1 Worksheet 9: Motion of Charged Particles in Electromagnetic Fields

In this worksheet we continue exploring electromagnetism using *Mathematica* and Fortran. We will now study the motion of charged particles in crossed electric and magnetic fields. This will necessitate solving ordinary differential equations (ODE's). This will be our last worksheet for this semester. We hope you enjoyed learning more about *Mathematica* and being introduced to Fortran. We have by no means covered either one exhaustively, and hopefully you will continue learning more about computational physics throughout your education and career.

You will have three class periods for this worksheet.

1.1 Problem Formulation

Consider the motion of a particle with charge q with initial velocity $\vec{v}_0$ in an electromagnetic field with electric field $\vec{E}(\vec{r},t)$ and magnetic field $\vec{B}(\vec{r},t)$. In general, these can be any function of coordinates and time, but in this worksheet we will restrict our attention to the case when the direction of the fields is fixed, and their magnitudes oscillate harmonically with a frequency ω and amplitudes δE and δB around E_0 and B_0. If we let the angle between $\vec{E}$ and $\vec{B}$ be θ and the phase displacement between their oscillation φ, then it is quite general to assume that:

$$\begin{aligned} \vec{B} &= B_0\left(1+\delta B\cdot\cos\omega t\right)\cdot\hat{z} \\ \vec{E} &= E_0\left[1+\delta E\cdot\cos\left(\omega t+\varphi\right)\right]\cdot\left(\hat{x}\sin\theta+\hat{z}\cos\theta\right) \end{aligned} \tag{1}$$

In the worksheet we will consider only specific simpler cases of this general case.

The classical motion of the particle in such a field is determined by Newton's equation of motion, where the force is given by:

$$\vec{a}=\frac{\vec{F}}{m}=\frac{q}{m}(\vec{E}+\vec{v}\times\vec{B}) \tag{2}$$

As usual, we will be interested mostly in qualitative understanding of the motion, so set $\frac{q}{m}=1$. We can now write this as an ODE for $\vec{r}=\vec{r}(t)$ with a given initial velocity $\vec{v}(t=0)=\vec{v}_0$ and starting position $\vec{r}(t=0)=\vec{r}_0$:

$$\begin{aligned} \frac{d^2\vec{r}(t)}{dt^2} &= \frac{q}{m}\left[\vec{E} + \frac{d\vec{r}(t)}{dt} \times \vec{B}\right] \\ \vec{r}(0) &= \vec{r}_0, \frac{d\vec{r}(0)}{dt} = \vec{v}_0 \end{aligned} \tag{3}$$

More formally, equation 2 is what is known as a *second order initial value problem* (IVP), and there are many numerical techniques aimed at solving this type of extremely common equation. *Mathematica*'s function `NDSolve` is equipped with a suite of such techniques, and there many Fortran libraries that solve problems of the type 2 as well.

1.2 Numerical Solution of IVP's: Euler Method

1.3 First Order IVP's

When approaching a problem numerically, it is usually wise to write in the most general (yet specific) way, so that the same numerical algorithm can be used for a variety of problems. A typical second order IVP has the form:

$$\begin{aligned} \frac{d^2\vec{y}}{dx^2} &= \vec{f}(x, \vec{y}, \frac{d\vec{y}}{dx}) \\ \vec{y}(x_o) &= \vec{y}_0, \frac{d\vec{y}(x_o)}{dx} = \vec{y}_0' \end{aligned} \tag{4}$$

Look at equation 3 and identify all the variables in the above equation before proceeding.

Most numerical methods are tailored for the *first order IVP*,

$$\begin{aligned} \frac{d\vec{y}}{dx} &= \vec{f}(x, \vec{y}) \\ \vec{y}(x_o) &= \vec{y}_0 \end{aligned} \tag{5}$$

since any problem of order n, such as 4 ($n = 2$), can be cast as a first order problem by introducing auxiliary variables for the $n - 1$ lower derivatives. For example, eq. 4 can be written in the form of eq. 5 by introducing a new variable for the first derivative $\vec{y}'$ (velocity):

$$\begin{aligned} \frac{d\vec{y}'}{dx} &= \vec{f}(x, \vec{y}, \vec{y}'),\ \vec{y}'(x_o) = \vec{y}'_0 \\ \frac{d\vec{y}}{dx} &= \vec{y}',\ \vec{y}(x_o) = \vec{y}_0 \end{aligned} \tag{6}$$

Although this is formally system of first order IVP's, it is in fact equivalent to a formal first order IVP for the variable that appends the first derivative with the variable itself, $\vec{Y} = [\vec{y}' \vdots \vec{y}]$,

$$\begin{aligned} \frac{d\vec{Y}}{dx} &= \vec{F}(x, \vec{Y}),\ \vec{Y}(x_o) = [\vec{y}'_0 \vdots \vec{y}_0] \\ \text{where } \vec{F}(x, \vec{Y}) &= \left[\vec{f}(x, \vec{y}, \vec{y}') \vdots \vec{y}'\right] \end{aligned} \tag{7}$$

so that any method that can solve 5 can also solve 6 and thus also 4. Before proceeding, rewrite equation 3 as a first order IVP (either in the form 6 or 7) and *show your result to your teacher or TA*.

1.4 Euler's Method

From here on we will only discuss problem 5, having in mind the previous discussion. The simplest numerical method for solving such an equation is Euler's method, which steps through the independent variable (time) in small time steps dx from the current value x, using a first-order Taylor expansion to approximate the dependent variable y at the next time step $x + dx$:

$$\vec{y}(x + dx) \approx \vec{y}(x) + \frac{d\vec{y}(x)}{dx} = \vec{y}(x) + \vec{f}\,[x, \vec{y}(x)]$$

It is clear that once we have an equation for $\vec{f}$ we can start from the initial point x_0 and step through time until any desired point to obtain an approximate solution for the IVP. Euler method is very slow and a very small step size dx is needed to achieve good results, but it is prototypical of the commonly used methods, such as Runge-Kutta methods (see Numerical Recipies for a description, or the file `/classes/phy201/RK4.f90`).

2 Solution in *Mathematica*

Mathematica's function `NDSolve` implements various algorithms for solving first and higher order IVP's. Unlike Fortran, it is a symbolic algebra tool so it can convert any higher order IVP's into first order automatically and involve the appropriate solver to obtain an approximate numerical solution. Look at the documentation pages under `Help` to see typical examples of usage.

Just as earlier, in *Mathematica* it is possible to follow the above physical formulation very closely without worrying much about the numerical aspects we discussed above. Our dependent variable is the position of the particle, `r={x[t],y[t],z[t]}`, with velocity `v={x'[t],y'[t],z'[t]}`. Write down the expressions in equations 1 and then calculate the force in 2 and then write the system of equations 3.

Now use `NDSolve` to solve for the orbit of the charged particle in these simplified cases, and plot the orbit using `ParametricPlot3D` in each case:

1. $B_0 = 1$, $E_0 = 0$, $\delta B = 0$, and choose any initial position and velocity. What is the physical situation in this case? What is the solution, as you learned in your EM class? Verify your guesses.
2. $B_0 = 1$, $E_0 = 0$, $\delta B = 1$, $\omega = 0.2$. Same as above.
3. $B_0 = 1$, $E_0 = 1$, $\theta = 0$, $\delta B = 0$, $\delta E = 0$. Explain the results.
4. $B_0 = 1$, $E_0 = 1$, $\theta = \frac{\pi}{2}$, $\delta B = 0$, $\delta E = 0$. Explain the results.
5. Experiment with other combinations that you find interesting.

3 Solution in Fortran

3.1 Euler's Method

The Fortran solution to this problem will of course take some more efford. You should write a program or better yet a procedure that uses the Euler method to integrate a first order IVP of the form 5. Assume that the function $\vec{f}$ already exists and will be passed as an argument along with number of unknowns `n_unknowns` in the dependent variable $\vec{y}$ (in our case this is 6—3 coordinates and 3 velocities), the number of time steps `n_points` to take and output the solution at, the initial conditions `x0` and `y0`, as well as the step size `dx`:

```
MODULE IVPSolve
USE Precision
...
   SUBROUTINE Euler(f,y,n_unknowns,n_points,dx,x0,y0)
      ! The function f is passed as an argument:
      INTERFACE
         SUBROUTINE f(x,y,y_prime,n_unknowns)
            USE Precision
            INTEGER n_points
            REAL(KIND=wp) :: x
            REAL(KIND=wp), DIMENSION(n_unknowns) :: y, y_prime
         END SUBROUTINE f
      END INTERFACE
      INTEGER :: n_unknowns,n_points
      ! Upon exit, this array should contain the solution
      REAL(KIND=wp), DIMENSION(n_unknowns,n_points) :: y
      REAL(KIND=wp) :: dx, x0 ! Step size and initial x
      REAL(KIND=wp), DIMENSION(n_unknowns) :: y0 ! Initial y
      ...
      ...Declare any needed local variables here...
      ...Initialize variables for the DO loop...
      DO i=1,n_points-1
         ...Calculate x here in steps of dx starting at x0...
         CALL f(x,y(:,i),y_prime,n_unknowns)
         y(:,i+1)=y(:,i)+y_prime*dx ! Euler's method
         ...
      END DO
      ...
   END SUBROUTINE Euler
END MODULE IVPSolve
```

Feel free to use any other approach. There is ample room for improvement in the above scheme. For example, why output the solution at every time step, and not every desired `n_output` steps? This will save on storage, especially if `n_points` is very large. Also, those that feel up to it can try coding a more sophisticated algorithm, such as Runge-Kutta and ask for assistance if needed (you can use the file `RK4.f90` as a template).

3.2 The function `f`

It may be wise to first test your `Euler` routine on a simple case, such as a simple harmonic oscillator, $\frac{d^2x}{dt^2} = -kx$, by printing (or plotting) your solution to verify correctness. It will then be one more step to modify the function $\vec{f}$ to reflect the problem of the motion of a particle in an electromagnetic field. The strategy is the same as usual. Make a separate module in which you will place $\vec{f}$, and make any physical parameters it may use, such as E_0 and B_0, public module variables, so that the main program can set their values later on. We leave it up to your ingenuity to code the vector expressions appearing in eq. 1 as you wish. We recommend using *Mathematica* to help you in simplifying the expressions before coding them in Fortran.

3.3 Plotting the Solution

Your main program should as usual first set the values of the physical parameters in the problem. Simply choose one of the cases you analyzed in *Mathematica*. Then call the integration routine `Euler` to obtain the approximate solution.

The module `SimpleGraphics` has been supplemented with a rouine for plotting points in 3D, `Plot3D`. The documentation in,

http://computation.pa.msu.edu/phy201/SimpleGraphics.html

contains the calling interface and an example. So a simple,

```
CALL Plot3D(x=y(4,:),y=y(5,:),z=y(6,:),...)
```

will plot the orbit of the particle in Cartesian space. The executable solution to this assignment is in our class directory. Look at it!

3.4 Advanced: The ready-made module `ODE`

There are many publicly available sophisticated Fortran libraries for solving IVP's. In Fortran 90, such a library is `RKSUITE90`, which implements adaptive Runge-Kutta methods. This library, or its FORTRAN 77 counterpart `RKSUITE`, are by no means easy to use, so we have again made a wrapper routine, `ODESolve` in the module `ODE`, found in our class directory. The interface for the routine is:

```
subroutine ODESolve(f,n_eq,x_range,n_points,y0,x,y,relerr,abserr)
```

```
    interface
      subroutine f(x,y,dy)
         integer, parameter :: wp=kind(0.0D0)
         real(kind=wp), intent(in) :: x
         real(kind=wp), dimension(*), intent(in) :: y
         real(kind=wp), dimension(*), intent(out) :: dy
      end subroutine f
    end interface
    integer, intent(in) :: n_eq ! Number of unknowns
    ! The range of values to solve in (as in NDSolve)
    ! Thus x_range(1)=x0
    real(kind=wp), dimension(2), intent(in) :: x_range
    integer, intent(in) :: n_points ! Number of output points
    real(kind=wp), dimension(n_eq), intent(in) :: y0 ! Initial y
    ! If you need them (for plotting), you can get the
    ! values of x for which the solution was outputed.
    ! But note that x=(/(x0+(i-1)*dx,i=1,n_points)/) always:
    real(kind=wp), dimension(n_eq), intent(out), optional :: x
    real(kind=wp), dimension(n_eq,n_points) :: y ! The solution
    real(kind=wp), intent(in) :: relerr ! Desired relative error
    ! The absolute error is optional:
    real(kind=wp), dimension(n_eq), intent(in), optional :: abserr
  end subroutine ODESolve
```

Looks complicated? Not at all. Take a look at the example `Harmonic.f90` in our class directory. To compile programs that use this module, first do a (not neccessary if your PC has been recently restarted),

```
#   source /classes/phy201/ODE.init
```

and then append a `$ODEvf90` to your compilation line. Notice that this routine requires using double precision numbers, while the plotting routines require single-precision values. Conversion will be neccessary!