A multifunctional hamiltonian mechanics simulator and some of its application examples

詹有丘 (Youqiu Zhan)

Friday 11th September, 2020

Abstract

We have developed convenient online software that can simulate hamiltonian mechanics. Users can see the motion of a system as long as he tells the simulator the hamiltonian of the system and the initial conditions of it. The simulator can also analyze the oscillation pattern of an oscillator by using FFT to derive the frequency domain of the motion. The simulator is small and fast and is convenient and easy to operate and customize. The user interface is simple (a graphics interface for simple basic operations and a console interface for other operations). The simulator can also output data of the simulated system to create datasets for other potential usages. There are a lot of applications that can be done with it.

Keywords— hamiltonian, simulation, mechanics, visualization

Contents

1 Introduction

Theoretical mechanics is a subject difficult to study. One can often find that he cannot find out what the motion of a mechanics system looks like, so he may probably want to see the graph of the motion. A software that can output the motion of a system as long as the mechanics system is defined is called a mechanics simulator.

There have been a lot of mechanics simulators available on the Internet, but most of them have one of the following disadvantages:

1. Too massive. Some simulators are really powerful, but the cost is its massive volume. This makes them unportable.

2. Not convenient enough. Most of the simulators require the user to download the program files into the disk. This is inconvenient because you have to reinstall the software when you use another device.

3. Not customizable enough. Some simulators focuses on usual models in real life. It may be useful when it comes to rigid body contact problems. However, it is usually not shipped with functions to simulate systems like arbitrary central force field or special relativity problems.

4. Unable to output data. Some simulators aims at present how the appearance of the system change, but they lack a convenient interface to output the very data of the motion of the system.

5. Too difficult to operate. Some powerful simulators have very complicated user interface, which requires the user to study for hours to being able to operate the simulator and check the result. This is not friendly to new users.

Because of this, we wanted to create a simulator to solve the problems above. To make it convenient to use, it should be hosted on a webpage so that everyone with a browser can have access to it as long as he can access the Internet.

With such a simulator, one can study theoretical mechanics more conveniently. He can have perceptual congnition about specific mechanics systems. The simulator can also be used as animated presentation for physics education or lecture.

2 List of symbols

Note that if a symbol has domain $A \to B$, which means a function from set A to set B, then it can sometimes represent the value of the function and lies in the domain B. In other words, if it says that $f : A \to B$ is a function w.r.t. x, then f can be a abbreviation of $f(x)$.

We always assume that functions we encounter have good enough properties as long as we need to use this property.

The list of symbols is shown in Table 1. The list of mathematical operations is shown in Table 2.

Since all quantities are implemented numerically in the computer program as floating numbers, the quantities do not necessarily share the same units as in the real world but use other more convenient units like pixel or tick, so the units are not mentioned in the lists.

There are some model-specific symbols mentioned in examples in Section 6. They are not included the the list of symbols, but their specific meanings are explained in the section.

Table 1: List of symbols

Symbol	Domain	Meaning	Value
t	$\mathbb{R}$	Time	
Δt	$\mathbb{R}^+$	The increment step of the ODE solver	5×10^{-4}
ι	$\{2\zeta \vert \zeta \in \mathbb{Z}^+\}$	The dimension of the vector in the ODE	
$\mathbf{x}$	$\mathbb{R} \to \mathbb{R}^\iota$	The state of a system w.r.t. t	$(\mathbf{q}, \mathbf{p})$
$\mathbf{q}$	$\mathbb{R} \to \mathbb{R}^{\iota/2}$	Generalized coordinates w.r.t. t	
$\mathbf{p}$	$\mathbb{R} \to \mathbb{R}^{\iota/2}$	Generalized momentum w.r.t. t	
$\mathcal{H}$	$\mathbb{R} \times \mathbb{R}^\iota \to \mathbb{R}$	Hamiltonian w.r.t. $(t, \mathbf{x})$	
$\boldsymbol{\omega}$	$\mathbb{R}^{\iota \times \iota}$	A matrix to make symplectic gradients	$\begin{bmatrix} \mathbf{O} & \mathbf{I}_{\iota/2} \\ -\mathbf{I}_{\iota/2} & \mathbf{O} \end{bmatrix}$
ξ	$\mathbb{R}$	Abscissa on the canvas, in pixel	
η	$\mathbb{R}$	Ordinate on the canvas, in pixel	
m_t	$\mathbb{R} \to \mathbb{R}$	The mapping from actual t to the ξ-coordinate on the canvas	
m_y	$\mathbb{R} \to \mathbb{R}$	The mapping from actual $\mathbf{x}$ component to the η-coordinate on the canvas	
w	$\mathbb{Z}^+$	The width of the graphics screen	1024
h	$\mathbb{Z}^+$	The height of the graphics screen	768
y	$\mathbb{R}$	The component of $\mathbf{x}$ to be plotted	
N	$\mathbb{Z}^+$	Number of samples to calculate DFT	1×10^5
W	$[0, 1) \to \mathbb{R}$	The window function	See Equation 6

Table 2: List of operations

Symbol	Name	Definition
$\dot{f}$	Complete derivative[1] of f w.r.t. t	$\frac{\mathrm{d}f}{\mathrm{d}t}$
Δf	Complete change[2] in f when t becomes $t + \Delta t$	$f(t + \Delta t) - f(t)$
$\sum_j^n r_j$	Sum of n numbers w.r.t. index j[3]	$\sum_{j=0}^{n-1} r_j$
$\zeta \,\%\, \chi$[4]	Remainder of ζ divided by χ	$\zeta - \chi \left\lfloor \frac{\zeta}{\chi} \right\rfloor$

3 Physics theory

3.1 Predicting dynamics

A dynamics system's motion can be predicted using some ordinary differential equations (ODE) in form of

$$\dot{\mathbf{x}} = \mathbf{f}(t, \mathbf{x}), \tag{1}$$

where $\mathbf{f} : \mathbb{R} \times \mathbb{R}^\iota \mapsto \mathbb{R}^\iota$ is a function related to the specific system, and ι is some positive integer which should unnecessarily be the degree of freedom (DOF) of the system (it is actually 2 times DOF in our case).

[1]Complete derivative means that: if f is a function w.r.t. g, and g is a function w.r.t. t, then $\dot{f}$ denotes $\frac{\mathrm{d}}{\mathrm{d}t} f(g(t))$.

[2]Complete change is similar to complete derivative. See Footnote 1.

[3]By conventions in computer science, indices start from 0 instead of 1. The convention will be followed in the article.

[4]This notation is from conventions in computer science.

3.2 ODE solver

This ODE (Equation 1) can be solved numerically using the Runge–Kutta method[5]

$$\Delta\mathbf{x} \approx \Delta t \sum_{j}^{s} b_j \mathbf{K}_j, \tag{2}$$

where $\mathbf{K}_j$ is defined recursively as [10, p. 907]

$$\mathbf{K}_j := f\left(t + \Delta t \sum_{k}^{j} a_{j,k}, \mathbf{x} + \Delta t \sum_{k}^{j} a_{j,k}\mathbf{K}_k\right). \tag{3}$$

The smaller Δt is, the more precise and the less efficient the solver is.

The order number s and the coefficients b_j and $a_{j,k}$ are specific for different Runge–Kutta methods. Here the 3/8-rule [5, p. 138] is adopted. The coefficients of it is shown in Table 3.

Table 3: The coefficients of Runge–Kutta method 3/8-rule

	$a_{j,k}$				b_j
j \ k	0	1	2	3	
0					$1/6$
1	$1/3$				$1/3$
2	$-1/3$	1			$1/3$
3	1	-1	1		$1/8$

The ODE solver should store t and $\mathbf{x}$, and every time it increments, log $(t, \mathbf{x})$ and let $\mathbf{x} \leftarrow \mathbf{x}+\Delta\mathbf{x}$ and $t \leftarrow t+\Delta t$.

The ODE solver can give the numerical value of $\mathbf{x}$ at any t as long as the numerical form of $\mathbf{f}$ and a initial value $\mathbf{x}(0)$ is given.

3.3 Constructing ODE

Only an ODE solver does not help with simulating a dynamics. The ODE is required. According to the theorems in physics, there are a lot of methods to construct the ODE of a dynamics system. Here the hamiltonian method is adopted.

The hamiltonian mechanics states that, for some dynamics system, there exists a function $\mathcal{H} : \mathbb{R}^\iota \times \mathbb{R} \to \mathbb{R} : (t, \mathbf{x}) \mapsto \mathcal{H}(t, \mathbf{x})$ such that the motion of the system satisfy the equation called **canonical equation**[6]

$$\dot{\mathbf{x}} = \boldsymbol{\omega}\nabla_{\mathbf{x}}\mathcal{H}, \tag{4}$$

where $\boldsymbol{\omega}\nabla_{\mathbf{x}}$ denotes the **symplectic gradient** w.r.t. $\mathbf{x}$.

Under the circumstance of hamiltonian mechanics, the vector $\mathbf{x} \in \mathbb{R}^\iota$ can be separated into two vectors $\mathbf{q} \in \mathbb{R}^{\iota/2}$ and $\mathbf{p} \in \mathbb{R}^{\iota/2}$, which are respectively the **generalized coordinates** and **generalized momentum** of the system, so the components of $\mathbf{x}$ can be called `q0`, `q1`, `p0`, `p1`, etc..

[5]The algorithm can be expressed more briefly using Ruby programming language:

```ruby
dx = b.zip(a).each_with_object([]).sum do |(bj, aj), ary|
  bj * ary.push(f.(t+aj.sum*dt, x+aj.zip(ary).sum{_1*_2}*dt)).last
end * dt
```

[6]The canonical equation is usually denoted as

$$\dot{\mathbf{q}} = \frac{\partial\mathcal{H}}{\partial\mathbf{p}}, \quad \dot{\mathbf{p}} = -\frac{\partial\mathcal{H}}{\partial\mathbf{q}}$$

in other books [6][2, p. 65][9, p. 132].

Since the gradient $\nabla_{\mathbf{x}}\mathcal{H}$ can be calculated numerically easily, we can give the numerical form of $\mathbf{f}$ in Equation 1 according to

$$\mathbf{f}(t, \mathbf{x}) := \boldsymbol{\omega}\nabla_{\mathbf{x}}\mathcal{H} \tag{5}$$

and can thus solve Equation 1 numerically according to the method described in Section 3.2.

3.4 Analyzing the frequency domain of the motion

When we study the periodical motion of a dynamics system, it is usually interesting to study its frequency domain. Therefore, we want the simulator to be shipped with the ability to show the Fourier transformation (FT) of the motion (on an interval of time $[0, N\Delta t)$). Because we do this numerically, and t is actually discrete, so what we calculate is actually the discrete Fourier transformation (DFT).

The FFT library mentioned in Section 4.2 provides the method to calculate the DFT. Although we can just pick a time interval long enough and calculate its DFT, the operation can result in some loss in the frequency domain [7]. We should apply a window function to the motion on the interval before calculating DFT.

There are various window functions candidates, each of which have its unique application scenes. Here the Hamming window function [7]

$$W(\zeta) := \frac{25}{46} - \frac{21}{46}\cos(2\pi\zeta) \tag{6}$$

is adopted because it fits with most cases.

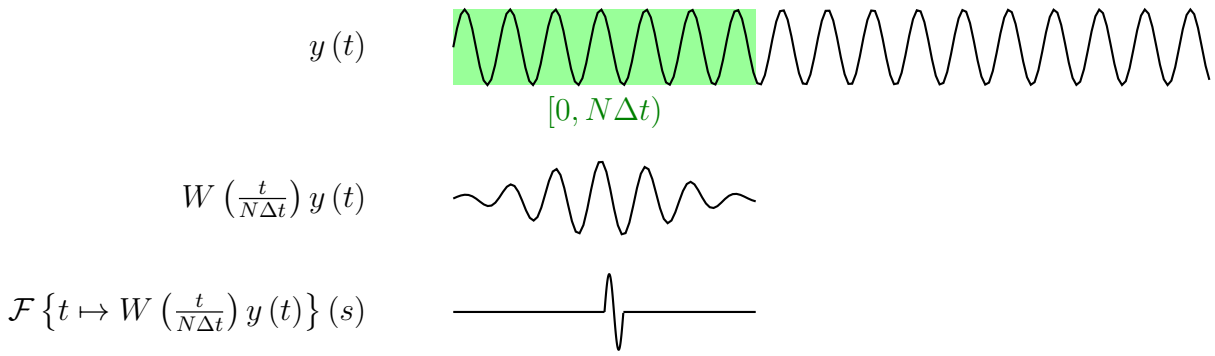


Figure 1: The process of deriving the frequency domain

4 Libraries used by the simulator

Our simulator depends on some third-party libraries.

The graphics library is used to show graphs on the screen.

The FFT library is used to calculate the DFT of the motion of the system.

The ODE constructor and ODE solver is written by us, independent to third-party libraries. The theories about them are explained in Section 3.

4.1 Graphics library

We use rpg_core.js to draw and show graphs. It is a web game engine based on PixiJS. Although rpg_core.js is shipped with RPG Maker MV, which is not a free software, it is open-source on GitHub.

In rpg_core.js, a `Bitmap` object is used to store the info of a picture, and a `Sprite` object is used to present the picture depicted by a `Bitmap`. The coordinates info etc. are also in the `Sprite` [1]. Figure 2 shows how rpg_core.js shows a picture.

The `x` and `y` property of a `Sprite` can be adjusted to move a picture.

rpg_core.js also provides methods to fill a rectangular region on a `Bitmap` with a certain color. This enables us to set the color of pixels on a `Bitmap` and can thus draw graphs.

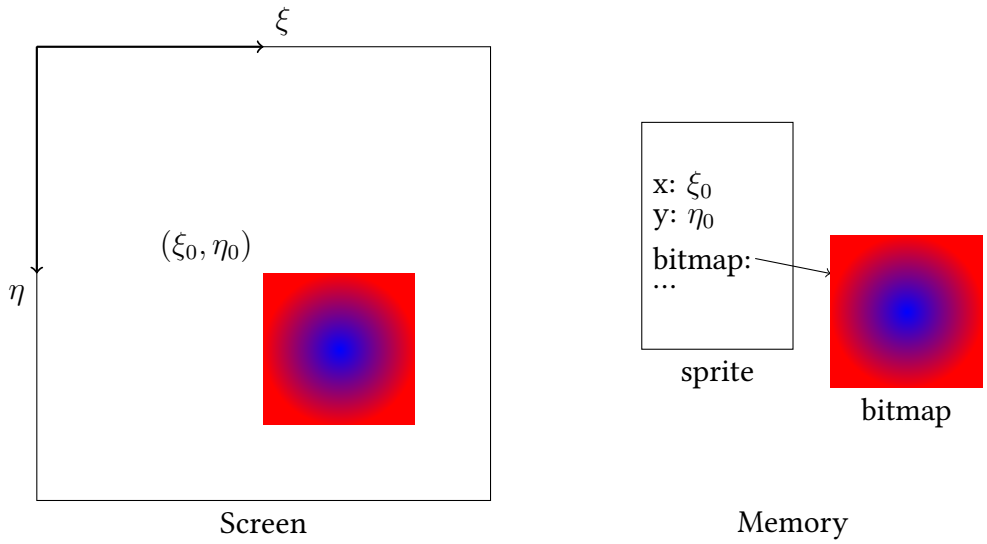


Figure 2: How rpg_core.js shows a picture

4.2 FFT library

A FFT library implements the fast Fourier transformation (FFT) algorithm. The FFT library we use is FFTW, which is written in C. Since we need to use it on the webpage, we used emscripten to import it.

5 Plotting the graph

According to the theories described in Section 3, a program can be designed to give the generalized coordinates $\mathbf{q}$ and generalized momentum $\mathbf{p}$ at any t according to the input hamiltonian $\mathcal{H}$ and initial values $\mathbf{q}(0)$ and $\mathbf{p}(0)$.

However, a person can hardly figure out the patterns in the motion by just looking at a bunch of $(t, \mathbf{x})$ pairs. To make it easier to find out the patterns, the simulator should be capable of plotting a graph according to the $(t, \mathbf{x})$ pairs.

To be more specific, for each component y of $\mathbf{x}$, on the ξ-η plane (the canvas), the point $(m_t(t), m_y(y))$ should be plotted. The introduction of m_t and m_y is because the coordinates on the canvas are in pixel, which is a small unit. Another purpose of m_t and m_y is to make it possible to use nonlinear scales like logarithmic scale.

5.1 Scrolling the graph

Since people often want to simulate a system for a long time, which makes the graph very wide, so the canvas should be much wider than the screen. Then we must make the canvas scroll as the simulator simulates the system.

Although nowadays computers can draw pictures on the screen very fast and can redraw it every $1/60$ seconds, the read-write operations to the `Bitmap` is time-consuming. Therefore, here we implement a scroll algorithm similar to Carmack scroll algorithm. Using this method, the computer only need to change one pixel on the `Bitmap` instead of tens of thousands of them every time when the ODE solver increments.

The algorithm requires 2 sprites, respectively called sprite 1 and sprite 2, both of which shows a bitmap of width w and height h (so there are 2 bitmaps altogether), where w and h are also the width and height of the graphics screen.

When the ODE solver increments, sprite 1 and sprite 2 move left by Δm_t. Now, the coordinates of sprite 1 are $(w - (m_t(t) \,\%\, w), 0)$, and the coordinates of sprite 2 are $(-(m_t(t) \,\%\, w), 0)$. Fill the pixel at $(m_t(t) \,\%\, w, m_y(y))$ on the bitmap of sprite 1. This process is shown in Figure 3.

When the two sprites move left enough, sprite 1 touches the left side of the screen. At this moment, sprite 2 suddenly moves to the right to sprite 1, clears its bitmap, and exchange its name with sprite 1. This process is shown in Figure 4.

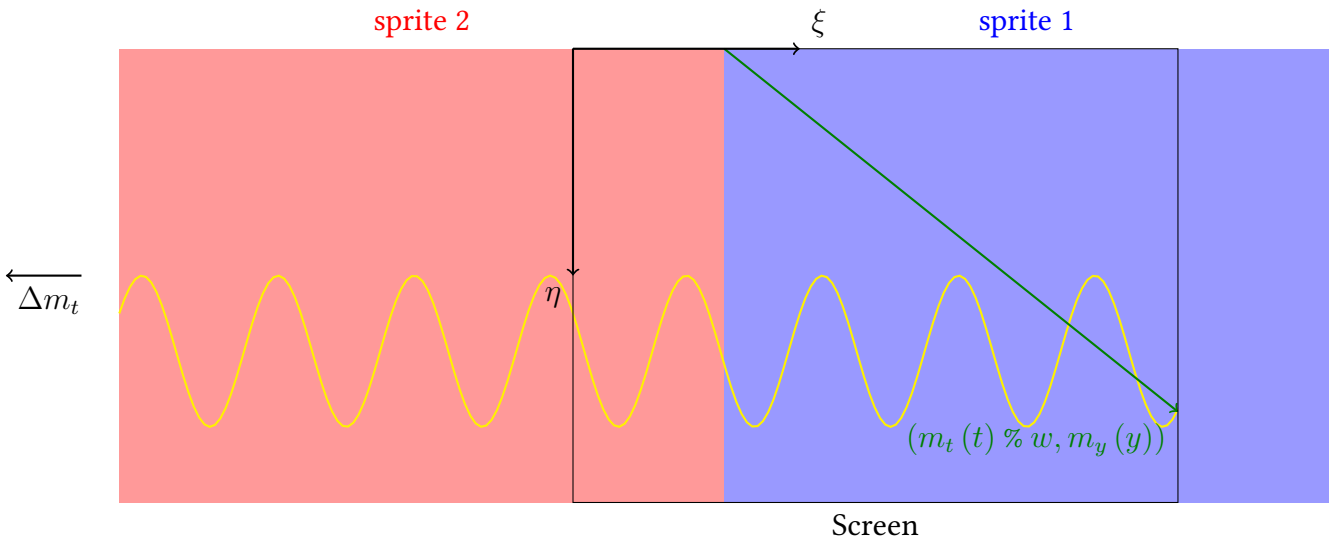


Figure 3: How the sprites move and the bitmaps are plotted as the ODE solver increments

5.2 Presenting the frequency domain of the motion

In Section 3.4, it is said that we need to be able to analyze the frequency domain of the motion. Since the frequency domain is often discrete and sparse, and the high-frequency region is often almost zero, it is better to spread out the low-frequency region across the width of the screen and connect adjacent points with lines.

Because rpg_core.js does not ship with a method drawing a straight line on the bitmap, we need to implement an algorithm to draw straight lines. Here Bresenham's line algorithm [4] is adopted.

6 Guides for operating

When you open the webpage, it will start simulating the default model, which is a 1-dimensional vibration with parametric vibration [9, p. 82] and alternating external force [9, p. 61]

$$\mathcal{H}(t,q,p) = \frac{p^2}{2} + \omega^2 (1 + u\cos(\gamma t))\frac{q^2}{2} - fq\cos(\kappa t + \beta), \tag{7}$$

where

$$(u, \gamma, \beta, \kappa, \omega, f) = (0.3, 21.7, 0.2, 8, 10, 20),$$

and the initial conditions are

$$(q, p) = (2, 0).$$

The frequency domain calculator feature is enabled. After some time, when there have accumulated enough samples, the simulator will create buttons at the up-left corner of the screen clicking which will make it show the time domain to be FFTed and the calculated frequency domain.

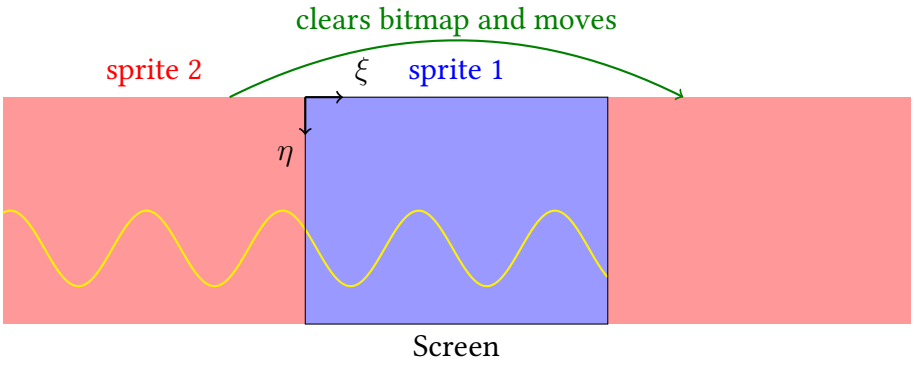
In the interface of frequency domain, the line colored the same as the button represents the real part of the FFT result, and the line colored gray represents the imaginary part of the FFT result.

Hit [ ] to pause or resume the simulation.

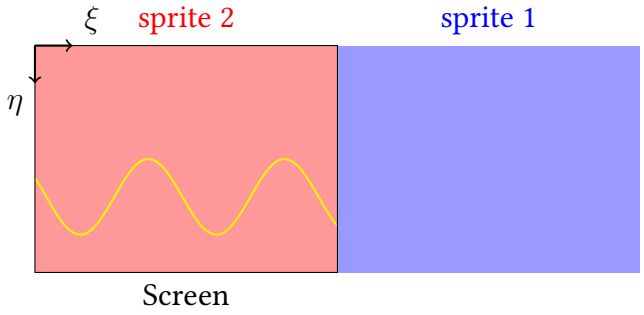
Figure 6 shows the graphics interface of the simulator. From the result of the simulation, it seems that we can indeed learn something about the pattern of the motion. The frequency domain is very clear.

As described in Section 1, the simulator is highly customizable. You can customize the simulation by coding in the console[7].

[7]For most browsers, hit [F12] to have access to the console. The interface of the console panel usually looks like Figure 5.



(a) Sprite 2 suddenly clears its bitmap and moves



(b) Sprite 1 and sprite 2 swap their names

Figure 4: How the sprites suddenly move and swap

6.1 Downloading the motion data

The simulator will not record the history of the ODE solver by default. To ask the simulator to record the history, run the following codes.

```
rungeKutta.recordHistory = true;
restart();
```

Wait the ODE solver for some time for it to accumulate enough data, and then run the following codes to download the simulated data.

```
rungeKutta.downloadHistory(0, 30);
```

Replace `0` and `30` with the ends of interval of time during which your desired data was simulated out. Omitting the 2 parameters will make it download all data accumulated so far.

6.2 Changing the parameters of the default model

For instance, now we want to study the pattern of parametric resonance. For a parametric vibration model, when $\gamma \approx 2\omega$, the condition for it to reach parametric resonance is[8] [9, p. 82]

$$|\gamma - 2\omega| < \frac{1}{2}\omega u. \tag{8}$$

We want to study how the motion looks like when it reaches parametric resonance. We can let $f := 0$ to cancel the external force and let $\gamma := 21.3$ to make the system meet the condition of parametric resonance. Then, run `restart()`.

[8] Precision to $\mathrm{O}(u)$.

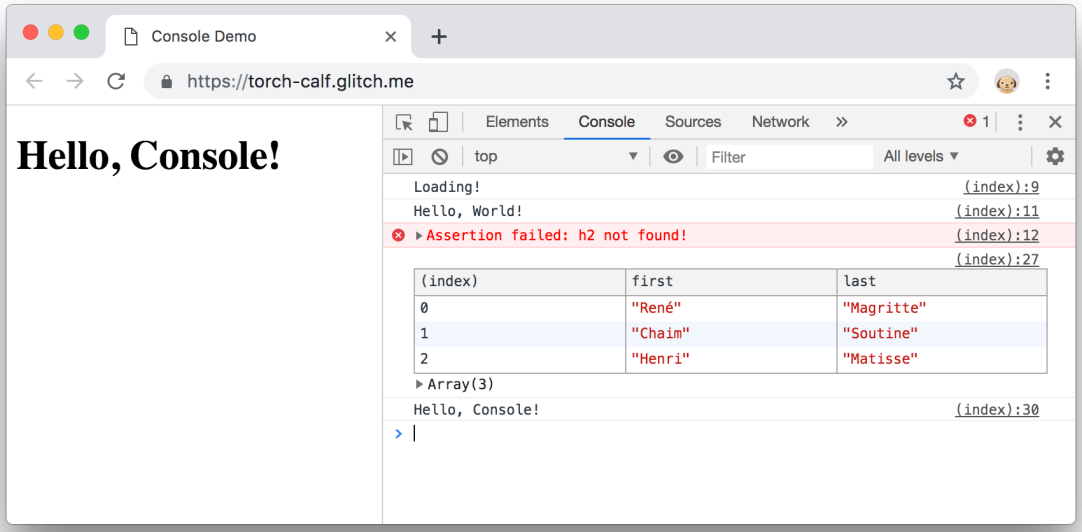


Figure 5: The console panel of Chrome browser [3]

```
f = 0;
gamma = 21.3;
restart();
```

6.3 Changing the initial conditions

The initial conditions $\mathbf{x}(0)$ can be set to a customized one by running the following codes.

```
rungeKutta.initial = [2, 0];
restart();
```

Change `[2, 0]` into any initial conditions you like.

6.4 Changing the scale

It can be found that the amplitude indeed grows in an exponential pattern as predicted by theorems. Since exponential growth is very fast, the graph soon exceeds the screen. We can use logarithmic scale to prevent this. This can be done by changing m_y using the following codes.

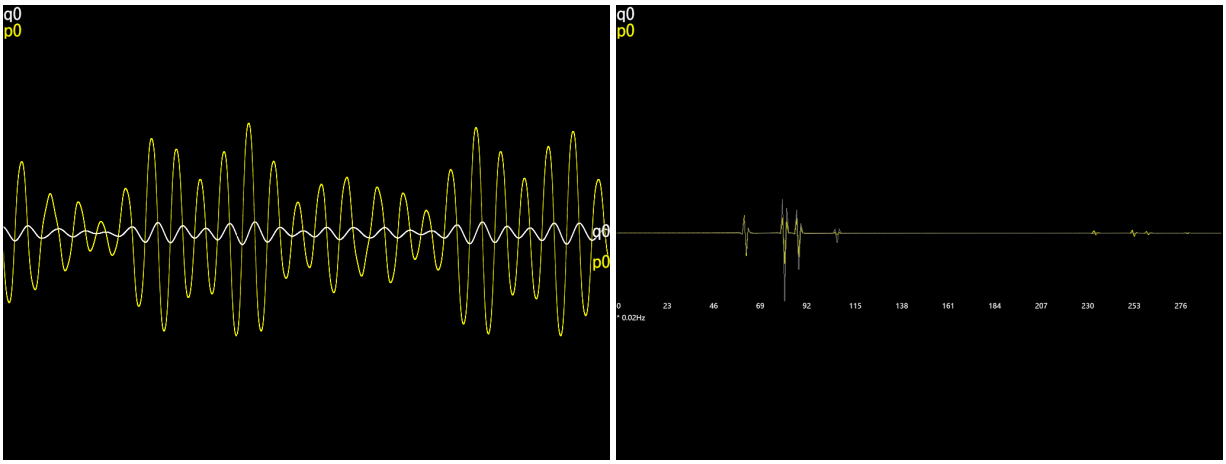
```
canvas.mappingY = y => 20 * log1p(abs(y)) * sign(y) + Graphics.height/2;
restart();
```

Now $m_y(y) := 20 \ln(1 + |y|) \operatorname{sgn} y + h/2$. Note that in the codes, `Graphics.width` stands for w and `Graphics.height` stands for h.

The result of the simulation of the parametric resonance can be seen in Figure 7. It can be seen that the amplitude is indeed growing exponentially.

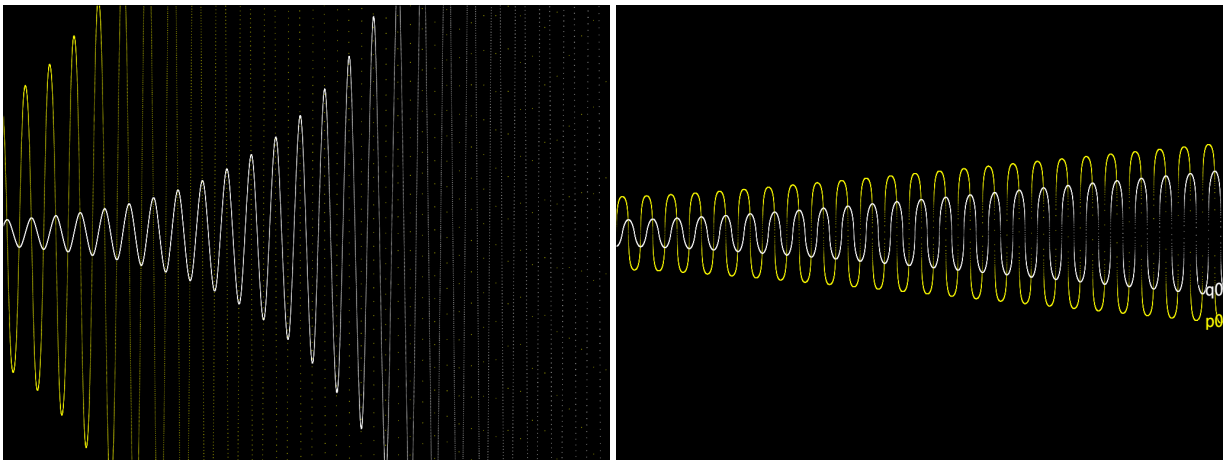
6.5 Disabling analysis of frequency domain

You can also disable the frequency domain analysis using the following codes. After running the codes, there will not be buttons appearing at the up-left corner to indicate the availability of the frequency domain.



(a) The simulated motion of the default model (b) The frequency domain of p of the default model

Figure 6: The screenshots of the simulator simulating the default model



(a) The parametric resonance phenomenon is simulated (b) Using logarithmic scale to prevent the amplitude growing too fast

Figure 7: The simulator simulating the parametric resonance

```
canvas.detectPeriod = false;
restart();
```

6.6 Advanced: drawing the phase path

A phase path is the graph representing the motion of the phase point $\mathbf{x}$ [9, p. 146][2, p. 68].

The simulator has an API to allow the user to run custom codes as the ODE solver increments. To draw the phase path, first you need to create a `Sprite` and a `Bitmap` by following the codes below.

```
var phaseSprite = new Sprite();
phaseSprite.bitmap = new Bitmap(Graphics.width, Graphics.height);
scene.addChild(phaseSprite);
```

Then, change `canvas.onTrace` to run custom codes when a new point is added, and `restart()`.

```
canvas.onTrace = (t, qp) => {
  phaseSprite.bitmap.setPixel(...qp.map(canvas.mappingY), 'white');
  return true;
```

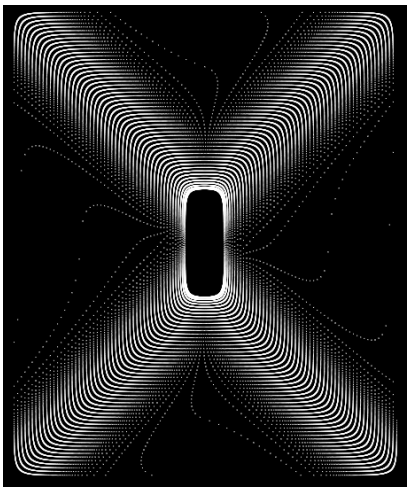


Figure 8: The phase path of the parametric resonance in logarithmic scale

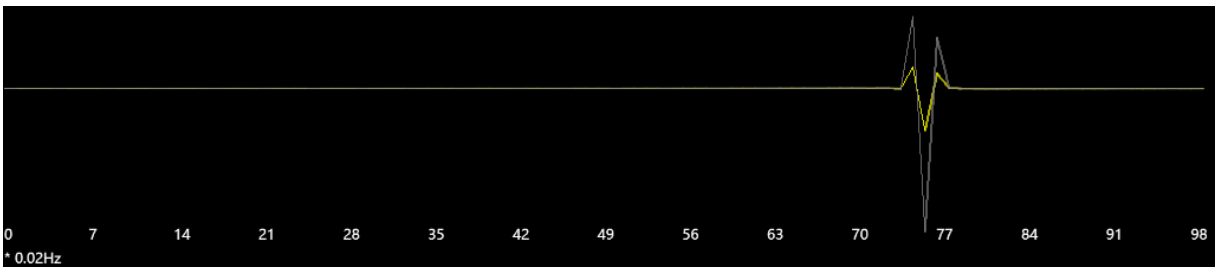


Figure 9: The frequency spectrum of a nonlinear oscillator (higher frequencies are not shown)

```
};
restart();
```

If you want, you can hide the original canvas by running `canvas.visible = false;`, and you can clear the phase path at any time by running `phaseSprite.bitmap.clear();`.

Figure 8 shows the result of the codes above.

The line should be continuous at the edge of each period of the phase path, but the phase point moves so fast that the simulator cannot trace it continuously. The discrete points seem to form curves across the phase path, as can be seen in Figure 8. What are the equations for them? It seems that the simulator can inspire us to ask such questions and encourage us to study something like this about physics.

6.7 Change the hamiltonian into a nonlinear oscillator

The user can assign a function value to the variable `rungeKutta.func` to change the **f** in Equation 1. To create the **f** according to the hamiltonian using Equation 5, using the function `canonicalEquation`, with the first argument be the DOF, and the second argument be the hamiltonian function.

One example is to change the hamiltonian of the default model into that of a nonlinear oscillator

$$\mathcal{H}(t,q,p) := \frac{p^2}{2} + \frac{\omega_0^2 q^2}{2} + \alpha q^3 + \beta q^4$$

by running the following codes. Before applying the codes, if you have runned some codes above, refresh the webpage[9] to have a clean environment to prevent the modifications above affecting.

[9]For most browsers, the shortcut of refreshing is to hit F5.

```
rungeKutta.func = canonicalEquation(1, (t, qp) => {
  let [q, p] = qp;
  return p**2/2 + omega0**2*q**2/2 + alpha*q**3 + beta*q**4;
});
```

The parameters $(\omega_0, \alpha, \beta)$ should be defined. Assume that we take

$$(\omega_0, \alpha, \beta) := (10, -2, -3),$$

then we can run the following codes.

```
var omega0 = 10;
var alpha = -2;
var beta = -3;
```

If we take the initial conditions as $(q, p) = (1, 0)$ using the following codes, the amplitude is now taken as $b = 1$.

```
rungeKutta.initial = [1, 0];
```

We can use the simulator to verify the formula for calculating the frequency of the nonlinear oscillator [9, p. 87][10]

$$\omega = \omega_0 + \left(\frac{3\beta}{2\omega_0} - \frac{15\alpha^2}{4\omega_0^3}\right) b^2 = 9.535.$$

The frequency spectrum is shown in Figure 9. As can be seen, the frequency is roughly the theoretically predicted value $\frac{\omega}{2\pi} = 1.518$, different from the linear one $\frac{\omega_0}{2\pi} = 1.592$.

7 Examples

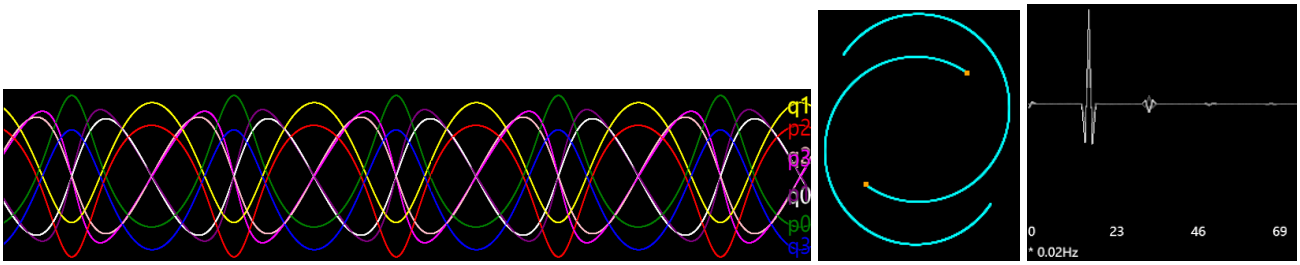
The default model and the typical customizations to it presented in Section 6 are good examples of the application of the simulator.

Here are some other examples. The codes of the examples can also be seen on the webpage.

7.1 Kepler's 2-body problem

2-body problems are systems with 4 DOF. The hamiltonian of the model to be simulated is

$$\mathcal{H}(t, q_0, q_1, q_2, q_3, p_0, p_1, p_2, p_3) := p_0^2 + p_1^2 + p_2^3 + p_3^2 - \frac{500}{\sqrt{(q_0 - q_2)^2 + (q_1 - q_3)^2}}.$$



(a) The graph of $\mathbf{x}$, 8 curves in total (b) Visualization (c) Frequencies of q_0

Figure 10: The simulator simulating Kepler's 2-body problem

Run the following codes and you will get 2 stars move around each other.

[10] Precision to $O(b^2)$.

```
// There are 8 curves to be drawn on the canvas
canvas.n = 8;
// Setting the hamiltonian of the system
rungeKutta.func = canonicalEquation(4, (t, qp) => {
  let [x1, y1, x2, y2, px1, py1, px2, py2] = qp;
  return px1**2 + py1**2 + px2**2 + py2**2 - 500 / hypot(x1-x2, y1-y2);
});
// Setting the initial conditions of the system
rungeKutta.initial = [-3, -3, 3, 3, 2, -3, -2, 3];
// Setting the scale of the graph
canvas.mappingY = y => 20 * y + Graphics.height/2;
// Setting the colors for graphing; there are 8 curves and thus 8 colors
canvas.colors = ["white", "yellow", "pink", "blue",
                 "green", "purple", "red", "magenta"];

// Following codes creates the sprites and bitmaps for visualization
// (Using API provided by rpg_core.js, see [1] for help)
var traceSprite = new Sprite();
var star1 = new Sprite();
var star2 = new Sprite();
scene.addChild(traceSprite);
scene.addChild(star1);
scene.addChild(star2);
traceSprite.bitmap = new Bitmap(Graphics.width, Graphics.height);
star1.bitmap = star2.bitmap = new Bitmap(4, 4);
star1.bitmap.fillAll('orange');
star1.anchor.x = star1.anchor.y = 0.5;
star2.anchor.x = star2.anchor.y = 0.5;

// Updating the state of the sprites when a new sample comes out
canvas.onTrace = (t, qp) => {
  // Setting the position of sprites of stars
  [star1.x, star1.y, star2.x, star2.y] =
      qp.slice(0, 4).map(canvas.mappingY);
  // Plotting the trajectory
  traceSprite.bitmap.setPixel(star1.x, star1.y, 'cyan');
  traceSprite.bitmap.setPixel(star2.x, star2.y, 'cyan');
  return true;
};

// Restarting to apply the changes
restart();
```

The simulated result is shown in Figure 10.

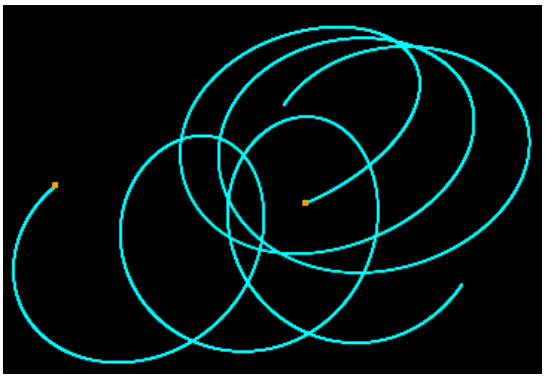
If the user is curious, the hamiltonian or the initial conditions can be modified a little to create interesting graphs.

The line `rungeKutta.initial = [-3, -3, 3, 3, 2, -3, -2, 3];` can be modified to apply different initial conditions. For different initial conditions, the trajectory can appear as a different shape. If we change `2, -3, -2, 3` into `5, -2, -5, 2`, the trajectory is hyperbola as shown in Figure 11.

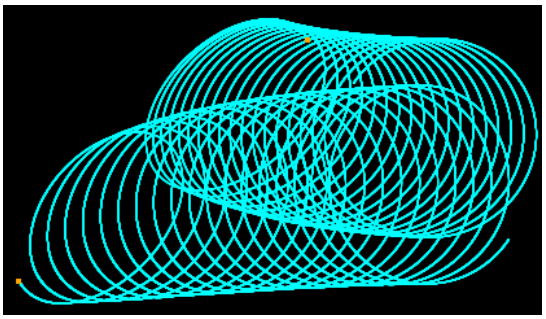
The line `return px1**2 + py1**2 + px2**2 + py2**2 - 500 / hypot(x1-x2, y1-y2);` can be modified to apply a different hamiltonian to the system. If you change `x1-x2` into `1.5*x1-x2`, what you get is a chaotic motion as shown in Figure 11.

7.2 Adiabatic invariants

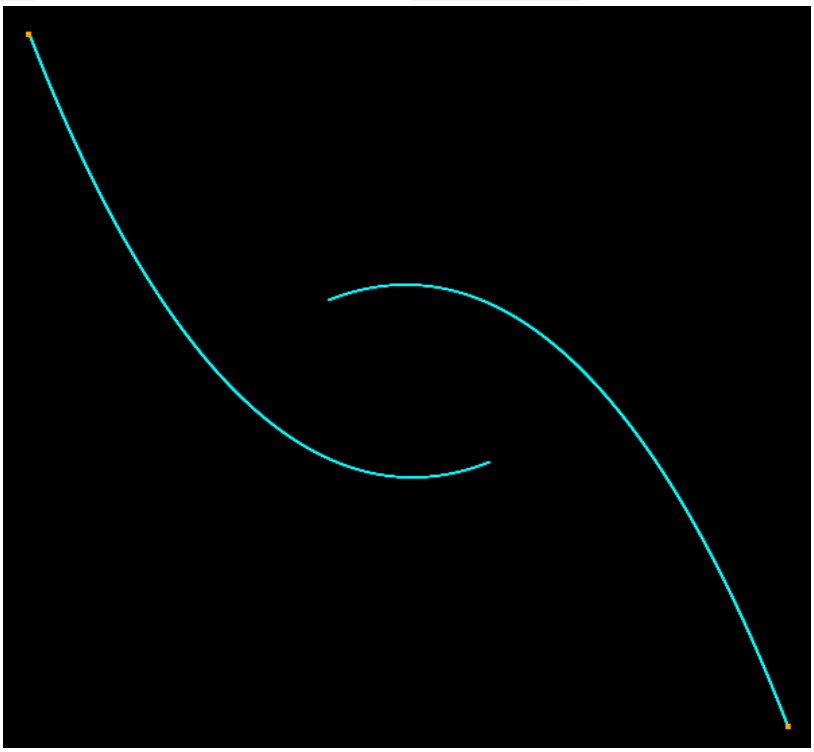
The action variable of a periodic 1-dimensional system is an adiabatic invariant, which does not vary when the parameters in hamiltonian change slowly [2, p. 298][9, p. 156].



(a) The code `x1-x2` is changed into `1.5*x1-x2`



(b) The code `x1-x2` is changed into `1.1*x1-x2`



(c) The code `2, -3, -2, 3` is changed into `5, -2, -5, 2`

Figure 11: The Kepler's 2-body problem model is modified

A system related to adiabatic invariance is a little difficult to imagine. We want to use the simulator to show how adiabatic invariants work. The codes shown below can be modified to simulate other systems related to adiabatic invariance.

Here the example adopted is the harmonic oscillator with its frequency altering slowly. The hamiltonian is

$$\mathcal{H}(t,q,p) = \frac{p^2}{2} + \omega(t)^2 \frac{q^2}{2},$$

where $\omega(t) := 4 + 0.05t$ changes slowly w.r.t. t.

By the definition of the action variable, we can derive that the action variable for the harmonic oscillator is [2, p. 300][9, p. 157]

$$I = \frac{\mathcal{H}}{\omega}.$$

We are going to make the canvas draw graph for I and $\mathcal{H}$. Run the following codes.

```
// The altering parameter omega mentioned above; it should change slowly
var omega = t => 4 + 0.05 * t;
// The hamiltonian of the system
var hamiltonian = (t, qp) => qp[1]**2/2 + omega(t)**2 * qp[0]**2/2;
// Setting the hamiltonian of the system
rungeKutta.func = canonicalEquation(1, hamiltonian);
// There are 8 curves to be drawn on the canvas
canvas.n = 4;
// The labels of the curves will be q, p, H, I
canvas.getLabelString = i => 'qpHI'[i];
// The colors for the curves
canvas.colors = ["white", "yellow", "pink", "blue"];
// Tracing the H and I data when a new sample comes out
canvas.trace = function (t, data) {
  // Calculating the hamiltonian at this time
  let h = hamiltonian(t, data);
  // To be drawn on the graph at this time: [q, p, h, h/omega]
  data = data.concat([h, h / omega(t)]);
  // Calling old method, a JavaScript trick
  return this.__proto__.trace.call(this, t, data);
};
// Restarting to apply the changes
restart();
```

The definition of the slowly altering parameter and the hamiltonian is marked with comments. Feel free to modify them to simulate other systems with adiabatic invariants.

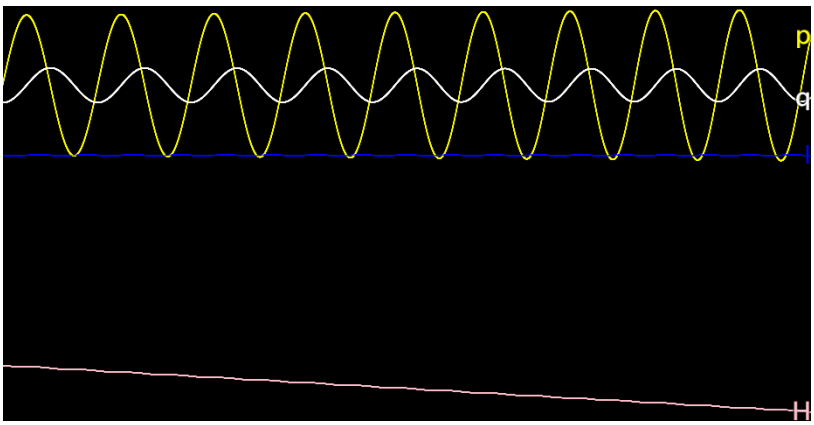


Figure 12: The adiabatic invariant of a harmonic oscillator with slowly changing parameter

Figure 12 shows the result. As can be seen, the same as theoretically predicted, while ω and thus $\mathcal{H}$ changes slowly, I does not vary.

7.3 Scattered beam of particles

Suppose there is a beam of identical particles being shot toward a central force field. Each of the particles of the beam has hamiltonian

$$\mathcal{H}(t, q_0, q_1, p_0, p_1) = p_0^2 + p_1^2 + \frac{30}{\sqrt{q_0^2 + q_1^2}}.$$

The beam will be scattered, and particles with different impact parameter will have different angle of scattering [9, p. 49]. We want to study this phenomenon using the simulator.

The simulation can be done using the codes below. Note that on low-performance devices, the simulation is slow because there are 30 motions to be simulated at the same time.

```
// The total number of particles in the beam
var n = 30;
// The array of the n ODE solvers
rungeKuttas = [];
for (let i = 0; i < n; i++) {
  // Create an ODE solver
  rungeKuttas[i] = RungeKutta.solveHamiltonian(
    // Parameters list:
    2,                              // DOF
    [-20, (i - n/2)*0.3, 4, 0],     // Initial conditions
    Number.POSITIVE_INFINITY,       // Maximum time
    null,                           // Canvas; null for no canvas
    (t, qp) => {                    // The hamiltonian
      let [x, y, px, py] = qp;
      return px**2 + py**2 + 30/hypot(x,y);
    }
  );
}

// Create the sprite and bitmap for visualization (rpg_core.js API)
var traceSprite = new Sprite();
scene.addChild(traceSprite);
traceSprite.bitmap = new Bitmap(Graphics.width, Graphics.height);

// The scale to be used for graphing
var my = y => 20 * y + Graphics.height/2;
// This is the function to be called at each frame refreshing
update = function () {
  for (let i = 0; i < n; i++) {
    // Calculate the coordinates of the point to be plotted
    let xy = [0, 1].map(j => my(rungeKuttas[i].current[j]))
    // Plotting the trajectory
    traceSprite.bitmap.setPixel(...xy, 'white');
    // Incrementing ODE solvers
    rungeKuttas[i].update();
  }
};

// Restarting to apply the changes
restart();
// Hiding the original canvas
canvas.visible = false;
```

The result of the codes above can be seen in Figure 13.

The effect of scattering forms a graph rather beautiful. It can be seen that the envelope of the trajectories of the particles looks like a conic section, which is not thought about before we simulate it. Then, we can be curious about what is the equation for the envelope because it is meaningful as it indicates the safe place where the particles will not reach.

Simple simulation like this can inspire us to study about physics. This is exactly one of the major usages of the simulator.

One can also study how the initial velocity of the beam of the particle and the field intensity affects the trajectories. We can use the simulator to verify that as the initial velocity decreases or the field intensity increases, the trajectories will be more bending.

The line `[-20, (i - n/2)*0.3, 4, 0],` specifies the initial conditions for the `i` th particle, and the line `return px**2 + py**2 + 30/hypot(x,y);` specifies the hamiltonian of a particle. We can change the codes `4, 0` into `3, 0` to make trajectories more bending, and change the codes `30/hypot(x,y)` into `20/hypot(x,y)` to make trajectories less bending, as can be seen in Figure 14.

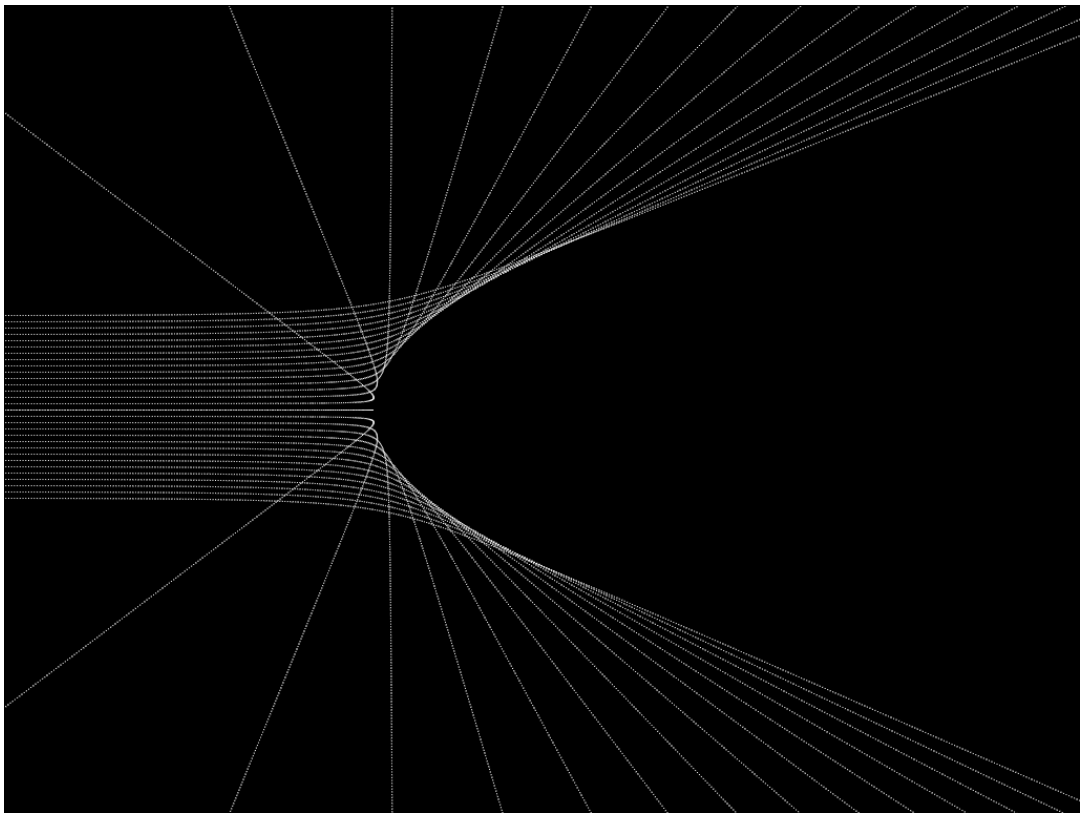


Figure 13: The trajectories of the scattered particles

7.4 Special relativity

Not only classical mechanics, the simulator can simulate special relativity mechanics because relativity mechanics can be depicted by hamiltonian mechanics. Consider a relativity particle in a uniform gravitational field, which has hamiltonian [8, p. 28]

$$\mathcal{H}(t,q,p) := \sqrt{p^2+10} - 0.8q$$

and initial conditions $(q,p) = (-10,-10)$.

The result of the simulation is shown in Figure 15. The particle travels with its momentum uniformly growing, and its speed is nearer and nearer to that of light as $t \to \infty$, as predicted theoretically [8, p. 24].

8 Conclusion

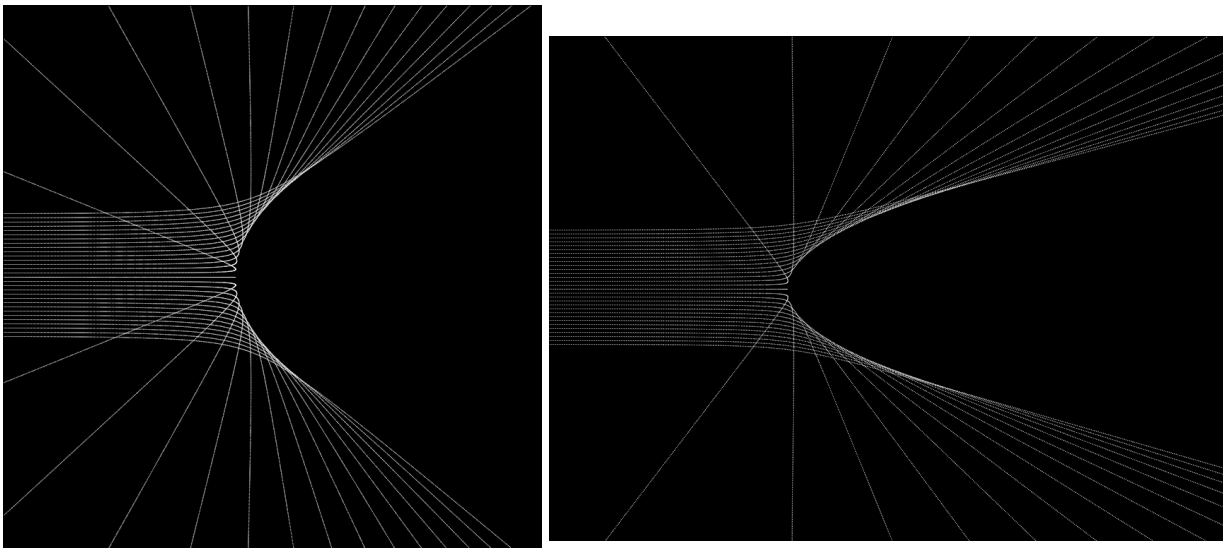
We have developed a convenient online software that can simulate hamiltonian mechanics. There are a lot of applications that can be done with it.

The simulator is a mechanics simulator. After the user input a hamiltonian and the initial conditions of the system, the simulator can simulate the motion of the system, graph the motion of it on the screen. If a vibrating system is being simulated, the simulator can also calculate the frequency domain of the motion using FFT after enough samples have been calculated out.

The simulator has the following advantages:

1. Very small and fast. The simulator is rather simple but efficient. A user with a well-connected computer can start simulating mechanics in seconds without any preparation.

2. Very convenient. Anyone with browser and Internet access can have access to the simulator without the requirement of downloading program files into the disk. The simulator is based on HTML and the programs is written in JavaScript which is supported by almost all browsers.



(a) The codes `4, 0` are replaced with `3, 0`, and the curves become more bending

(b) The codes `30/hypot(x,y)` are replaced with `20/hypot(x,y)`, and the curves become less bending

Figure 14: The parameters of the model of scattered beam of particles is modified

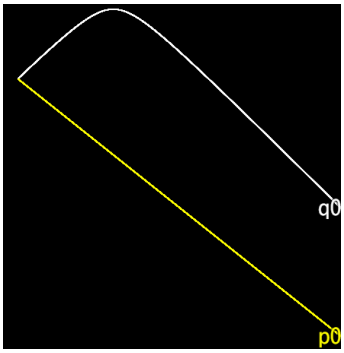


Figure 15: The motion of a relativity particle in uniform gravity field

3. Highly customizable. Everything of the simulator can be customized at ease. The simulated model, the parameters of the ODE solver, the way the simulator presents the system, etc. can all be customized.

4. Being able to output data. The simulated data can be output to create mechanics datasets. The created datasets can be used to study the pattern of a system or be analyzed by a third-party software.

5. Being easy to operate. All that the user need to do to operate the system is to write simple JavaScript codes in the console and click on the screen. The coding is very easy. If advanced usage is not needed, a user without programming experience only needs a few minutes to learn to use it.

The simulator can be used to study motions of hamiltonian systems, learn classical theoretical mechanics, create animation image for presentation related to physics or online physics courses, and create dataset of motions of a hamiltonian system.

We wrote the default model for illustrating the basic applications, mentioned in Section 6, and mentioned some other examples for proposing some further applications in Section 7.

It is available on a webpage.

References

[1] Rpg maker mv help.

[2] V. I. Arnol′d, K. Vogtmann, and A. Weinstein. *Mathematical methods of classical mechanics*. Springer, 2nd edition, 1989.

[3] K. Basques. Console overview, 2020.

[4] F. L. Gaol. Bresenham algorithm: Implementation and analysis in raster shape. *Journal of Computers*, 8(1), 2013.

[5] E. Hairer, S. P. Nørsett, and G. Wanner. *Solving ordinary differential equations I: Nonstiff problems*. Springer, 2008.

[6] L. N. Hand and J. D. Finch. *Analytical mechanics*. Cambridge University Press, 2008.

[7] F. J. Harris. On the use of windows for harmonic analysis with the discrete fourier transform. *Proceedings of the IEEE*, 66(1):51–83, 1978.

[8] L. D. Landau and L. E. M. *The classical theory of fields*. Butterworth Heinemann, 2010.

[9] L. D. Landau, L. E. Mikhaĭlovich, J. B. Sykes, and J. S. Bell. *Mechanics*. Butterworth-Heinemann, 3rd edition, 1976.

[10] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical recipes: the art of scientific computing*. Cambridge University Press, 2007.

Acknowledgements

The subject is originally from my study in vibration. I needed to write a convenient mechanics simulator to study its motion. I want to give thanks to my instructor 李晟 (Sheng Li) because in the process, he proposed me some ideas on the functions of the simulator like the FFT analysis.

I want to thank my parents for their supporting me spiritually when I stuck on my study.