模拟电子在匀强磁场之中的运动

周骏锋 17307110073

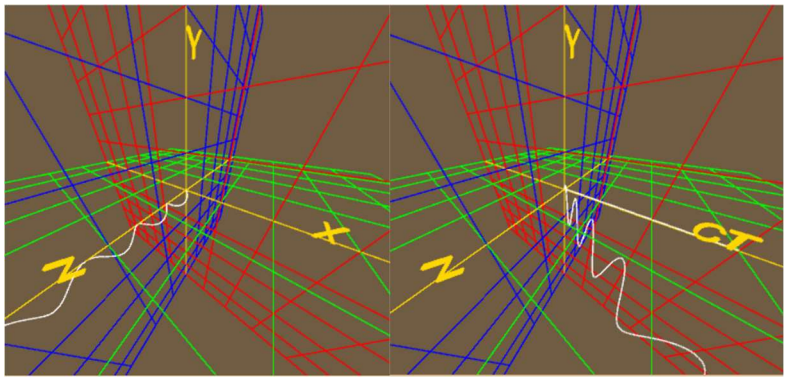
摘要

本实验用 HTML 和 ThreeJs 编写了一个模拟电子在匀强磁场运动的模拟程序，并展示了其在三维空间和四维时空之中的运动轨迹。验证了螺旋线运动的半径计算公式的正确性。同时展示了普通物质在四维时空之中沿类时测地线运动的性质。实验的源代码可以在如下网址获取：https://share.weiyun.com/MDOdFLYR。密码是：cq4pov。源代码可以直接用浏览器运行。

一：引言

电子在匀强磁场之中受到洛伦茨力的作用，在牛顿第二定律的支配下做螺旋线运动。这个运动就是本实验的模拟对象。使用常见的微分方程的数值模拟的办法，可以较好的模拟出电子在三维空间 x-y-z 之中的运动轨迹，从而可以进一步得到其在四维时空 ct-x-y-z 的运动轨迹。通过分析三维运动轨迹，就可以实时的算出螺旋运动的半径，从而验证半径公式。通过分析四维运动轨迹，就可以验证其是否出于光锥之内而属于类时测地线。

展示模拟结果有多种手段，本实验使用 HTML 编写一个网页来展示模拟出来的运动。实际模拟效果如下：



图一：模拟效果

左图是三维空间之中模拟出来的螺旋线运动。右图是四维时空之中的相应运动轨迹。左图和右图模拟的运动所有物理参数均是相同的。在左图之中，螺距看上去似乎在逐渐变大，这是由于电子接近了相机。并不是物理模拟出现了错误。

二：具体实现

1. HTML 与 ThreeJS

HTML（Hyper Text Mark Language）是一种用于编写网页的语言。本实验就是使用了 HTML 编写了一个网页来展示模拟效果。其实展示模拟效果的方法有很多，比如 Python、Matlab、Mathematica，这些工具都可以用于展示数值模拟的结果。HTML 相比之下倒是一个比较冷门的工具了。根据我在完成本实验过程之中的实际体会，这一点情有可原。使用 HTML 来进行物理模拟，主要的麻烦之处有如下三点：

- 展示模拟效果是用 HTML 编写网页，可是编写模拟程序却要内插一段 JavaScript 代码来完成。两者不在一处，甚至都不是用同一种语言编写，要分别处理。这一点就不如很多的专业模拟软件。
- HTML 自带的绘制功能相对于上述的几大热门物理模拟工具而言十分原始。

- JavaScript 几乎没有封装好的函数，所有的代码都要自己手动编写。

本实验要模拟的螺旋线运动是三维运动。为了绘制三维效果，就需要用到 HTML 的一个三维绘制协议即所谓的 WebGL。上个世纪的网页只能展示文字与图片，如要显示三维效果，就需要外挂插件，且编写起来相当繁琐。自从 WebGL 协议在本世纪初出现以后，直接使用 HTML 语言进行三维绘制变成了可能。当然，前提是浏览器要支持 WebGL。然而 WebGL 的原生借口学习成本高。所以我转而去学习一个封装好的使用 WebGL 的库——ThreeJS。本实验所有的具体绘制都是使用 ThreeJS 来完成的。在 ThreeJS 之中，进行绘图有三个要素：场景、相机、渲染器。场景又可以分为几何与材料。所以绘制的一般流程就是，先编写几何体，再赋予几何体以具体的材料，然后指定相机位置与渲染器，最后使用渲染器渲染即可。至于本实验要实现的运动模拟，其逻辑就是使用一个缓冲几何体即 BufferGeometry 将模拟的结果动态的放入这个缓冲几何体之中，然后控制绘制范围随着输入的填入变化，从而实现动态的运动绘制。具体的教程，也就是我使用的教程，见参考文献 1。

图二：WebGL 与 ThreeJs 的 logo

2．数值模拟方法

下面来介绍我使用的模拟螺旋线运动的方法。由于支配螺旋线运动的方程是熟知的洛伦茨力和牛顿第二定律，并且这两个方程都是一阶微分方程，所以可以使用一阶微分方程数值模拟的方法来模拟螺旋线运动。模拟的方法主要有简单步进，欧拉法，还有龙格-库塔方法。这三种方法之中最后一种效果最好。所以本实验采用最后一种方法即龙格-库塔方法。实验中具体使用的是四阶龙格-库塔方法。我之所以说其效果好，是因为我在使用了三种方法以后对结果进行了比较，然后才得出这样的结论。使用简单步进和欧拉法来模拟的时候，累积误差导致了螺距以可见的速度增长，而使用龙格-库塔方法就没有这样的担忧。其原理和具体实现如下：

$$k_1 = \mathbf{h}F(X(t_n), n)$$

$$k_2 = \mathbf{h}F\left(X(t_n) + \frac{k_1}{2}, t_n + \frac{\mathbf{h}}{2}\right)$$

$$k_3 = \mathbf{h}F\left(X(t_n) + \frac{k_2}{2}, t_n + \frac{\mathbf{h}}{2}\right)$$

$$k_4 = \mathbf{h}F(X(t_n) + k3, t_n + \mathbf{h})$$

$$X(t_{n+1}) = X(t_n) + \frac{k_1}{6} + \frac{k_2}{3} + \frac{k_3}{3} + \frac{k_4}{6}$$

and

with $t_{n+1} = t_n + \mathbf{h}$

```
//更新物理参数,使用RK4方法
function getChangeRate(velocity, field){
    //RK4方法需要调用的基础函数，用于计算导数
    return [dt*velocity[0],
            dt*velocity[1],
            dt*velocity[2],
            dt*(q/m)*(velocity[1]*field[2]-velocity[2]*field[1]),
            dt*(q/m)*(velocity[2]*field[0]-velocity[0]*field[2]),
            dt*(q/m)*(velocity[0]*field[1]-velocity[1]*field[0])];
}
function RK4(){
    //RK4方法的实现
    k1 = getChangeRate(velocity, field);
    k2 = getChangeRate([velocity[0]+k1[3]/2,
                        velocity[1]+k1[4]/2,
                        velocity[2]+k1[5]/2], field);
    k3 = getChangeRate([velocity[0]+k2[3]/2,
                        velocity[1]+k2[4]/2,
                        velocity[2]+k2[5]/2], field);
    k4 = getChangeRate([velocity[0]+k3[3],
                        velocity[1]+k3[4],
                        velocity[2]+k3[5]], field);
    for(var i=0; i<3; i++){
        position[i] = position[i]+k1[i]/6+k2[i]/3+k3[i]/3+k4[i]/6;
        velocity[i] = velocity[i]+k1[i+3]/6+k2[i+3]/3+k3[i+3]/3+k4[i+3]/6;
    };
}
RK4();
```

图三：龙格-库塔方法的原理（左）与程序实现（右）

左图中的 F 函数的含义是导数，比如牛顿第二定律中就有 F(v)=F/m

3．半径实时计算

为了验证螺旋线运动的熟知的半径计算公式，我设计了一个简单的方法实时计算模拟出来的螺旋线运动的半径。方法简述如下：用施密特正交化方法，构造一个磁场为其中之一的正交向量组，再将剩下的两个向量归一化。如此则剩下的两个向量就成为了与磁场垂直的平面的基。把运动之中某一点的位置矢量与这个基做内积操作。就可以得到该点在于磁场垂直的平面上的投影坐标。对三个点重复这样的操作，最后使用三点确定一个圆的算法确定圆心的位置从而得到半径。确定圆心的算法的来源见参考文献 2。具体原理和实现如下：

```
function dotProduct(vectorA, vectorB){
    //矢量点乘
    return vectorA[0]*vectorB[0]+vectorA[1]*vectorB[1]+vectorA[2]*vectorB[2];
}
function vectorProduct(vectorA, vectorB){
    //矢量叉乘
    return [vectorA[1]*vectorB[2]-vectorA[2]*vectorB[1],
            vectorA[2]*vectorB[0]-vectorA[0]*vectorB[2],
            vectorA[0]*vectorB[1]-vectorA[1]*vectorB[0]];
}
//然后是施密特正交化函数
function Schmidt(){
    var vector1 = [Math.random(), Math.random(), Math.random()];
    var vector2 = vectorProduct(field, vector1);
    vector1 = [vector1[0]-(dotProduct(vector1, field)/dotProduct(field, field))*field[0],
               vector1[1]-(dotProduct(vector1, field)/dotProduct(field, field))*field[1],
               vector1[2]-(dotProduct(vector1, field)/dotProduct(field, field))*field[2]];
    vector2 = [vector2[0]-(dotProduct(vector2, field)/dotProduct(field, field))*field[0]-(dotProduct(vector2, vector1)/dotProduct(vector1, vector1))*vector1[0],
               vector2[1]-(dotProduct(vector2, field)/dotProduct(field, field))*field[1]-(dotProduct(vector2, vector1)/dotProduct(vector1, vector1))*vector1[1],
               vector2[2]-(dotProduct(vector2, field)/dotProduct(field, field))*field[2]-(dotProduct(vector2, vector1)/dotProduct(vector1, vector1))*vector1[2]];
    vector1 = [vector1[0]/Math.sqrt(dotProduct(vector1, vector1)),
               vector1[1]/Math.sqrt(dotProduct(vector1, vector1)),
               vector1[2]/Math.sqrt(dotProduct(vector1, vector1))];
    vector2 = [vector2[0]/Math.sqrt(dotProduct(vector2, vector2)),
               vector2[1]/Math.sqrt(dotProduct(vector2, vector2)),
               vector2[2]/Math.sqrt(dotProduct(vector2, vector2))];
    return [vector1, vector2];
}
```

图四：计算半径第一步-施密特正交化

得到一个包含磁场的正交向量组，并将剩下的两个向量归一化

```
//然后是获取投影函数
function getProjection(){
    var coordinateAxes = Schmidt();
    var pool = line.geometry.attributes.position;
    var coordinateVector1 = [pool.getX(count-1)/lengthRate,
                             pool.getY(count-1)/lengthRate,
                             pool.getZ(count-1)/lengthRate];
    var coordinateVector2 = [pool.getX(count-1-sampleGap)/lengthRate,
                             pool.getY(count-1-sampleGap)/lengthRate,
                             pool.getZ(count-1-sampleGap)/lengthRate];
    var coordinateVector3 = [pool.getX(count-1-2*sampleGap)/lengthRate,
                             pool.getY(count-1-2*sampleGap)/lengthRate,
                             pool.getZ(count-1-2*sampleGap)/lengthRate];
    coordinateVector1 = [dotProduct(coordinateVector1, coordinateAxes[0]),
                         dotProduct(coordinateVector1, coordinateAxes[1])];
    coordinateVector2 = [dotProduct(coordinateVector2, coordinateAxes[0]),
                         dotProduct(coordinateVector2, coordinateAxes[1])];
    coordinateVector3 = [dotProduct(coordinateVector3, coordinateAxes[0]),
                         dotProduct(coordinateVector3, coordinateAxes[1])];
    return [coordinateVector1, coordinateVector2, coordinateVector3];
}
```

图五：计算半径第二步-投影

这一步是为了获取运动的某点在与磁场垂直的方向上面的投影坐标

$$a = x_1 - x_2$$
$$b = y_1 - y_2$$
$$c = x_1 - x_3$$
$$d = y_1 - y_3$$
$$e = \frac{(x_1^2 - x_2^2) - (y_2^2 - y_1^2)}{2}$$
$$f = \frac{(x_1^2 - x_3^2) - (y_3^2 - y_1^2)}{2}$$

$$x_0 = -\frac{de - bf}{bc - ad}$$
$$y_0 = -\frac{af - ce}{bc - ad}$$

```
//最后才是计算半径,计算原理见http://blog.sina.com.cn/s/blog_67a4b6b10102x166.html
function getRadius(){
    var coordinateVectors = getProjection();
    var a = coordinateVectors[0][0]-coordinateVectors[1][0];
    var b = coordinateVectors[0][1]-coordinateVectors[1][1];
    var c = coordinateVectors[0][0]-coordinateVectors[2][0];
    var d = coordinateVectors[0][1]-coordinateVectors[2][1];
    var e = ((coordinateVectors[0][0]**2-coordinateVectors[1][0]**2)-
             (coordinateVectors[1][1]**2-coordinateVectors[0][1]**2))/2;
    var f = ((coordinateVectors[0][0]**2-coordinateVectors[2][0]**2)-
             (coordinateVectors[2][1]**2-coordinateVectors[0][1]**2))/2;
    var x0 = -(d*e-b*f)/(b*c-a*d);
    var y0 = -(a*f-c*e)/(b*c-a*d);
    return Math.sqrt((x0-coordinateVectors[0][0])**2+(y0-coordinateVectors[0][1])**2);
}
```

图六：计算半径最后一步-得到圆心

4．四维时空轨迹绘制

本实验之中除了在三维 x-y-z 空间之中绘制运动轨迹，还尝试着在四维时空 ct-x-y-z 之中绘制运动轨迹。从而展示粒子的测地线。所谓的测地线就是粒子在四维时空之中画出的轨迹。光子画出的轨迹称为类光测地线。由类光测地线包围的时空区域就是光锥。所有的普通物质，包括机械物质和电磁场，运动画出的测地线都只能出现在光锥之内，被称为类时测地线。光锥之外的测地线叫做类空测地线，是不能现实存在的。如果两个四维事件通过类空测地线连接，那么就说明这两个事件之间绝对没有可能存在因果关系。只有通过类时测地线和类光测地线连接起来的两个事件才有可能存在因果关系。所谓的黑洞之所以称为黑洞就是因为这个时空区域之中任何一点都不存在从这一点出发的且可以离开这个区域的类时或者类光测地线。本实验要尝试在三维之中画出四维时空之中的测地线，采用办法就是放弃一个坐标。通过绘制 ct-x-y、ct-y-z、ct-x-z 来简介的展示四维测地线的样子。并显示其处于光锥之内。同时为了方便观察，绘制时对事件坐标和空间坐标采用了不同的放大倍数。具体来说，垂直于磁场方向的空间坐标放大 1000 倍；平行与磁场方向的空间坐标放大 100 倍；时间坐标缩小 10 倍。空间坐标的放大策略的目的是为了展示螺旋线运动，不让运动看上去是一条直线，而时间坐标由于 c 的量级过大必须加以缩小。因此实际的真实测地线并没有看上去的离开时间轴的程度那么大，真实的测地线几乎紧贴着时间轴。而在这样的缩放比例之下光锥的边界几乎与 x-y 平面重合

三：参数列表与输入控件

参数名	含义	单位	是否可变	下限	上限
B	磁场强度	uT	是	1	100
B_direction	磁场方向	无	否，z 轴正向	无	无
V	初速大小	10^5m/s	是	1	5
theta	初速 theta 角	度	是	0	180
fai	初速 fai 角	度	是	0	360
c	光速	m/s	否，$3x10^8$	无	无
m	电子质量	Kg	否，$9.1x10^{-31}$	无	无
q	电子电荷	C	否，$1.6x10^{-19}$	无	无

表一：物理参数列表

参数名	含义	单位	是否可变	下限	上限
dt	帧与帧之间的时间间隔	Ns	否，10	无	无
numOfFrames	模拟的总帧数	帧	否，1000	无	无
LengthRate	三维绘制时的放大比例	无	否，200	无	无
spacetimeLengthRateT	四维绘制时空间坐标的放大比例	无	否，0.1	无	无
spacetimeLengthRateZ	四维绘制时平行磁场的空间坐标放大比例	无	否，100	无	无
spacetimeLengthRateX	四维绘制时垂直磁场的空间坐标放大比例	无	否，1000	无	无
type	绘制类型	无	是，在 1-4 之间	1-(x,y,z) 2-(ct,y,z)	3-(x,ct,z) 4-(x,y,ct)

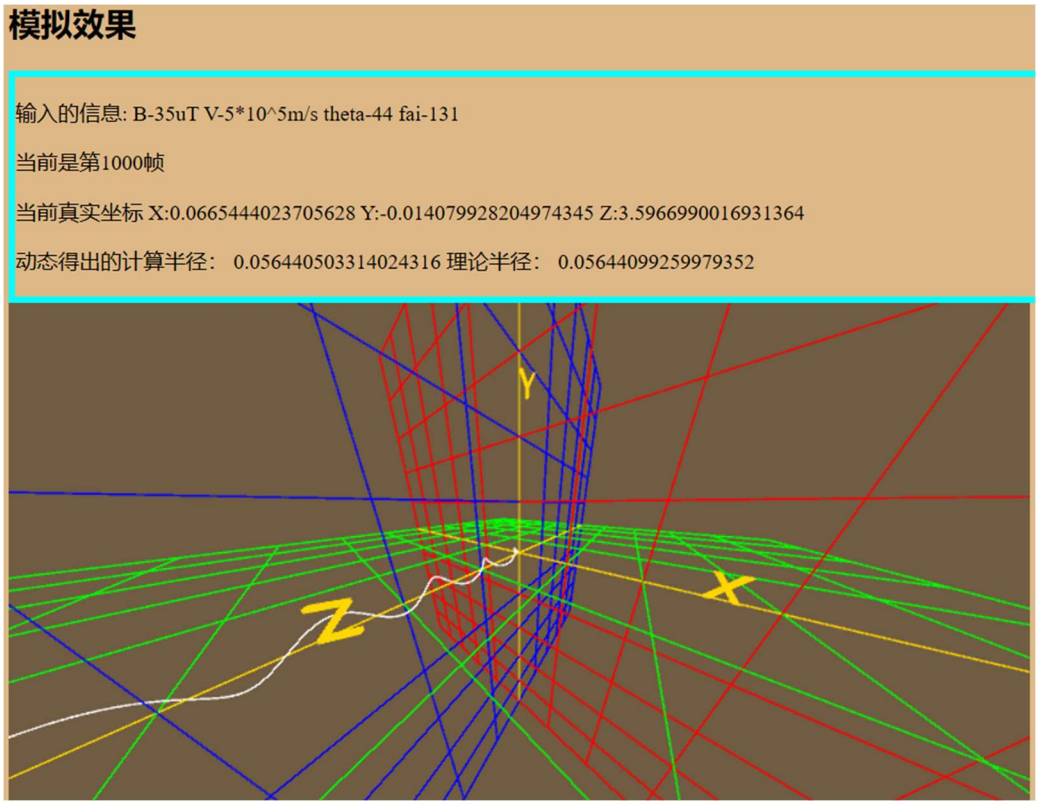
参数名	含义	单位	是否可变	下限	上限
cameradistance	相机相对于原点距离	像素	否，300		
cameratheta	相机相对于原点的 theta 角	度	是	0	90
camerafai	相机相对于原点的 fai 角	度	是	0	90
needGrid	是否需要绘制坐标轴	无	是，取 0 或 1	0-不需要	1-需要
coordinateRange	绘制坐标轴的范围	像素	否，[-500,500]	无	无
coordinateGap	绘制坐标轴的铺线间隔	像素	否，100	无	无
sampleGap	计算半径时候的取点间隔	无	否，1	无	无

表二：程序参数列表

图七：输入可变参数的控件页面

四：实验结果与结论

一次典型的三维实验结果展示如下图：



图八：典型的实验结果

1. 实验成功的在小数点后三位的精度上验证了计算螺距的公式。并且这个精度随着参数的变化具有鲁棒性。
2. 实验成功的展示了粒子的测地线运动（见图一）。确实处于光锥之内。因为在我上述的放大倍数的前提下，光锥边界几乎重合于 x-y 平面。而绘制出来的测地线（见图一）明显在 x-y 平面和 z 轴正向之间的空间区域之中。所以肯定是类时测地线。
3. 各项参数调整功能正常

总之，本实验模拟了电子在匀强磁场之中的运动，并以此出发验证了一些熟知的结论。为了完成这个实验，学习了 HTML，ThreeJS 以及 Javascript。

实验的不足之处有两点，第一是网页设计还过于初级，第二是参数调整了以后必须重新开始模拟才能看到效果。无法实现参数调控实时对模拟的实时控制。这两点有待后续改进。

五：参考文献

1. ThreeJS 教程：https://threejs.org/docs/index.html#manual/en/introduction/Creating-a-scene
2. 三点确定圆心算法：http://blog.sina.com.cn/s/blog_67a4b6b10102x166.html