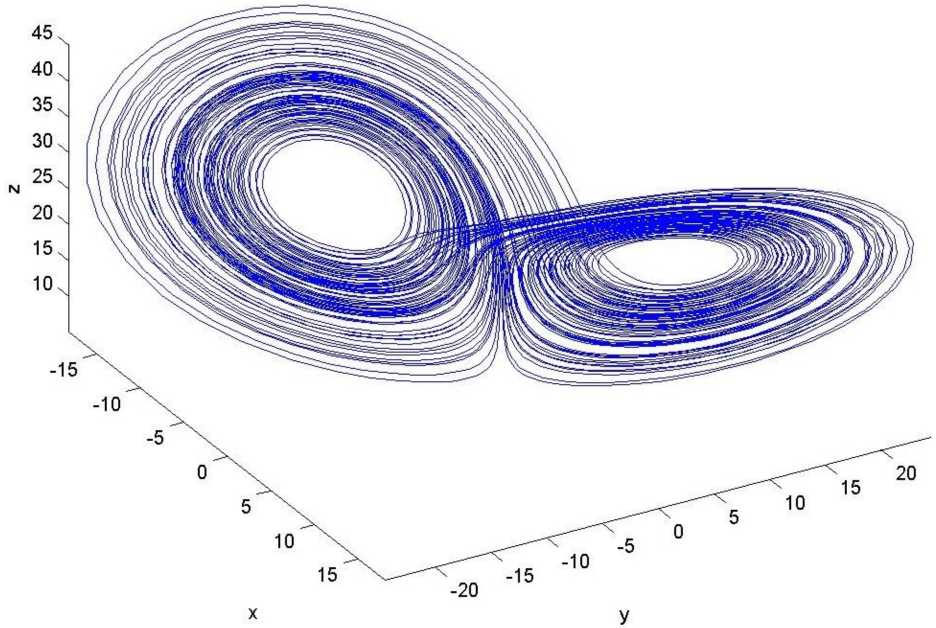
Physics Tutorial 2: Numerical Integration



Housekeeping

- Aardvark Swift competitions (Search for a Star, etc.) are now open and taking applications
- http://gradsingames.com/competitions/
- If you're interested, go for it!

Housekeeping

- Reminder, 2 lectures today
- Second is at 13:00
- Immediately followed by the practical until 16:00
- One lecture tomorrow, 10:00am

New Concepts

- Moving Objects in Physics Engine
- Integration Refresher
  - Linear Motion
  - Angular Motion
- Numerical Integration
- Integration Methods
  - Euler
  - Verlet
  - RK2
- Implementation

Moving Objects with our Physics Engine

- We recall from the previous tutorial that the first thing our physics engine should be capable of doing is updating our object's physical location
- In amongst that, we include our object's physical orientation
- We emphasised the importance of applied forces to compute both of these characteristics
- But how do we take force and convert it into motion?

Moving Objects with our Physics Engine

- Force is the product of mass and acceleration. Its units, therefor, are $kgms^{-2}$ or $kg \cdot \frac{m}{s^2}$ - kilogram-metres, per second squared.
- You'll often hear physicists or other scientists use "metres per second per second" when describing acceleration, instead of "per second squared"
- This is to emphasise that acceleration is a change of a variable in time (that variable being velocity which is, itself, a change of a variable in time)
- As a result, time lies at the heart of our entire physics simulation

Moving Objects with our Physics Engine

- Time is an analogue concept – it isn't digital, or discrete. By its nature, time is a continuum

- But we're working on a digital medium, and everything we do with the computer has to be discrete

- In other words, before we can decide how our system will react to the passage of time, we have to decide just how our system will *represent* the passage of time

Moving Objects with our Physics Engine

- We’ll cover just how we actually do this later in the lecture – but you should keep it in your mind while we discuss the concepts we’re about to address.
- Calculus, yay!
- Specifically, we cover integration. This is because our system is based around force and, as such, acceleration – meaning that we’re moving inwards towards position.
- If, for some crazy reason, our system were computing acceleration reactively based on change in position, we’d differentiate, instead.

A note on Inverse Mass

- You’ll notice in the Physics Framework that we store a property of an object called `m_InvMass` and multiply by it whenever converting a force into an acceleration
- This is, as the sharp-eyed amongst you have already gleaned, the inverse mass of the object (e.g., $\frac{1}{m}$)
- We do this for efficiency’s sake. Multiplication takes approximately half as long as division in cycles – so by computing `m_InvMass` when we first create an object, and multiplying it by force to obtain acceleration, we half the amount of time we spend computing acceleration

Integration Refresher

Integration Overview

- Integration is the process by which we compute how much a variable has changed, based on its rate of change and the passage of time

- For example, someone earns £250 per week:
- $\frac{d£}{dt} = 250$ where $dt =$ one week

- Integrating this very simple example means that, assuming they start with £0, after four weeks they have £1000 ($\frac{d£}{dt} \cdot dt = 200 \ \times 4$)

- How much do they have after 3.5 weeks? Depends on whether or not our system is continuous or discrete

Linear Motion

- The process for resolving linear motion on an object is relatively straightforward:
  - Resolve all forces acting on the object into a single force ($F_{Total} = F_1 + F_2 + \cdots + F_N$)
  - Calculate the object's acceleration ($F = ma$)
  - Integrate acceleration over time to obtain velocity
  - Integrate velocity over time to obtain position

Linear Motion

- Acceleration is rate of change of velocity over time:

$$a = \frac{dv}{dt}$$

- As such, velocity is obtained by integrating acceleration with respect to time:

$$v = \int a.dt$$

- Velocity is rate of change of displacement over time:

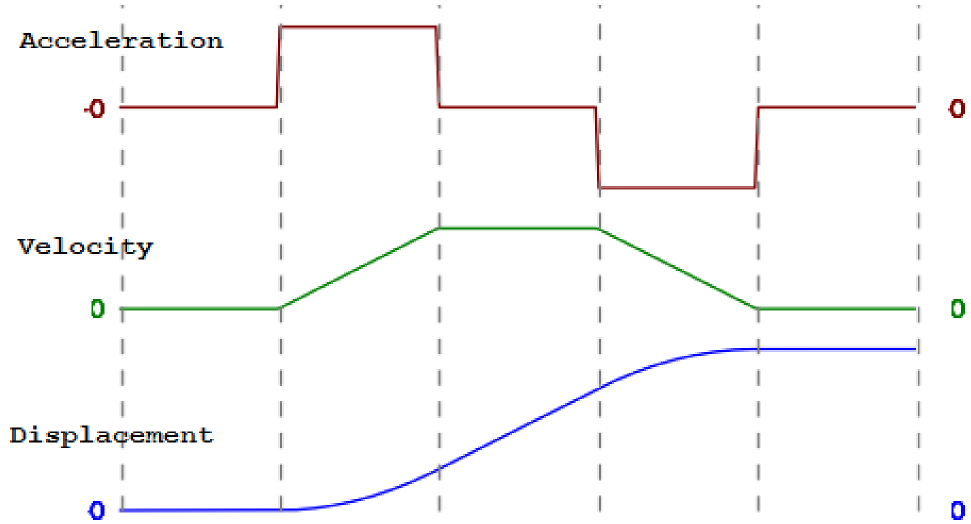
$$v = \frac{ds}{dt}$$

- As such, displacement is obtained by integrating velocity with respect to time:

$$s = \int v.dt$$

Linear Motion

- Consider the figure below
- This illustrates how the three variables interact with one another



Angular Motion

- The process for resolving angular motion on an object is also relatively straightforward:

- Option A: Resolve the net torque acting on an object (this is a bit more complex than net force, and is often overlooked as a result)
- Option B: Handle each torque independently, and it'll come out in the wash

- Calculate the acceleration from ($\tau = I\alpha$)
- Integrate angular acceleration over time to obtain angular velocity
- Orientation needs special care when handled in this manner, see lecture notes for more detail

Numerical Integration

Wibbly Wobbly Timey Wimey Stuff

- Promised we'd revisit the question of 'time'.
- So, because physics in the real world is a continuous process with no discernible discrete interval.
- If it makes you feel better, the smallest recognised unit of time (Planck time), is of the order of a few billion times smaller than the smallest duration technology can detect (and may ever be able to detect: see Uncertainty)

Wibbly Wobbly Timey Wimey Stuff

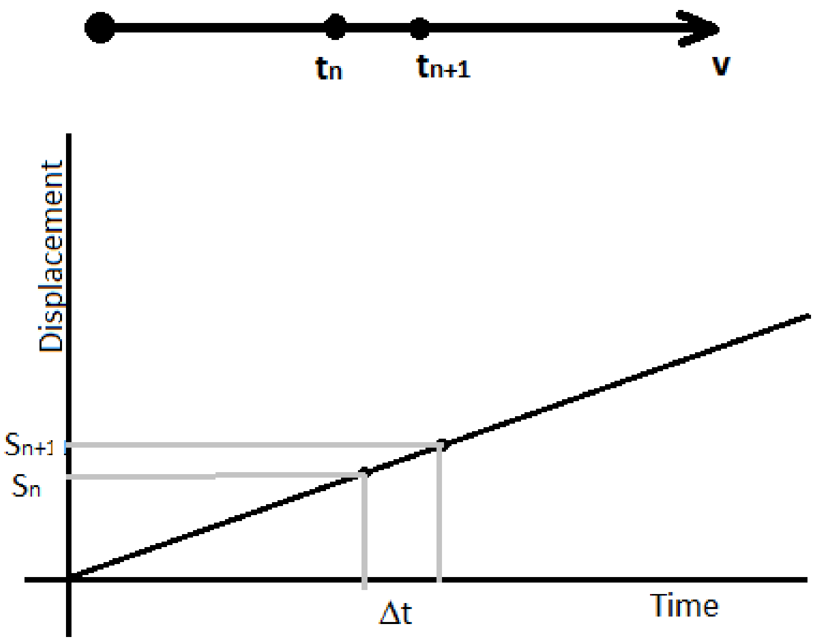
- We can't model things in Planck time ($\sim 1.85 \times 10^{43}$fps)

- Instead, we fix on a time step which will define a discrete interval between updates of our system (normally ~120fps for a physics engine). We call this the time step, and (as in graphics) we refer to a specific time step (t_n) as a frame

- The problem with this approach is that it inherently introduces errors…

Wibbly Wobbly Timey Wimey Stuff

- Consider the diagram below. This outlines in simple terms the displacement of an object over time, moving at a constant velocity:



$$s_{n+1} = s_n + v_n \Delta t$$

Wibbly Wobbly Timey Wimey Stuff

- The error comes in the form of unpredictability. What happens if v suddenly changes between t_n and t_{n+1}?
- This is compounded by the fact that our answer for the object's position at t_{n+2} is predicated on the result for s_{n+1}.
- The concept of the time series comes from here. The time series is the record of an object's state (in our diagram, its displacement s)
- Because s_n is predicated on the result s_{n-1}, for all values of n above 0, we consider this process to be iterative

Wibbly Wobbly Timey Wimey Stuff

- We can reduce the error creeping into our system over time by minimising our time step – in many ways, continued development of swifter algorithms to handle physical computation is directly related to this
- But we can't get rid of it – our system will always be spiralling away from accuracy
- It doesn't matter, though – so long as our system is stable (doesn't explode) and believable to the player, it's doing its job
- Different numerical integration techniques can help maintain the stability of our system over large numbers of updates, and potentially indefinitely
- Consider a level of a first-person shooter: if the physics system performs 120 updates per second, and the level takes 20 minutes to complete, the physics system needs to be stable for 144,000 updates

Numerical Integration Algorithms

- Explicit Euler Integration
- Implicit Euler Integration
- Semi-implicit (or Symplectic) Euler Integration
- Second Order
  - Verlet Integration
  - RK2

Explicit Euler Integration

$$v_{n+1} = v_n + a_n \Delta t$$
$$s_{n+1} = s_n + v_n \Delta t$$

- Uses the current frame's values for velocity and acceleration to compute the next frame's displacement and velocity
- Very quick, with minimal thought required to implement
- This tends to be the example used when first explaining numerical integration for that very reason
- Highly unstable for larger time steps
- Only really useful for scenarios where updates will occur very quickly as a result
- There are better (more stable) approaches that have a similar number of operations per object per update

Implicit (Backward) Euler Integration

$$v_{n+1} = v_n + a_{n+1}\Delta t$$
$$s_{n+1} = s_n + v_{n+1}\Delta t$$

- Uses the next frame's value for acceleration to compute the next frame's value for velocity, and that velocity to update displacement
- Very stable, as consideration of future events reduces error
- This tends to be used in highly predictable systems, or systems which don't need to operate in real time
- Computationally expensive to work out next frame's acceleration (as forces need predicting, and forces can be a function of predicted movement)
- Not really appropriate for gaming scenarios due to computational expense
- Player actions can be highly unpredictable, making estimation of future acceleration difficult

Semi-Implicit (Symplectic) Euler Integration

$$v_{n+1} = v_n + a_n \Delta t$$
$$s_{n+1} = s_n + v_{n+1} \Delta t$$

- Uses the current frame's value for acceleration to compute the next frame's value for velocity, and that velocity to update displacement
- Very stable
- Same speed as Explicit – just need to ensure the computations are performed in the right order (update velocity, then use new velocity to update displacement).
- Has a silly name
- That's about it for down sides
- It isn't as stable as fully implicit, but due to its speed and stability it sees a lot of use
- Ordering of the equations is vital

Verlet Integration

- Begin with semi-implicit equations:

$$v_{n+1} = v_n + a_n \Delta t$$
$$s_{n+1} = s_n + v_{n+1} \Delta t$$

- Substitute in v_{n+1} to get:

$$s_{n+1} = s_n + (v_n + a_n \Delta t)\Delta t$$

- Which rearranges to:

$$s_{n+1} = s_n + v_n \Delta t + a_n \Delta t^2$$

- If:

$$v_n = \frac{s_n - s_{n-1}}{\Delta t}$$

- Then:
- $s_{n+1} = s_n + (s_n - s_{n-1}) + a_n \Delta t^2 = \boldsymbol{2s_n - s_{n-1} + a_n \Delta t^2}$

Verlet Integration

$$s_{n+1} = 2s_n - s_{n-1} + a_n \Delta t^2$$

- This is a very stable solution
- About as many operations as Explicit/Symplectic
- Reversible in time
- Doesn't rely explicitly on velocity
- Neatly applicable to GPU computation
- Complex to implement properly – requires more thought than other methods
- Requires start conditions to be set up prior to updating, since it has to look back in time

Runge-Kutta “Midpoint”

- In Euler integration, we assume everything is constant between time *t* and time *t + 1*.
- Our simulation is more accurate if we don’t!
- Under changing forces, an object’s acceleration might change over the course of a time-step

- **Under constant acceleration, velocity changes over the course of a time-step**

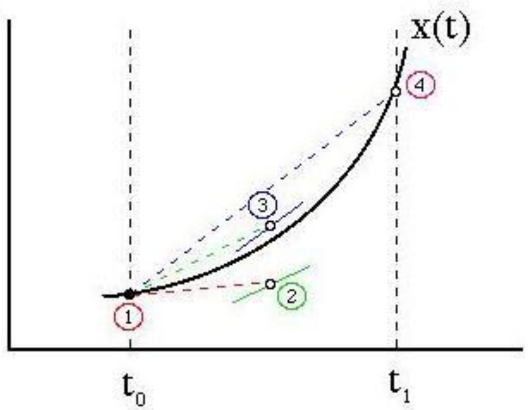
Runge-Kutta "Midpoint"

- If we know how the variables are changing, and have a quick way to guess how they're changing
- AND we have spare processor cycles
- We can substitute in a mid-point value for the changed variable, for a closer-to-accurate result.
- For example:

$$s_{t+1} = s_t + ((v_t + v_{t+0.5})\Delta t \times 0.5)$$

More Advanced Numerical Integration

- Solving differential equations
- Runge Kutta (*RK4*)



$$
\begin{aligned}
dx1 &= \Delta t\, v_{x,n} \\
dv_x1 &= \Delta t\, a_x(x_n, y_n, t) \\
dx2 &= \Delta t\, (v_{x,n} + \frac{dv_x1}{2}) \\
dv_x2 &= \Delta t\, a_x(x_n + \frac{dx1}{2}, y_n + \frac{dy1}{2}, t + \frac{\Delta t}{2}) \\
dx3 &= \Delta t\, (v_{x,n} + \frac{dv_x2}{2}) \\
dv_x3 &= \Delta t\, a_x(x_n + \frac{dx2}{2}, y_n + \frac{dy2}{2}, t + \frac{\Delta t}{2}) \\
dx4 &= \Delta t\, (v_{x,n} + dv_x3) \\
dv_x4 &= \Delta t\, a_x(x_n + dx3, y_n + dy3, t + \Delta t) \\
x_{n+1} &= y_n + \frac{dx1}{6} + \frac{dx2}{3} + \frac{dx3}{3} + \frac{dx4}{6} \\
v_{x,n+1} &= v_{x,n} + \frac{dv_x1}{6} + \frac{dv_x2}{3} + \frac{dv_x3}{3} + \frac{dv_x4}{4}
\end{aligned}
$$

Summary

- Calculus, calculus, and more calculus
- The theories underpinning numerical integration
- Several numerical integration algorithms
- Extension to angular motion from linear motion

Implementation

- See first couple of tasks on the Practical Tasks hand out for today
- Have some fun with it!