Physics - Constraints and Solvers

Introduction

In the tutorial series so far, we've seen how to apply forces to rigid bodies, to produce both linear and angular motion. We've also used collision detection to see if objects are overlapping, and used collision resolution to separate them back out again. So far, our objects have been otherwise completely separate from one another - we've never created an object that is attached to another one in some way, like the ball bearings of a Newton's cradle attached to the frame with a wire.

To enhance our physics engine to support connected rigid bodies, we will in this tutorial be investigating *constraints*, and *solvers*. Constraints allow us to formally define connections between objects in terms of an *equality equation*, and allow us to apply impulses to keep rigid bodies a certain distance apart, or a certain relative orientation to each other. Using these we can simulate ropes, hinges, and ball & socket joints. As more constraints are added between objects, we will find ourselves with cases where one rigid body is being manipulated by *multiple* constraints, which pull objects in different ways. These combinations of constraints must be calculated and made correct using a *solver* - something which takes in one world state, and applies the correct calculations to maintain the correct motion of our objects with as little error as possible.

Constraints

You might think that the answer to this problem is just to move the objects away from each other a bit, by changing their position. While this would work, its not particularly physically accurate. Doing this doesn't take into consideration the masses of the objects, or the directions they are currently travelling in. We'd essentially be 'teleporting' the objects, something which might mean that they move through objects they shouldn't be, and will effectively add energy to the object (if we move an object from point A to point B instantly, it won't have had as much drag or friction applied to it as if it got to point B under its own forces, so it can now move further than it should have as these forces are expended).

To take into consideration all of these things, constraints are usually maintained by instead manipulating the derivatives of position; that is, either the velocity and acceleration of the object. So rather than 'teleporting' to a position/orientation that satisfies the rules of the constraint, we instead apply forces or impulses to quickly adjust the object, so that the changes go through our physics system and maintain consistency as much as possible.

Constraints are incredibly useful when writing physics code. Game physics is all about resolving interactions between objects in some physically accurate way, whether they be collisions between objects, a constant amount of force being added towards a particular point, or a rope tying two objects together; all of these things and more can be written as a constraint. In a way, just as shaders are the building block of adding unique graphical features to our game, constraints are the method by which we add unique physical features to our game.

Degrees of Freedom

As the name would suggest, a constraint will constrain, or limit, our physics object in some way. Constraints will typically in some way need to adjust the orientation or position of an object - we call these modifiable aspects *degrees of freedom*, so our objects start with 6 degrees of freedom (sometimes referred to as *6DoF*) - the x,y and z axis positions, along with the rotations around the x, y and z axes. The job of a constraint is to in some way limit each of these degrees of freedom's ability to

change, such that overall, the constraint is considered 'satisfied'. Note that although we've so far been using quaternions to store a rotation, and thus using 4 values, that's not '4 degrees of freedom' - quaternions are tool for storing the 3 axes of rotation efficiently and in a way that lets us transform and store them efficiently, but unlike a position, where every value is entirely separate, the values of a quaternion are all related, and just changing one won't just change 'one axis' the way a position does.

Constraints as an equation

During the last two modules, we have on occasion seen the plane equation, usually formulated like so:

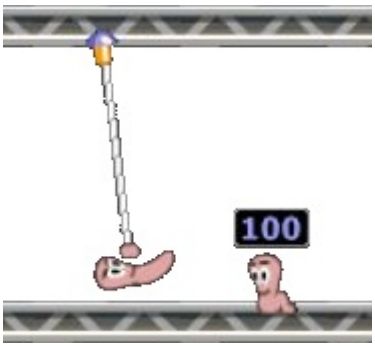
$$ax + by + cz + d = 0$$

This is saying that for any point (xyz), if the result is 0, then the point is on the plane. We can then further say that if the value is greater than 0, the point must move along the plane's normal to reach the plane, and if it less than 0, the point should move the other way along that normal to reach the plane. Why is the plane equation being brought up again? Broadly speaking, the same logic for moving that point to sit exactly on a plane, is how our constraints work - we usually have some sort of condition that we want to equal 0. When dealing with rigid bodies, those constraint conditions will generally be related to either the position or orientation of the rigid body, and we want to change them so that their difference from what the constraint says they should be is 0 - if the properties of the rigid body match up to a constraint, we say that the constraint is *satisfied*

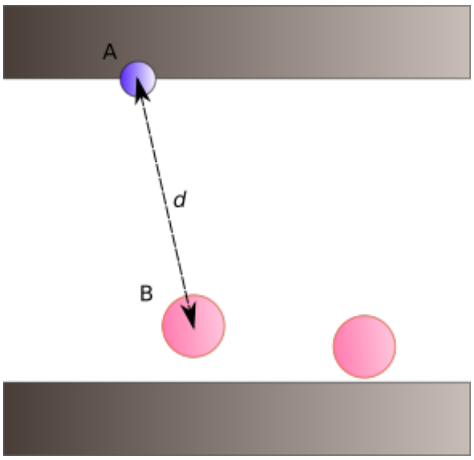
Distance Constraint

The classic example of a distance constraint in a game is in character 'ragdoll' physics - when an enemy 'dies' they collapse in a heap and can be pushed around by the player, or roll down hills and fall off platforms with amusing results. Under the hood, we can imagine the normal skeletal animation system of the enemy has been replaced by physics calculations, with each bone in the skeleton given a mass and a collision volume just like a 'normal' rigid body. To prevent the limbs then flying off or otherwise becoming disconnected, they are connected with constraints on their position - their orientation can change freely, but their position relative to their 'parent' connector cannot change, keeping the body together, but letting it flop about. In this case, we're really describing a constraint that limits the degrees of freedom to 3 - allowing the orientation around the 3 axis to be changed at will, but locking the 3 positional axes completely. Such a constraint is sometimes known as a 'ball and socket' joint, as it mimics its real life counterpart.

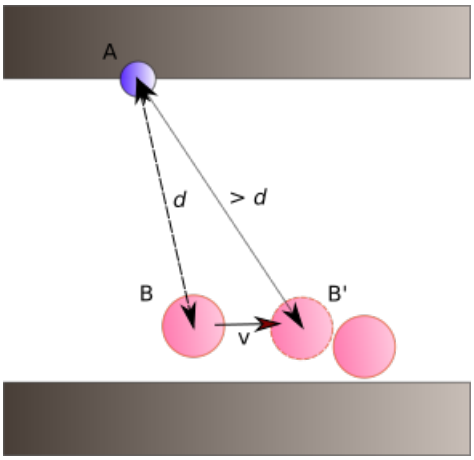
To examine distance constraints in a bit more detail, let's consider a fairly simple constraint that we might see in games - the ninja rope utility as seen in *Worms*:



Here, the player has fired a rope which has collided with the ceiling (the raycasting techniques we saw earlier in the tutorial series could be used to find a suitable surface for the rope's attachment), allowing the player's worm character to swing around, and eventually release the rope and launch themselves to another part of the level. If we take more a 'game logic' view of what's happening, we might have something like this:



From the point of view of the data structures and techniques we've seen so far on this tutorial, we could replicate this scene with a couple of AABB rigid bodies with an inverse mass of 0 to represent the floor and ceiling girders, and a couple of sphere rigid bodies for the player characters. To implement the ninja rope, we want to add a distance constraint, between point A and our player character B, with a maximum length of d. To make things a little easier to follow, let's assume that point A is a temporary rigid body with an inverse mass of 0, so that it also cannot move. When the player moves their character left or right, eventually they will be at a distance of greater than d from the connected point A:



To put it another way, at some point, the player will apply a force that results in a velocity that would break the constraint, if it were to be integrated into the player character's position. Therefore, to not break the constraint, we have to change that velocity so that when it is integrated, the constraint of being distance d away from rigid body A is still maintained, or *satisfied*.

We can represent this constraint more generally with the following constraint equation:

$$P = ||B - A||$$
$$C = P - d$$

We will usually see our constraint noted as being C in this way, and we can see that following through the calculation that if object B is exactly distance B away from A, the result of C will be 0, and thus satisfied; if however, it is **not** the same, the constraint will in some way be broken, by either a positive or negative amount.

We can say that $P - d$ represents the *displacement* of the objects in *constraint space*, a coordinate system where the origin represents a 'satisfied' constraint. Therefore, to satisfy the constraint, we must in some way modify the first derivative of the constraint. As P-d is formed from the object's position, we can therefore say that the first derivative of C (which we'll denote $\dot{C}$) is formed from the first derivative of the object positions; their *velocities*. To satisfy the constraint, we must in some way *transform* the object's relative velocities, to make sure that once integrated into the object's position, they move them closer to 0 in the constraint's space. As the velocities are vectors, we can transform them using a matrix:

$$\dot{C} = J \cdot v$$

This matrix J is known as the *Jacobian* matrix, which more formally defines the partial derivatives of a vector valued function - in this case the function is changing the constrained object's position vectors by v over time. We generally don't want to lose the velocity being applied (we want our objects to conserve their momentum when possible), but transform it such that it satisfies the constraint while still behaving physically consistently - this is why in a constraint, we don't just set the positions such that the constraint becomes satisfied.

Let's take a closer look at what the equation above is formulated. The vector value v is our constrained object's velocities (we're dealing with the *first derivatives* of our constraint):

$$v = \begin{bmatrix} v_A \\ v_B \end{bmatrix}$$

Our Jacobian matrix is therefore what the change in one object's velocity will do to the other object's velocity if the constraint is to remain satisfied:

$$J = \begin{bmatrix} V_A - V_B \\ V_B - V_A \end{bmatrix}^T$$

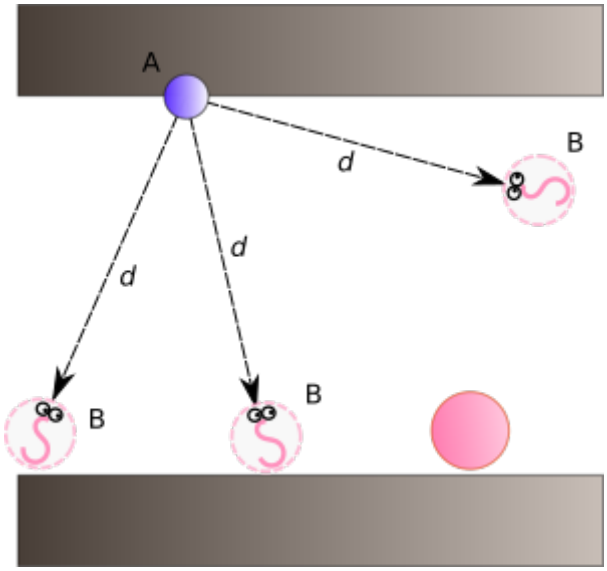
It's common to see a Jacobian denoted as being a *transpose*, as above; this is because to transform our 2 x 1 matrix v, we need a 1 x 2 matrix - which is the same as a 2 x 1 matrix transposed. In the wider field of vector valued functions, Jacobians can be tricky to *look* at (there's lots of the *del* operator: $\frac{\partial u_1}{\partial x_1}$), and hard to *visualise* (that is, what they're trying to represent). Fortunately, we're dealing entirely with concepts we're very familiar with by now - positions in 3D space, and velocities that move through that 3D space; if we think of our constraint transforms this way, then our Jacobian just boils down to how much the position and orientation of one object affects the position and orientation of another object when trying to keep the constraint satisfied.

Hinged Constraint

It's not just the relative position of an object we might want to constrain, but sometimes the relative orientation, too. Consider the case of the front door of a house - we want the door to be able to swing outwards to open up and let us in, but we don't want it to fall over, or be lifted up and taken away. In real life, this is achieved by connecting the door to the doorframe via a *hinge*, which can angle in and out to allow the door to move. In a simulated environment like a game, we would usually have a wall object with a door-shaped 'hole' in it (or multiple objects making up a wall, each of which can

then be fully convex), a 'door' object, and then a constraint between the two to allow some rotational changes, but no positional ones. In other words, we're limiting the degrees of freedom to 1 - the 'yaw' of the door; all other changes are disallowed.

We can continue our *Worms* example to see how a hinge constraint might work. Perhaps instead of a maximum rope distance d, the Ninja rope is instead constraining the worm so that its local 'up' direction always faces its attachment point, as in each of the potential locations B for our worm here:



Broadly speaking, the worm is our door and point A is our hinge - the concept is exactly the same, in that we're constraining the relative orientation; although of course, our door would also have its *position* constrained, too!
Our constraint calculation starts off looking exactly the same as in the previous example:

$$\dot{C} = J \cdot v$$

The difference is this time, v contains the *angular* velocities of the worm and the attachment point:

$$v = \begin{bmatrix} \omega_A \\ \omega_B \end{bmatrix}$$

Therefore, the Jacobian matrix for our constraint is then:

$$J = \begin{bmatrix} (\omega_A - \omega_B) \times n \\ (\omega_B - \omega_A) \times n \end{bmatrix}^T$$

In this case, we're trying to find an axis by which to rotate the orientation around so that the up and down axes of the worm and its attachment point coincide.

Collision resolution as a constraint

In our distance and orientation constraints, we were formulating them as an equation that was satisfied when it became 0. If we think back to the collision resolution we did earlier in the tutorial series, we actually needed to do something similar to a distance constraint - we had to force objects that are intersecting one another to be at a distance of at least 0 away from each other, taking into consideration the intersection depth. In other words, we were forming a temporary distance constraint that will in some way change the positions of the objects such that they satisfy a constraint upon their positions such that their volumes do not overlap. However some intersections are also solved by changing the orientations of objects - an OBB falling at an angle onto the floor will get rotated such that it lies flat on the floor. So as well as being a constraint on the object's *positions*, we can model collision resolution as also being a constraint on the object *orientations*.

Let's break this idea down to see how the constraint ends up being formulated. If we assume our now-standard constraint calculation:

$$\dot{C} = J \cdot v$$

Our resolution constraint would have a v vector as follows:

$$v = \begin{bmatrix} v_A \\ v_B \\ \omega_A \\ \omega_B \end{bmatrix}$$

We saw in the earlier tutorial that we resolve collisions by trying to add forces along the *collision normal*, which we'll denote n. We also assumed the objects were colliding at a specific point p in the world. To encode the same collision resolution behaviour in a constraint, we end up with both n and p inside our Jacobian matrix:

$$J = \begin{bmatrix} -n \\ n \\ -(p - s_A) \times n \\ (p - s_B) \times n \end{bmatrix}$$

From there, we get the following full constraint calculation:

$$\dot{C} = \begin{bmatrix} -n \\ n \\ -(p - s_A) \times n \\ (p - s_B) \times n \end{bmatrix} \begin{bmatrix} v_A \\ v_B \\ \omega_A \\ \omega_B \end{bmatrix}$$

To solve our object intersection constraint calculation, we need the penetration distance to become zero, so the derivative of a constraint must make the answer *approach* zero, meaning we have to form a Jacobian matrix that takes in our colliding object's first derivatives, and changes them to suit our constraint - we've really been doing this particular form of constraint in a limited form since the collision resolution tutorial, its just now we're seeing it in a slightly differently formulated way. Our object positions should be separated out along the collision normal, and our object orientations should change by an angular change proportional to the collision normal and the relative position of the collision - we saw in the collision resolution tutorial how taking the cross product of these provides the axis of rotation for that object.

In this particular constraint we saw that the partial derivatives of the constraint might end up with extra data in them (in this case the collision normal and the relative position). We can think of this as being the programmable element of the constraint - different constraints will have different Jacobian matrices, and may require the calculation of additional information to properly formulate in a way such that they can be satisfied correctly.

Lagrange multiplier and constraint force

The basic formulation of a Jacobian matrix doesn't change for a particular 'type' of constraint - the collision resolution constraint above will *always* have a Jacobian matrix that transforms the positions and orientations of two objects, based on the collision normal. What we haven't seen so far is the *magnitude* of how those velocities should change, only which *direction* they should change in. To correctly satisfy our constraint, we must therefore scale the Jacobian in some way so as to make it result in C = 0 when applied. This scaling factor is known as the *Lagrange multiplier*, often denoted as λ (and so sometimes just referred to as 'lambda', too). We know our Jacobian, and we know our input variables, we just don't know (yet) the correct values for λ, giving us an equation that we must in some way *solve*.

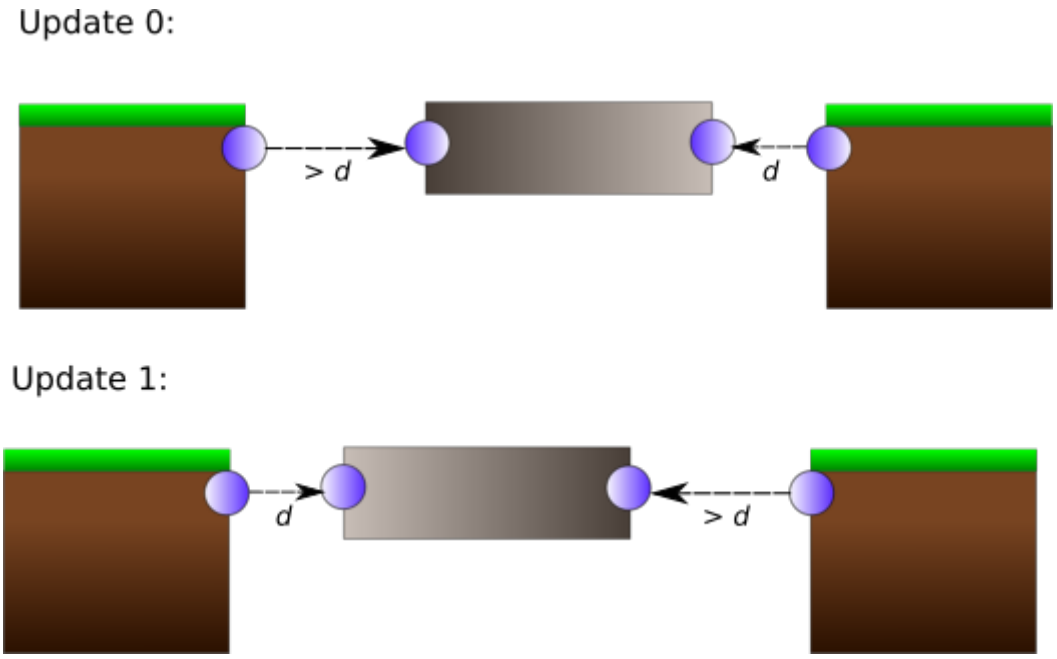
The true purpose of a constraint is to add some forces to our objects so that they end up in a state that satisfies that constraint. We can therefore think of our constraint as being the process of determining the linear and angular *constraint force* to apply to our objects, in the form of:

$$F = J^T \lambda$$

Solving systems of constraints

It is often the case in video games that we may want to have multiple constraints operating on the same objects. For a quick 'game' example, think of a small rope bridge, of boards linked together with distance constraints, as in the image below:

If any forces are enacted upon one of the links, the rest will be pulled around by the constraints, creating a suitably wobbly rope bridge, at least in theory. What is more likely to happen is that even without any forces directly applied, the bridge links wobble about anyway, and might even start to build up velocity in an unrealistic manner. To understand why this is so, we need to think about how our constraints are satisfied. If we have a list of constraints, and solve the constraint forces for each separately, our rope bridge will misbehave:



If we let the first distance constraint calculate its result, the bridge pieces moves left, causing the constraint on the right to become violated. If we calculate the constraints the other way around, we instead violate the left constraint. What do we do to minimise the error in our two constraints? To fully integrate constraints into a physics engine, not only does the constraint force have to be determined for one constraint, but for an entire *system* of constraints simultaneously.

Calculating λ

When dealing with interactions between our rigid bodies, we've in previous tutorials seen how this can be achieved by modifying a body's position, its first derivative of position (velocity), or its second derivative of position (acceleration). Therefore, to satisfy our constraints, we can do the same (along with the orientation and its derivatives, of course). Generally though, constraints are solved with changes in velocity, for much the same reason as the collision resolution we saw earlier - we can easily conserve momentum, and we quickly converge upon a correct answer (springs might let a constraint remain violated if they are not encoded correctly for their k value, for instance). To solve our constraints, we generally then determine which *impulses* to apply to our constrained objects - an impulse is just a force over time, so it maps easily onto our idea of a constraint force. Knowing this, solving

any constraint's λ is just to calculate the correct impulse to accommodate for the mass of the objects, in the correct direction. Our Jacobian matrix already contains the correct *direction*, so our Lagrange multiplier needs to just work on object linear and angular velocities, and how much those velocities map onto solving the constraint, divided by the mass of the constraint.

For our distance constraint, where only the object masses matter (rather than the moment of inertia at a particular point), we get a final λ like so:

$$\lambda = (\frac{1}{{m_a}^{-1} + {m_b}^{-1}})J \cdot v$$

The multiplication of our object vector v and our Jacobian matrix v gives us the relative velocity along the axis that will solve the constraint (for a distance constraint, that'll be the direction vector between the objects), and by scaling by mass, we get a resulting λ value that, when applied in that direction as an impulse (which then takes into consideration *inverse* mass), we get a correct physically accurate response that conserves momentum - you should remember adding object masses together from when collision resolution was added to the codebase in an earlier tutorial, and this is no coincidence, as that impulse can be seen as a single frame's worth of a constraint between the two objects.

We can calculate the same result for λ by forming a *mass matrix* for the constraint rigid bodies like so (assuming the constraint is to take both linear and angular velocity into account):

$$M^{-1} = \begin{bmatrix} {m_a}^{-1} \\ {m_b}^{-1} \\ {\mathbf{I}_a}^{-1} \\ {\mathbf{I}_b}^{-1} \end{bmatrix}$$

From there, we can determine λ as follows:

$$\lambda = (\frac{1}{JM^{-1}J^T})J \cdot v$$

This allows us to encode all of the properties of our constraints as a set of matrix multiplications in a generic form - as long as we form the correct Jacobian, and scale the result by the correct mass matrix, our resulting constraint force will be of sufficient magnitude to bring our objects towards a constraint satisfaction state.

Global vs Iterative solvers

Roughly speaking, we now know how to solve any single constraint - we scale its Jacobian by the constraint mass to determine some impulses to apply to our objects. But what about when there's multiple constraints? In the example earlier we saw how solving any one constraint violates the other, so one solution is to calculate all of the constraints together in one single large calculation. To do so, we'd need an expanded v vector containing all of our constraint rigid body information for constraints 0 to n, and a larger n x n matrix of all of constraint Jacobians down the diagonal.

So far so good - by encoding constraints in Jacobian matrix form, they are easily combinable into larger matrices. However, each constraint can invalidate another, so they are not entirely independent - in matrix form that means we don't just have a diagonal matrix of Jacobians, but a mapping of the *changes* of those Jacobian combinations - more derivatives! Also, as the number of constraints grows, the size of this combined Jacobian matrix goes up by the *square* of the increase; as each Jacobian is itself a matrix with a number of elements, it can quickly be the case that a combined matrix has 1000s of entries, many of which are just zeroes (for constraints that don't affect each other), but which still need processing. Together, this is known as a *global solver*, and while it will produce the correct answer, it can't do so quickly, and its not even guaranteed that there *is* a solution - if two distance constraints both say that object B needs to be 20 units from object A or B, but A and B are 5000 units apart, those constraints will never be satisfied anyway, and trying to solve them as part of a larger whole is going to cause other dependent constraints to go wrong along with them.

Instead of a global solver, most video game physics systems will instead use an *iterative* solver for its constraints. In an iterative solver, each constraint is looped over and solved on its own, with the results fed back in to the next iteration of the solver, so that over multiple iterations the constraints that influence multiple objects converge towards all being as close to being satisfied as possible. This has the benefit of allowing each constraint to be calculated in the simplest way possible, rather than plugging in generic values into a global solver - for example, in our distance constraint we can use vector multiplication rather than matrix multiplication if we know that the velocity in one axis has no bearing on another axes' velocity, and will never influence any other constraint directly (only indirectly through the iterative process).

As the results of our constraint solution involve the calculations of impulses, which change the velocities of rigid bodies, the calculations that each iteration of the solver make must be based on the velocity - as we'll see in the code later, this is no problem, as our constraints operate on velocity anyway. The most common form of iterative solver used in physics engines is known as *Sequential Impulse*, which is in turn a form of *Projected Gauss-Seidel*. Whatever we call our iterative solver, the general form is the same:

```
while(!finished) {
   for(each constraint in system) {
      Calculate Jacobian
      Calculate lambda
      Apply resulting impulse
   }
}
```

Iterative solver pseudo code

As each iteration results in different impulses, the next iteration of the solver sees different relative velocities (either from that same constraint's previous iteration, or as the result of some other constraint operating on the same objects), and so different ways in which the constraint is being violated, so that after a number of iterations, the objects are pushed towards an acceptable answer for their linear and angular velocity. The exit condition for the loop above might be a simple maximum number of iterations, or each constraint could become inactive for that frame after a given threshold of closeness is reached - a maximum number of iterations is recommended for the same reason as the global solver example above, in that there could be some constraints that simply cannot be satisfied for whatever reason.

Adding bias

While most constraints are solved via their first derivative, the solution might not actually move the constraint that close towards being satisfied. For instance, the impulses applied to keep a distance constraint together might be slowly beaten by gravity, resulting in the constraint becoming more violated over time. The solution to this is to in some way bias the results of the constraint calculation in some way:

$$\dot{C} = J \cdot v + b$$

A common method of adding bias is *Baumgarte stabilisation*, which takes the following form:

$$b = \frac{\beta}{dt} C$$

Where β is a bias factor within the range 0 to 1 (usually very close to 0). This will take the current amount our constraint C is being violated by, and add it to our components of v, artificially inflating them, and thus causing the resulting impulses to be larger the more the constraint is violated by. By adding this in to each iteration of the iterative solver, any error that occurs due to external forces or numerical instability will hopefully be minimised, and the system correctness as a whole maintained as much as possible.

Tutorial Code

To see a practical demonstration of constraints in action, we're going to make a typical use case for a persistent constraint - we're going to make a simple 'rope bridge' out of cubes, connected with distance constraints. The ability to add constraints to the game world is already present in the codebase - the **GameWorld** class stores a collection of pointers to *Constraint* objects, which are declared as follows:

```
1  namespace NCL {
2     namespace CSC8503 {
3        class Constraint  {
4        public:
5           Constraint() {}
6           ~Constraint() {}
7
8           virtual void UpdateConstraint(float dt) = 0;
9        };
10    }
11 }
```

Constraint class header

There's not much we can say about what a constraint does other than that over time it will try and apply impulses to try and satisfy its requirements, so all the base class has is an *UpdateConstraint* method, which takes in the frame's timestep (or, as we'll see later, a *subdivision* of that timestep to allow the constraint to iterate multiple times, as an iterative solver).

To actually create an implementation of a constraint, we're going to make a new subclass, **PositionConstraint**, with a header file that looks like this:

```
12 #pragma once
13 #include "Constraint.h"
14
15 namespace NCL {
16    namespace CSC8503 {
17       class GameObject;
18
19       class PositionConstraint : public Constraint {
20       public:
21          PositionConstraint(GameObject* a, GameObject* b, float d) {
22             objectA     = a;
23             objectB     = b;
24             distance    = d;
25          }
26          ~PositionConstraint() {}
27
28          void UpdateConstraint(float dt) override;
29
30       protected:
31          GameObject* objectA;
32          GameObject* objectB;
33
34          float distance;
35       };
36    }
37 }
```

Constraint class header

This will let us represent a constraint that tries to maintain a set distance of d between two **GameObjects** a and b. The workings of the constraint are entirely within the override *UpdateConstraint* method:

```
1  void PositionConstraint::UpdateConstraint(float dt)   {
2     Vector3 relativePos =
3        objectA->GetConstTransform().GetWorldPosition() -
4        objectB->GetConstTransform().GetWorldPosition();
5
6     float currentDistance = relativePos.Length();
7
8     float offset = distance - currentDistance;
```

PositionConstraint::UpdateConstraint method

The first thing we need to do is work out if the two objects are breaking the rules of the constraint or not. As this is only a distance constraint, all we need to check is their relative position to each other, and then work out how far off this is from matching the constraint's preset distance variable (line 8).

In the case where the objects are either too close or too far away (either way breaks this constraint, so we can imagine the constraint as being a solid rod connecting our objects), we need to work out how much of the two object's velocities are working against the satisfaction of the constraint (that is, how much of the velocity is resulting in a change in the constraint, which in this case will be almost *any* relative velocity), and use that as a scaling factor for the corrective forces (line 24). To try and come up with a solution for the lagrange multiplier, we're also going to add in a small amount of bias, using the Baumgarte stabilisation method outlined earlier. The *biasFactor* on line 26 is a tuneable variable - different constraints may require different amounts of bias added to make their constraint stable.

We then just add on our bias to the effective velocity, and divide the answer by the constraint mass (line 31) - by dividing by the (inverse) mass, we work out a correct full amount of force to try and apply, as the *ApplyLinearImpulse* method will divide the value back down by that object's actual mass. This means our constraint still obeys the laws of motion - there's an equal and opposite force applied to both objects, and momentum will be conserved.

Note that we never actually make the Jacobian matrix directly, we just construct the equivalent vectors and scale them by lambda. Our distance constraint should have position partial derivatives equivalent to vectors that bring the positions closer together (the normalised vector between the two objects), and as we're trying to solve the constraint using velocity, we then apply impulses, which over the course of the iterative solver, work to satisfy our constraint in a realistic manner.

```
10    if (abs(offset) > 0.0f) {
11       Vector3 offsetDir = relativePos.Normalised();
12
13       PhysicsObject* physA = objectA->GetPhysicsObject();
14       PhysicsObject* physB = objectB->GetPhysicsObject();
15
16       Vector3 relativeVelocity = physA->GetLinearVelocity() -
17                                  physB->GetLinearVelocity();
18
19       float constraintMass =  physA->GetInverseMass() +
20                               physB->GetInverseMass();
21
22       if (constraintMass > 0.0f) {
23    //how much of their relative force is affecting the constraint
24          float velocityDot =Vector3::Dot(relativeVelocity, offsetDir);
```

PositionConstraint::UpdateConstraint method

```
25          float biasFactor  = 0.01f;
26          float bias        = -(biasFactor / dt) * offset;
27
28          float lambda = -(velocityDot + bias) / constraintMass;
29
30          Vector3 aImpulse =    offsetDir * lambda;
31          Vector3 bImpulse =   -offsetDir * lambda;
32
33          physA->ApplyLinearImpulse(aImpulse);//multiplied by mass here
34          physB->ApplyLinearImpulse(bImpulse);//multiplied by mass here
35      }
36   }
37 }
```

PositionConstraint::UpdateConstraint method

TutorialGame Class

To demonstrate our new constraint, we're going to make the classic distance constraint test - a rope bridge. To do this we're going to have two rigid bodies fixed in position using an inverse mass of 0, and have a chain of addition cube rigid bodies between them, each linked with a distance constraint. The code for this we're going to add to the empty *BridgeConstraintTest* method - this can be called inside *InitWorld* much like the *InitCubeGridWorld* that created the very first physics environment we saw back in tutorial 1. Here's how the code looks:

```
1  void TutorialGame::BridgeConstraintTest() {
2     Vector3 cubeSize = Vector3(8, 8, 8);
3
4     float    invCubeMass  = 5; //how heavy the middle pieces are
5     int      numLinks     = 10;
6     float    maxDistance  = 30; //constraint distance
7     float    cubeDistance = 20; //distance between links
8
9     Vector3 startPos = Vector3(500, 500, 500);
10
11    GameObject* start = AddCubeToWorld(startPos + Vector3(0, 0, 0)
12                  , cubeSize, 0);
13    GameObject* end = AddCubeToWorld(startPos + Vector3((numLinks + 2)
14                  * cubeDistance, 0, 0), cubeSize, 0);
15
16    GameObject* previous = start;
17
18    for (int i = 0; i < numLinks; ++i) {
19       GameObject* block = AddCubeToWorld(startPos + Vector3((i + 1) *
20                cubeDistance, 0, 0), cubeSize, invCubeMass);
21       PositionConstraint* constraint =new PositionConstraint(previous,
22                block, maxDistance);
23       world->AddConstraint(constraint);
24       previous = block;
25    }
26    PositionConstraint* constraint = new PositionConstraint(previous,
27                end, maxDistance);
28    world->AddConstraint(constraint);
29 }
```

TutorialGame::BridgeConstraintTest() method

We start by making our fixed objects with infinite mass (line 11 and 13), and then progressively add more objects between them (line 18) - by using the previous pointer we always know the other object to connect our newly added block to. All we have to do is make sure our last box gets constrained to the far side (line 26), or our rope bridge will just flop down due to gravity - that's fine if we're making the swinging rope in *Pitfall*, but for a proper bridge we'll need to attach it properly.

The mechanism for updating each constraint is already present within the *PhysicsSystem::Update* method, like so:

```
1 float constraintDt = iterationDt /  (float)constraintIterationCount;
2 for (int i = 0; i < constraintIterationCount; ++i) {
3    UpdateConstraints(constraintDt);
4 }
5 IntegrateVelocity(iterationDt);
```

PhysicsSystem::Update() method

The *constraintIterationCount* variable starts being set to 10, but is easy enough to change on the fly with the addition of some keyboard checks - you should find that more iterations results in an overall more stable constraint chain, at the cost of more computation.

GameWorld Class

The **GameWorld** class needs a little adjustment - when the world is cleared, the constraints should also be removed, and when erased should also be deleted:

```
6  void GameWorld::Clear() {
7     gameObjects.clear();
8     constraints.clear(); //new line!
9  }
10
11 void GameWorld::ClearAndErase() {
12    for (auto& i : gameObjects) {
13       delete i;
14    }
15    for (auto& i : constraints) {
16       delete i; //new for loop!
17    }
18    Clear();
19 }
```

PhysicsSystem::Update() method

Conclusion

If you've managed to do everything correctly, you should find a line of cubes in your game world. Enabling gravity should cause that line to flop down in the middle, and stabilise into A 'U' shape - the ends are fixed in place due to their inverse mass of 0, but the middle elements are free to fall under gravity, until they are kept together by the constraints adding corrective impulses.

Constraints are the inner mechanisms by which physics engines maintain the stability of the game world, and allow us to define further relationships between our objects, so that they don't necessarily just collide, but in some way cause changes in each other's position and orientation over time. In a way, we've been doing constraints since the collision resolution tutorial, but now we've seen the skeleton of an overall framework for constraints, and how they can be used to define changes in our world's rigid bodies. We've also learned a little bit about solvers - mostly that global solvers are hard, and iterative solvers need a little bit of help via Baumgarte stabilisation to keep the world in a correct

state. With the Jacobian matrix, and relationship between two object's position and orientation can be represented, and from there we can just solve them as part of the overall iterative process.

Further Work

1) The code so far only provides a single 'position constraint' type. Try adding in an additional constraint that also constrains the orientation of the two attached objects. Doing so should then be able to limit the ability to spin around the cubes freely by adding forces to them; the torque will instead be spread out throughout the entire rope bridge.

2) Another variation of constraint can be something which purposefully *adds* force and torque, as a motor effect. This should be able to be achieved as a slight modification of the position constraint (for linear motion) or the above orientation constraint (for angular motion) to modify the distance check to add on an offset, and calculating a new normal value to where the object should be moving to / facing towards.

3) As well as *persistent* constraints, its also possible to make *temporary* ones that fulfil some purpose and are then removed. We can use this to provide a collision resolution constraint, that replaces the current *ImpulseResolveCollision* method with the adding of a **ResolutionConstraint** to the **PhysicsSystem**. You'll need to add some method of determining when a constraint should be removed from the system (in this case, when there's no collision between the objects any more), and a new class to represent a constraint that will calculate the correct impulses to separate out the objects such that momentum remains conserved.