GPU Constraint Solver - Part 1

This post contains my derivation notes for the Jacobi iteration of a system of spherical rigid bodies constrained together in a chain by "point" constraints (ie. ball joints) - ie. a pearl necklace :).

I've been working on a little framework project for comparing different GPU based physics solvers, and this is meant to be a solver/scene which can function as a baseline by which to compare the stability and performance of various solvers. I've called this post "Part 1", because I intend to build on the basis which I'm introducing here.

I'm sticking with a specific, very simple constraint type for the scope of this derivation for concreteness, and so that it can immediately map to an actual simulation, although the concepts can be generalized to other equality constraint types.

The result: 2.5k chains of 40 beads (100k beads and constraints total) at 60fps on a RTX4060.

Glossary of Terms

Derivation

State Definitions

We will start by defining our terms. In a system with N_b rigid bodies, the ith body will have a position x_i and quaternion rotation q_i. Its linear and angular velocities are $\dot{x}_i$ and ω_i.

The configuration and velocity states of the system are

$$X = \begin{bmatrix} x_0 & q_0 & x_1 & q_1 & ... & x_{N_b-1} & q_{N_b-1} \end{bmatrix}^T \quad (1)$$

$$V = \begin{bmatrix} \dot{x}_0 & \omega_0 & \dot{x}_1 & \omega_1 & ... & \dot{x}_{N_b-1} & \omega_{N_b-1} \end{bmatrix}^T \quad (2)$$

$X \in \mathbb{R}^{7N_b}$ uses unit quaternions for rotation (7 components per body). $V \in \mathbb{R}^{6N_b}$ is the generalized velocity (6 components per body), using angular velocity ω_i rather than $\dot{q}_i$. Note that $V \neq \dot{X}$ - they live in spaces of different dimension. V is an element of the tangent space of the constraint configuration, not of $\mathbb{R}^{7N_b}$. The relationship between $\dot{q}_i$ and ω_i is

$$\dot{q}_i = \tfrac{1}{2} q_i \otimes (0, \omega_i)$$

where ω_i is embedded as a pure quaternion. This is used for position integration but does not appear in the constraint solve.

Point Constraint Definition

To simplify the constraint expression we want to construct, we'll define $R(q)$ to be the operation which rotates a vector by the unit quaternion q.

Remember, we're trying to make a "point constraint" - that is, we want to constrain specific points on two objects to always be collocated. Let's say there are N_c constraints. The ℓth constraint references two bodies via indices i_ℓ and j_ℓ, and "pins" their local-space attachment points r^0_ℓ and r^1_ℓ together.

$$c_\ell(x_{i_\ell}, q_{i_\ell}, x_{j_\ell}, q_{j_\ell}) = x_{i_\ell} - x_{j_\ell} + R(q_{i_\ell}) r^0_\ell - R(q_{j_\ell}) r^1_\ell \quad (3)$$

$$= 0 \quad (4)$$

Then the full constraint vector of the system is the stack of all c_ℓ

$$C(X) = \begin{bmatrix} c_0 & c_1 & ... & c_{N_c-1} \end{bmatrix}^T = 0$$

We define the *constraint Jacobian* J, a $3N_c \times 6N_b$ matrix, by the relation

$$\dot{C} = JV \quad (1)$$

Note that $J \neq \frac{\partial C}{\partial X}$ in the strict sense: since $X \in \mathbb{R}^{7N_b}$ and $V \in \mathbb{R}^{6N_b}$ have different dimensions, no such equality is possible. Instead, we obtain J by differentiating C directly with respect to time and reading off the coefficients of V, which gives us a gradient in the tangent space of the constraint configuration.

If we did want J to be a Jacobian of the constraint function in a stricter sense, we could add a constraint per body which enforces normalization of the quaternion state variables, and treat the quaternion's fourth value as another degree of freedom with its own corresponding time derivative. This would be a very interesting exercise and probably carries its own advantages, but the obvious major disadvantage is that it adds an extra unknown and an additional constraint *per body*, which could increase computation - not worth it for the improved aesthetic of the derivation.

Since $C(X) = 0$ must hold for all time, $\dot{C} = 0$. Combining this with (1), the velocity-level constraint is

$$JV = 0 \tag{2}$$

J is sparse - every block of 3 rows corresponds to one constraint, and the ℓth block J_ℓ is a $3 \times 6N_b$ matrix with only 12 nonzero columns (the 6-DOF blocks for bodies i_ℓ and j_ℓ). We compute those columns by differentiating c_ℓ with respect to time and reading off the velocity coefficients.

$$\begin{aligned}\dot{c}_\ell &= \dot{x}_{i_\ell} - \dot{x}_{j_\ell} + \omega_{i_\ell} \times R(q_{i_\ell}) r_\ell^0 - \omega_{j_\ell} \times R(q_{j_\ell}) r_\ell^1 && (5)\\ &= \dot{x}_{i_\ell} - \dot{x}_{j_\ell} - R(q_{i_\ell}) r_\ell^0 \times \omega_{i_\ell} + R(q_{j_\ell}) r_\ell^1 \times \omega_{j_\ell} && (6)\\ & && (7)\end{aligned}$$

From this, we can split out the velocities and infer the constraint Jacobian

$$\begin{aligned}\dot{c}_\ell &= \begin{bmatrix} I_3 & -\left[R(q_{i_\ell}) r_\ell^0\right]_\times & -I_3 & \left[R(q_{j_\ell}) r_\ell^1\right]_\times \end{bmatrix} \begin{bmatrix} \dot{x}_{i_\ell} \\ \omega_{i_\ell} \\ \dot{x}_{j_\ell} \\ \omega_{j_\ell} \end{bmatrix} && (8)\\ &= J_\ell V_\ell && (9)\end{aligned}$$

where $V_\ell = \begin{bmatrix} \dot{x}_{i_\ell}^T & \omega_{i_\ell}^T & \dot{x}_{j_\ell}^T & \omega_{j_\ell}^T \end{bmatrix}^T \in \mathbb{R}^{12}$ is the velocity state of the two bodies involved in constraint ℓ, in the same interleaved order as the global V. Also, I'm using the syntax $[v]_\times$ to represent the cross-product matrix, defined by $[v]_\times u = v \times u$.

We can infer the Jacobian

$$J_\ell = \begin{bmatrix} I_3 & -\left[R(q_{i_\ell}) r_\ell^0\right]_\times & -I_3 & \left[R(q_{j_\ell}) r_\ell^1\right]_\times \end{bmatrix}$$

Equations of Motion

Recalling the Lagrangian L as

$$\begin{aligned}L &= K - U - \lambda C(X) && (10)\\ &= \frac{1}{2} V^T M V - U(X) - \lambda C(X) && (11)\end{aligned}$$

where $M \in \mathbb{R}^{6N_b \times 6N_b}$ is the block-diagonal generalized mass matrix, with blocks $m_i I_3$ for linear inertia and the world-space inertia tensor I_i for angular inertia.

Since J is defined in the tangent space of the constraint configuration (rather than as $\frac{\partial C}{\partial X}$ literally), the constraint force term $\left(\frac{\partial C}{\partial X}\right)^T \lambda$ in the Euler-Lagrange equation is equivalent to $J^T \lambda$ when both are evaluated in this tangent space. The constrained Euler-Lagrange equation is then

$$\begin{aligned}0 &= \frac{d}{dt} \frac{\partial L}{\partial V} - \frac{\partial L}{\partial X} && (12)\\ &= M\dot{V} - F + \left(\frac{\partial C}{\partial X}\right)^T \lambda && (13)\\ &= M\dot{V} - F + J^T \lambda && (14)\end{aligned}$$

where $F = -\frac{\partial U}{\partial X}$ is the total force on the system.

Discretizing Time

We now discretize the equations of motion over the time domain by introducing h, the "delta time", and let n denote the current timestep index. The acceleration of the system $\dot{V}$ then becomes

$$\dot{V} = (V_{n+1} - V_n)\, h^{-1}$$

We treat force integration and constraint solving as separate passes by defining the free velocity $V^* = V_n + hM^{-1}F$, i.e. the velocity after applying forces but before enforcing constraints. When F does not depend on velocity, this splitting introduces no additional error beyond the first-order time discretization already present.

Then the equation of motion becomes a system of $6N_b$ equations with $6N_b + 3N_c$ unknowns. The unknowns are $V_{n+1} \in \mathbb{R}^{6N_b}$, the velocity state, and $\lambda \in \mathbb{R}^{3N_c}$, the constraint multipliers (3 per constraint, one per scalar constraint equation).

$$MV_{n+1} + hJ^T\lambda = MV^* \tag{3}$$

We close the system with the discretized velocity constraint

$$JV_{n+1} = 0 \tag{4}$$

Equations (3) and (4) give $6N_b + 3N_c$ equations in $6N_b + 3N_c$ unknowns. This system includes the constraint at the velocity level only - it does not directly enforce $C(X) = 0$, so position-level drift can accumulate over time.

We'll solve (3) for V_{n+1}, and substitute the result into (4), and rearrange to find λ. The result can then be plugged back into (3) to solve for V_{n+1}.

First, solve (3) for V_{n+1}.

$$MV_{n+1} + hJ^T\lambda = MV^* \tag{15}$$
$$V_{n+1} = V^* - hM^{-1}J^T\lambda \tag{5}$$

Now, plug this into (4) and rearrange to get a linear system in λ.

$$JV_{n+1} = 0 \tag{16}$$
$$JV^* - hJM^{-1}J^T\lambda = 0 \tag{17}$$
$$JM^{-1}J^T\lambda = h^{-1}JV^* \tag{18}$$
$$\tag{19}$$

To simplify things, we define A and b

$$A = JM^{-1}J^T \tag{20}$$
$$b = h^{-1}JV^* \tag{6}$$

so that our system becomes

$$A\lambda = b \tag{21}$$
$$\tag{22}$$

The matrix A is known as the *Delassus matrix* and represents the compliance of the system in constraint space - how much constraint velocity changes in response to a unit constraint impulse. It is effectively the inverse inertia (ie. inverse mass), but in constraint space.

The Jacobi Step

I'll rely on the Jacobi method for solving the linear systems. This is a good choice as a starting point because it is well suited for parallelization, and I am targeting GPU simulation. I should note, however, that I'm aware that this method of solving linear systems converges about half as quickly as Gauss Seidel, but I'll save that for a later optimization since for parallel execution it requires constraint graph coloring machinery.

The Jacobi method begins by breaking A into its block-diagonal and off-block-diagonal parts, D and E, where D consists of the 3×3 diagonal blocks $A_{\ell\ell}$.

$$A = D + E \tag{23}$$

We then plug this into the original equation and rearrange so that λ appears on both sides, but so that it is alone on one side and so that the only inverse is of the diagonal matrix on the other side.

$$(D + E)\,\lambda = b \tag{24}$$
$$D\lambda = b - E\lambda \tag{25}$$
$$\lambda = D^{-1}\,(b - E\lambda) \tag{26}$$

This is just a rearrangement of our original equation, but we can interpret it as a recurrence relation. That is, the λ on the *left* side is the "new" value and the λ on the *right* side is the "previous". At iteration k, this recurrence relation looks like

$$\lambda^{k+1} = D^{-1}\,(b - E\lambda^k) \tag{27}$$
$$\tag{28}$$

Now, we eliminate E by substituting back in our equation for A.

$$\lambda^{k+1} = D^{-1}\,(b - (A - D)\,\lambda^k) \tag{29}$$
$$= D^{-1}\,(b - A\lambda^k) + \lambda^k \tag{30}$$

The innermost expression $b - A\lambda^k$ is known as the *residual*. Intuitively, b is the constraint violation of the free velocity and $A\lambda^k$ is the correction the current impulse estimate would produce, so the residual is the remaining violation that λ^k has not yet accounted for - the iteration converges when it reaches zero.

Using Sparsity for Parallelism

Since each c_ℓ is a 3D vector equation (a difference between two positions), A has dimensions $3N_c \times 3N_c$ with 3×3 blocks. A is generally sparse, containing nonzero blocks only where constraints share a body. We now devise an update for the ℓth constraint's Lagrange multiplier λ_ℓ, which is a 3-vector. Each λ_ℓ can be updated independently.

The diagonal blocks of D are $A_{\ell\ell} = J_\ell M^{-1} J_\ell^T$, where $A_{\ell\ell}$ are 3×3 matrices. Although J_ℓ and M^{-1} both depend on orientation and change each timestep, they are fixed for the duration of a single solve. So $A_{\ell\ell}^{-1}$ can be precomputed once per timestep, before the iteration loop begins.

Note that J_ℓ is technically a $3 \times 6N_b$ matrix, where we typically only store the 12 of the $6N_b$ columns which are nonzero, the 6-DOF blocks for bodies i_ℓ and j_ℓ.

We track the velocity estimate across iterations, writing V^k for the velocity implied by the current λ^k, initialized as $V^0 = V^*$.

Recalling the definitions for A and b, the per-constraint update is then

$$\lambda_\ell^0 = 0 \tag{31}$$

$$\lambda_\ell^{k+1} = \lambda_\ell^k + A_{\ell\ell}^{-1}(b_\ell - (A\lambda^k)_\ell) \tag{32}$$

$$= \lambda_\ell^k + h^{-1} A_{\ell\ell}^{-1} J_\ell V^k \tag{33}$$

where $V^k = V^* - hM^{-1}J^T\lambda^k$ comes from (5). The second line can be verified by substituting the definitions of A, b, and V^k directly. Since J_ℓ is nonzero only for bodies i_ℓ and j_ℓ, each thread only needs to read the velocities of those two bodies.

To then get the next estimate for body velocities, we need to express (5) in terms of per-constraint λ_ℓ, and substitute in our new value λ_ℓ^{k+1}. We use $J^T\lambda = \sum_\ell J_\ell^T \lambda_\ell$ to obtain the velocity iteration

$$V^{k+1} = V^* - hM^{-1} \sum_\ell J_\ell^T \lambda_\ell^{k+1} \tag{34}$$

We can see that V^k requires summing impulse contributions from all constraints touching each body. But if we break down the summation to obtain a *per-body* velocity update, we can see from the sparsity of J_ℓ that the velocity change on body i depends only on constraint ℓ if $i = i_\ell$ or $i = j_\ell$. In the case of our exemplary chain, each body is referred to in at most two constraints, so this summation is cheap.

Therefore, the Jacobi iteration splits naturally into two kernel passes, initializing with $V^0 = V^*$:

1. **Lambda pass** (one thread per constraint): read V^k for bodies i_ℓ and j_ℓ, produce λ_ℓ^{k+1}
2. **Velocity pass** (one thread per body): accumulate V_i^{k+1}, given λ_ℓ^{k+1}

Connectivity

The velocity pass only needs to sum over constraints touching each body, not all constraints. To make this efficient, we precompute a flat array of constraint indices sorted by body, so that each body's connected constraints are contiguous. Each body stores a single integer offset marking where its section begins. Each constraint appears twice - once per body it connects. This structure is built once per topological change (not per frame).

Numeric Drift

The derivation up to this point is technically correct. Now say we go through the long process of coding it all up, testing every step to make sure there were no mistakes, and finally building a scene with a simple chain of spheres. This is what we see:

Close but not quite right! Note the paragraph immediately after equation (4). We've got a solid physical basis for the derivation of equations which enforce $\dot{C}(X) = 0$, but *not* $C(X) = 0$. This means that over time, drift is introduced by imperfect floating point calculations at each step.

There are a number of ways to address this. In real-time sims, the most common solution is called Baumgarte stabilization. Thankfully it's very simple to implement and to understand intuitively. However, it is not a physically based method - while it does typically improve numeric accuracy, it can also inject kinetic energy into the system and cause instabilities.

The idea comes from control theory. In reality, $C(X) \neq 0$ because constraint error is introduced over time. To control for it, we can introduce a term to the constraint velocity which opposes violation. With $\beta > 0$, we have

$$\dot{C}(X) = -\beta C(X) \tag{7}$$

This differential equation has the solution $C(t) = C(0)e^{-\beta t}$, which goes to zero smoothly over time.

We now carry (7) through each step of the derivation. This starts with equation (4), where we close the sys-

tem of equations of motion. We come to a system identical to (6), but with a new term for b.

$$A\lambda = b \tag{35}$$
$$A = JM^{-1}J^T \tag{36}$$
$$b = h^{-1}\left(JV^* + \beta C(X)\right) \tag{8}$$

This trickles down to the per-constraint update to the multiplier.

$$\lambda_\ell^{k+1} = \lambda_\ell^k + h^{-1}A_{\ell\ell}^{-1}\left(J_\ell V^k + \beta C_\ell(X)\right) \tag{37}$$

It's a simple change to make, but one question remains: since β is not a real physical parameter, what value should we assign to it? In section 4.2 of Erin Catto's famous paper, he derives bounds of $[0, 2]h^{-1}$ for convergence and $[0, 1]h^{-1}$ for smooth convergence. In practice, values in the range $[0.1, 0.3]h^{-1}$ are common.

Finally, with this update, we have a much better looking chain simulation:

Algorithm

Let's begin by defining our data. We have the following structs and buffers. Note that even though the Jacobian should be a 3x12 matrix, we only store two 3x3 matrices. This is because in the way that we have formulated the point constraint, the two 3x3 matrices corresponding to positions are always just the identity matrix, and can be dropped entirely from our representation.

```
struct Jacobian
{
    mat3 w0;
    mat3 w1;
};

// per body data
int ic0[NB + 1]; // index to the first element
 ↪  in the constraint list; ic0[NB] is a
 ↪  sentinel equal to NC*2
vec3 pos[NB];
quat rot[NB];
vec3 lin_vel[NB];
vec3 ang_vel[NB];
vec3 lin_vel_delta[NB]; // the working lin vel
 ↪  delta during jacobi iterations
vec3 ang_vel_delta[NB]; // the working ang vel
 ↪  delta during jacobi iterations
float lin_mass[NB];
float lin_mass_inv[NB];
vec3 lin_acc[NB]; // linear acceleration from
 ↪  external forces (e.g. gravity)
vec3 ang_acc[NB]; // angular acceleration from
 ↪  external torques
mat3 ang_mass_local[NB];
mat3 ang_mass[NB];
mat3 ang_mass_inv[NB];

// per constraint data
int ib0[NC]; // index of body 0
int ib1[NC]; // index of body 1
vec3 r0[NC]; // offset in body 0 local space
vec3 r1[NC]; // offset in body 1 local space
vec3 multiplier[NC];
Jacobian jacobian[NC];
vec3 violations[NC]; // constraint violation
 ↪  values, ie. C(X)
mat3 delassus_inv[NC];
int constraint_list[NC * 2]; // constraint
 ↪  indices, ordered by body
```

At a high level, the algorithm to do a time step looks like this:

```
void update(float dt) {

    // recompute the constraint topology
     ↪  mappings
    // this is NOT a parallel algorithm, and
     ↪  should run once per topological update.
    if (/* topology has changed */) {
        compute_connectivity();
    }

    // update world space rotational inertia
     ↪  tensors
    parallel_for (int ib : body_indices)
        compute_inertia(ib);

    // integrate forces to get "free" velocities
    parallel_for (int ib : body_indices)
        compute_free_velocity(ib, dt);

    // update constraint violation, Jacobian
     ↪  matrix, and Delassus matrix
    parallel_for (int ic : constraint_indices)

 ↪  compute_violation_and_jacobian_and_delassus(ic);

    // note: at this point we could zero out the
     ↪  multipliers before iterating,
    // but convergence will generally improve if
     ↪  we "warm-start" the system
    // with the values from the previous frame.

    // begin Jacobi iterations
    for (int iter = 0; iter < NI; ++iter) {

        // update lagrange multipliers (gather)
```

```
        parallel_for (int ic :
↪   constraint_indices)
            update_multipliers(ic, dt);

        // update velocities (scatter)
        parallel_for (int ib : body_indices)
            update_velocities(ib, dt);
    }

    // integrate velocities
    parallel_for (int ib : body_indices)
        compute_positions(ib);
}
```

Next, I'll walk through the sequential connectivity builder, which runs only when there's a topological change.

```
void compute_connectivity() {

    // clear the number of constraints per body.
    // this is a scratch array which we'll reuse
    int ncs[NB];
    for (int ib = 0; ib < NB; ++ib)
        ncs[ib] = 0;

    // increment constraint count for each body
     ↪  in each constraint
    for (int ic = 0; ic < NC; ++ic) {
        ++ncs[ib0[ic]];
        ++ncs[ib1[ic]];
    }

    // compute index of first constraint in each
     ↪  body's constraint list.
    // the first constraint for each body will
     ↪  be one after the last constraint
    // of the previous body.
    ic0[0] = 0;
    for (int ib = 1; ib <= NB; ++ib)
        ic0[ib] = ic0[ib-1] + ncs[ib-1]; //
↪   ic0[NB] = NC*2 serves as a sentinel

    // clear the constraint per body count -
     ↪  we'll reuse this now
    for (int ib = 0; ib < NB; ++ib)
        ncs[ib] = 0;

    // fill constraint lists
    for (int ic = 0; ic < NC; ++ic) {
        constraint_list[ic0[ib0[ic]] +
↪   (ncs[ib0[ic]]++)] = ic;
        constraint_list[ic0[ib1[ic]] +
↪   (ncs[ib1[ic]]++)] = ic;
    }
}
```

Now we'll break down each step which runs as a compute kernel

```
void compute_inertia(int ib) {

    // if geometry changed, update lin_mass and
     ↪  ang_mass
    //...

    // update inverse mass
    lin_mass_inv[ib] = 1.f / lin_mass[ib];

    // update global rotational inertia tensor:
     ↪  I_world = R * I_body * R^T
    mat3 R = to_mat3(rot[ib]);
    ang_mass[ib] = R * ang_mass_local[ib] *
↪   transpose(R);
    ang_mass_inv[ib] = inverse(ang_mass[ib]);
}

void compute_free_velocity(int ib, float dt) {

    // these integrations are separable, and it
     ↪  may be desirable
    // to swap out different methods of
     ↪  integration for each in the
    // case that forces are position-dependent
     ↪  or require higher
    // precision. For example, we may want an
     ↪  integrator which does
    // not fully lose the nonlinear rotation
     ↪  terms.
    lin_vel[ib] += lin_integrate(lin_acc[ib],
↪   dt);
    ang_vel[ib] += ang_integrate(ang_acc[ib],
↪   dt);

    // clear the velocity deltas, in preparation
     ↪  for jacobi iterations
    lin_vel_delta[ib] = vec3(0.0f);
    ang_vel_delta[ib] = vec3(0.0f);
}

void
↪   compute_violation_and_jacobian_and_delassus(int
↪   ic) {

    // compute world space body points
    int b0 = ib0[ic];
    int b1 = ib1[ic];
    vec3 r0 = to_mat3(rot[b0]) * r0[ic]
    vec3 r1 = to_mat3(rot[b1]) * r1[ic]

    // compute current constraint violation
    violations[ic] = pos[b0] - pos[b1] + r0 -
↪   r1;

    // set up the Jacobian matrix for this
     ↪  constraint in the current orientations
    mat3 jw0 = -cross_mat(r0);
    mat3 jw1 =  cross_mat(r1);
    jacobian[ic].w0 = jw0;
    jacobian[ic].w1 = jw1;
```

```
    // set up the Delassus matrix A_ll = J_l
     ↪ M^-1 J_l^T and store its inverse.
    // the angular terms are jw * I^-1 * jw^T =
     ↪ -jw * I^-1 * jw since jw^T = -jw
    // (jw is skew-symmetric), so they are
     ↪ subtracted.
    mat3 A_ll
        = mat3(lin_mass_inv[ib0[ic]] +
         ↪ lin_mass_inv[ib1[ic]])
        - (jw0 * ang_mass_inv[ib0[ic]] * jw0)
        - (jw1 * ang_mass_inv[ib1[ic]] * jw1);
    delassus_inv[ic] = inverse(A_ll);
}

void update_multipliers(int ic, float dt) {

    // get current body velocities
    int b0 = ib0[ic];
    int b1 = ib1[ic];
    vec3 lin_vel0 = lin_vel[b0] +
↪   lin_vel_delta[b0];
    vec3 lin_vel1 = lin_vel[b1] +
↪   lin_vel_delta[b1];
    vec3 ang_vel0 = ang_vel[b0] +
↪   ang_vel_delta[b0];
    vec3 ang_vel1 = ang_vel[b1] +
↪   ang_vel_delta[b1];

    // compute constraint velocity: J_l * X_dot
     ↪ = (v0 - v1) + (jw0 * w0) + (jw1 * w1) +
     ↪ ( * C_l)
    vec3 constraint_vel
        = (lin_vel0 - lin_vel1)
        + (jacobian[ic].w0 * ang_vel0)
        + (jacobian[ic].w1 * ang_vel1)
        + (baumgarte * violations[ic]);

    // increment the multiplier: lambda +=
     ↪ A_ll^-1 * h^-1 * J_l * X_dot
    multiplier[ic] += delassus_inv[ic] *
↪   constraint_vel * (1.f/dt);
}

void update_velocities(int ib, float dt) {

    // initially clear the velocity deltas
    lin_vel_delta[ib] = vec3(0.f);
    ang_vel_delta[ib] = vec3(0.f);

    // iterate over this body's constraints
     ↪ using the CSR structure.
    // ic0[ib] is the start index, ic0[ib+1] is
     ↪ the start of the next body (used as
     ↪ end).
    int start = ic0[ib];
    int end = ic0[ib + 1]; // sentinel ic0[NB] =
     ↪ NC*2 makes this safe for the last body

    for (int iic = start; iic < end; ++iic) {
        int ic = constraint_list[iic];
        bool is_body0 = (ib == ib0[ic]);

        // angular Jacobian block for this body
        mat3 jw = is_body0 ? jacobian[ic].w0 :
↪   jacobian[ic].w1;

        // linear delta: -h * m^-1 * lambda for
         ↪ body 0, +h * m^-1 * lambda for body
         ↪ 1
        // (from the +I_3 and -I_3 blocks in
         ↪ J_l, transposed)
        if (is_body0)
            lin_vel_delta[ib] -= dt *
↪   lin_mass_inv[ib] * multiplier[ic];
        else
            lin_vel_delta[ib] += dt *
↪   lin_mass_inv[ib] * multiplier[ic];

        // angular delta: +h * I^-1 * jw *
         ↪ lambda for both bodies
        // (jw^T = -jw, so -h * I^-1 * jw^T *
         ↪ lambda = +h * I^-1 * jw * lambda)
        ang_vel_delta[ib] += dt *
↪   ang_mass_inv[ib] * jw * multiplier[ic];
    }
}
```

N_b	Number of rigid bodies in the system
N_c	Number of constraints
h	Timestep (delta time)
n	Current timestep index
k	Jacobi iteration index
ℓ	Constraint index
i, j	Body indices (generic)
i_ℓ, j_ℓ	Indices of the two bodies connected by constraint ℓ
x_i	World-space position of body i
q_i	Unit quaternion orientation of body i
$\dot{x}_i$	Linear velocity of body i
ω_i	Angular velocity of body i (world space)
m_i	Scalar (linear) mass of body i
I_i	World-space rotational inertia tensor of body i
$R(q)$	Rotation operator: rotates a vector by unit quaternion q
$[v]_\times$	Cross-product matrix of vector v, satisfying $[v]_\times u = v \times u$
$\otimes$	Quaternion multiplication
X	System configuration state, $X \in \mathbb{R}^{7N_b}$
V	Generalized velocity vector, $V \in \mathbb{R}^{6N_b}$
M	Block-diagonal generalized mass matrix, $M \in \mathbb{R}^{6N_b \times 6N_b}$
F	Generalized force vector
L	Lagrangian
K	Kinetic energy
U	Potential energy
r_ℓ^0, r_ℓ^1	Local-space attachment points of constraint ℓ on bodies i_ℓ and j_ℓ
c_ℓ	Constraint function for constraint ℓ (3-vector)
$C(X)$	Full constraint vector (stack of all c_ℓ), $C \in \mathbb{R}^{3N_c}$
J	Constraint Jacobian, $J \in \mathbb{R}^{3N_c \times 6N_b}$, defined by $\dot{C} = JV$
J_ℓ	Rows of J for constraint ℓ, $J_\ell \in \mathbb{R}^{3 \times 6N_b}$
λ	Lagrange multiplier vector, $\lambda \in \mathbb{R}^{3N_c}$
λ_ℓ	Lagrange multiplier for constraint ℓ (3-vector)
λ^k	Lagrange multiplier estimate at Jacobi iteration k
V^*	Free velocity: velocity after force integration, before constraint solve
V^k	Velocity estimate at Jacobi iteration k
A	Delassus matrix, $A = JM^{-1}J^T \in \mathbb{R}^{3N_c \times 3N_c}$
$A_{\ell\ell}$	Diagonal 3×3 block of A for constraint ℓ
b	Constraint RHS vector, $b \in \mathbb{R}^{3N_c}$
D	Block-diagonal part of A (Jacobi splitting)
E	Off-block-diagonal part of A (Jacobi splitting)