Lancaster University

Computational lattice design

Numerical methods II

Dr Robert Apsimon
r.apsimon@lancaster.ac.uk

In this lecture

- We will now look at the considerations for writing your own tracking code.

- While there are plenty of tracking codes available, it is good to understand how to write your own.
  - Aside from giving a good conceptual understanding, codes written by someone else don't always do exactly what you want it to do…
    - ASTRA changes your coordinate system if you use dipoles
    - MAD/MADX doesn't allow you to import field maps
    - PARMILA/PARMELA is difficult to use and computationally limited
    - …

General strategy for particle tracking

Lancaster University

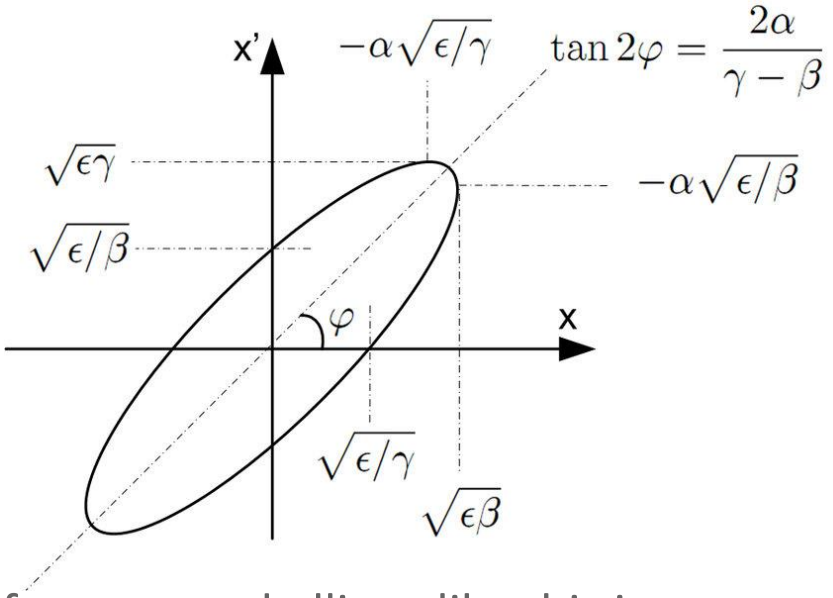
1. Import/generate your particle distribution
2. Import/generate your field map or beam line
3. Integrate your trajectory along the field map/beam line

- Conceptually, writing a tracking code is much easier than it sounds
  - There are, of course, plenty of little fiddly bits, but nothing too strenuous!

Generating a particle distribution

- You should all be familiar with the Twiss parameters.
- From this, the beam ellipse can be written as:

  $\gamma x^2 + 2\alpha x x' + \beta x'^2 = \varepsilon_g$
- Where $\varepsilon_g = \frac{\varepsilon_N}{(\beta\gamma)_{rel}}$



- Generating a random particle distribution for a rotated ellipse like this is difficult, what would be better is to generate a particle distribution for a circle.
  - This is essentially what we do. We define the particle distribution in normalised phase space coordinates and transform it into actual phase space coordinates.

Generating a particle distribution

- If we take the transformation:

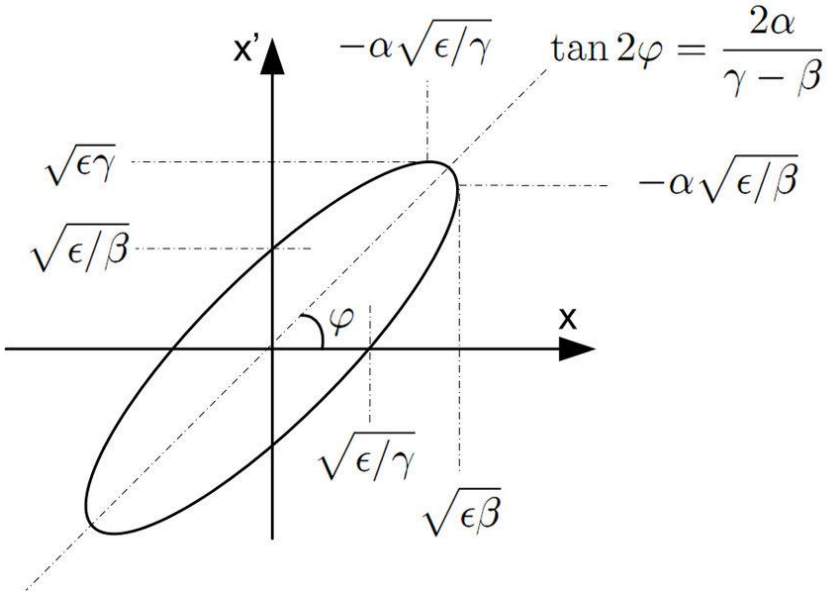
$$\begin{pmatrix} x \\ x' \end{pmatrix} = \begin{pmatrix} \sqrt{\beta} & 0 \\ -\frac{\alpha}{\sqrt{\beta}} & \frac{1}{\sqrt{\beta}} \end{pmatrix} \begin{pmatrix} X_N \\ X_N' \end{pmatrix}$$

- Then the beam ellipse:

$\gamma x^2 + 2\alpha x x' + \beta x'^2 = \varepsilon_g$

Turns into:

$X_N^2 + X_N'^2 = \varepsilon_g$



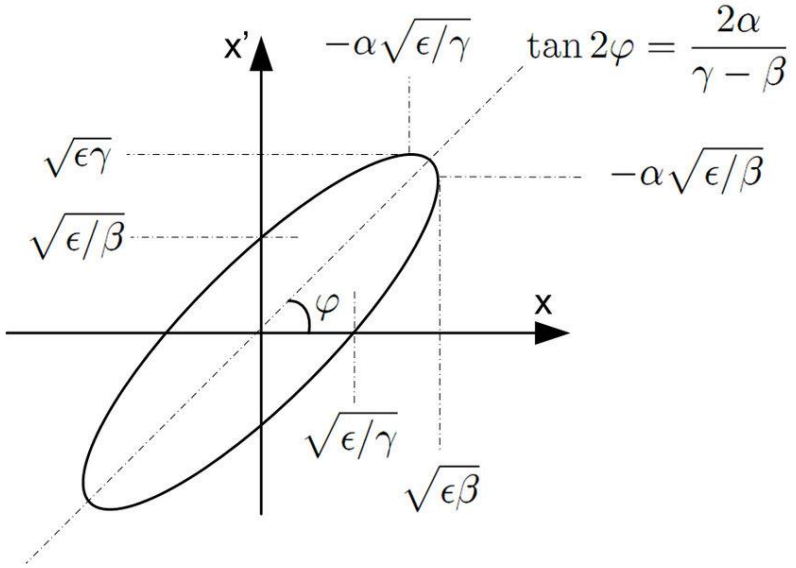
- So now we can generate our normalised phase space distribution easily and the real phase space coordinates are given as:

$$x = \sqrt{\beta} X_N$$

$$x' = \frac{(-\alpha X_N + X_N')}{\sqrt{\beta}}$$

Generating a particle distribution

- There are many different particle distributions we could generate
  - Most commonly a Gaussian
  - Could do uniform distribution
    - This is slightly more complicated



- Gaussian:
  - Let X_N and X'_N be Gaussian distributed arrays of random numbers with a standard deviation of $\sqrt{\varepsilon_g}$

$$x = \sqrt{\beta} X_N$$

$$x' = \frac{(-\alpha X_N + X'_N)}{\sqrt{\beta}}$$

Generating a particle distribution

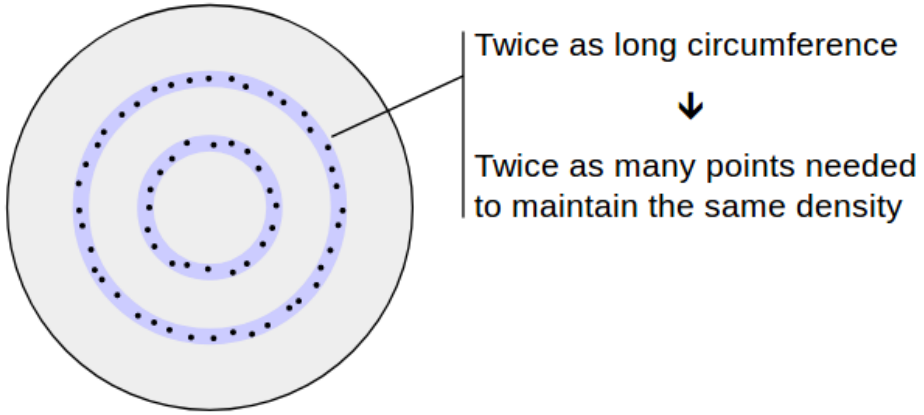
- Uniform:
  - Let m and n be uniformly distributed random in the range of [0, 1]

$$r = \sqrt{\varepsilon_g m}$$
$$\theta = 2\pi n$$

  - Then

$$X_N = r\cos\theta$$
$$X'_N = r\sin\theta$$

  - Why do we need sqrt(m) for r?



Generating a particle distribution

Lancaster University

- We can in fact create any distribution we want with just 3 arrays uniformly distributed random numbers (m, n, p):
  - Assume our required distribution is a function $f(x, x')$
    m is a random number in the range $[x_{min}, x_{max}]$
    n is a random number in the range $[x'_{min}, x'_{max}]$
    p is a random number in the range $[0, 1]$
  - If $p \leq \frac{f(m,n)}{\max(f(x,x'))}$, then $x = m, x' = n$ and this is added to the particle distribution
  - Otherwise m, n, p are rejected and we generate new values for m, n, p
  - We continue this algorithm until we have enough particles in our distribution.

Generating/importing a field map

Lancaster University

- A field map can be described in different ways, depending on what you need.
  - If an analytical field distribution exists and can be easily described then a field map can be considered as a function:
    - E.g. quadrupoles, dipoles…
  - For more complicated systems, we need to define the electric and magnetic fields at specific points
    - This can be 1-, 2-, 3- or even 4D
      - 4D is very rare and much more difficult as the amount of data grows rapidly with the number of dimensions!
    - The obvious downside with discrete field maps is that you lose accuracy between grid points.

Generating/importing a field map

- Formulaic field maps
  - Most tracking codes have commands to class common classes of magnetic and electric elements:
    - Dipole, quadrupole, multipole, kickers…
    - As we saw in the previous lecture, if we have a formula to describe the magnetic and electric fields, we can easily define an equation of motion (which we can either solve analytically or numerically):

$$\begin{pmatrix} \ddot{x} \\ \ddot{y} \\ \ddot{z} \end{pmatrix} = \frac{q}{\gamma m} \begin{pmatrix} E_x(x,y,z,t) + v_y B_z(x,y,z,t) - v_z B_y(x,y,z,t) \\ E_y(x,y,z,t) - v_x B_z(x,y,z,t) + v_z B_x(x,y,z,t) \\ E_z(x,y,z,t) + v_x B_y(x,y,z,t) - v_y B_x(x,y,z,t) \end{pmatrix}$$

    - Later we will come back to this equation

Generating/importing a field map

Lancaster University

- Discrete field maps
  - Most codes will also allow you to import a field map as a text file
    - Each code will have its own format for the text file, but they all require the same information:
      - Positions in x, y, z
      - Real and imaginary components of the field in x, y and z
      - Usually the electric (E) and magnetic (H) fields are imported as separate files

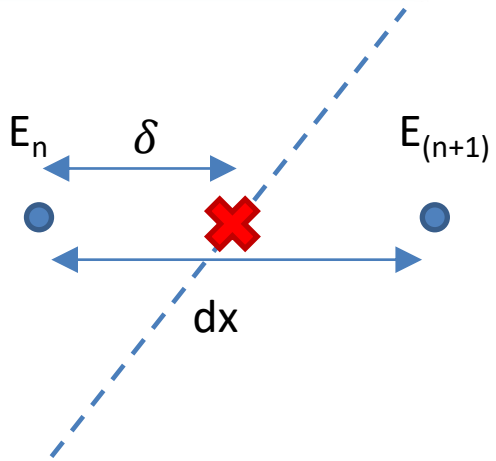
        (Recall $B = 4\pi \times 10^{-7} H$)

  - Since discrete field maps miss information between the grid points, we need to interpolate between points in general.

Interpolation of a field map

- As we only have limited data, we need to find a way to estimate the field between points:
  - Linear interpolation
    - Simplest approach, good for slowly varying fields

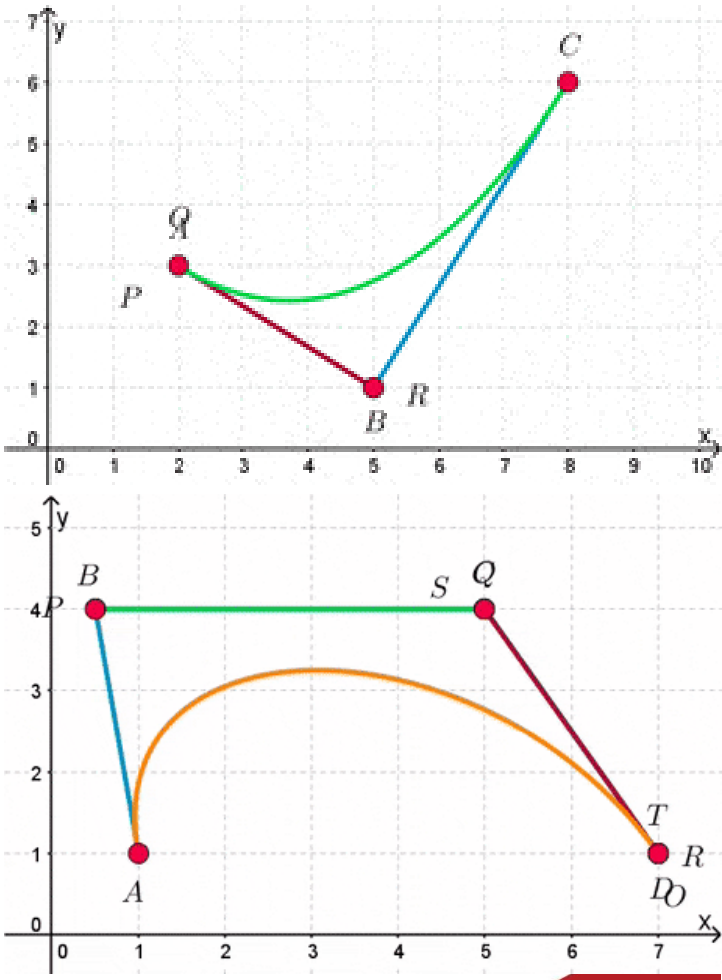
$$E(x) = \left(1 - \frac{\delta}{dx}\right) E_n + \frac{\delta}{dx} E_{(n+1)}$$



  - Polynomial interpolation
    - Similar to linear interpolation, but requires more data points
    - Better accuracy, but more computationally expensive
  - Spline fitting
    - High accuracy, but computationally expensive, especially for 3D interpolation
    - Most commonly Bezier curves (plenty of information about these online)

Bezier curves

- Bezier curves are at the core of almost all spline fitting.
  - Let's start by thinking about a 2nd order Bezier curve:
  - Define 3 points, A, B and C
  - Draw a line from A to B (L1), and B to C (L2)
  - We will define a parameter t, such that when it is zero, we are at A on L1 and B on L2.
  - Now draw a line (L3) from L1(t) to L2(t)
  - The point L3(t) along our new line describes our 2nd order Bezier curve

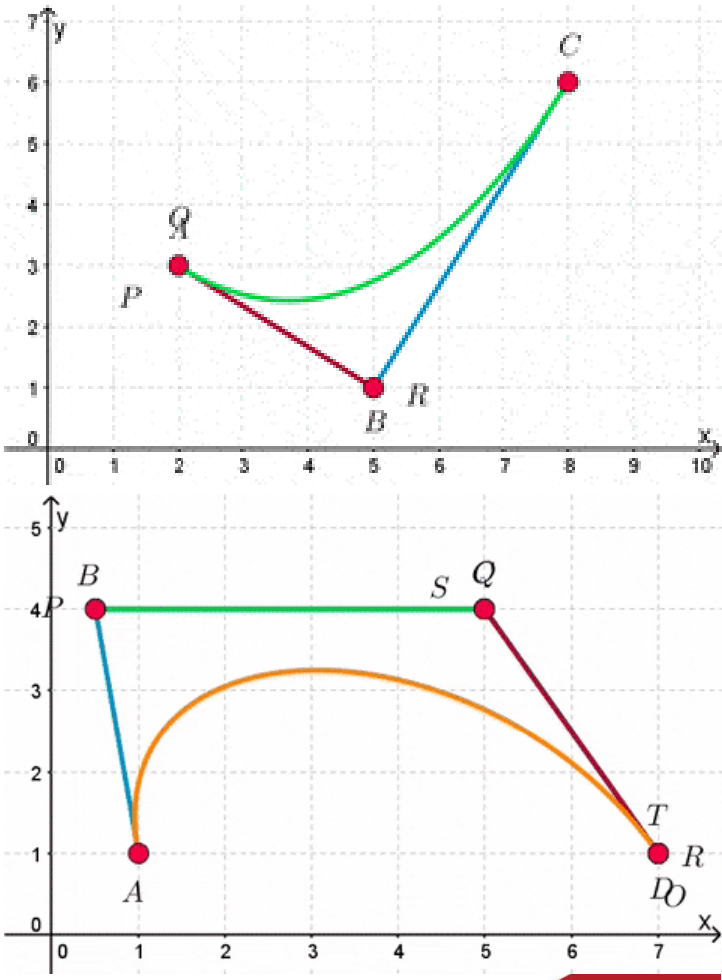


Bezier curves

- Writing this all out as equations:
  $x_B(t) = (1-t)^2 x_0 + 2t(1-t)x_1 + t^2 x_2$
  $y_B(t) = (1-t)^2 y_0 + 2t(1-t)y_1 + t^2 y_2$

- Now increase the order of the Bezier curve, we need more points, so an n[th] order Bezier curve needs n+1 points
  - We use these to generate n generations of Bezier curves (1[st] order curves are lines!).
  - The functional form of the n[th] order Bezier curve forms a binomial expansion:

$$P_B(t) = \sum_{k=0}^{n} \binom{n}{k} (1-t)^{n-k} t^k P_k$$

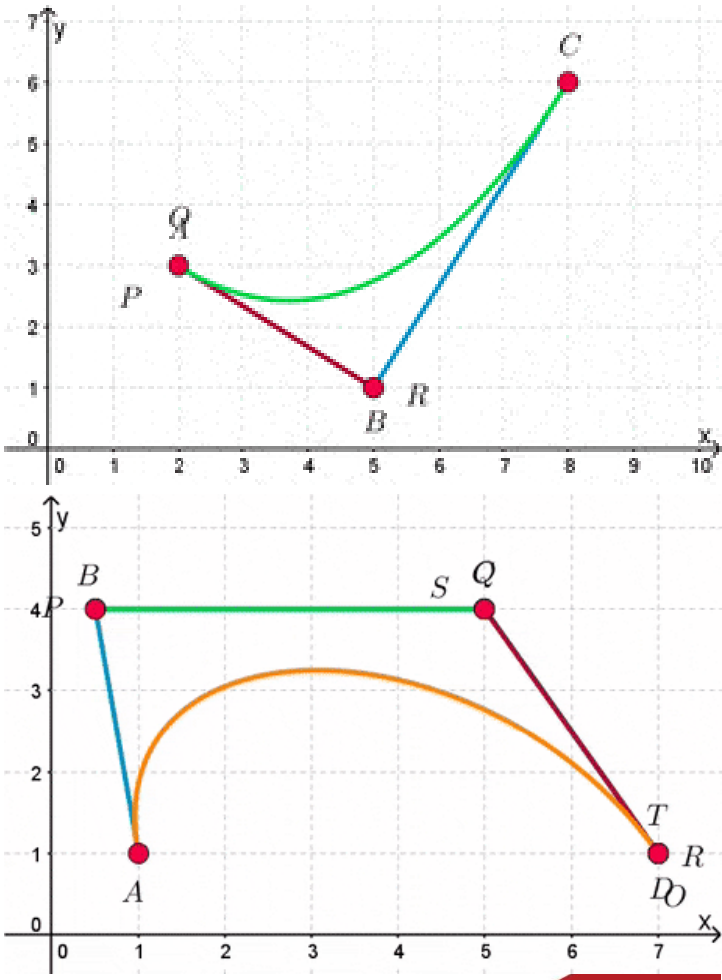


Bezier curves

- Writing this all out as equations:
  $x_B(t) = (1-t)^2 x_0 + 2t(1-t)x_1 + t^2 x_2$
  $y_B(t) = (1-t)^2 y_0 + 2t(1-t)y_1 + t^2 y_2$

- While Bezier curves may have a simple looking form, they can describe very complicated shapes in a computationally efficient manner.
  - However, a Bezier curve can never perfectly describe a circle (I'll leave that as an exercise for you to find out why)

- As we move to higher and higher dimensions, any spline fitting method becomes computationally expensive and something to note for any tracking code.



Integrating trajectories

- So far, we have looked at:
  - Generating particle distributions
  - Field maps and interpolation
- Now we need to move on to figuring out the particles' trajectories through our system.
  - Recall from a few slides ago, we said that the equation of motion we get is:

$$\begin{pmatrix} \ddot{x} \\ \ddot{y} \\ \ddot{z} \end{pmatrix} = \frac{q}{\gamma m} \begin{pmatrix} E_x(x,y,z,t) + v_y B_z(x,y,z,t) - v_z B_y(x,y,z,t) \\ E_y(x,y,z,t) - v_x B_z(x,y,z,t) + v_z B_x(x,y,z,t) \\ E_z(x,y,z,t) + v_x B_y(x,y,z,t) - v_y B_x(x,y,z,t) \end{pmatrix}$$

  - While this is true, for relativistic systems, this is not the most appropriate method.

Integrating trajectories

$$\begin{pmatrix} \ddot{x} \\ \ddot{y} \\ \ddot{z} \end{pmatrix} = \frac{q}{\gamma m} \begin{pmatrix} E_x(x,y,z,t) + v_y B_z(x,y,z,t) - v_z B_y(x,y,z,t) \\ E_y(x,y,z,t) - v_x B_z(x,y,z,t) + v_z B_x(x,y,z,t) \\ E_z(x,y,z,t) + v_x B_y(x,y,z,t) - v_y B_x(x,y,z,t) \end{pmatrix}$$

- This method relies on us using position and velocity, but in relativistic systems, $v \leq c$
  - Therefore, as we accelerate our particles, the increase in velocity gets smaller.
    - A numerical error could push the velocity over the speed of light and the tracking code would break down.
  - If we remember the Lorentz force:

$$F = q(E + v \times B) = \frac{dp}{dt}$$

    - Although the velocity change varies, for a given force, the momentum increases linearly with time!

Integrating trajectories

- The first thing we need to do is describe all velocity-related variables in terms of momentum. It's useful to remember:

$$pc = \beta\gamma mc^2$$
$$p_k c = \beta_k \gamma mc^2$$
$$E = \gamma mc^2$$
$$E^2 - p^2c^2 = m^2c^4$$

- Where p_k means the momentum in the k-direction
- Note that in here $c = 1$ if we are working in natural units (elsewhere in the tracking code it won't be, which is a common cause of errors that even I fall foul of!).
  - To avoid this confusion, we will ignore the c's and rewrite these as

$$p = \beta\gamma m$$
$$p_k = \beta_k \gamma m$$
$$E = \gamma m$$
$$E^2 - p^2 = m^2$$

Integrating trajectories

$$p = \beta\gamma m$$
$$p_k = \beta_k\gamma m$$
$$E = \gamma m$$
$$E^2 - p^2 = m^2$$

- We will now use these to help us write velocity in terms of momentum:

$$v_k = \frac{p_k c}{E} = \frac{p_k c}{\sqrt{p^2 + m^2}}$$

- Note: in this equation, $\boldsymbol{c = 3 \times 10^8}$ which is subtle but very important!

Integrating trajectories

- Finally we can rewrite out equation of motion into a more appropriate form:

$$\frac{d\boldsymbol{p}}{dt} = q(\boldsymbol{E} + \boldsymbol{v} \times \boldsymbol{B}) = q\left(\boldsymbol{E}(x,y,z,t) + \frac{c}{\sqrt{p^2 + m^2}}\boldsymbol{p} \times \boldsymbol{B}(x,y,z,t)\right)$$

- The charge is given in units of electrons, so for electrons or protons, we can take it to be -1 or +1 respectively, allowing us to simplify our equation to:

$$\begin{pmatrix} \dot{p_x} \\ \dot{p_y} \\ \dot{p_z} \end{pmatrix} = \begin{pmatrix} E_x + \frac{c}{\sqrt{p^2 + m^2}}(p_y B_z - p_z B_y) \\ E_y + \frac{c}{\sqrt{p^2 + m^2}}(p_z B_x - p_x B_z) \\ E_x + \frac{c}{\sqrt{p^2 + m^2}}(p_x B_y - p_y B_x) \end{pmatrix}$$

- For simplicity, I will just use the vector form of this equation from now on, but please not that in reality is a set of 3 coupled differential equations.

Integrating trajectories – simplest integrator

$$\frac{\Delta \boldsymbol{p}}{\Delta t} \approx \dot{\boldsymbol{p}} = \boldsymbol{E} + \frac{c}{\sqrt{p^2 + m^2}} \boldsymbol{p} \times \boldsymbol{B}$$

- We will assume that we are dealing with a system that is either DC (e.g. dipole magnet) or single frequency (e.g. RF cavity), so we can pull out the time dependence as:

$$\frac{\Delta \boldsymbol{p}}{\Delta t} \approx \left(\boldsymbol{E} + \frac{c}{\sqrt{p^2 + m^2}} \boldsymbol{p} \times \boldsymbol{B} \right) e^{i\omega t}$$

- We will use our field maps and interpolation to estimate the electric and magnetic fields (usually the magnetic field is given as $H = \frac{B}{\mu_0} \Rightarrow B = 4\pi \times 10^{-7} H$)

$$\frac{p_{(n+1)} - p_n}{\delta t} \approx \left(E_n + \frac{c}{\sqrt{p_n^2 + m^2}} p_n \times B_n \right) e^{i\omega t_n}$$

Integrating trajectories – simplest integrator

$$\frac{\boldsymbol{p}_{(n+1)} - \boldsymbol{p}_n}{\delta t} \approx \left(\boldsymbol{E}_n + \frac{c}{\sqrt{p_n^2 + m^2}} \boldsymbol{p}_n \times \boldsymbol{B}_n \right) e^{i\omega t_n}$$

- Rearranging, this gives us:

$$\boldsymbol{p}_{(n+1)} \approx \boldsymbol{p}_n + \delta t \left(\boldsymbol{E}_n + \frac{c}{\sqrt{p_n^2 + m^2}} \boldsymbol{p}_n \times \boldsymbol{B}_n \right) e^{i\omega t_n} = \boldsymbol{p}_n + e^{i\omega t_n} \boldsymbol{F}_n \delta t$$

$$\boldsymbol{v}_n = \frac{\boldsymbol{p}_n c}{\sqrt{p_n^2 + m^2}}$$

$$\boldsymbol{v}_{(n+1)} = \frac{\boldsymbol{p}_{(n+1)} c}{\sqrt{p_{(n+1)}^2 + m^2}}$$

$$\boldsymbol{x}_{(n+1)} = \boldsymbol{x}_n + \frac{\left(\boldsymbol{v}_n + \boldsymbol{v}_{(n+1)} \right)}{2} \delta t$$

- This method is called an Euler integrator
- Note: δt is called the timestep

Integration methods

- **Integrator order**: this refers to order of the numerical error
  - E.g. an Euler integrator is a 1st order integrator, so it has errors that are 2nd order or higher.
  - Some integrators can limit the maximum error to a certain order, while others allow errors to propagate and grow over time.
- **Symplectic integrators**: the total energy of a system is conserved, which in turn conserves the phase space emittance
  - Non-symplectic integrators can lose or gain energy over many iterations, but this is only really an issue if we want to track particles for very long times.
- If we want to improve our tracking accuracy, we can:
  - Reduce the timestep
  - Increase the integrator order

Most common integrators

Lancaster University

- Euler integrator:
  - Very basic integrator, but very poor accuracy, almost never used
- 4^{th} order Runge-Kutta integrator:
  - Quite easy to set up, good accuracy, almost always the method of choice
  - Non-symplectic, so not appropriate for long-term simulations
  - RK integrators can come in higher orders, but RK4 is most popular.
- Leap frog algorithms:
  - Describes a class of methods of different order, can be symplectic or not.
  - Position is evaluated at the timesteps, velocity is between timesteps.
  - Velocity calculations prone to divergences, which is bad for relativistic applications!
  - The Boris “push” algorithm is a leap frog-like algorithm that’s second order and can overcome this velocity divergence issue (a popular choice)

RK4 – 4th order Runge-Kutta

Lancaster University

- The algorithm works for equations of the form:

$$\frac{dy}{dt} = f(t, y), y(t_0) = y_0$$

- We get:

$$y_{n+1} = y_n + \frac{\delta t}{6}(k_1 + 2k_2 + 2k_3 + k_4)$$

- Where

$$k_1 = f(t_n, y_n)$$
$$k_2 = f\left(t_n + \frac{\delta t}{2}, y_n + k_1\frac{\delta t}{2}\right)$$
$$k_3 = f\left(t_n + \frac{\delta t}{2}, y_n + k_2\frac{\delta t}{2}\right)$$
$$k_4 = f(t_n + \delta t, y_n + k_3\delta t)$$

RK4 – 4th order Runge-Kutta

Lancaster University

- For our case, we have a second order differential equation, so we can write:

$$\frac{dp}{dt} = f(t,p) = \boldsymbol{E} + \frac{c}{\sqrt{p^2 + m^2}}\boldsymbol{p} \times \boldsymbol{B}$$

$$\frac{dx}{dt} = g(t,x) = \frac{p_x c}{\sqrt{p^2 + m^2}} = v_x$$

- If velocity doesn't change much (as we are relativistic), then we can solve the second equation with a simple integrator, like Euler, no need for RK4.
  - Hard to write $\frac{dx}{dt}$ as an explicit function of x, so RK4 isn't much better than Euler in this case.