires only a component-wise `min` / `max` over points of the enclosed object.

Bounding sphere `SphereBB`

```
data SphereBB = SphereBB
  { sCenter :: Vec3
  , sRadius :: Double
  } deriving (Eq, Show)
```

Spheres are rotation-invariant and cheap to intersect; they are often used as a first culling stage in molecular or particle simulations.

Oriented bounding box `OBB`

```
data OBB = OBB
  { oCenter    :: Vec3
  , oHalfSizes :: Vec3                  -- (hx,hy,hz)
  , oRotation  :: Quaternion            -- world←body rotation
  } deriving (Eq, Show)
```

Anyway, we'll be moving on to the test between each collision type, starting with AABB-AABB.

An OBB is a rectangular prism in a local frame, rotated into the world frame by `oRotation`. The half-sizes h_x, h_y, h_z define the extent along the local axes.

Two axis-aligned boxes intersect iff they overlap on **all three** axes:

$$x_1^{\max} \geq x_2^{\min} \wedge x_2^{\max} \geq x_1^{\min}$$
$$y_1^{\max} \geq y_2^{\min} \wedge y_2^{\max} \geq y_1^{\min}$$
$$z_1^{\max} \geq z_2^{\min} \wedge z_2^{\max} \geq z_1^{\min}$$

```
intersectsAABB :: AABB -> AABB -> Bool
```

It should be obvious that all six comparisons are independent; branch prediction is favourable because each clause exits early on the first separating axis.

And the sphere-sphere

Overlap implies that squared centre distance $\leq (r_1 + r_2)^2$ (and vice versa):

```
intersectsSphere (SphereBB c1 r1) (SphereBB c2 r2) =
  vdot d d <= (r1+r2) * (r1+r2)
  where d = vsub c1 c2
```

`vdot` and `vsub` are reused from the leapfrog thing we built.

- `aabbUnion a b` returns the minimal AABB containing both boxes by component-wise `min` / `max`. Used in sweep-and-prune when merging child volumes in a BVH.
- `aabbFromSphere s` converts a sphere into a conservatively enclosing AABB. This is required by the uniform-grid broad phase, which stores only AABBs for fast cell indexing.

```
aabbFromSphere (SphereBB (cx,cy,cz) r) =
  AABB (cx-r, cy-r, cz-r) (cx+r, cy+r, cz+r)
```

Any scenario beyond this involves me doing more work, which is obviously outrageous, so we will pretend like these are the only scenarios (don't tell anyone).

Broad Phase Algorithms

Let $B = \{B_1, \ldots, B_N\}$be the current set of **bounding volumes** (AABBs or spheres). The broad phase builds a subset $\mathcal{P} \subseteq \{(i,j) \mid 1 \leq i < j \leq N\}$ that satisfies

$$(B_i, B_j) \notin \mathcal{P} \implies B_i \cap B_j = \varnothing.$$

The smaller $\mathcal{P}$ is, the less work the narrow phase performs. Two complementary filters are implemented.

Sweep-and-Prune

For an axis x define the interval

$$I_{x,i} = [\,\underline{x}_i,\ \overline{x}_i\,], \qquad \underline{x}_i = \min\{x\text{-coord of } B_i\},\ \overline{x}_i = \max\{\ldots\}.$$

Write each interval as two tagged endpoints $E = (i,\ \underline{x}_i,\ \text{low})$ and $E = (i,\ \overline{x}_i,\ \text{high})$. Sorting the $2N$ endpoints gives an ordered list $E_1, \ldots, E_{2N}$ with non-decreasing coordinate.

During a left-to-right scan keep an **active set** $A \subseteq \{1, \dots, N\}$ containing indices whose *low* endpoint has appeared but *high* endpoint has not yet been processed. When a *low* endpoint $E = (k, \underline{x}_k, \text{low})$ is encountered, k must overlap on the xx-axis with **every** $j \in A$. All pairs $(k, j) : j \in A\{(k, j) : j \in A\}$ are added to the candidate set $\mathcal{P}_x$. A *high* endpoint simply removes k from A.

Uniform Grid

Fix a cell size $s > 0$. Map a point (x, y, z) to the integer **cell coordinate**

$$\lfloor x/s \rfloor, \quad \lfloor y/s \rfloor, \quad \lfloor z/s \rfloor.$$

For B_i with component-wise extrema $(x_{\min}, y_{\min}, z_{\min})$, $(x_{\max}, y_{\max}, z_{\max})$ the set of covered cells is the Cartesian product

$$\left[\lfloor x_{\min}/s \rfloor,\ \lfloor x_{\max}/s \rfloor\right] \times \left[\lfloor y_{\min}/s \rfloor,\ \lfloor y_{\max}/s \rfloor\right] \times \left[\lfloor z_{\min}/s \rfloor,\ \lfloor z_{\max}/s \rfloor\right],$$

a rectangle with

$$n_i = (\lfloor x_{\max}/s \rfloor - \lfloor x_{\min}/s \rfloor + 1)\,(\lfloor y_{\max}/s \rfloor - \lfloor y_{\min}/s \rfloor + 1)\,(\lfloor z_{\max}/s \rfloor - \lfloor z_{\min}/s \rfloor + 1)$$

cells. For spheres $n_i = 1$ when $2r \leq s$.

Two hash maps are maintained:

map	key	value
σ	component id	list of occupied cells
γ	cell coord	list of components in cell

Building these maps is $O\left(\sum_{i=1}^{N} n_i\right)$.

For enumerating candidates, for each component i

1. obtain its cell list $\sigma(i)$;
2. collect $\bigcup_{c \in \sigma(i)} \gamma(c)$;
3. remove i and eliminate duplicates;
4. filter with `intersectsAABB`.

If bodies have diameter $d \ll s$ and are distributed with spatial density ρ (Poisson assumption), the expected cell occupancy is $\lambda = \rho s^3$ and the expected number of neighbour checks per body is λ. Total expected work: $O(N\lambda)$.

`Collision.Manager` façade

```
buildManager   :: BroadPhase -> [ComponentBB] -> CollisionManager
updateManager  :: CollisionManager -> [ComponentBB] -> CollisionManager
runBroadPhase  :: CollisionManager -> Set (Component,Component)
```

The façade allows experiments such as

```
cm0 = buildManager (BP_Grid 0.25) initialBBs
cm1 = updateManager cm0 frameBBs
pairs = runBroadPhase cm1           -- feed to narrow phase
```

without touching downstream collision code.

Narrow Phase Algorithms

`Collision.NarrowPhase` → `RRune`

Let

$$\mathcal{P} = \{(a, b)\} \subseteq \mathcal{C} \times \mathcal{C}, \qquad a < b$$

be the unordered candidate pairs from the broad phase. For each component cc a *shape function* $\text{shape} : \mathcal{C} \to \{\text{sphere}, \text{plane}\}$ is obtained from the `(Component,Shape)` table.

The current code retains **only sphere ⊗ sphere pairs**:

$$\widetilde{\mathcal{P}} = \{(a, b) \in \mathcal{P} \mid \text{shape}(a) = \text{shape}(b) = \text{sphere}\}.$$

All planes are handled separately by composing the ground-plane rune `contactGroundF`.

Sphere ⊗ Sphere impulse

Consider one pair $(1, 2)$ with masses m_1, m_2, body-space inertia tensors I_{B1}, I_{B2} (diagonal if principal-axis aligned at rest), world-space orientations Q_1, Q_2, radii r_1, r_2.

For penetration and depth contact frame,

$$\mathbf{d} = \mathbf{p}_1 - \mathbf{p}_2, \qquad d = \|\mathbf{d}\|, \qquad \mathbf{n} = \frac{\mathbf{d}}{d}, \qquad \delta = r_1 + r_2 - d > 0.$$

```
p1    = rsPos M.! c1
p2    = rsPos M.! c2
d     = vsub p1 p2
dist  = vnorm d
rSum  = radMap M.! c1 + radMap M.! c2
n     = vscale (1/dist) d
pen   = rSum - dist
```

Contact points in world space

$$\mathbf{c}_1 = \mathbf{p}_1 - r_1\mathbf{n}, \qquad \mathbf{c}_2 = \mathbf{p}_2 + r_2\mathbf{n}.$$

Lever arms

$$\mathbf{r}_1 = \mathbf{c}_1 - \mathbf{p}_1 = -r_1\mathbf{n}, \qquad \mathbf{r}_2 = \mathbf{c}_2 - \mathbf{p}_2 = +r_2\mathbf{n}.$$

```
r1 = vscale (-radMap M.! c1) n
r2 = vscale ( radMap M.! c2) n
```

And the velocity at contact is,

$$\mathbf{v}_c = (\mathbf{v}_1 + \boldsymbol{\omega}_1 \times \mathbf{r}_1) - (\mathbf{v}_2 + \boldsymbol{\omega}_2 \times \mathbf{r}_2).$$

Split into normal/tangential

$$v_n = \mathbf{v}_c \cdot \mathbf{n}, \qquad \mathbf{v}_t = \mathbf{v}_c - v_n \mathbf{n}, \qquad v_t = \|\mathbf{v}_t\|.$$

```
vRel1 = vadd (rsVel M.! c1) (cross (rsAngVel M.! c1) r1)
vRel2 = vadd (rsVel M.! c2) (cross (rsAngVel M.! c2) r2)
vRel  = vsub vRel1 vRel2
vn    = vdot vRel n
vt    = vsub vRel (vscale vn n)
vtMag = vnorm vt
```

I defined the *effective mass* (inverse of the constraint matrix) along a generic direction $\mathbf{u}$ as

$$K^{-1}(\mathbf{u}) = \frac{1}{m_1} + \frac{1}{m_2} + (\mathbf{r}_1 \times \mathbf{u})^\top R_1 I_{B1}^{-1} R_1^\top (\mathbf{r}_1 \times \mathbf{u}) + (\mathbf{r}_2 \times \mathbf{u})^\top R_2 I_{B2}^{-1} R_2^\top (\mathbf{r}_2 \times \mathbf{u}),$$

where $R_i = \texttt{quatToMatrix}(Q_i)$.

For the **normal axis** set $\mathbf{u} = \mathbf{n}$; for **tangential** use a unit vector $\mathbf{t} = \mathbf{v}_t / v_t$ (choose any orthonormal complement when $v_t = 0$).

We define the *effective mass* (inverse of the constraint matrix) along a generic direction $\mathbf{u}$ as

$$K^{-1}(\mathbf{u}) = \frac{1}{m_1} + \frac{1}{m_2} + (\mathbf{r}_1 \times \mathbf{u})^\top R_1 I_{B1}^{-1} R_1^\top (\mathbf{r}_1 \times \mathbf{u}) + (\mathbf{r}_2 \times \mathbf{u})^\top R_2 I_{B2}^{-1} R_2^\top (\mathbf{r}_2 \times \mathbf{u}),$$

where $R_i = \texttt{quatToMatrix}(Q_i)$.

For the **normal axis** set $\mathbf{u} = \mathbf{n}$; for **tangential** use a unit vector $\mathbf{t} = \mathbf{v}_t / v_t$ (choose any orthonormal complement when $v_t = 0$).

```
termN1 = vdot n (applyMat (invIMap M.! c1) (cross r1 n))
termN2 = vdot n (applyMat (invIMap M.! c2) (cross r2 n))
invKn  = invM1 + invM2 + termN1 + termN2

tDir = if vtMag < 1e-9 then (0,0,0) else vscale (1/vtMag) vt
termT1 = vdot (cross r1 tDir) (applyMat (invIMap M.! c1) (cross r1 tDir))
termT2 = vdot (cross r2 tDir) (applyMat (invIMap M.! c2) (cross r2 tDir))
invKt = invM1 + invM2 + termT1 + termT2
```

Then, for the normal impulse:

For coefficient of restitution $e = e(v_n) \in [0, 1]$.

$$j_n = \begin{cases} 0, & v_n \geq 0, \\ -(1+e)\,\dfrac{v_n}{K^{-1}(\mathbf{n})}, & v_n < 0. \end{cases}$$

Impulse vector $\mathbf{J}_n = j_n\,\mathbf{n}$.

Then, the coulomb-friction impulse:

Maximum tangential magnitude $j_t^{\max} = \mu(v_t)\,|j_n|$. Sliding impulse (no limit)

$$j_t^{\text{slide}} = -\frac{v_t}{K^{-1}(\mathbf{t})}.$$

Final scalar

$$j_t = \operatorname{clip}\big(j_t^{\text{slide}},\, -j_t^{\max},\, j_t^{\max}\big), \qquad \mathbf{J}_t = j_t\,\mathbf{t}.$$

```
jn | vn < 0 && invKn > 0 = -(1 + e) * vn / invKn
   | otherwise           = 0

jtMax = mu * abs jn
jtSlide | invKt > 0 = -vtMag / invKt
        | otherwise = 0
jt = max (-jtMax) (min jtSlide jtMax)

impN = vscale jn n
impT = vscale jt tDir
imp1 = vadd impN impT
imp2 = vscale (-1) imp1
```

We have the velocity update:

$$\begin{aligned}
\Delta\mathbf{v}_1 &= +\tfrac{1}{m_1}(\mathbf{J}_n + \mathbf{J}_t),\\
\Delta\boldsymbol{\omega}_1 &= +I_1^{-1}\big(\mathbf{r}_1 \times (\mathbf{J}_n + \mathbf{J}_t)\big),\\
\Delta\mathbf{v}_2 &= -\tfrac{1}{m_2}(\mathbf{J}_n + \mathbf{J}_t),\\
\Delta\boldsymbol{\omega}_2 &= -I_2^{-1}\big(\mathbf{r}_2 \times (\mathbf{J}_n + \mathbf{J}_t)\big).
\end{aligned}$$

Here I_i are world tensors $I_i = R_i I_{Bi} R_i^\top$.

These increments are *accumulated* during the Jacobi sweep, then merged into `RigidState` after every iteration.

```
dv1 = vscale invM1 imp1
dw1 = applyMat (invIMap M.! c1) (cross r1 imp1)

dv2 = vscale invM2 imp2
dw2 = applyMat (invIMap M.! c2) (cross r2 imp2)
```

Then, for the convergence of the jacobi scheme:

Define the contact constraint function

$$C(\mathbf{p}_1, \mathbf{p}_2) = \delta = r_1 + r_2 - d \geq 0.$$

The impulse update is equivalent to one step of Projected Gauss–Jacobi on the linearised complementarity problem

$$\mathbf{J}K^{-1}\mathbf{J}^\top\lambda = -b, \qquad \lambda \geq 0,$$

where $\lambda = (j_n, j_t)$.

```
iterate impulsePass (zeroMap, zeroMap) !! it

foldl' solver (dvAcc,dwAcc) [(c1,c2) | (c1:c2:_) <- tails comps]
```

For frictionless contacts the matrix is 2×2 positive definite, and the spectral radius of the Jacobi preconditioner is < 1 whenever $K^{-1} > 0$; therefore convergence is linear. Empirically, 8 – 10 iterations reduce the residual penetration below $10^{-4}\,\mathrm{m}$ for $N \le 10^{3}$ randomly packed spheres.