Simulating Dynamic Systems Under Special Relativity

Will Ebmeyer

May 5, 2021

Introduction:

The Theory of Special Relativity:

The theory of special relativity (SR) describes the apparent motion of objects through spacetime in the absence of gravity. At its core, the theory is based around two basic postulates:

1. The laws of physics are the same in all inertial reference frames. In other words, there is no "privileged" viewpoint, so long as it's not accelerating.

2. The speed of light, c, is constant in all inertial reference frames.

Consequently, space and time must be considered entirely relative to the observer. Moreover, two observers can even disagree on their definitions of space and time, as distances and time intervals contract or dilate depending on their relative velocity. For example, two events that are simultaneous in one frame may be asynchronous in another. Or, perhaps, an observer will see a moving object as bizarrely compressed.

Motivation and Goals:

Unfortunately, for those of us who don't regularly travel at relativistic speeds, much of special relativity appears unintuitive. As such, this project aimed to build a general framework for simulating the evolution of dynamic, relativistic systems.

To this end, the first step was to support simple systems where all velocities are constant. Fortunately, well-studied examples of such systems are readily available, rendering it easy to qualitatively verify the simulation against analytic predictions. The next goal, then, was to extend support to systems where objects do not move at constant velocities.

Of course, another important goal of this program was the visual aspect—the ability for the user to see the consequences of special relativity. For this, the program includes an interactive window where they may switch between any arbitrary reference frame as the system evolves.

Lorentz Transformations:

In low-velocity (non-relativistic) situations, the classical Galilean transformations hold for switching between frames of reference. Unlike in special relativity, these transformations do not assume the speed of light is constant. Conventionally, they take the form:

$$x' = x - vt$$

$$t' = t$$

In this form, these equations take the position of some object (t, x) as seen by an "unprimed" reference frame and return that position as seen by some moving "primed" reference frame.

However, the Galilean transformations fail at sufficiently high velocities, and it instead becomes necessary to use the Lorentz transformations. These transformations form the basis for relating inertial reference frames in special relativity, as they constructed such that all inertial frames observe a constant speed of light. Conventionally, they take the form:

$$\mathrm{x}' = \gamma(x - vt)$$
$$t' = \gamma\left(t - \frac{vx}{c^2}\right)$$

Where:

$$\gamma = \frac{1}{\sqrt{1 - \frac{v^2}{c^2}}}$$

Or, in a vectorized form that does not assume *x* and *v* are in the same direction:

$$\overrightarrow{r'} = \vec{r} + (\gamma - 1)[\vec{r} \cdot \hat{n}]\hat{n} - \gamma \mathrm{t}\vec{v}$$
$$\mathrm{t}' = \gamma\left(\mathrm{t} - \frac{\vec{\mathrm{v}} \cdot \vec{\mathrm{r}}}{\mathrm{c}^2}\right)$$

Where $\hat{n} = \frac{\vec{v}}{||\vec{v}||}$

Unfortunately, these transformations are still flawed, as they assume an unchanging velocity over time. For the Galilean transformations, this problem is solved by taking the differential form and integrating:

$$\int \mathrm{d}x' = \int \mathrm{d\,x} - \int \mathrm{v}(t)\mathrm{dt}$$
$$\int dt' = \int dt$$

Which simplifies to:

$$\mathrm{x}'(t) = \mathrm{x(t)} - \int \mathrm{v(t)dt}$$
$$t'(t) = t$$

The analogous process for the Lorentz transforms, then, is:

$$\int d\vec{r}' = \int d\vec{r} + \int (\gamma(t) - 1)[d\vec{r} \cdot \hat{n}]\hat{n} - \int \gamma(t)dt\,\vec{v}(t)$$
$$\int dt' = \int \gamma(t)dt - \int \frac{1}{c^2}\gamma(t)\vec{v}(t) \cdot d\vec{r}$$

Which simplifies to:

$$\overrightarrow{r'}(t) = \vec{r}(t) + (\gamma(t) - 1)[\vec{r}(t) \cdot \hat{n}(t)]\hat{n}(t) - \int_0^t \gamma(t)\vec{v}(t)dt$$

$$t'(t) = \int_0^t \gamma(t)dt - \frac{1}{c^2}\gamma(t)\vec{v}(t)\cdot\vec{r}(t)$$

These equations give us the entire history of an object, its "worldline" as seen in one frame, given its history in another.

Length Contraction:

One consequence of the Lorentz transforms is length contraction: where an object appears "compressed" in the direction of its velocity. Consider some object that's at rest in the unprimed frame but has some velocity in the primed frame:

$$t_1' = \gamma\left(t_1 - \frac{\vec{v}\cdot\overrightarrow{r_1}}{c^2}\right) \qquad t_2' = \gamma\left(t_2 - \frac{\vec{v}\cdot\overrightarrow{r_2}}{c^2}\right)$$

$$\overrightarrow{r_1'} = \overrightarrow{r_1} + (\gamma - 1)[\overrightarrow{r_1}\cdot\hat{n}]\hat{n} - \gamma t_1\vec{v} \qquad \overrightarrow{r_2'} = \overrightarrow{r_2} + (\gamma - 1)[\overrightarrow{r_2}\cdot\hat{n}]\hat{n} - \gamma t_2\vec{v}$$

If the primed frame makes its measurements of the object's two endpoints at the same time:

$$\Delta t' = t_2' - t_1' = 0$$

Which, when substituting in the above equations for t_2' and t_1' yields:

$$\Delta t = \frac{\vec{v}\cdot\overrightarrow{L_0}}{c^2}$$

Where $\overrightarrow{L_0} = \overrightarrow{r_2} - \overrightarrow{r_1}$, the length vector of the object in its own frame.

Applying a similar process to obtain $\overrightarrow{L'} = \Delta\overrightarrow{r'} = \overrightarrow{r_2'} - \overrightarrow{r_1'}$ eventually yields:

$$\overrightarrow{L'} = \overrightarrow{L_0} + \left(\frac{1}{\gamma} - 1\right)\left(\overrightarrow{L_0}\cdot\hat{n}\right)\hat{n}$$

Relativistic Force and Acceleration:

In classical Newtonian dynamics, force is defined as the time-derivative of momentum:

$$\vec{F} = \frac{d\vec{p}}{dt}$$

In special relativity, this equation still holds true, and can be used to derive a concept of relativistic force. The equation for relativistic momentum takes the form:

$$\vec{p} = \gamma m\vec{v}$$

So, taking the time derivative, one may obtain:

$$\vec{F} = \gamma m\left(\frac{\vec{v}\cdot\vec{a}}{c^2}\gamma^2\vec{v} + \vec{a}\right)$$

Which, when solved for acceleration yields:

$$\vec{a} = \frac{1}{m\gamma}\left(\vec{F} - \frac{(\vec{F}\cdot\vec{v})\vec{v}}{c^2}\right)$$

Time Delay of Forces:

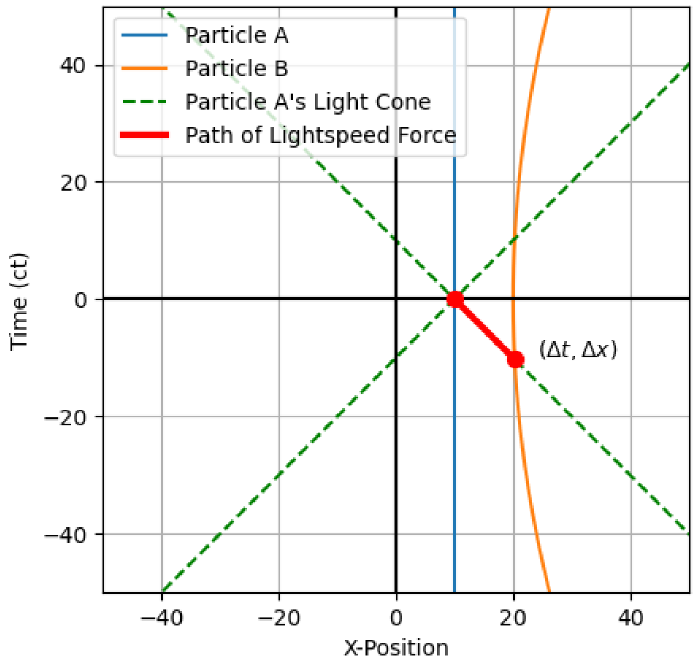
In special relativity, information cannot travel faster than light—including forces. As such, unless the force field is assumed to have always existed, it will be necessary to factor in a time delay when calculating forces. In other words, a particle is only capable of experiencing an "outdated" version of another particle's influence.

If this force propagates at the speed of light, this problem requires finding the position on the other particle's worldline such that, were it to emit a light-speed force carrier, would strike the present particle. This translates to finding the point where the other particle's worldline crossed the current particle's "light cone" of influence. This "outdated" position must satisfy:

$$c^2\Delta t^2 - \Delta x^2 - \Delta y^2 = 0$$

Where Δt, Δx, and Δy are the coordinate separations between the particle's present position, and the other particle's past position.

Represented graphically:



The present particle experiences the force that left the other particle back at (Δt, Δx)

Program Manual:

Dependencies:

This program requires the *numpy*, *pandas*, and *matplotlib* libraries for performing computations and loading/saving data. Additionally, the *pygame* library is required, as it is used extensively in the graphical segments of the program. Note, however, this program was written with *pygame 1.9.6*, and has not been tested in newer versions.

The remaining required library, *typing*, is already included in python's standard library.

Simulation Objects:

The *Simulation* class contains the bulk of the code for creating, running, and viewing simulations.

Since any meaningful observation about a system requires a frame of reference, all physics calculations are run from the perspective of a main, inertial origin frame. Consequently, all worldlines generated by the simulation are also taken from this frame.

The worldlines themselves are contained in lists of numpy arrays named: time, x_history, y_history, vx_history and vy_history. All four of these arrays take follow the same format:

↑ Time (First Index) ↓	← Object (Second Index) →		
	Object 1 position/velocity	Object 2 position/velocity	…
	Object 1 position/velocity	Object 2 position/velocity	…
	…	…	…

The first index of these arrays refers to the simulation's state at a particular instance, while the second index refers to the state of an individual object at that instance.

Building Simulations:

Declaring Simulations Objects:

Preparing a simulation starts with calling the Simulation() constructor. At minimum, this constructor must be provided a time-step and a maximum time to simulate to.

Optionally, it may be provided a function representing the forces acting on the particles—the default for which is "None" (indicating there are no forces). This function must follow the form:

```
func(time-history, x-history, y-history, vx-history, vy-history, mass, charge)
```

And must return two numpy arrays representing the forces in the X and Y directions acting on each of the particles. Naturally, these arrays must be equal in length to the number of objects in the simulation.

Expect the format of these arguments as being the same as described in the previous section.

Also optionally, the constructor may be provided a speed of light, c, so that its calculations are performed in units where c is the given value. The default value for this is 10.0 so that relativistic effects are easier to observe.

Once declared, the Simulation object can be populated with "points" and/or "polygons." Several functions are included to assist with this, though it is possible to do so manually by directly accessing *x*, *y*, *vx*, *vy*, *mass*, *charge*, and *polygons* attributes—which act as the initial state of the simulation.

Adding Points to the Simulation:

A singular point may be added to the simulation with the function call *sim.add_point()*. This is the simplest method of populating the simulation with objects, the syntax for which is:

```
sim.add_point(x, y, vx, vy, mass, charge)
```

This function must be provided the X, Y position, velocity, mass, and charge of the particle. Calling it will append this information into the *sim.x*, *sim.y*, *sim.vx*, *sim.vy*, *sim.mass*, and *sim.charge* arrays.

Adding Polygons to the Simulation:

The *sim.add_polygon()* function allows the user to add an entire polygon to the simulation. A polygon's vertices are treated as a collection of independent points by the simulation—with the purely visual exception that the vertices are connected when *sim.show()* is run. The syntax for this function is:

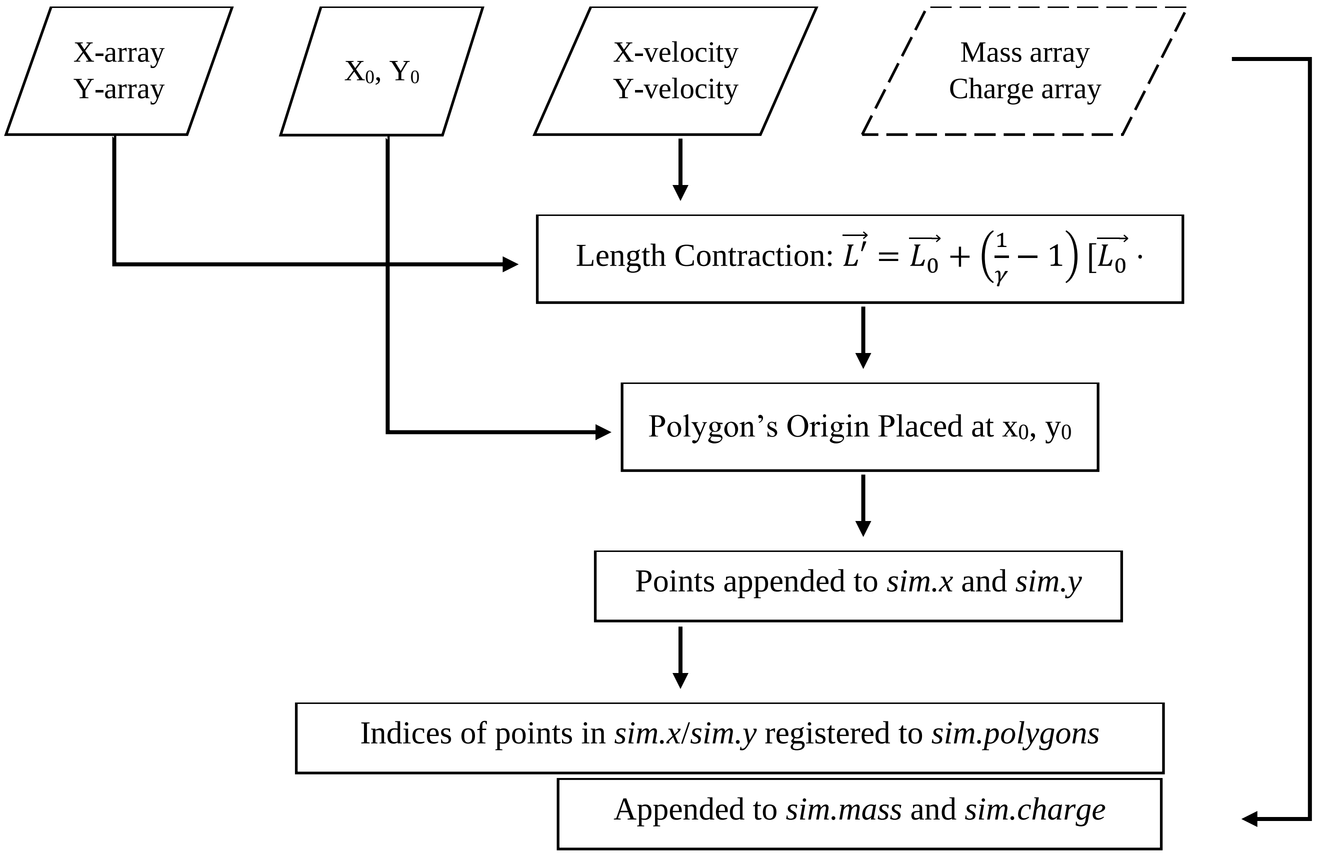
```
sim.add_polygon(x_points, y_points, x0, y0, vx, vy, masses, charges)
```

This function is provided an x-array and y-array, representing the vertices of the polygon relative to its personal origin. The x-velocity and y-velocity arguments, the velocity of all vertices relative to the main origin, are then used to compute the length-contracted version of the polygon with equation. This modified polygon's origin is then placed at the provided x_0 and y_0 coordinates.

Optionally, this function may be provided a mass-array and a charge-array for the masses and charges of each individual vertex. Left unspecified, the masses default to 1.0 units and the charges default to uncharged.

Finally, calling this function will compile a list of the vertices' indices, and append it to the *sim.polygon* attribute. This attribute contains a record of all registered polygons.

Graphically, this process is represented as:


X-array
Y-array
X_0, Y_0
X-velocity
Y-velocity
Mass array
Charge array
Length Contraction: $\overrightarrow{L'} = \overrightarrow{L_0} + \left(\frac{1}{\gamma} - 1\right) [\overrightarrow{L_0} \cdot$
Polygon's Origin Placed at x_0, y_0
Points appended to *sim.x* and *sim.y*
Indices of points in *sim.x*/*sim.y* registered to *sim.polygons*
Appended to *sim.mass* and *sim.charge*

Loading (or Saving) Data from a File:

The Simulation class contains methods for saving and loading data to/from a file: *sim.save()* and *sim.load()*. Such functions may be particularly useful for time-consuming simulations, so that they may be analyzed or viewed afterwards without having to re-run said simulation. The files they generate/read from are gzip-compressed .csv files—though compression may be disabled via the *compressed=False* keyword argument. Meanwhile, their contents are structured as:

Index	*Array of values, length equal to object count times time array length*
X-history	*Array with length of object count × time array length*
Y-history	*Array with length of object count × time array length*
X-velocity history	*Array with length of object count × time array length*
Y-velocity history	*Array with length of object count × time array length*
Time	*Array with length equal to time array length (below)*
Mass	*Array with length equal to object count*

Charge	*Array with length equal to object count*		
Parameters	*Speed of light*	*Time array length*	*Object count*

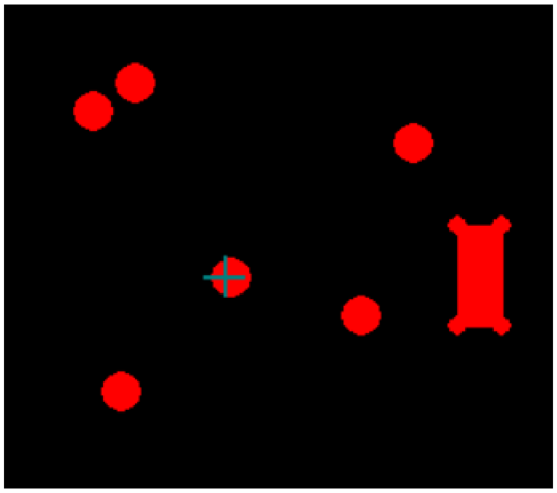
Note that, while in this format the history arrays are flattened into a 1D array. Upon being loaded, they must be reshaped into 2D arrays of shape (time array length, object count). This is only relevant if the user needs to process these files outside of the program, as *sim.load()* will handle the conversions itself.

Example of Declaring a Simulation:

The following example makes use of the above functions to declare and populate a simulation.

```
# Import the simulation class
from SpecialRelativity import *

# Declare a simulation.
# This simulation will run until t=50, with a time-step of 0.1 and a speed-of-light of 10.0
sim = Simulation(0.1, 50, c=10.0)

# Add a single particle to the simulation
# Place this particle at x=1, y=1, with a velocity of vx=1.0, vy=0.0, a mass of 1.0, and a charge of 0.0
sim.add_point(1.0, 1.0, 1.0, 0.0, 1.0, 0.0)

# Randomly place several five, motionless, uncharged particles
# Spread them over an x and y range of -100.0 to +100.0, with a maximum mass of 10.0
sim.basic_random_start((-100.0, 100.0), (-100.0, 100.0),
                       (0.0, 0.0), (0.0, 0.0),
                       (0.0, 0.0),
                       10.0,
                       5)

# Place a 50x50 square at x=100.0, y=0.0, with a velocity of vx=9.0, vy=0.0
# Assume default masses and charges
sim.add_polygon([-25, 25, 25, -25], [25, 25, -25, -25], 100.0, 0.0, 9.0, 0.0)
```

If run, this simulation produces the following initial state:

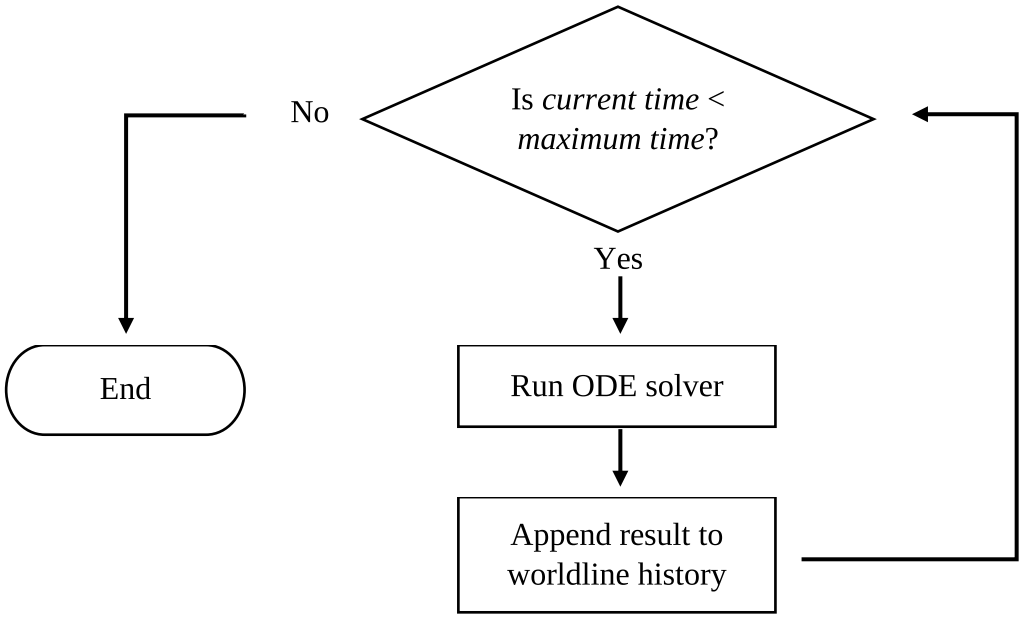


Running Simulations:

Once the initial conditions are prepared, calling *sim.run()* will run a simulation until the origin's time reaches the maximum time specified in the original declaration. By default, the simulation uses a fourth order Runge-Kutta method, though Euler's method is available for testing purposes via the keyword argument *method='euler'*. For convenience, the *run()* method also includes the keyword argument *print_progress*, which will print the simulation's progress over time to the console for monitoring purposes. Put together, the syntax—with default values—is:

```
sim.run(method='rk4', print_progress=False)
```

The implementation of the *run* method can be summarized with:


No
Is *current time* < *maximum time*?
Yes
End
Run ODE solver
Append result to worldline history

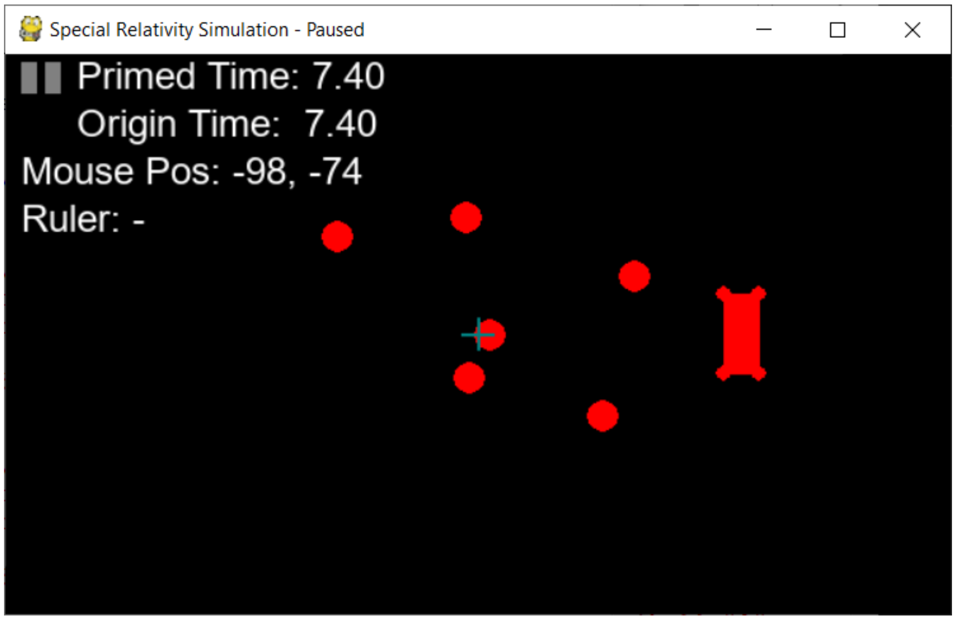
The simulation will continue to iterate until it reaches the maximum time, extending the worldlines as it goes.

There is, however, a caveat hidden within the ODE solver. Recall that the user cannot specify accelerations manually, only the forces. This is where the equation for relativistic acceleration comes in (see *Relativistic Force and Acceleration*). In the included ODE solvers, this equation computes the relativistic accelerations that are used to update the velocities. This process is handled entirely internally with the *sim.force_to_acceleration()* method.

Viewing Simulations:

Interactive Viewing:

The final key portion of the Simulation class is the ability to view the results. The first means for doing so is the *sim.show()* method. This method takes three optional arguments: the start and end times—for only showing a segment of interest—, as well as the maximum frames-per-second with which to render the screen. Upon calling *sim.show()*, the user will be greeted with a window will look something like:



The basic controls for interacting with this window are:

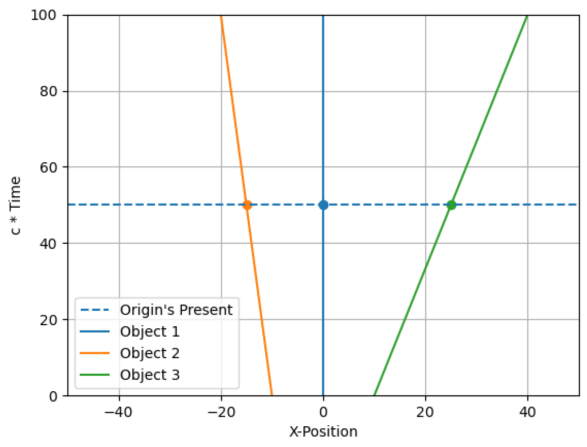
Spacebar	Pause/unpause playback
Right click and drag	Use ruler
Left click on point	Select reference frame
Escape key	Return to origin's frame

Meanwhile, the top right displays some potentially useful information:

Pause indicator	Indicates whether the simulation is currently paused
Origin time	The current time according to the origin
"Primed" time	The coordinate time reported by the currently selected reference frame, as calculated by the integrated Lorentz transforms.
Mouse position	The current coordinates of the mouse on the screen. The (0, 0) position is located at the direct center, indicated by the crosshair.
Ruler	The distance covered by the ruler when said ruler is active.

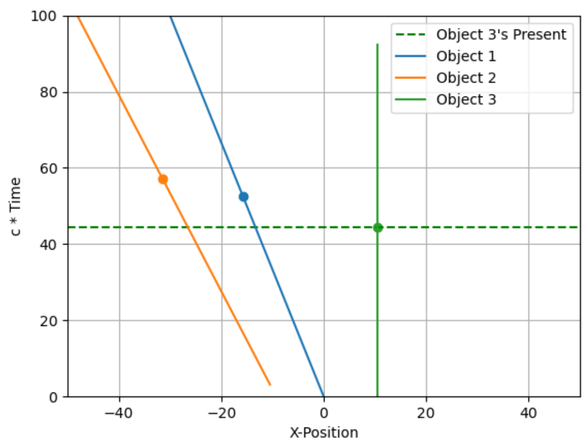
Left clicking on one of the points will allow the user to switch to that point's reference frame. This is accomplished by performing the integrated Lorentz transforms to obtain the primed coordinates of all worldlines. However, recall that simultaneous events in one frame may not be simultaneous in another. As such, it is necessary to obtain the positions of all objects in this new frame's version of "present." This process can be summarized graphically:

Unprimed Frame:



Start in the origin's coordinate system.

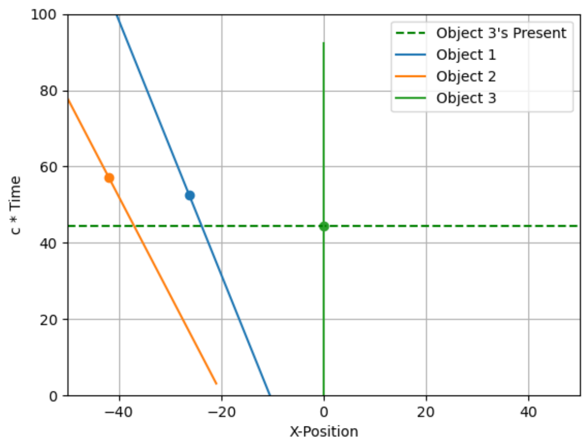
Primed Frame:



Using the integrated Lorentz transformations, switch to the target reference frame's coordinate system.

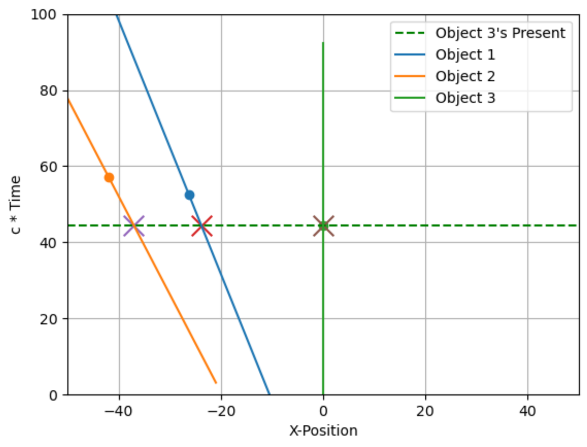
Primed coordinates are calculated by the *sim.lorentz_boost()* method.

Shifted Primed Frame:



Shift the screen such that the new frame is, spatially, at the center.

Present Shifted Primed Frame:



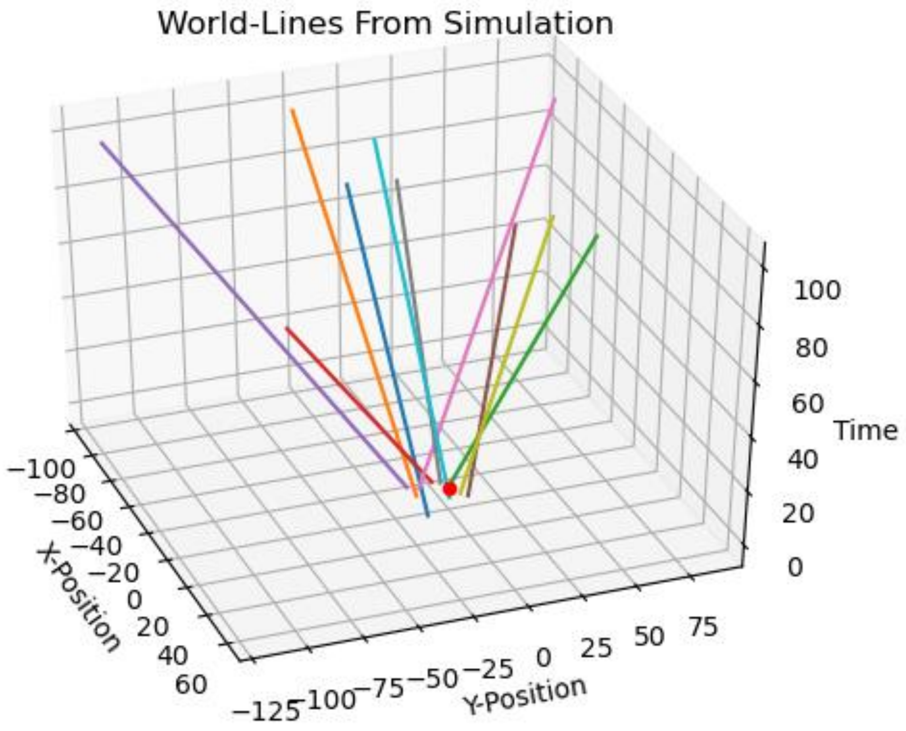
Locate the point in each object's worldline that corresponds with the new frame's version of "now."

This is handled by the *sim.obtain_present()* method.	This is also handled by the *sim.obtain_present()* method.

A limitation of this, however, is that the switch will not occur if the particle is travelling at (or somehow exceeding), the speed of light. At the speed of light, the Lorentz factor approaches infinity, indicating that lightspeed trajectories do not have a proper reference frame. For faster than light speeds, the Lorentz factor takes an imaginary value. This, of course, is non-physical, so such reference frames are disallowed by the program.

Plotted Viewing:

The other method for viewing a simulation results is *sim.plot()*. This method constructs a full 3D plot of all worldlines in the simulation:



By default, this plot is from the perspective of the origin frame. However, the reference frame of any other point may be used by supplying the optional keyword argument *reference_frame* with the index of said point. For additional configuration, the keyword arguments *x_lim*, *y_lim*, and *t_lim* may be supplied tuples of ranges to narrow in on a particular region. The full syntax for this method is:

```
sim.plot(reference_frame=-1, x_lim=(min x, max x), y_lim=(min y, max y), t_lim=(min t, max t))
```

Results:

Brief Note:

The code for running all the following simulations is contained within *Example Simulations.py*, which imports the main special relativity program.

Collection of Inertial Points:

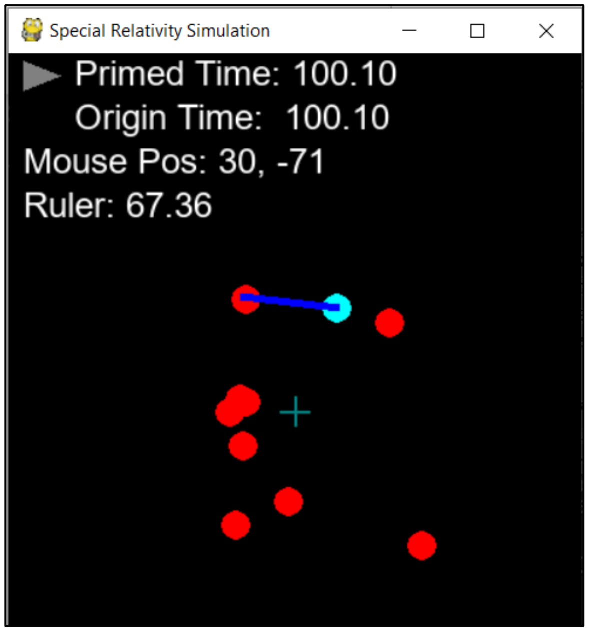
Setup:

The first test performed involved a simple collection of moving points. All points in this simulation were moving at a constant velocity—i.e. inertial—for the entire duration. This test aimed to check whether the space/time-distorting effects of special relativity were, in fact, working as expected.

Results:

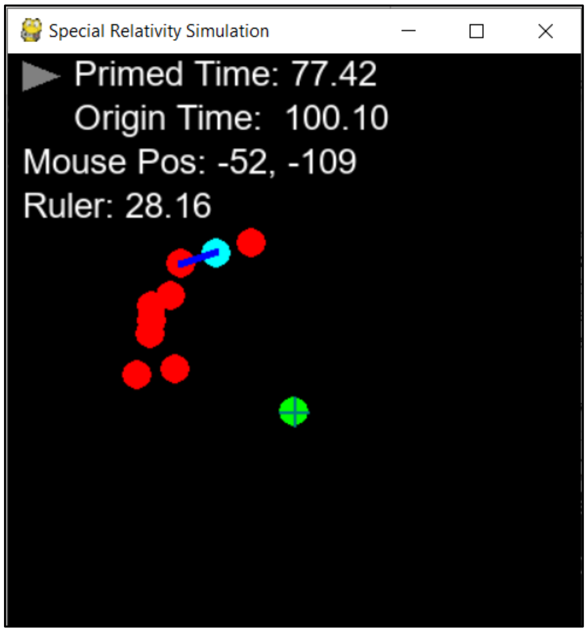
A brief inspection of the simulation's results reveals that spacetime does, in fact, appear to be distorting as expected. Upon selecting a reference frame, all other particles appear to group closer together. Additionally, the clock of any moving reference frame is running behind the origin's reference frame. These two effects are consistent with time dilation and space contraction: distances are compressed, and time runs slower relative to a moving reference frame.

Shown below is a comparison between the origin's view, and the view of the fastest object in the simulation:



```
Primed Time: 100.10
Origin Time:  100.10
Mouse Pos: 30, -71
Ruler: 67.36
```

Origin's frame



```
Primed Time: 77.42
Origin Time:  100.10
Mouse Pos: -52, -109
Ruler: 28.16
```

Frame of fastest object

In this example, notice that the selected frame's clock reads 77.42 seconds, whereas the origin's clock reads 100.10 seconds. This discrepancy indicates that the object's clock is running slower than the origin's—hence time dilation. Additionally, notice that the ruler reports a longer distance between two points in the origin's frame—67.36 units—than it does between the same two points in the object's frame—28.16 units.

Though this inspection isn't entirely rigorous, it does at least demonstrate that the program is working.

Length Contraction:

Setup:

The next test placed a simple square next to the origin with a constant velocity taking it towards the bottom right. The goal of this test was to both demonstrate how objects look under length contractions, as well as to verify that length contraction was working exactly as predicted.

The square's size was 50 units by 50 units, and its velocity was 7 units/sec right and 7 units/sec down. Looking at just the diagonal (parallel to the velocity) of the square:

$\overrightarrow{L_0}$	Diagonal of the square in box's reference frame	$(50\hat{x} + 50\hat{y})$ units
$\vec{v}$	Velocity of square relative the origin	$(7\hat{x} + 7\hat{y})$ units/sec
c	Speed of light	10 units/sec
$\hat{n}$	Unit velocity vector relative to the origin	$\left(\frac{1}{\sqrt{2}}\hat{x} + \frac{1}{\sqrt{2}}\hat{y}\right)$ units/sec
γ	Lorentz factor, $\frac{1}{\sqrt{1-v^2/c^2}}$	$5\sqrt{2}$

Substituting these values into the vectorized length contraction equation (*see the Length Contraction section*) yields:

$$\overrightarrow{\mathrm{L}'} = (50\hat{x} + 50\hat{y}) + \left(\frac{1}{5\sqrt{2}} - 1\right)\left(50 * \frac{1}{\sqrt{2}} + 50 * \frac{1}{\sqrt{2}}\right)\left(\frac{1}{\sqrt{2}}\hat{x} + \frac{1}{\sqrt{2}}\hat{y}\right)$$

Which simplifies to:

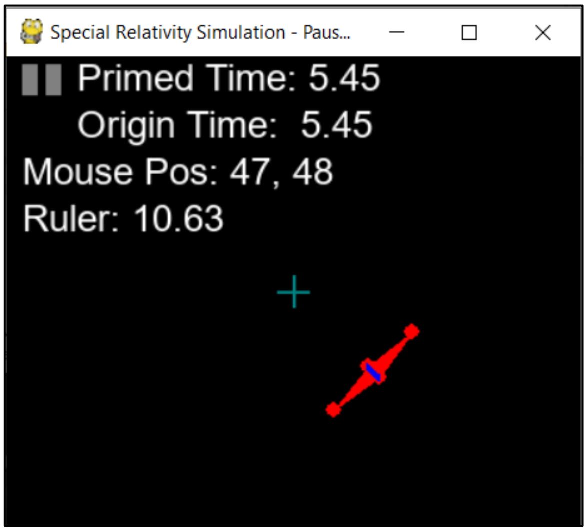
$$\overrightarrow{L'} = 5\sqrt{2\hat{x}} + 5\sqrt{2\hat{y}} \approx 7.07\hat{x} + 7.07\hat{y}$$

Comparing the two diagonal vectors, the lengths should be:

$$\left\|\overrightarrow{L'}\right\| = 10 \text{ units} \qquad \left\|\overrightarrow{L_0}\right\| \approx 70.7 \text{ units}$$

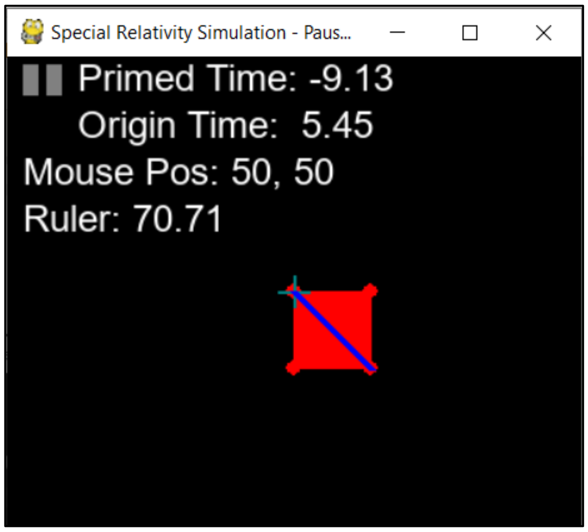
Results:

As expected, the resulting square in the origin's frame appeared compressed in the direction of the velocity, while the in the box's frame it appeared as a perfect square. Moreover, the length of the diagonals—measurement error aside—were exactly as predicted: about 10 units in the origin's frame, and about 70.7 units in the box's frame.


Special Relativity Simulation - Paus...

```
▮▮ Primed Time: 5.45
   Origin Time:  5.45
Mouse Pos: 47, 48
Ruler: 10.63
```

Origin's frame


Special Relativity Simulation - Paus...

```
▮▮ Primed Time: -9.13
   Origin Time:  5.45
Mouse Pos: 50, 50
Ruler: 70.71
```

Box's frame

Evidently, the implemented Lorentz transforms are giving rise to the correct length contraction.

Barn and Ladder Paradox:

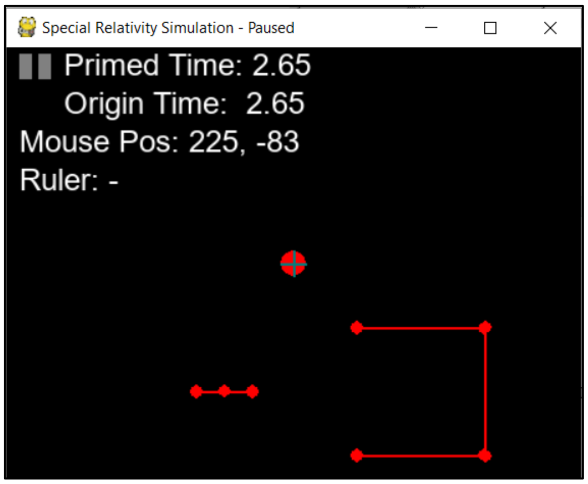
Setup:

The "barn and ladder paradox" is a classic example of special relativity. In this setup, a "ladder" is carried into a "barn" at a significant fraction of the speed of light. According to a stationary observer relative to the barn, the ladder will appear shortened by length contraction such that it could fit inside the barn. However, in the ladder's reference frame, it is the barn that's moving (and therefore contracted). As such, according to whoever is carrying the ladder, said ladder will *not* fit inside the barn. This, then, begs the question, does the ladder fit inside the barn or not?

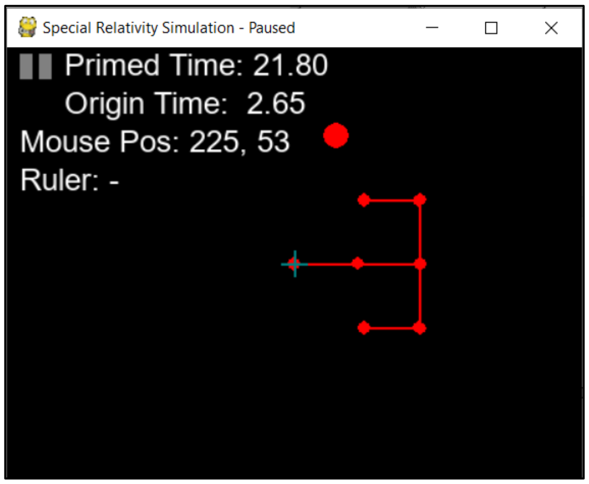
In this simulation, a line, representing the ladder, is placed to the left of the origin traveling right at 90% lightspeed. To the left of the origin, meanwhile, a stationary open box to represent the barn. Finally, the stationary observer is placed at the origin, represented by a singular point.

Results:

Based on the simulation's results, it appears that this question is fundamentally flawed. According to the first postulate of special relativity, all inertial reference frames are equivalent. Consequently, both reference frames here—one where the ladder fits, and one where it does not—are technically correct. However, the ladder never does both *simultaneously*. The below captures were taken at the instant the ladder hits the back of the barn in its own frame:



Origin's frame



Ladder's frame

In the ladder's frame, it cannot move forward any further—unless it desires to annihilate the barn—nor does it fully fit. In the observer's frame, however, the ladder never fully entered the barn anyways. Put another way: though the ladder could *technically* fit inside the barn, it never actually does so. As such, there is no paradox where the ladder simultaneously fits and doesn't fit, as the situation never actually occurs.

Constant Force:

Setup:

In this simulation, a singular particle is subjected to a constant rightwards force. For reference, several particles with velocities of zero, one-quarter lightspeed, one-half lightspeed, and lightspeed are also included. Given the equation for relativistic acceleration due to a force (*see Relativistic Force and Acceleration*), consider a force directed parallel to an object's velocity:

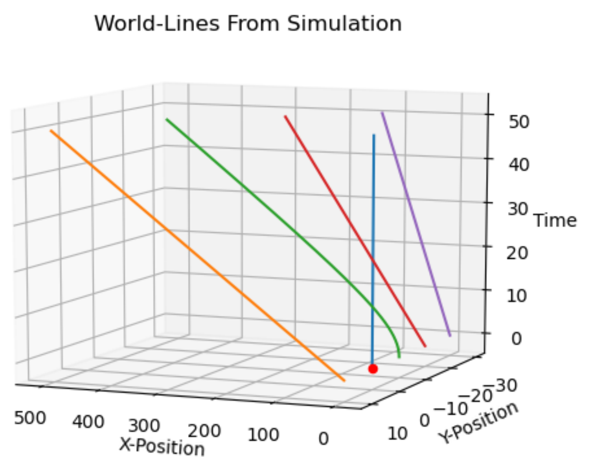
$$a = \frac{F}{m\gamma^3} = \frac{F}{m}\left(1 - \frac{v^2}{c^2}\right)^{3/2}$$

As the object approaches the speed of light, the observed acceleration will approach zero due to the inverse Lorentz factor. As such, the object will only be able to approach the speed of light, but never reach or surpass it.

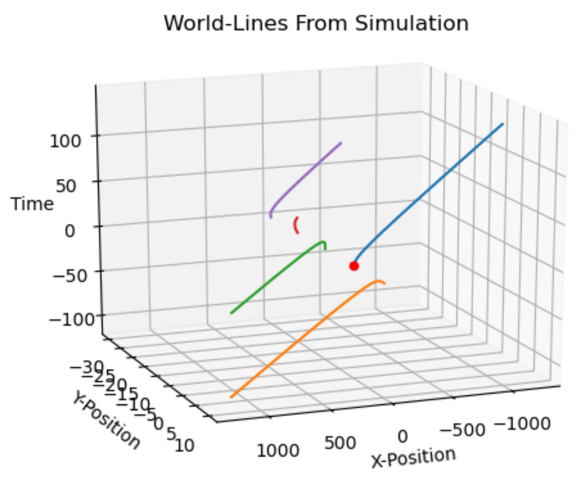
Results:

From the perspective of the origin, the accelerating object behaves exactly as expected. The slope of its worldline (shown below in green) approaches that of light (shown in orange), forming a hyperbola.

However, unless travelling backwards in time is suddenly possible, the accelerating object's perspective appears entirely unphysical. Indeed, this is due to a significant flaw in how coordinate transforming into these frames is handled. See the *Conclusions* section for details as to the nature of this flaw.



Worldlines as viewed from the origin's frame



Worldlines as viewed from the accelerating particle.

The results are non-physical, and indicative of an error.

Electrostatic Forces Between N-Bodies:

Setup:

In this simulation, ten initially stationary particles are randomly placed around the screen. These are all charged and exert an electrostatic force on all other particles. Of course, these forces do not act instantaneously, as their information cannot exceed the speed of light without breaking causality. For more details on how the resulting forces are calculated, see *Relativistic Force and Acceleration*.

Results:

As expected, the particles do not immediately respond to the force fields produced by other particles. Instead, time is required for the information to reach them. Notice, then, in the included worldline plot, closer objects tend to react to one another faster than isolated ones, as there is less distance for said information to traverse.

There is a caveat with this simulation's accuracy, however. If the forces are strong enough, it is possible for a particle to jump past the light barrier. As for why this occurs, consider a particle initially at rest. For such a particle, the equation for acceleration reduces to:

$$a = \frac{F}{m}$$

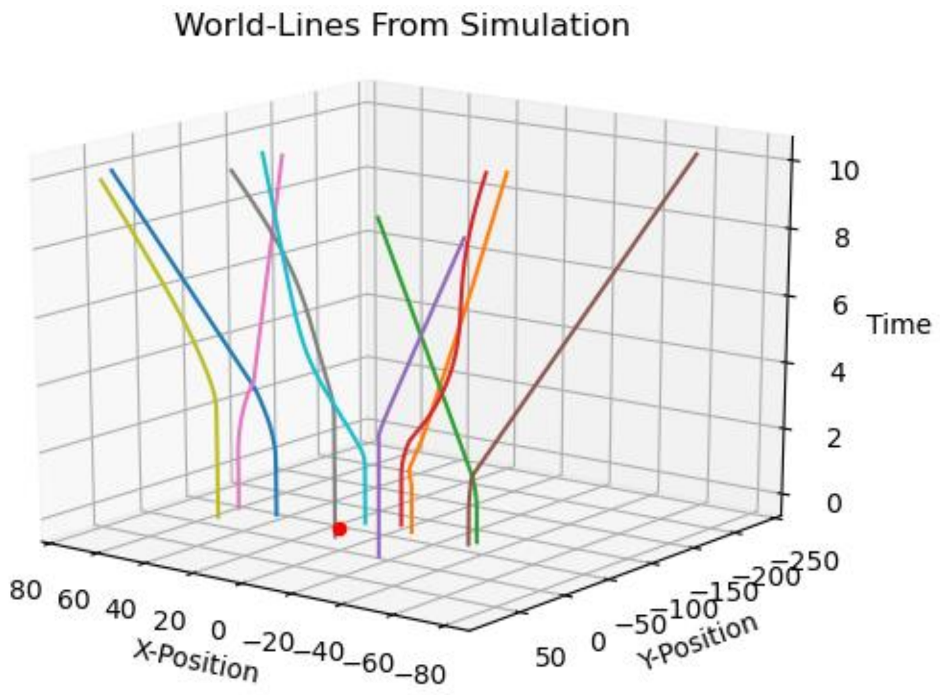
Assuming Euler's method:

$$\Delta v = a \, \Delta t$$

All that would be required for the velocity to jump above lightspeed would be:

$$\frac{F}{m} \Delta t \geq c$$

For superior ODE solving methods, such as RK4, the force required to accomplish this will be higher, but the basic reasoning still stands. In short, objects managing to surpass lightspeed should be taken as a result of insufficient accuracy—an effect that can be partially suppressed with smaller time-steps.



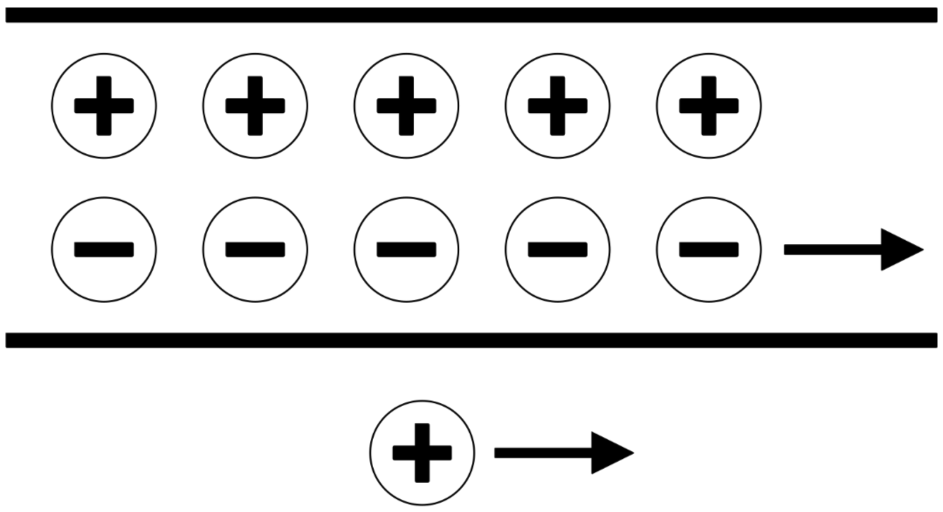
As this simulation took somewhat longer to run than others, its data is included in *Electrostatic Simulation.gz*. It can be loaded with the *load()* method as described in the program manual.

Moving Charges in a Wire:

Setup:

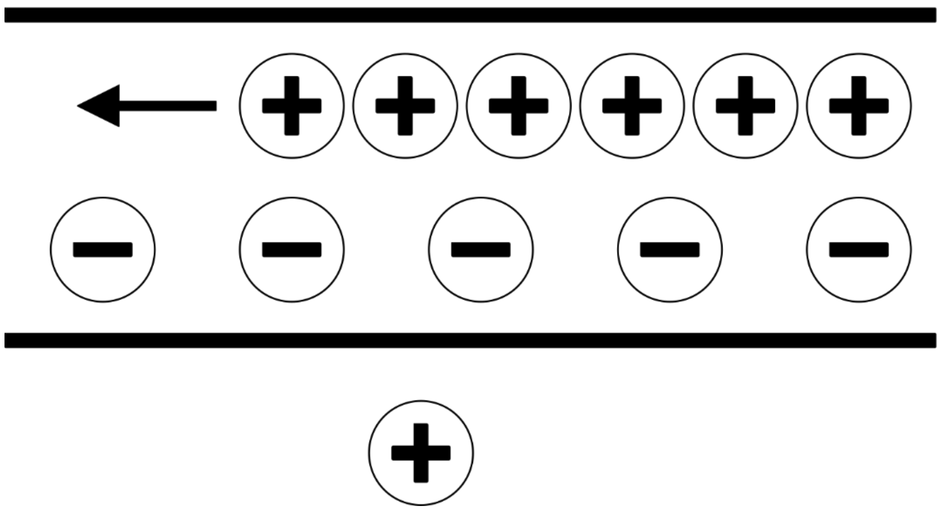
The final simulation involved testing the relationship between the electrostatic and magnetic forces. According to special relativity, these two forces are, in fact entirely observer dependent. In other words, one frame may observe the force as entirely electrostatic in nature, while another might view it as entirely magnetic.

For a brief explanation as to why, consider a moving charge next to a wire with a current. For simplicity's sake, the test charge's velocity will be equal to the electron's drift velocity:



In this reference frame, the negative charges are moving through the wire, and the positive test ion is moving next to it. Both the positive charges and the negative charges have a spacing of L, so the wire has a net charge of zero.

However, consider the situation in the negative charge's frame:



In this frame, it is the positive charges that are moving, not the negative ones. Due to this, length contraction will cause the positive charges to appear more closely packed, with a spacing of L^{+}. Meanwhile, the loss of velocity will cause the negative charges to appear more loosely packed, with a spacing of L^-. This creates an imbalance in charge density that ultimately gives the wire a positive net charge. This positive net then repels the positive test ion via the electrostatic force.

As a function for relativistic electrostatic forces was already developed for the previous simulation, this simulation will re-use that code by taking the negative charge's reference frame, where the forces are entirely electrostatic. The outside observer's reference frame, then, will be represented by a point moving with the same leftward velocity as the positive charges.

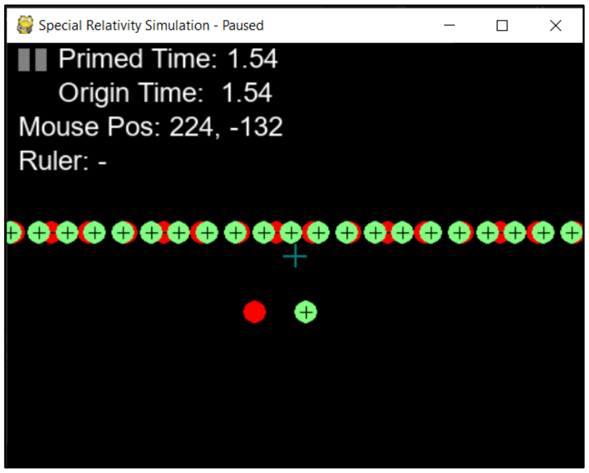
To ensure the spacing of the charges in each of the frames is correct, a simplified variant of the length contraction formula may be used:

$$L^+ = \frac{L}{\gamma} \qquad\qquad L^- = \gamma L$$

Results:

Just as expected, the force exerted on the positive test charge appears electrostatic in one frame, while magnetic in the other.

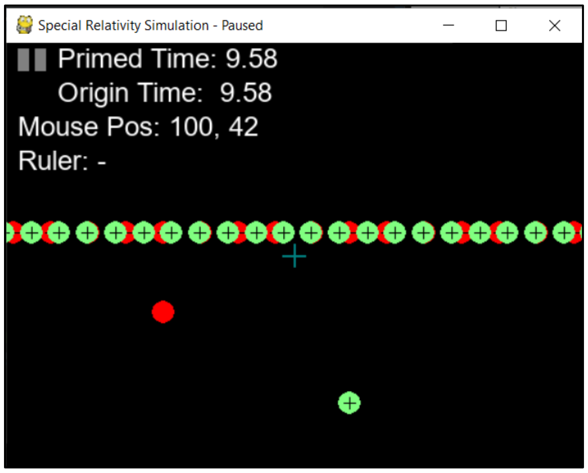
In the negative charge's frame:



Special Relativity Simulation - Paused

Primed Time: 1.54
Origin Time: 1.54
Mouse Pos: 224, -132
Ruler: -

In this frame, only the positive charges in the wire appear to be moving. Due to length contraction, these charges appear condensed, giving the wire a net positive charge.

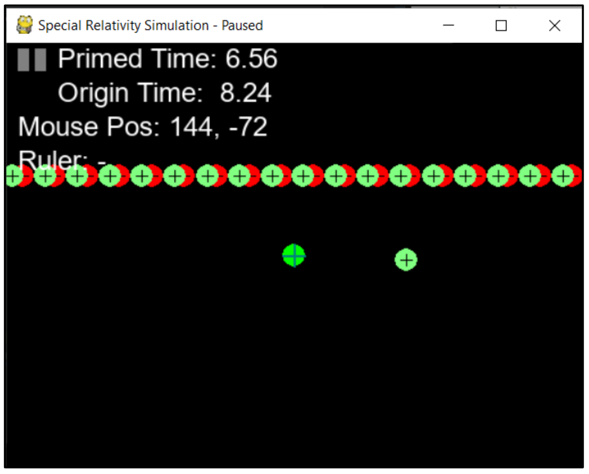


Special Relativity Simulation - Paused

Primed Time: 9.58
Origin Time: 9.58
Mouse Pos: 100, 42
Ruler: -

The wire's positive charge repels the positively charged ion with the electrostatic force.

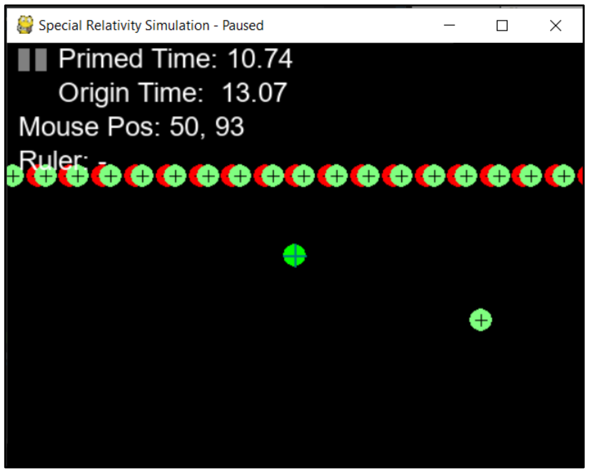
However, in the external observer's frame:



```
Primed Time: 6.56
Origin Time:  8.24
Mouse Pos: 144, -72
Ruler: -
```

In the observer's frame, both negative and positive charges are equally spaced, leaving the wire neutral. Only the negative charges in the wire appear to be moving.

The positive test ion initially moves towards the right.



```
Primed Time: 10.74
Origin Time:  13.07
Mouse Pos: 50, 93
Ruler: -
```

Over time, the flow of current exerts a downwards magnetic force on the positive charge, pushing it away.

This simulation also took somewhat longer to run than others, so its data is included in *Line of Charge Simulation.gz*.

Conclusion:

Overall, the program appears mostly consistent with the predictions of special relativity. Moreover, the framework is general enough to test a multitude of scenarios—ranging from the barn-and-ladder paradox to the relativity of electromagnetism. Furthermore, the ability to visually see these systems develop from arbitrary points of view offers a useful sense for how different observers report their surroundings. In each of these respects—simulation and visualization—the program succeeded in its initial goes. Unfortunately, it is not without its flaws.

Problems with Accuracy and Lightspeed:

Firstly, as mentioned previously, this program experiences an accuracy problem with sufficiently high forces, where particles may jump over the speed of light. A possible fix for this issue could be to use relativistic *rapidity* in favor of velocity. Rapidity is defined as:

$$w = \tanh^{-1}\left(\frac{v}{c}\right)$$

Unlike regular velocity, rapidity is additive between reference frames. A consequence of this is that rapidity can be simulated as normal, given an acceleration. From there, a velocity can be recovered for use in updating position—without ever exceeding the speed of light, due to the nature of the hyperbolic tangent function.

Problems with Non-Inertial Reference Frames:

Secondly, as mentioned in the *Constant Force* test, the viewing of accelerating reference frames is entirely unreliable. As it turns out, adding support for such frames is significantly more difficult than previously thought—potentially involving Fermi-Walker transport, comoving tetrads, and nightmarish notation that would likely require an entire course to learn. Fortunately, this issue *only* affects viewing non-inertial reference frames. Viewing inertial frames, and even the entire simulation aspect of the program, are unaffected.

References:

David Morin. (2017). *Special Relativity: For the Enthusiastic Beginner*. CreateSpace.

Derek Muller [Veritasium]. (2013, September 23). *How Special Relativity Makes Magnets Work* [Video]. Youtube. https://www.youtube.com/watch?v=1TKSfAkWWN0.

Rod Nave. (1998). Time dilation/length contraction. Georgia State University Department of Physics and Astronomy. http://hyperphysics.phy-astr.gsu.edu/hbase/Relativ/tdil.html

Robert Nelson. (1994). "Erratum: Generalized Lorentz transformation for an accelerated, rotating frame of reference". Journal of Mathematical Physics. Retrieved from https://aip.scitation.org/doi/pdf/10.1063/1.530669.

Krzysztof Rebilas. (2019). "A straightforward method for deriving the Fermi-Walker transport law". European Journal of Physics. Retrieved from https://iopscience.iop.org/article/10.1088/1361-6404/aaff32/pdf

Wolfgang Rindler. (1977). *Essential Relativity: Special, General, and Cosmological*. Springer. Retrieved from https://www.google.com/books/edition/Essential_Relativity/0J_dwCmQThgC?hl=en&gbpv=0.

Proper reference frame (flat spacetime). (2021, January 13). In *Wikipedia*. https://en.wikipedia.org/wiki/Proper_reference_frame_(flat_spacetime).

Rapidity. (2021, March 23). In *Wikipedia*. https://en.wikipedia.org/wiki/Rapidity.

Lorentz transformation. (2021, April 27). In *Wikipedia*. https://en.wikipedia.org/wiki/Lorentz_transformation.