## nBody Solvers: serial Implementation

- In outline, a serial n-body solver can be based on the following pseudocode:

```
1.    Get input data;
2.    for each timestep {
3.          if (timestep output) Print positions and velocities of particles;
4.    for each particle q
5.          Compute total force on q;
6.    for each particle q
7.          Compute position and velocity of q;
8.    }
9.    Print positions and velocities of particles;
```

- We can use our formula for the total force on a particle (Formula.2) to refine our pseudocode for the computation of the forces in Lines 4–5:

```
for each particle q {
   for each particle k != q {
      x_diff = pos[q][X] - pos[k][X];
      y_diff = pos[q][Y] - pos[k][Y];
      dist = sqrt(x_diff*x_diff + y_diff*y_diff);
      dist_cubed = dist*dist*dist;
      forces[q][X] -= G*masses[q]*masses[k]/dist_cubed * x_diff;
      forces[q][Y] -= G*masses[q]*masses[k]/dist_cubed * y_diff;
   }
}
```

- We're assuming that the forces and the positions of the particles are stored as two-dimensional arrays, **forces** and **pos**, respectively. We're also assuming we've defined constants X = 0 and Y = 1. So the x-component of the force on particle q is **forces[q][X]** and the y-component is **forces[q][Y].** Similarly, the components of the position are **pos[q][X]** and **pos[q][Y].**

$$\begin{bmatrix} 0 & f_{01} & f_{02} & \cdots & f_{0,n-1} \\ -f_{01} & 0 & f_{12} & \cdots & f_{1,n-1} \\ -f_{02} & -f_{12} & 0 & \cdots & f_{2,n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ -f_{0,n-1} & -f_{1,n-1} & -f_{2,n-1} & \cdots & 0 \end{bmatrix}.$$

- We can use Newton's third law of motion, that is, for every action there is an equal and opposite reaction, to halve the total number of calculations required for the forces. If the force on particle ***q*** due to particle ***k*** is $\mathbf{f}_{qk}$, then the force on k due to q is $-\mathbf{f}_{qk}$. Using this simplification we can modify our code to compute forces, as shown in Program below:

```
for each particle q
   forces[q] = 0;
for each particle q {
   for each particle k > q {
      x_diff = pos[q][X] - pos[k][X];
      y_diff = pos[q][Y] - pos[k][Y];
      dist = sqrt(x_diff*x_diff + y_diff*y_diff);
      dist_cubed = dist*dist*dist;
      force_qk[X] = G*masses[q]*masses[k]/dist_cubed * x_diff;
      force_qk[Y] = G*masses[q]*masses[k]/dist_cubed * y_diff

      forces[q][X] += force_qk[X];
      forces[q][Y] += force_qk[Y];
      forces[k][X] -= force_qk[X];
      forces[k][Y] -= force_qk[Y];
   }
}
```

- For each particle q will add the entries in row 0 into forces[0]. It will also add the kth entry in column 0 into forces[k] for k = 1, 2, : : : ,n-1. In general, the qth iteration will add the entries to the right of the diagonal (that is, to the right of the 0) in row q into forces[q], and the entries below the diagonal in column q will be
  added into their respective forces, that is, the kth entry will be added in to forces[k].
- It's necessary to initialize the forces array in a separate loop, since the qth iteration of the loop that calculates the forces will, in general, add the values it computes into forces[k] for k = q+1,q+2, : … , n-1, not just forces[q].

- In order to distinguish between the two algorithms, we'll call the n-body solver with the original force calculation, the basic algorithm, and the solver with the number of calculations reduced, the reduced algorithm. The position and the velocity remain to be found. We know that the acceleration of particle q is given by

$$\mathbf{a}_q(t) = s_q''(t) = \mathbf{F}_q(t)/m_q,$$

  where $S_q$''(t) is the second derivative of the position $S_q(t)$ and $F_q(t)$ is the force on particle q. We also know that the velocity $V_q(t)$ is the first derivative of the position $S_q$'(t) so we need to integrate the acceleration to get the velocity, and we need to integrate the velocity to get the position.

- In Euler's method, we use the tangent line to approximate a function. The basic idea is that if we know the value of a function $g(t_0)$ at time $t_0$ and we also know its derivative $g'(t_0)$ at time $t_0$, then we can approximate its value at time t0 C1t by using the tangent line to the graph of $g(t_0)$ See Figure 1 for an example. Now if we know a point $(t_0,g(t_0))$ on a line, and we know the slope of the line $g'(t_0)$, then an equation for the line is given by

$$\mathbf{y = g(t_0)+g'(t_0)(t-t_0)}$$

- Since we're interested in the time $t = t_0 + \Delta t$, we get

$$g(t+\Delta t) \approx g(t_0) + g'(t_0)(t+\Delta t - t) = g(t_0) + \Delta t g'(t_0).$$

- Now we know the value of $S_q(t)$ and $S_q'(t)$ at time 0, so we can use the tangent line and our formula for the acceleration to compute $S_q(\Delta t)$ and $V_q(\Delta t)$

$$\mathbf{s}_q(\Delta t) \approx \mathbf{s}_q(0) + \Delta t \mathbf{s}_q'(0) - \mathbf{s}_q(0) + \Delta t \mathbf{v}_q(0),$$

$$\mathbf{v}_q(\Delta t) \approx \mathbf{v}_q(0) + \Delta t \mathbf{v}_q'(0) = \mathbf{v}_q(0) + \Delta t \mathbf{a}_q(0) = \mathbf{v}_q(0) + \Delta t \frac{1}{m_q} \mathbf{F}_q(0).$$

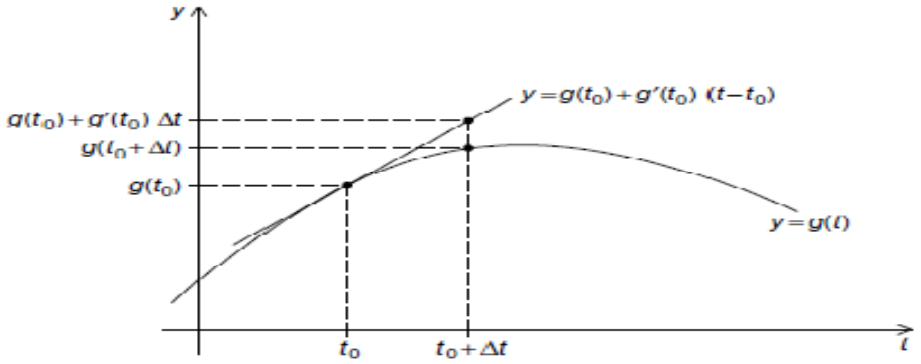


Fig 1: Using tangent line to approximate the function

- When we try to extend this approach to the computation of Sq (2Δt) and S'q(2Δt) we see that things are a little bit different, since we don't know the exact value of Sq(Δt) and S'q(Δt) However, if our approximations to Sq(Δt) and S'q(Δt) are good, then we should be able to get a reasonably good approximation to Sq (2Δt) and S'q(2Δt) using the same idea. This is what Euler's method does Fig 2.

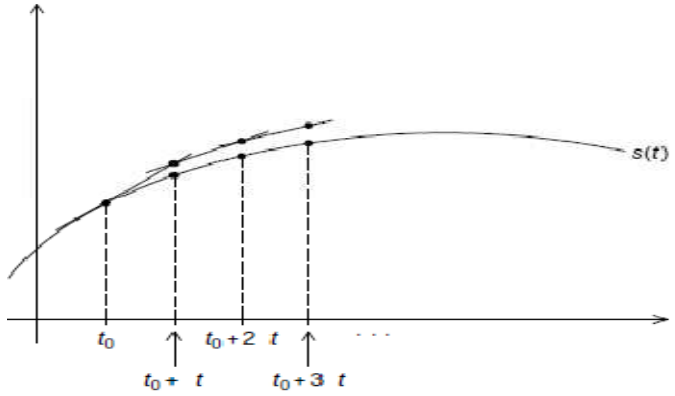


Fig2 : Euler's Method

- The pseudocode for the two n-body solvers by adding in the code for computing position and velocity:

```
pos[q][X] += delta_t*vel[q][X];
pos[q][Y] += delta_t*vel[q][Y];
vel[q][X] += delta_t/masses[q]*forces[q][X];
vel[q][Y] += delta_t/masses[q]*forces[q][Y];
```

- Here, we're using pos[q], vel[q], and forces[q] to store the position, the velocity,
  and the force, respectively, of particle q.

- Before moving on to parallelizing our serial program, let's take a moment to look
  at data structures. We've been using an array type to store our vectors:

```
#define DIM 2
typedef double vect_t[DIM];
```

- A struct is also an option. However, if we're using arrays and we decide to change our program so that it solves three-dimensional problems, in principle, we only need to change the macro DIM. If we try to do this with structs, we'll need to rewrite the code that accesses individual components of the vector.

- For each particle, we need to know the values of
  - its mass,
  - its position,
  - its velocity, .
  - its acceleration, and .
  - the total force acting on it.
- Since we're using Newtonian physics, the mass of each particle is constant, but the other values will, in general, change as the program proceeds.

- With the forces stored in contiguous memory, we can use a fast function such as ***memset*** to quickly assign zeroes to all of the elements at the beginning of each iteration:

```
#include <string.h> /*For memset*/
. . .
Vect_t* forces = malloc(n*sizeof(vect t));
. . .
for (step = 1; step <= n_steps; step++) {
. . .
/* Assign 0 to each element of the forces array */
forces = memset(forces, 0, n*sizeof(vect t);
for (part = 0; part < n-1; part++)
Compute_force(part, forces, . . .)
```

```
...
}
```