

Constraints for EA

Indented remarks aren't needed to understand the main text.

Equations of motion in matrix form

[TW08] § 2, [TW18] § 1, [TW20] § 2

For a single rigid body, the equations of motion are

$$\begin{aligned} m\ddot{\mathbf{p}} &= \mathbf{f} \\ \mathbf{I}\ddot{\boldsymbol{\theta}} + \dot{\boldsymbol{\theta}} \times (\mathbf{I}\dot{\boldsymbol{\theta}}) &= \boldsymbol{\tau} \end{aligned}$$

where $\mathbf{p}$ is the position of the centre of mass ($\dot{\mathbf{p}}$ is its velocity, $\ddot{\mathbf{p}}$ is its acceleration), $\boldsymbol{\theta}$ is the orientation ($\dot{\boldsymbol{\theta}}$ is the angular velocity, $\ddot{\boldsymbol{\theta}}$ is the angular acceleration), m is the mass, $\mathbf{I}$ is the inertia tensor, $\mathbf{f}$ is the external force, and $\boldsymbol{\tau}$ is the external torque with respect to the centre of mass.

We'll drop the gyroscopic term $\dot{\boldsymbol{\theta}} \times (\mathbf{I}\dot{\boldsymbol{\theta}})$ for the time being. We'll see how to get it back later.

For a system of rigid bodies, the equations of motion (minus the gyroscopic term), can be put in matrix form:

$$\mathbf{M}\ddot{\mathbf{x}} = \mathbf{F}$$

where $\mathbf{M}$ is the mass matrix, $\mathbf{x}$ is the position vector, and $\mathbf{F}$ is the vector of external forces.

$$\begin{pmatrix} \begin{bmatrix} [m_1 \mathbb{1}] & \mathbb{0} \\ \mathbb{0} & [\mathbf{I}_1] \end{bmatrix} & & & \\ & \begin{bmatrix} [m_2 \mathbb{1}] & \mathbb{0} \\ \mathbb{0} & [\mathbf{I}_2] \end{bmatrix} & & \\ & & \ddots & \\ & & & \begin{bmatrix} [m_n \mathbb{1}] & \mathbb{0} \\ \mathbb{0} & [\mathbf{I}_n] \end{bmatrix} \end{pmatrix} \begin{pmatrix} \begin{bmatrix} \mathbf{p}_1 \\ \boldsymbol{\theta}_1 \end{bmatrix} \\ \begin{bmatrix} \mathbf{p}_2 \\ \boldsymbol{\theta}_2 \end{bmatrix} \\ \vdots \\ \begin{bmatrix} \mathbf{p}_n \\ \boldsymbol{\theta}_n \end{bmatrix} \end{pmatrix} = \begin{pmatrix} \begin{bmatrix} \mathbf{f}_1 \\ \boldsymbol{\tau}_1 \end{bmatrix} \\ \begin{bmatrix} \mathbf{f}_2 \\ \boldsymbol{\tau}_2 \end{bmatrix} \\ \vdots \\ \begin{bmatrix} \mathbf{f}_n \\ \boldsymbol{\tau}_n \end{bmatrix} \end{pmatrix}$$

The position vector $\mathbf{x}$ contains the (centre of mass) position and orientation of rigid bodies.

The forces vector $\mathbf{F}$ contains the generalised external forces (external forces and external torques with respect to the centre of mass) that affect the system.

The mass matrix $\mathbf{M}$ contains the generalised masses of the system as blocks in its diagonal: for each rigid body, a block containing diagonally a diagonal block with the body's mass and a block with the body's inertia tensor.

The algorithm described here can also work with single particles or with a mixture of single particles and bodies, by simply removing the rotation part for single particles.

If n_b is the number of bodies and n_p the number of particles, vectors $\mathbf{x}$ and $\mathbf{F}$ have dimension $n = 6n_b + 3n_p$, and $\mathbf{M}$ has dimension $n \times n$.

$\mathbf{M}$ converts accelerations into forces, or velocities into impulses.

Constraints

[TW08] § 2, [TW18] § 1, [TW20] § 2

Now, we introduce constraints on the position: $\mathbf{c}(\mathbf{x}; t) = \mathbf{0}$ (constraints like this, that only depend on time and position and not on velocity, are called *holonomic* constraints). The vector $\mathbf{c}(\mathbf{x})$ is a list of functions on the position of the system. We want the system to satisfy the conditions at all times. This will generate generalised additional constraint forces $\mathbf{F}_c$, still to be determined, so that now

$$\mathbf{M}\ddot{\mathbf{x}} = \mathbf{F} + \mathbf{F}_c$$

In general, the constraint forces introduce more unknowns than the constraints introduce equations. We need to restrict the degrees of freedom of $\mathbf{F}_c$. This is done by requiring that the constraints be ideal, i.e., that they do not inject energy in the system: for *any* virtual velocity compatible with the constraints $\dot{\mathbf{x}}_v$, the virtual work done by the constraint forces must be zero (d'Alembert's form of the principle of virtual work). A virtual velocity is the difference between two possible velocities.

Possible velocities can be obtained by deriving the constraints with respect to time:

$$\dot{\mathbf{c}} = \frac{\partial \mathbf{c}}{\partial \mathbf{x}} \dot{\mathbf{x}} + \frac{\partial \mathbf{c}}{\partial t} = \mathbf{0}$$

Virtual velocities are then those that satisfy

$$\frac{\partial \mathbf{c}}{\partial \mathbf{x}} \dot{\mathbf{x}}_v \stackrel{\text{def}}{=} \mathbf{J} \dot{\mathbf{x}}_v = \mathbf{0}$$

where $\partial \mathbf{c} / \partial \mathbf{x}$ is the *Jacobian* of $\mathbf{c}$, which we'll call $\mathbf{J}$.

$\mathbf{J}$ is a matrix that contains all the terms $\partial c_i / \partial x_j$, where c_i is the i -th constraint function and x_j is the j -th coordinate of $\mathbf{x}$. It represents how badly each constraint is violated when each coordinate moves. Each row is the gradient of one constraint with respect to the position: $\mathbf{J}_i = \nabla_{\mathbf{x}} c_i$. For a given constraint i , the non-zero elements of $\mathbf{J}_i$ show what the bodies affected by the constraint are.

A constraint that has several dimensions can be broken into several scalar constraints. If, after converting all constraints to scalar constraints, the number of scalar constraints is n_c , then the dimension of $\mathbf{c}$ as a vector of functions is n_c , and the dimension of $\mathbf{J}$ is $n_c \times n$.

The virtual power (virtual work divided by time) of a force $\mathbf{F}_c$ for a virtual velocity $\dot{\mathbf{x}}_v$ is $\mathbf{F}_c \cdot \dot{\mathbf{x}}_v$, which in matrix form is $\mathbf{F}_c^\top \dot{\mathbf{x}}_v$.

The ideal constraint condition is then

$$\forall \dot{\mathbf{x}}_v \text{ such that } \mathbf{J} \dot{\mathbf{x}}_v = \mathbf{0}, \mathbf{F}_c^\top \dot{\mathbf{x}}_v = 0$$

which implies that $\mathbf{F}_c^\top$ is a linear combination of the rows of $\mathbf{J}$:

$$\mathbf{F}_c = \mathbf{J}^\top \boldsymbol{\lambda} \quad \text{for some vector } \boldsymbol{\lambda}$$

The dimension of $\boldsymbol{\lambda}$ is n_c .

The elements of $\boldsymbol{\lambda}$ are called *Lagrange multipliers*.

With this expression for $\mathbf{F}_c$, the introduction of n_c ideal scalar constraints adds n_c new unknowns (the elements of $\boldsymbol{\lambda}$) but also n_c new equations (the constraint conditions).

$\mathbf{J}$ converts displacements into constraint function violations. $\mathbf{J}^\top$ converts Lagrange multipliers into forces.

Lagrange multipliers in analytic mechanics

In theory, in order to determine $\boldsymbol{\lambda}$, we could introduce the expression of $\mathbf{F}_c$ in the equations and solve for $\ddot{\mathbf{x}}$

$$\begin{aligned}\ddot{\mathbf{x}} &= \mathbf{M}^{-1} (\mathbf{F} + \mathbf{F}_c) \\ &= \mathbf{M}^{-1} (\mathbf{F} + \mathbf{J}^\top \boldsymbol{\lambda})\end{aligned}$$

and then derive $\mathbf{c}$ with respect to time twice:

$$\begin{aligned}\ddot{\mathbf{c}} &= \dot{\mathbf{J}}\dot{\mathbf{x}} + \mathbf{J}\ddot{\mathbf{x}} + \frac{\partial^2 \mathbf{c}}{\partial t^2} \\ &= \mathbf{J}\mathbf{M}^{-1} (\mathbf{F} + \mathbf{J}^\top \boldsymbol{\lambda}) + \dot{\mathbf{J}}\dot{\mathbf{x}} + \frac{\partial^2 \mathbf{c}}{\partial t^2} \\ &= \mathbf{0}\end{aligned}$$

which would give

$$\boldsymbol{\lambda} = -(\mathbf{J}\mathbf{M}^{-1}\mathbf{J}^\top)^{-1} \left[\mathbf{J}\mathbf{M}^{-1}\mathbf{F} + \dot{\mathbf{J}}\dot{\mathbf{x}} + \frac{\partial^2 \mathbf{c}}{\partial t^2} \right]$$

but *in practice* we won't use $\mathbf{F}$ in our algorithm and we usually violate the constraints, so we can't apply this.

Why won't we use $\mathbf{F}$ in our algorithm? Wouldn't it give us better anticipation to disturbances? Couldn't we do a first pass where we get constraint forces directly from this equation, add them to the regular forces, and run the solver with them, then run a

regular constraint correction pass which would hopefully yield smaller impulses and corrections and require fewer iterations? If this works, it would require also the evaluation of the $\dot{\mathbf{J}}\dot{\mathbf{x}}$ term.

The dimension of $(\mathbf{J}\mathbf{M}^{-1}\mathbf{J}^\top)^{-1}$ is $n_c \times n_c$. When you see the term $(\mathbf{J}\mathbf{M}^{-1}\mathbf{J}^\top)^{-1}$, resist the urge to "simplify" it into $(\mathbf{J}^\top)^{-1}\mathbf{M}\mathbf{J}^{-1}$: there's no such thing as $\mathbf{J}^{-1}$ because $\mathbf{J}$ isn't a square matrix.

The term $\dot{\mathbf{J}}\dot{\mathbf{x}}$ can be developed as

$$\begin{aligned}\dot{\mathbf{J}}\dot{\mathbf{x}} &= \frac{\partial \mathbf{J}}{\partial t} \dot{\mathbf{x}} + \left(\frac{\partial \mathbf{J}}{\partial \mathbf{x}} \dot{\mathbf{x}} \right) \dot{\mathbf{x}} \\ &= \frac{\partial \mathbf{J}}{\partial t} \dot{\mathbf{x}} + \left(\sum_j \frac{\partial \mathbf{J}}{\partial x_j} \dot{x}_j \right) \dot{\mathbf{x}}\end{aligned}$$

where $\partial \mathbf{J} / \partial \mathbf{x}$ is the $(n_c) \times (n \times n)$ Hessian tensor of $\mathbf{c}(\mathbf{x}; t)$ with respect to $\mathbf{x}$.

The expression for $\dot{\mathbf{J}}\dot{\mathbf{x}}$ can become very complicated. I wonder if it wouldn't be easier to simply replace it with $\frac{1}{\Delta t} (\mathbf{J}(\mathbf{x} + \dot{\mathbf{x}} \Delta t; t + \Delta t) - \mathbf{J}(\mathbf{x}; t)) \dot{\mathbf{x}}$ if the first pass of constraint forces described above works.

$(\mathbf{J}\mathbf{M}^{-1}\mathbf{J}^\top)$ converts Lagrange multipliers into [forces into accelerations into] accelerations of constraint violations. $(\mathbf{J}\mathbf{M}^{-1}\mathbf{J}^\top)$ does the opposite: it converts accelerations of constraint violations into Lagrange mul $(\mathbf{J}\mathbf{M}^{-1})$ converts forces into [accelerations into] accelerations of constraint violations.

Lagrange multipliers in simulation

[TW08] § 2, [TW18] § 1, [TW20] § 2, § 3, § 4

Instead, we measure the constraint violation and apply a correction in the form of a position increment to satisfy the constraint equations, using impulses that don't inject energy in the system (same condition as for the constraint forces).

Linearising the constraint equation with respect to the position, we get

$$\mathbf{c}(\mathbf{x} + \Delta \mathbf{x}; t) = \mathbf{c}(\mathbf{x}; t) + \mathbf{J} \Delta \mathbf{x}$$

Before we apply the correction, we get a value of the constraint function that may not be zero. Let's call it the *constraint error*: $\mathbf{e} = \mathbf{c}(\mathbf{x}; t)$. The position correction is the result of integrating the impulse $\mathbf{K} = \mathbf{J}^\top \boldsymbol{\lambda} \Delta t$ with respect to time, assuming the impulse is applied at the beginning of the time step,

$$\begin{aligned}\Delta \ddot{\mathbf{x}} &= \mathbf{M}^{-1} \mathbf{J}^\top \boldsymbol{\lambda} \\ \Delta \dot{\mathbf{x}} &= \Delta \ddot{\mathbf{x}} \Delta t = \mathbf{M}^{-1} \mathbf{J}^\top \boldsymbol{\lambda} \Delta t = \mathbf{M}^{-1} \mathbf{K} \\ \Delta \mathbf{x} &= \Delta \dot{\mathbf{x}} \Delta t = \mathbf{M}^{-1} \mathbf{J}^\top \boldsymbol{\lambda} \Delta t^2 = \mathbf{M}^{-1} \mathbf{K} \Delta t\end{aligned}$$

(we're only concerned here with the increment due to the constraint impulse, so we leave out everything else that the solver is already integrating). If we want the constraint error *after the correction* to be zero, i.e. $\mathbf{c}(\mathbf{x} + \Delta \mathbf{x}) = \mathbf{0}$, we need

$$\mathbf{c}(\mathbf{x} + \Delta \mathbf{x}; t) = \mathbf{c}(\mathbf{x}; t) + \mathbf{J} \Delta \mathbf{x} = \mathbf{c}(\mathbf{x}; t) + (\mathbf{J} \mathbf{M}^{-1} \mathbf{J}^\top) \boldsymbol{\lambda} \Delta t^2 = \mathbf{0}$$

therefore

$$\boldsymbol{\lambda} = -(\mathbf{J} \mathbf{M}^{-1} \mathbf{J}^\top)^{-1} \mathbf{e} \frac{1}{\Delta t^2}$$

and the resulting impulse and position increment are

$$\begin{aligned}\mathbf{K} &= \mathbf{J}^\top \boldsymbol{\lambda} \Delta t = -\mathbf{J}^\top (\mathbf{J} \mathbf{M}^{-1} \mathbf{J}^\top)^{-1} \mathbf{e} \frac{1}{\Delta t} \\ \Delta \mathbf{x} &= \mathbf{M}^{-1} \mathbf{K} \Delta t = -\mathbf{M}^{-1} \mathbf{J}^\top (\mathbf{J} \mathbf{M}^{-1} \mathbf{J}^\top)^{-1} \mathbf{e}\end{aligned}$$

$\mathbf{K}$ is a vector containing the impulse caused by the constraints on $\mathbf{x}$. Its dimension is n .

We'll refer to the matrix $(\mathbf{J} \mathbf{M}^{-1} \mathbf{J}^\top)$ as *acceptance* $\mathbf{A}$, and to its inverse as *resistance* $\mathbf{R}$.

$$\begin{aligned}\mathbf{A} &\stackrel{\text{def}}{=} (\mathbf{J} \mathbf{M}^{-1} \mathbf{J}^\top) \\ \mathbf{R} &\stackrel{\text{def}}{=} (\mathbf{J} \mathbf{M}^{-1} \mathbf{J}^\top)^{-1}\end{aligned}$$

The dimensions of $\mathbf{A}$ and $\mathbf{R}$ are $n_c \times n_c$. The rôle of $\mathbf{R}$ is similar to that of a mass quantity.

Gauss-Seidel

The equations are too hard to solve exactly in real time, so we use the Gauss-Seidel method, which is hard to describe and understand (and get right!) formally but easy to explain in English: for a number of iterations, we solve for each Lagrange multiplier alone in turn, supposing all other multipliers are solved. In a given iteration, a particular multiplier is solved for using the corrected values for those that have already been solved, and the uncorrected values for those that are yet to be solved.

This method can be mechanised simply by solving each constraint in turn as if it was the only active constraint, and keeping as intermediate state for the correction the per-body impulses and position corrections, rather than the Lagrange multipliers.

Gauss-Seidel with constraint re-evaluation

The full equation to solve using Gauss-Seidel is

$$(\mathbf{J}\mathbf{M}^{-1}\mathbf{J}^\top)\boldsymbol{\lambda} = -\mathbf{e}\frac{1}{\Delta t^2}$$

This equation is solved in turns for each constraint as explained above. Normally, constraints are unidimensional, for a multidimensional constraint (usually 3D or 6D), we can choose to solve the constraint as a whole. That's why we'll keep writing quantities such as $\mathbf{c}_i$, $\mathbf{e}_i$, and $\boldsymbol{\lambda}_i$ as vectors (in bold typeface).

For constraint i , the equation to solve when applying Gauss-Seidel is

$$\mathbf{A}_i\boldsymbol{\lambda} = -\mathbf{e}_i\frac{1}{\Delta t^2}$$

Solving for $\boldsymbol{\lambda}_i$:

$$\boldsymbol{\lambda}_i = -(\mathbf{A}_{ii})^{-1} \left(\mathbf{e}_i\frac{1}{\Delta t^2} + \sum_{j \neq i} \mathbf{A}_{ij}\boldsymbol{\lambda}_j \right)$$

By iterating several times through all constraints, we could thus obtain a final approximation of λ from which to compute the impulses and position corrections (but we won't: read on).

Note that in the solution above, the value of e stays the same across iterations:
 $e = c(x; t)$, where x is still the position without any constraint correction.

Up until now, we've always worked with the linearised equations: we measure the error and its Jacobian at the beginning, and then iterate on them, never re-evaluating them. Let's see how to replace this linearisation with exact computations. This will make non-linear parts of our model (rotations and non-linear constraint functions) more accurate, and allow us to better partition the problem, arranging the solve state into separate per-constraint and per-body state in such a way that each constraint doesn't need to access state from other constraints.

The basic idea is to create a new updated problem every time we update λ , identical to the original one but for $x + \Delta x$ instead of x , i.e., with the position corrected to the most up-to-date set of impulses. Since that new problem was reached with the previous value of λ , the new value of λ , with its updated λ_i , represents an increment to bring $c_i(x + \Delta x; t)$ to 0 using constraint i . This will only require an evaluation of the constraint function for the particular constraint we're solving for.

Let's consider the update of a single λ_i at any iteration: the value of λ goes from λ^0 to λ^+ , where λ^+ is identical to λ^0 except for λ_i :

$$\begin{aligned}\lambda_i^+ &= -(\mathbf{A}_{ii})^{-1} \left(\mathbf{e}_i \frac{1}{\Delta t^2} + \sum_{j \neq i} \mathbf{A}_{ij} \lambda_j^0 \right) \\ &= \lambda_i^0 - (\mathbf{A}_{ii})^{-1} \frac{1}{\Delta t^2} \left(\mathbf{e}_i + \sum_j \mathbf{A}_{ij} \lambda_j^0 \Delta t^2 \right) \\ &= \lambda_i^0 - (\mathbf{A}_{ii})^{-1} \frac{1}{\Delta t^2} (\mathbf{e}_i + \mathbf{A}_i \lambda^0 \Delta t^2)\end{aligned}$$

where the term in brackets can be recognised as the linearisation

$$\left(c(x; t) + (\mathbf{J} \mathbf{M}^{-1} \mathbf{J}^\top) \lambda^0 \Delta t^2 \right)_i = c_i(x + \Delta x^0; t)$$

We can thus compute the Lagrange multiplier increment $\delta\lambda_i$ for constraint i for this iteration with the single non-linearised constraint function for constraint i , and apply it to λ_i :

$$\begin{aligned}\delta\lambda_i &= -(\mathbf{A}_{ii})^{-1} \frac{1}{\Delta t^2} \mathbf{c}_i(\mathbf{x} + \Delta\mathbf{x}^0; t) \\ \lambda_i^+ &= \lambda_i^0 + \delta\lambda_i\end{aligned}$$

and do the same to the impulse vector $\mathbf{K}_i$ for constraint i and to the position correction $\Delta\mathbf{x}$:

$$\begin{aligned}\delta\mathbf{K}_i &= (\mathbf{J}_i)^\top \delta\lambda_i \Delta t \\ \mathbf{K}_i^+ &= \mathbf{K}_i^0 + \delta\mathbf{K}_i \\ \delta\Delta\mathbf{x} &= \mathbf{M}^{-1} \delta\mathbf{K}_i \Delta t \\ \Delta\mathbf{x}^+ &= \Delta\mathbf{x}^0 + \delta\Delta\mathbf{x}\end{aligned}$$

$\mathbf{K}$ is a vector containing the impulse caused by all constrains on all of $\mathbf{x}$. We're concerned here with $\mathbf{K}_i$, which is not the i -th element of $\mathbf{K}$, but the impulse caused by constraint i on all of $\mathbf{x}$. All vectors $\mathbf{K}_i$ add up to $\mathbf{K}$.

For added accuracy, we should recompute $\mathbf{J}$ and $\mathbf{A}$ each time, but it would be very expensive. Just undoing the linearisation of $\mathbf{c}$ and that of the rotations in $\Delta\mathbf{x}$ is already much better than the fully linearised version.

Partitioning Gauss-Seidel

[TW20] § 5

In the form described in the previous section, the solution can be partitioned efficiently.

Let's call $\beta_i = \{\beta_{ik}\}$ the set of bodies affected by constraint i (those whose state the constraint function i depends on). We'll call $\mathbf{J}_{i\beta_i}$ the submatrix of $\mathbf{J}$ formed by selecting for constraint i and the set of bodies affected by it β_i . Similarly, $\mathbf{M}_{\beta_i}$ is the square submatrix of $\mathbf{M}$ where only the elements of the set of bodies β_i are kept.

Different constraints may be contributing to the positions of the same bodies; for instance, if constraint 1 affects bodies 2 and 3 and constraint 4 affects bodies 3 and 5,

then we have $\beta_{1,2} = \beta_{4,1} = 3$. This is how constraints affect each other and why we need to use Gauss-Seidel in the first place.

When evaluating constraint i and solving for its Lagrange multiplier and impulse as shown above, we can restrict ourselves to the relevant submatrices of $\mathbf{J}$ and $\mathbf{M}$, and the relevant subvectors of $\boldsymbol{\lambda}$, $\mathbf{K}_i$ and $\Delta \mathbf{x}$:

- computing $\delta \boldsymbol{\lambda}_i$ and $\boldsymbol{\lambda}_i^+$ only requires
 - precomputable $(\mathbf{A}_{ii})^{-1} = \left(\mathbf{J}_{i\beta_i} (\mathbf{M}_{\beta_i})^{-1} (\mathbf{J}_{i\beta_i})^\top \right)^{-1}$
 - $\mathbf{c}_i(\mathbf{x} + \Delta \mathbf{x}^0; t)$: re-evaluation of constraint i requiring only $\mathbf{x}_{\beta_i}^0 + \Delta \mathbf{x}_{\beta_i}^0$
 - past value $\boldsymbol{\lambda}_i^0$
- computing $(\delta \mathbf{K}_i)_{\beta_i}$ and $(\mathbf{K}_i^+)_{\beta_i}$ only requires
 - precomputable $\mathbf{J}_{i\beta_i}$
 - $\delta \boldsymbol{\lambda}_i$
 - past value $(\mathbf{K}_i^0)_{\beta_i}$
- computing $\delta \Delta \mathbf{x}_{\beta_i}$ and $\Delta \mathbf{x}_{\beta_i}^+$ only requires
 - precomputable $(\mathbf{M}_{\beta_i})^{-1}$
 - $(\delta \mathbf{K}_i)_{\beta_i}$
 - past value $\Delta \mathbf{x}_{\beta_i}^0$

Unilateral constraints

[TW08] § 4, [TW18] § 2.2.5, § 2.2.6, [TW20] § 9

Unilateral constraints, or *inequality* constraints, are scalar constraints whose expression is $c(\mathbf{x}; t) \leq 0$ instead of $c(\mathbf{x}; t) = 0$, or constraints that have limitations in the generalised constraint forces they can generate. They can easily be integrated with the Gauss-Seidel method: for each unilateral constraint, depending on the constraint definition,

- either skip the constraint if it's already satisfied,
- or solve it but clamp the impulse or the correction before applying it.

Algorithm for constraints in the displacement-based solver

Data:

- input
 - position $\mathbf{x}$ determined by the solver without the constraints (per body)
- solve state
 - impulse $\mathbf{K} = \mathbf{J}^\top \boldsymbol{\lambda} \Delta t$ (per constraint)
 - position correction $\Delta \mathbf{x}$ (per body)
- output
 - impulse $\mathbf{K} = \mathbf{J}^\top \boldsymbol{\lambda} \Delta t$ (per constraint)
 - position correction $\Delta \mathbf{x}$ (per body)

Algorithm:

- $\Delta \mathbf{x} \leftarrow \mathbf{0}$
- for each constraint i
 - "compile"
 - for the affected bodies β_i compute and store

$$(\mathbf{A}_{ii})^{-1} = \left(\mathbf{J}_{i\beta_i} (\mathbf{M}_{\beta_i})^{-1} (\mathbf{J}_{i\beta_i})^\top \right)^{-1}$$
 - compute and store other magnitudes that won't change across iterations (depends on the kind of constraint)
 - initialise constraint solve state (part of the "compile" function)
 - $(\mathbf{K}_i)_{\beta_i} \leftarrow \mathbf{0}$
- iterate for a number of iterations
 - for each constraint i
 - "solve" (update solve state)
 - $\mathbf{e}_i = \mathbf{c}_i(\mathbf{x}_{\beta_i} + \Delta \mathbf{x}_{\beta_i}; t)$
 - $\delta \boldsymbol{\lambda}_i = -(\mathbf{A}_{ii})^{-1} \mathbf{e}_i \frac{1}{\Delta t^2}$
 - $(\delta \mathbf{K}_i)_{\beta_i} = (\mathbf{J}_{i\beta_i})^\top \delta \boldsymbol{\lambda}_i \Delta t$
 - $(\mathbf{K}_i)_{\beta_i} \leftarrow (\mathbf{K}_i)_{\beta_i} + (\delta \mathbf{K}_i)_{\beta_i}$
 - for each affected body β_{ik}
 - $\delta \Delta \mathbf{x}_{\beta_{ik}} = (\mathbf{M}_{\beta_{ik}})^{-1} (\delta \mathbf{K}_i)_{\beta_{ik}}$ (displacement increment to body β_{ik} due to $\delta \mathbf{K}_{i\beta_{ik}}$)
 - $\Delta \mathbf{x}_{\beta_{ik}} \leftarrow \Delta \mathbf{x}_{\beta_{ik}} + \delta \Delta \mathbf{x}_{\beta_{ik}}$ (apply new displacement to body; another constraint may apply a new displacement in the same

iteration)

Introducing other terms

In the previous discussion, we've only used position terms that can be put as the solution of a linear matrix equation on $\Delta \mathbf{x}$. Let's see how we can introduce other kinds of terms.

Getting the gyroscopic term back —speculative—

[TW20] § 3

In the development above we've left out the gyroscopic term

$$\mathbf{T}_\Gamma \stackrel{\text{def}}{=} \dot{\boldsymbol{\theta}} \times (\mathbf{I} \dot{\boldsymbol{\theta}})$$

To introduce it back in the algorithm, we use skew symmetric matrices for the cross product:

$$[\mathbf{a}]_{\times} = [a_1 \quad a_2 \quad a_3]_{\times} = \begin{pmatrix} 0 & -a_3 & a_2 \\ a_3 & 0 & -a_1 \\ -a_2 & a_1 & 0 \end{pmatrix}$$
$$\mathbf{a} \times \mathbf{b} = [\mathbf{a}]_{\times} \mathbf{b} = [\mathbf{b}]_{\times}^{\top} \mathbf{a} = -[\mathbf{b}]_{\times} \mathbf{a}$$

therefore

$$\dot{\boldsymbol{\theta}} \times (\mathbf{I} \dot{\boldsymbol{\theta}}) = [\dot{\boldsymbol{\theta}}]_{\times} (\mathbf{I} \dot{\boldsymbol{\theta}}) = -[\mathbf{I} \dot{\boldsymbol{\theta}}]_{\times} \dot{\boldsymbol{\theta}}$$

The increment of $\mathbf{T}_\Gamma$ due to the constraint correction on $\Delta \boldsymbol{\theta}$ is

$$\begin{aligned} \Delta \mathbf{T}_\Gamma &= (\dot{\boldsymbol{\theta}} + \Delta \dot{\boldsymbol{\theta}}) \times [\mathbf{I} (\dot{\boldsymbol{\theta}} + \Delta \dot{\boldsymbol{\theta}})] - \dot{\boldsymbol{\theta}} \times (\mathbf{I} \dot{\boldsymbol{\theta}}) \\ &= \left([\dot{\boldsymbol{\theta}}]_{\times} \mathbf{I} - [\mathbf{I} \dot{\boldsymbol{\theta}}]_{\times} \right) \Delta \dot{\boldsymbol{\theta}} \\ &= \left([\dot{\boldsymbol{\theta}}]_{\times} \mathbf{I} - [\mathbf{I} \dot{\boldsymbol{\theta}}]_{\times} \right) \Delta \ddot{\boldsymbol{\theta}} \Delta t \end{aligned}$$

We just have to compute $\Delta \mathbf{T}_\Gamma$ here because the solver is already taking care of $\mathbf{T}_\Gamma$.

We can introduce this into the equations for the constraint corrections:

$$\mathbf{M}(\Delta \mathbf{x} + \Delta \mathbf{T}_\Gamma \Delta t) = -\mathbf{J}^\top (\mathbf{J} \mathbf{M}^{-1} \mathbf{J}^\top)^{-1} \mathbf{e} \frac{1}{\Delta t^2}$$

We turn this into

$$\tilde{\mathbf{M}} \Delta \mathbf{x} = -\mathbf{J}^\top (\mathbf{J} \mathbf{M}^{-1} \mathbf{J}^\top)^{-1} \mathbf{e} \frac{1}{\Delta t^2}$$

with

$$\tilde{\mathbf{M}} = \mathbf{M} + ([\dot{\mathbf{x}}_\Gamma]_\times \mathbf{M} + [\mathbf{M} \dot{\mathbf{x}}_\Gamma]_\times) \Delta t$$

where $\dot{\mathbf{x}}_\Gamma$ is $\dot{\mathbf{x}}$ with all components that aren't angular velocities turned into zeros, and the skew symmetric matrix is formed by diagonal blocks which are the skew symmetric matrices for the components corresponding to angular velocities, and zero everywhere else.

This equation can be solved the same as before, just changing the value of the mass matrix, which now depends on $\dot{\mathbf{x}}_\Gamma$.

When computing $\tilde{\mathbf{M}}$, we can use the uncorrected value or the corrected value for $\dot{\mathbf{x}}_\Gamma$: we're computing the derivative of the gyroscopic term, not the term itself, which has already been taken care of by the solver. I guess using the corrected value of $\dot{\mathbf{x}}_\Gamma$ results into a more stable algorithm, because it's kind of like an implicit integration scheme, where you evaluate the function to integrate at $t + \Delta t$ rather than at t .

Potential generalised forces —speculative—

[TW20] § 3, § 10

—FIXME—write—

Examples of constraints

Linear joint

[TW08] § 4.1, [TW18] § 3.1, [TW20] § 6

The constraint for a linear joint is that two points, each on a rigid body, have the same world position. For each body b , let $\mathbf{r}_b$ be the position of the joint point in the mass frame. Then, its world position is $\mathbf{p}_{jb} = \mathbf{p}_b + \boldsymbol{\theta}_b \times \mathbf{r}_b$, where $\mathbf{p}_b$ and $\boldsymbol{\theta}_b$ are the centre of mass position and orientation of the body. We can write the constraint as

$$\begin{aligned} c(\mathbf{p}_1, \boldsymbol{\theta}_1, \mathbf{p}_2, \boldsymbol{\theta}_2) &= |\mathbf{p}_{j2} - \mathbf{p}_{j1}| \\ &= |\mathbf{p}_2 + \boldsymbol{\theta}_2 \times \mathbf{r}_2 - \mathbf{p}_1 - \boldsymbol{\theta}_1 \times \mathbf{r}_1| \end{aligned}$$

Let $\mathbf{p} = \mathbf{p}_{j2} - \mathbf{p}_{j1}$ and $\mathbf{u} = \mathbf{p} / |\mathbf{p}|$. Using $\nabla_{\mathbf{p}} |\mathbf{p}| = \mathbf{p} / |\mathbf{p}| = \mathbf{u}$, the partial derivatives of c are

$$\begin{aligned} \frac{\partial c}{\partial \mathbf{p}_1} &= \frac{\partial c}{\partial \mathbf{p}} \frac{\partial \mathbf{p}}{\partial \mathbf{p}_1} = -\mathbf{u} \\ \frac{\partial c}{\partial \boldsymbol{\theta}_1} &= \frac{\partial c}{\partial \mathbf{p}} \frac{\partial \mathbf{p}}{\partial \boldsymbol{\theta}_1} = +[\mathbf{r}_1]_{\times} \mathbf{u} \\ \frac{\partial c}{\partial \mathbf{p}_2} &= \frac{\partial c}{\partial \mathbf{p}} \frac{\partial \mathbf{p}}{\partial \mathbf{p}_2} = +\mathbf{u} \\ \frac{\partial c}{\partial \boldsymbol{\theta}_2} &= \frac{\partial c}{\partial \mathbf{p}} \frac{\partial \mathbf{p}}{\partial \boldsymbol{\theta}_2} = -[\mathbf{r}_2]_{\times} \mathbf{u} \end{aligned}$$

The Jacobian is

$$\mathbf{J} = \begin{pmatrix} -\mathbf{u} \\ +[\mathbf{r}_1]_{\times} \mathbf{u} \\ +\mathbf{u} \\ -[\mathbf{r}_2]_{\times} \mathbf{u} \end{pmatrix}^{\top}$$

And the acceptance is a scalar

$$\begin{aligned} A &= \mathbf{J} \mathbf{M}^{-1} \mathbf{J}^{\top} \\ &= \mathbf{u}_{12} \left(\frac{1}{m_1} + [\mathbf{r}_1]_{\times}^{\top} \mathbf{I}_1 [\mathbf{r}_1]_{\times} + \frac{1}{m_2} + [\mathbf{r}_2]_{\times}^{\top} \mathbf{I}_2 [\mathbf{r}_2]_{\times} \right) \mathbf{u} \\ &= \mathbf{u}_{12} \left(\frac{1}{m_1} \frac{1}{m_2} - [\mathbf{r}_1]_{\times} \mathbf{I}_1 [\mathbf{r}_1]_{\times} - [\mathbf{r}_2]_{\times} \mathbf{I}_2 [\mathbf{r}_2]_{\times} \right) \mathbf{u} \end{aligned}$$

Having computed A , the impulses for the (scalar) error $e = |\mathbf{p}|$ are

$$\mathbf{J}^\top \boldsymbol{\lambda} \Delta t = -\mathbf{J}^\top \frac{e}{A} \frac{1}{\Delta t} = -\frac{e}{A} \begin{pmatrix} -\mathbf{u} \\ +[\mathbf{r}_1]_\times \mathbf{u} \\ +\mathbf{u} \\ -[\mathbf{r}_2]_\times \mathbf{u} \end{pmatrix} \frac{1}{\Delta t}$$

and the corrections are

$$\Delta \mathbf{x} = -\mathbf{M}^{-1} \mathbf{J}^\top \frac{e}{A} = \frac{e}{A} \begin{pmatrix} +\frac{1}{m_1} \mathbf{u} \\ -\mathbf{I}_1^{-1} [\mathbf{r}_1]_\times \mathbf{u} \\ -\frac{1}{m_2} \mathbf{u} \\ +\mathbf{I}_2^{-1} [\mathbf{r}_2]_\times \mathbf{u} \end{pmatrix} = \frac{e}{A} \begin{pmatrix} +\frac{1}{m_1} \mathbf{u} \\ +\mathbf{I}_1^{-1} (\mathbf{u} \times \mathbf{r}_1) \\ -\frac{1}{m_2} \mathbf{u} \\ -\mathbf{I}_2^{-1} (\mathbf{u} \times \mathbf{r}_2) \end{pmatrix}$$

which is equivalent to the result of applying a pair of (force) impulses $+\boldsymbol{\kappa}/\Delta t$ and $-\boldsymbol{\kappa}/\Delta t$ to bodies 1 and 2 respectively at the joint point, with

$$\boldsymbol{\kappa} = \frac{e}{A} \mathbf{u} \frac{1}{\Delta t} = \frac{1}{A} (\mathbf{p}_{j2} - \mathbf{p}_{j1}) \frac{1}{\Delta t}$$

Box joint

—FIXME—write—

Cone joint

—FIXME—write—

Angular axle

—FIXME—write—

Drive

—FIXME—write—

Contact

—FIXME—write—

Contact friction

—FIXME—write—

SIMD constraint implementations

A constraint implementation implements one iteration of the constraint solve for a particular kind of constraint `<Constraint>` . It consists of

- a class definition for `<Constraint>` in "JobArguments.cs" that contains, organised in streams, the input data (`Stream="Input"`), compiled data (`Stream="Compiled"`), and solve data (stored directly as streams with `Index="TemporaryIndex"`); and the string (why? are they treated as binary blobs?) streams `Input` and `Compiled` to hold the data
- a compile function `<Constraint>Compile()` in file " `<Compile>.h` " that computes and stores the compiled data (data that is used by the solve function and that remains constant for the frame) for all constraints of type `<Constraint>` ; this function is called once per frame, before the solve iterations begin
- a solve function `<Constraint>Solve()` in file " `<Compile>.h` " that runs one iteration of the solve for all constraints of type `<Constraint>` ; this function is called a number of times (iterations) per frame

Constraint implementations usually derive from base classes (contacts derive from `TwoBodyContact` , and constraints and `TwoBodyContact` derive from `TwoBodyConstraint`). Each of these classes contains its own streams and input definition.

The `<Constraint>Compile()` and `<Constraint>Solve()` functions have the following arguments:

- `JobArguments *arguments` to access the streams where the data are stored

- `start` and `count` , which define how many constraints must be looped over
- `worldBody` , which gives the index of the world body (when a body is fixed, its index is the world body index)

SIMD

The `<Constraint>Compile()` and `<Constraint>Solve()` functions must iterate over all constraints of type `<Constraint>` . This could be done by scanning the constraints with a regular for-loop, but it's faster if we use SIMD to run `SIMD_NUM_ELEMENTS` constraints in parallel. The streams are organised in such a way that in any given block of `SIMD_NUM_ELEMENTS` constraints, every constraint is in a separate island (they are all unrelated).

SIMD code is more constrained than regular code. The same exact operations are done on all data for all lanes simultaneously, so there can't be any branching. Branching is replaced with a combination of selecting functions whose result depends on a condition on its arguments. Loops are possible if the number of iterations doesn't depend on lane-specific data.

General structure

The general structure of the `<Constraint>Compile()` and `<Constraint>Solve()` functions is

1. set up local variables to access all required streams
2. loop over all the constraints in blocks of `SIMD_NUM_ELEMENTS`
 - read data from streams
 - operate on data
 - store data on streams

Stream access setup

Input streams are declared in different ways depending on where (the `<Constraint>` class, in one of its ancestors, or directly in the job arguments) and how (stream definition attributes) they're defined. They can require a index representing the kind of constraint, or an offset to the `start` argument, or an index argument to access past history, or nothing at

all. They are accessed with methods `Pop()` , `Gather()` , or `Load()` and updated with methods `Push()` or `Scatter()` .

I don't understand all of this, but i can probably work without knowing the details. In particular, i don't understand the logic behind the different methods to access and update stream, and why some streams use the pair `Pop()` - `Push()` while others use `Load()` - `Push()` .

References

From [Physics Core - Solver Maths](#):

- [TW08] “stasis_solver.pdf”
The Physics Stasis Solver, 2008 (Tom Waterson)
- [TW18] “solvern timer.pdf”
Physics Solver Maths, 2018 (Tom Waterson)
- [TW20] “Displacement_Based_Rigid_Body_Dynamics.pdf”
Displacement Based Rigid Body Dynamics, 2020 (Tom Waterson)