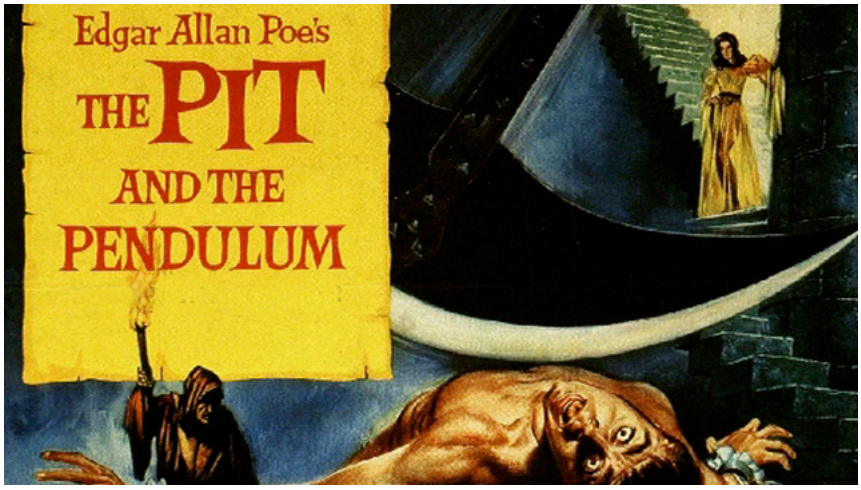
Physics Tutorial 7: Solvers



Summary

In this lecture we present the idea of constraint-based solvers. We explore constraint solutions in the context of physical systems with reference to the Jacobian is explored, and its connection to constraint force. We round off with the introduction of the global solver as a means to address the inter-related complexities of our simulation, with specific discussion of the solver implementation employed within the framework.

New Concepts

Differentiating Constraints, Jacobian solutions, the Constraint Force, Global Solvers, Linear Equations, Gauss-Seidel

Introduction

Up to now, we've primarily focused on the raw mathematics which underpins physics simulation. While we've addressed time-stepped solutions to acceleration-based position updates, and reactions to single collisions, we've not yet delved in to how we go about addressing the issues which arise once objects are capable of interacting in a more complex fashion.

Today, we introduce the concepts which will ultimately enable items represented by our physics engine to interact with each other in a meaningful and believable way. The idea of constraints is extended into the realm of solving multiple constraints simultaneously. We'll explore the motivation behind this approach, and include some (non-examinable) mathematics to explain its derivation. Following this, we revisit the Jacobian.

We move on to discuss the idea that we can represent our web of inter-related constraints as a set of simultaneous, linear equations. We refresh our knowledge of linear equations and introduce the idea of the matrix-based approach to their solution. We present the Gauss-Seidel method of obtaining a solution to our simulation which satisfies all constraints (as best we can), and discuss some implementation detail regarding the framework.

Constraints, Revisited

Up to now, we have discussed constraints in a very mechanical sense. Here, we discuss some more of the underlying mathematics.

We define a constraint using multivalued functions such as $\mathrm{C}(\mathbf{x}_1, \mathbf{q}_1, \mathbf{x}_2, \mathbf{q}_2)$, where $\mathbf{x}_1, \mathbf{x}_2$ are the positions of the two bodies and $\mathbf{q}_1, \mathbf{q}_2$ are the two quaternions representing the orientations of the two bodies.

We restrict the motion of the two bodies by maintaining the relationship:

$$\mathrm{C}(\mathbf{x}_1, \mathbf{q}_1, \mathbf{x}_2, \mathbf{q}_2) = 0$$

If the Constraint is satisfied the two bodies are held in the correct relative positioning in relation to each other. We use differentiation in order to calculate the quantities required to correct any error in the constraint.

Differentiating the Constraint (Non-Examinable)

The constraint function C is multivariate (i.e. has multiple input variables) and these variables are vector type quantities. To understand the properties of these constraints we need to know is how the constraint evolves over time given changes in position (i.e. velocities) or changes in orientation (i.e. angular velocities). Each of the variables $\mathbf{x}_1, \mathbf{q}_1, \mathbf{x}_2, \mathbf{q}_2$ are themselves determined by time. This means we can analyse the constraints by applying differentiation with respect to time to the function C and treat the variables as functions of time themselves.

When we differentiate a constraint with respect to time we can apply what is called the multivariate chain rule:

$$\frac{\mathrm{dC}}{\mathrm{d}t} = \frac{\partial \mathrm{C}}{\partial \mathbf{x}_1}\frac{\mathrm{d}\mathbf{x}_1}{\mathrm{d}t} + \frac{\partial \mathrm{C}}{\partial \mathbf{q}_1}\frac{\mathrm{d}\mathbf{q}_1}{\mathrm{d}t} + \frac{\partial \mathrm{C}}{\partial \mathbf{x}_2}\frac{\mathrm{d}\mathbf{x}_2}{\mathrm{d}t} + \frac{\partial \mathrm{C}}{\partial \mathbf{q}_2}\frac{\mathrm{d}\mathbf{q}_2}{\mathrm{d}t}$$

The multivariate chain rule uses what's called partial differentiation, $\frac{\partial \mathrm{C}}{\partial \mathbf{x}}$. Partial differentiation allows you to differentiate a multivariate function with respect to a single variable and hold all others constant. For example given a multivariate function:

$$\mathrm{f}(x, y) = 2x^2 + 3xy - y^2$$

then the results of the partial differentiation on x and y are:

$$\frac{\partial \mathrm{f}}{\partial x} = 4x + 3y$$
$$\frac{\partial \mathrm{f}}{\partial y} = 3x - 2y$$

and if x and y were functions of time the chain rule would give:

$$\frac{\mathrm{df}}{\mathrm{d}t} = (4x + 3y)\frac{\mathrm{d}x}{\mathrm{d}t} + (3x - 2y)\frac{\mathrm{d}y}{\mathrm{d}t}$$

Now consider the fact that the variables of the constraint C are vector type variables. The rules of calculus can be extended to vectors and there are often analogies between the one dimensional case and the vector case. Suppose $\mathbf{x}(t) = (x(t), y(t), z(t))$ then these are some simple examples of vector calculus:

$$\frac{\mathrm{d}\mathbf{x}}{\mathrm{d}t} = \left(\frac{\mathrm{d}x}{\mathrm{d}t}, \frac{\mathrm{d}y}{\mathrm{d}t}, \frac{\mathrm{d}z}{\mathrm{d}t}\right) = \mathbf{v}$$
$$\frac{\mathrm{d}\mathbf{x} \cdot \mathbf{n}}{\mathrm{d}t} = \mathbf{v} \cdot \mathbf{n}$$
$$\frac{\mathrm{d}\mathbf{q}}{\mathrm{d}t} \approx \tfrac{1}{2}\mathbf{q} \times \omega$$

Jacobian

Now that we have an idea about how we would like to constrain our system we now need to know how to maintain the relationship. By deriving the constraint by time we get an idea of how the constraint is changing over time. However the aim is that the constraint should never change, therefore the differentiation of the constraint should be equal to zero.

By re-arranging the quantities involved in the Constraint we can rewrite the differential of the Constraint in terms of a velocity vector $\mathbf{V}$ and a vector we call the Jacobian $\mathbf{J}$:

$$\dot{\mathrm{C}} = \mathbf{JV} = 0$$

The first vector $\mathbf{V}$ contains all variables associated with the velocity of the two objects, i.e. the velocity in the x,y,z-axis of the first object make 3 of the components of the vector $\mathbf{V}$. We simplify the representation of $\mathbf{V}$ by grouping common components into the 'sub-vectors' velocity $\mathbf{v}_1$, angular velocity ω_1, velocity $\mathbf{v}_2$ and angular velocity ω_2:

$$\mathbf{V} = \begin{pmatrix} \mathbf{v}_1 \\ \omega_1 \\ \mathbf{v}_2 \\ \omega_2 \end{pmatrix}$$

The Jacobian takes the form of a row vector which collates the coefficients which multiplied the velocity vectors, i.e. the Jacobian is made up of 4 vector valued functions of $\mathbf{x}_1, \mathbf{q}_1, \mathbf{x}_2, \mathbf{q}_2$ (or parameters related to these quantities):

$$\mathbf{J} = \begin{pmatrix} \mathbf{j}_1(\mathbf{x}_1, \mathbf{q}_1, \mathbf{x}_2, \mathbf{q}_2) & \mathbf{j}_2(\mathbf{x}_1, \mathbf{q}_1, \mathbf{x}_2, \mathbf{q}_2) & \mathbf{j}_3(\mathbf{x}_1, \mathbf{q}_1, \mathbf{x}_2, \mathbf{q}_2) & \mathbf{j}_4(\mathbf{x}_1, \mathbf{q}_1, \mathbf{x}_2, \mathbf{q}_2) \end{pmatrix}$$

This means that if we can find the differentiation of C with respect to time and re-write it as such:

$$\dot{\mathrm{C}} = \mathbf{j}_1 \cdot \mathbf{v}_1 + \mathbf{j}_2 \cdot \omega_1 + \mathbf{j}_3 \cdot \mathbf{v}_2 + \mathbf{j}_4 \cdot \omega_2$$

we can determine the Jacobian $\mathbf{J}$ by reading what values multiply $\mathbf{v}_1$, ω_1, $\mathbf{v}_2$ and ω_2.

Constraint Force

The Jacobian is the key to maintaining the constraints on the objects in our physics engine but to see this we first need to consider the concept of *Work Done*. An object is considered to have done work if it has been pushed through a non-zero distance by a constant force in the direction of the force. As such we calculate the work done by a force as:

$$W = \mathbf{F} \cdot \mathbf{s}$$

where the $\mathbf{F}$ is the force doing work and $\mathbf{s}$ is the 'displacement' (a vector indicating the direction and distance travelled) of the work done. Work done is measured in units of energy and represent the energy that has entered the system.

If you consider a box sitting on a table, the table exerts a force pushing up onto the box stopping it from falling through the table. In this situation however no energy enters the system since the box gains no velocity and has not moved from its spot on the table.

Now consider if the table suddenly disappeared. If the height of the table was h and the force applied by gravity is $m\mathbf{g}$ then the energy added to the system by the time the box hits the floor is $mh|\mathbf{g}|$ which is exactly the potential energy the box had in its original position on the table.

Constraints however are restrictions stopping objects moving in particular ways. Either by stopping an object occupying a certain location or by restricting the way an object rotates. This can be relative to other dynamic objects or static objects in the environment. This means that a constraint never adds energy to the system.

If we consider the rate of Work done (otherwise known as the Power of the force) we can reason that the Power of the system is zero. In other words there is zero rate of energy being introduced to the system. This can be written as:

$$P = \mathbf{F} \cdot \frac{\mathbf{s}}{\Delta t} = 0$$

for small enough time intervals the value $\mathbf{s}/\Delta t$ is equivalent to the velocity v of the object. This means that the velocity of the object is perpendicular to the constraint force:

$$\mathbf{F} \cdot \mathbf{v} = 0$$

This can be generalized to our Constraint equations. The two bodies in our system have four quantities which keep them constrained, one force for each object and one torque for each object:

$$\mathbf{F} = \begin{pmatrix} \mathbf{f}_1 \\ \tau_1 \\ \mathbf{f}_2 \\ \tau_2 \end{pmatrix}$$

By generalizing our earlier observation about constraint forces being perpendicular to the velocity vector we find that:

$$\mathbf{F} \cdot \mathbf{V} = 0$$

This is exactly the relationship the Jacobian $\mathbf{J}$ has with the velocity vector $\mathbf{V}$. We can use the Jacobian as a basis for our constraint force, by multiplying by an unknown quantity λ we set our force equal to:

$$\mathbf{F} = \mathbf{J}^T \lambda$$

We will demonstrate next lecture how to calculate an appropriate value for λ.

The Global Solver

The Problem

Constraints can be used to construct a large majority of the basic physical effects we see in computer games from an object resting on a flat surface to two objects connected by a ridged rod.

However as more constraints are implemented in the physics system, there becomes a need to be able to solve all constraints at the same time. This equates to solving all the linear equations at one time, and in fact (as can be seen from the implementation later) is implemented as such. We call this a global solver.

The reason some form of global solver is important, as opposed to sequentially solving each constraint, can be best described with the following example. Consider Figure 1. We have two balls trailing behind a box, constrained together by distance constraints (c1, c2).

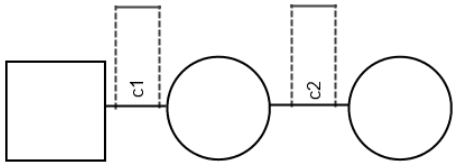


Figure 1: A box and two balls with two Constraints.

Now when we solve the constraints individually we can see that as the second constraint (c2) is solved, it itself invalidates the first constraint (c1).

So in order to truly solve this problem we would have to make a constraint that solved both c1 and c2 at the same time. However, what if this problem was extended to have 3 or more balls trailing

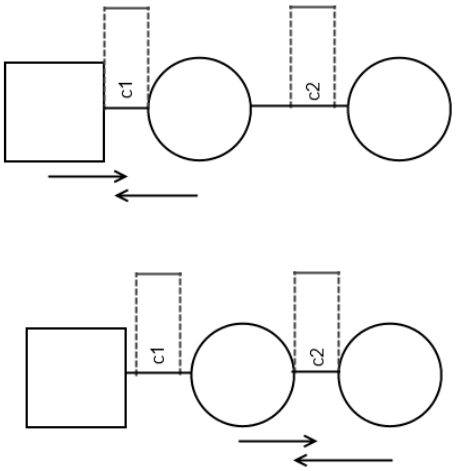


Figure 2: The difficulty in solving either constraint by itself.

behind the box? Rather than seeking to write bespoke constraints which take in more and more coupled objects, we can instead try to construct a global solver to produce a result that satisfies all individual constraints in the system at once as best we can.

Solving Linear Equations

You will recall from school that a set of equations with interdependent unknowns can be considered a set of linear equations. A simple example might be:

$$\begin{aligned} 4x + y &= 23 \\ x - z &= 6 \\ 2z + y &= 3 \end{aligned}$$

One approach to solving this problem is substitution, whereby you rearrange the equations to isolate one variable, resolve it, and the use that result to resolve the other variables. While this approach is very easy to comprehend, it becomes ungainly as the number of unknowns increases.

In the case of our problem, the unknowns are the resultant motion of each entity (this is where the interconnection comes in - how far object A moves is dependent upon the motion of object B, which might itself be dependent upon the motion of objects C and D, and so forth). Since we've already said that we don't want to construct bespoke solutions for each configuration of entities, and we acknowledge there could be many more than four or five entities in our environment (and probably will be), we need a more scalable and programmatically friendly approach.

The alternative approach, which is more computationally efficient, is known as the $\mathbf{Ax} = \mathbf{b}$ form. In this approach, you restructure the equations into a matrix - so the equations from our example adopt the form:

$$\begin{bmatrix} 4 & 1 & 0 \\ 1 & 0 & -1 \\ 0 & 1 & 2 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 23 \\ 6 \\ 3 \end{bmatrix}$$

$\mathbf{A}$ is the coefficient matrix (so the 3x3 in our example), $\mathbf{x}$ is the solution vector (containing the unknown variables x, y and z), and $\mathbf{b}$ is the constant vector (containing the 'answers' to the three equations in our system). The height of the coefficient matrix is always defined by the length of the

solution vector, and the width of the coefficient matrix is always defined by the length of the constant vector (as in our example, we substitute in 0s for any coefficient where the variable is unused for a given equation, e.g. 0 for x in the third equation).

For any system of i equations with j unknowns, the general form resembles:

$$\begin{bmatrix} a_{11} & a_{21} & \cdots & a_{i1} \\ a_{12} & a_{22} & \cdots & a_{i2} \\ \vdots & \vdots & \ddots & \vdots \\ a_{1j} & a_{2j} & \cdots & a_{ij} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_j \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_i \end{bmatrix}$$

There are numerous computational methods for solving linear equations in the $\mathbf{Ax} = \mathbf{b}$ form. Some notable computational methods are:

- Jacobi Method - this approach is parallelisable, but is slow to converge, which can negate some performance benefit of parallelisation
- Gauss-Seidel - this is the simplest to implement, and the approach taken within the basic framework you have been provided
- Successive-Over-Relaxation - this is a faster convergence variant of Gauss-Seidel
- Conjugate Gradient Method - this approach is complex, but is very quick to converge
- Many more besides...

For this tutorial - and to get you thinking about how you can simplify global solvers to fit inside your physics engine - we will be implementing the Gauss-Seidel Method. In its simplest form, this iterates through each row of the $\mathbf{A}$ matrix (object in our physics engine), solving any and all constraints relating to that object.

This generates a value which indicates how much the object's velocity should change to become 'closer' to likely solved solution. In this sense, the object is tending towards consistency with its multiple constraint requirements, rather than waiting until it has definitely achieved consistency (which is infeasible, given the ever-changing nature of the system coupled with its requirement for real-time computation).

In our physics engine, our constraints already automatically update each object's velocity, so this nicely just translates into:

```
1 do(Until all constraints satisfied):
2    for(all constraints):
3       compute total error
4       update velocities to compensate errors
```

So, assuming our global matrix is diagonally dominant and/or positively definitive - meaning all objects have non-zero mass - it will be solvable, though the time it may take to converge on a solution is undefined.

Imagine a wall with infinite mass constrained to another wall with infinite mass - we certainly don't want to wait infinite time for that to solve itself! For this reason, we set a maximum number of solver iterations, instead of waiting for the solution to be within a given threshold of the actual solution.

The reason we do not concern ourselves with waiting for convergence is that our physics engine is considered *eventually consistent* due to strong frame coherency. This concept will be revisited in later tutorials on networking. Simply put, this means that if our system is not completely solved in one frame it will have a chance the next frame to solve it again. One way to think of this is that our system is always 'playing catch-up' - and that if every object in our environment came to rest, it

would eventually converge.

One important note about the Gauss-Seidel Method (and Successive Over-Relaxation for that matter) is that the order of solving constraints matters! To converge towards the same answer, each iteration of the algorithm must iterate over the constraints in the same order every time. At the moment this just means we iterate over the list of constraints with the same for-loop each time, though it's something to keep in mind for the coursework if you fancy optimising the solver or any of its components. Yes... randomising the order of manifolds each physics time-step may lead to more accurate global solver with the aforementioned linear solvers. I'll leave it to you figure out why that may be.

Preventing Constraint Drift

We recall from the beginning of this tutorial series that our computed solutions for one frame are based on the solutions of previous frames - that our simulation is a time series. This, coupled with the eventually consistent nature of our solver, means that errors creep in over time, and that those errors feed forward to make the next computation 'even more erroneous'.

For instance, the distance constraint we built in an earlier tutorial can attempt to make the two objects share the same velocity around the constraint normal, but what would happen if one object was physically moved in a single frame. As our constraint is only based of velocity, it will continue to solve the constraint at its new distance. In fact, as it is only concerned with velocity it will never even know it has been extended. This can occur due to computational errors or due to approximations when solving the set of linear equations (as discussed above). The proof of these errors (known as constraint drift) are easily seen if objects are stacked; unless it is addressed, they will generally sink into each other.

Baumgarte Stabilization describes adding additional correctional forces to a system. These help compensate for previously accumulated errors. In our physics engine, this is managed through the addition of velocity based off the length of our distance constraint. This *eventually* compensates (in the same way that our system is *eventually* consistent) for any potential positional drift.

The amount of energy/velocity to be added to the system is not readily predictable - it isn't something you can easily compute (if it were, the offending error likely wouldn't be there in the first place). As such, this will usually require some fine-tuning specific to your own simulation to get correct.

If you adjust the velocities too little, the constraints will continue to drift. Too much (enough velocity to correct the positional error in a single timestep), and the system will explode - feel free to try it! Usually a factor of between 0.1-0.3 times the velocity needed to solve the positional error in a single time step works well, though this is dependant on many factors. You will have to find a value that works well for your simulation.

Implementation

We've now covered everything relating to real-time physics in the module. This final element, the Solver, is all that's required for you to grasp the framework in its entirety and begin exploring the coursework specification. Look at the tasks for today, explore the solver code, and consider how you intend to go about meeting the coursework specification.

Tutorial Summary

We have expanded the concept of the physics engine. We have introduced the concept of constraints as the basis for interactions between game items, and extended this to the detailed discussion of a generalised, constraint-based global solver.

```
PhysicsEngine::UpdatePhysics()
{
    //A whole physics engine in 6 simple steps =D
    //1. Broadphase Collision Detection (Fast and dirty)
    //2. Narrowphase Collision Detection (Accurate but slow)
    //3. Initialize Constraint Params (precompute elasticity/baumgarte
    //  factor etc)

    for (Manifold* m : manifolds) m->PreSolverStep(updateTimestep);
    for (Constraint* c : constraints) c->PreSolverStep(updateTimestep);

    //4. Update Velocities
    //5. Constraint Solver     Solve for velocity based on external
    //  constraints

    for (size_t i = 0; i < SOLVER_ITERATIONS; ++i)
    {
        for (Manifold* m : manifolds) m->ApplyImpulse();
        for (Constraint* c : constraints) c->ApplyImpulse();
    }

    //6. Update Positions (with final 'real' velocities)
}
```

PhysicsEngine.cpp

```
// Changes here are needed to ensure that:
// 1. The Collision never solves itself backwards (e.g. pushes itself
// all the way through the other side of the other object). As now if
// an object is colliding on both sides it could theoretically solve
// it by doing the above scenario.
// 2. The Friction force never exceeds the force applied by the main
// collision force. As now the main collision force will change over
// time.

// Add after float jn=
// and before jn = jn / constraintMass;

float oldSumImpulseContact = c.sumImpulseContact;
c.sumImpulseContact = max(c.sumImpulseContact + jn, 0.0f);
jn = c.sumImpulseContact - oldSumImpulseContact;

// Add after float jt=
// and before jt = jt / frictionalMass;

// Stop Friction from ever being more than frictionCoef * normal
// resolution impulse. Similar to above for SumImpulseContact, except
// for friction the direction of friction is potentially changing each
// time we solve the contact (as other objects are bouncing this
// object in different directions). So instead of a scalar, we need
// to clamp the total tangential impulse vector to the normal
// resolution impulse. This way, we can solve friction in any
// direction during the solving step without having to precompute and
// check a single tangent vector.

Vector3 oldImpulseFriction = c.sumImpulseFriction;
c.sumImpulseFriction = c.sumImpulseFriction + tangent * jt;
float len = c.sumImpulseFriction.Length();
```

```
33
34 if (len > 0.0f && len > c.sumImpulseContact)
35 {
36    c.sumImpulseFriction =
37       c.sumImpulseFriction / len * c.sumImpulseContact;
38 }
39
40 tangent = c.sumImpulseFriction - oldImpulseFriction;
41 jt = 1.0f;
```

Updating SolveContactPoint in Manifold.cpp

A Final Note on Improving Accuracy

As you have probably realised by now, a lot of making a 'stable' physics engine is about either solving without errors entirely (ideally) or dealing with errors in a way that doesn't explode the rest of the system. So we have to accept the fact that our solver still has errors, partly because we don't run it until it is fully solved but instead for a fixed number of iterations, and partly because of the way we solve constraints. Although, as you can hopefully see, the errors involved are very small.

However, if you watch the stacking scene example with the above solver code you may see the stack very slowly drift and fall over. This is because those tiny errors are all being solved in the same way each collision, and each collision (on the pyramid example) is colliding every frame, so the result is a consistent error in the same direction every frame. To alleviate this, we can randomise the order in which we solve the contact points in the manifold, and also randomise the order we solve the manifolds/constraints each frame.

For those of you keeping up with the above notes on Ax=B matrix form, this is fine as long as we don't randomise anything inside the solving of the above equation. It is equivalently randomising the row order.

```
1 // ... Directly after the Narrowphase call:
2
3 std::random_shuffle(manifolds.begin(), manifolds.end());
4 std::random_shuffle(constraints.begin(), constraints.end());
```

PhysicsEngine::UpdatePhysics()

```
1  void Manifold::PreSolverStep(float dt)
2  {
3     std::random_shuffle(contactPoints.begin(),
4        contactPoints.end()); //NEW
5
6     for (ContactPoint& contact : contactPoints)
7     {
8        UpdateConstraint(contact);
9     }
10 }
```

Manifold.cpp