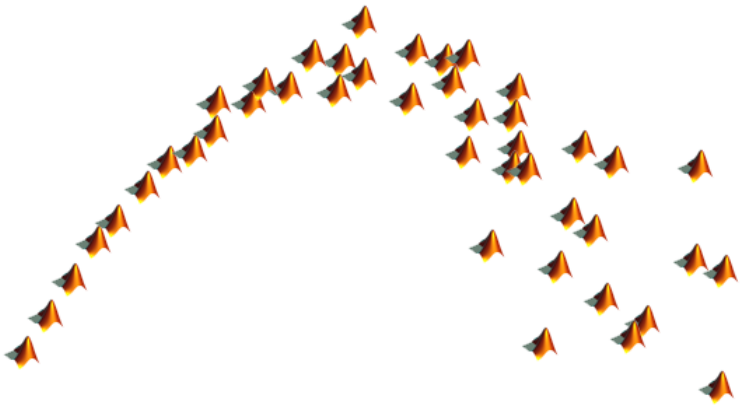


Matlab Particles 2.1

Jörg J. Buchholz

January 12, 2009

For Saskia

Contents

When Karl Sims did his award-winning computer animation "Particle Dreams" twenty(!) years ago, he tortured a Connection Machine CM-2 computer with as many as 65,536 processors, using one processor for the simulation of each particle.

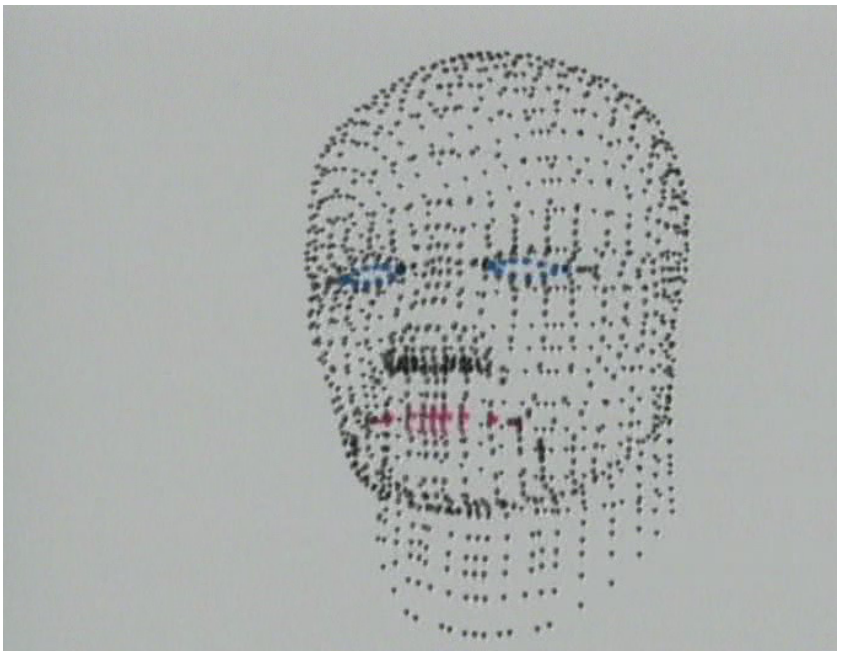


Figure 1: 1988 Computer Animation "Particle Dreams" [9]

Today we simulate tens of thousands of particles in real-time on a single cpu (Figure 2) – even in a browser plugin (Figure 3) – and advanced particle systems have become common practice for the simulation of snow, rain, dust, smoke, fire, and explosions in most computer games. Modern simulation environments like *Processing* [3] can be used to produce such astonishingly addicting games as *Falling Sand Game* [6], *sodaplay* [10], *BallDroppings* [7], and *Souptoys* [13].

In 2006, Traer Bernstein [2] wrote a pretty impressing particle physics library for *Processing*, which actually was the inspiration for this particle system toolbox in MATLAB. As a matter of fact, object oriented programming in MATLAB is not really the fastest lane on the particle system highway; we are back at the good old days of some ten or twenty real-time particles. But – the main purpose of this toolbox has never been to develop state-of-the-art computer games; it was rather planned as an educational, interactive learning-by-doing playground, with the aim to understand the mechanical interactions (and maybe the mathematical background) of the particle system components. Have fun!

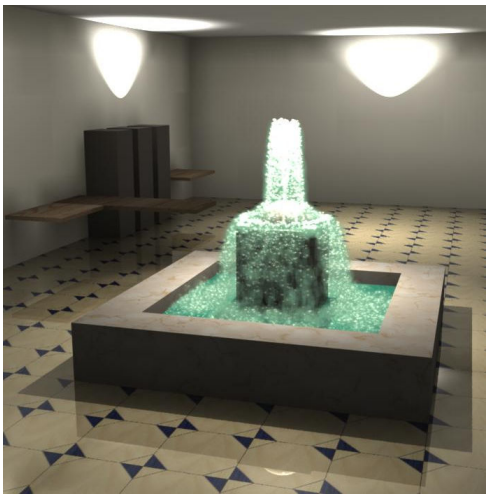


Figure 2: Particle System API [5]

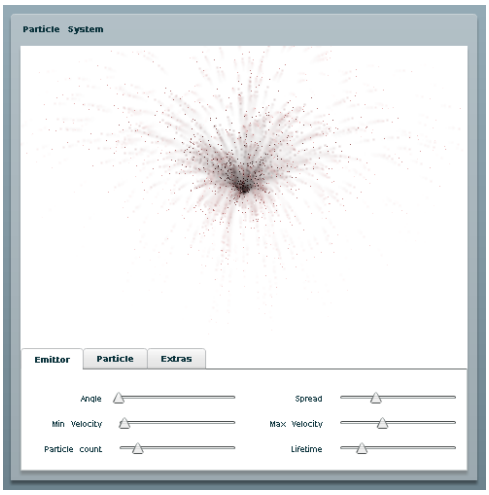


Figure 3: Flash 9 Particle System [1]

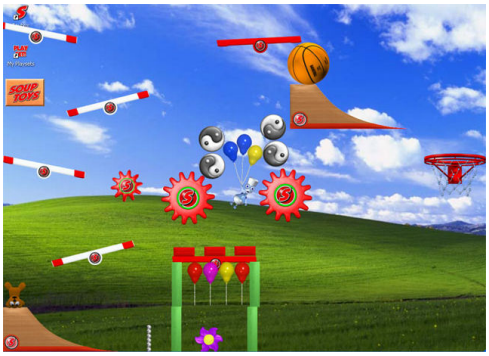


Figure 4: Souptoys [13]

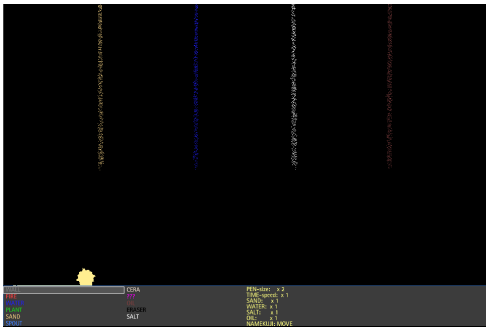


Figure 5: Falling Sand Game [6]

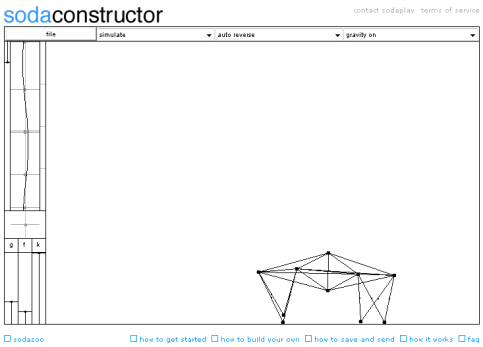


Figure 6: Sodaplay [10]

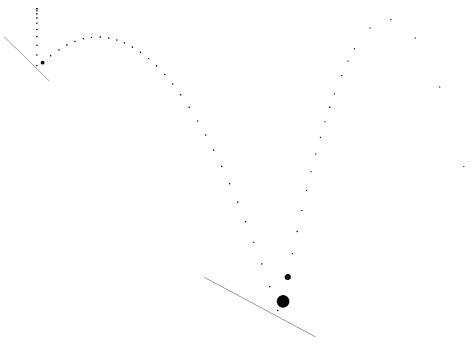


Figure 7: BallDroppings [7]

1 Genesis

The main purpose of this section is to give you a brief overview on how particles, springs, and attractions are created and implemented into the particle system.

1.1 Particle System Creation

Every particle system simulation begins with the creation of a particle system object[1] that maintains and manages all particles, springs, and attractions

```
>> Particle_System = particle_system

            gravity: [0 0 0]
               drag: 0
          particles: []
            springs: []
       attractions: []
               time: 0
    graphics_handle: 1
```

The command `particle_system` opens an empty 3-D axes in a maximized window and creates an empty particle system with default properties

- Acceleration due to gravity: $\begin{bmatrix}0 & 0 & 0\end{bmatrix}$
- Aerodynamic/fluid resistance (drag): 0
- Axes limits: ± 1

After the creation of the particle system object its properties can be defined

```
>> gravity = [0 0 -9.81];
```

and set

```
>> Particle_System.gravity = gravity

            gravity: [0 0 -9.8100]
               drag: 0
          particles: []
            springs: []
       attractions: []
               time: 0
    graphics_handle: 1
```

[1] As a convention throughout this toolbox, MATLAB objects names begin with uppercase letters.

In many cases it might be more convenient to set all three particle system properties (`gravity`, `drag`, and `limits`) directly during the creation of the object

```
>> Particle_System = particle_system ([0 0 -9.81], 0, 1)

            gravity: [0 0 -9.8100]
               drag: 0
          particles: []
            springs: []
        attractions: []
               time: 0
    graphics_handle: 1
```

1.2 Particle Creation

A particle object with default properties is created and incorporated into the particle system by the `particle` command using the particle system (handle) as its only parameter

```
>> Particle = particle (Particle_System)

               mass: 1
           position: [0 0 0]
           velocity: [0 0 0]
              fixed: 0
           lifespan: Inf
                age: 0
              force: [0 0 0]
    graphics_handle: 191.0077
```

Again, the particle properties (mass, initial position and velocity, fixed/free, and lifespan) can either be set individually with an existing particle object

```
>> Particle.mass = 42

               mass: 42
           position: [0 0 0]
:
```

or at once during the creation of the object

```
>> Particle = particle ...
(Particle_System, 42, [0 0 0], [0 0 0], false, inf)

               mass: 42
```

```
            position: [0 0 0]
:
```

The particle is automatically incorporated into the particle system and a red dot (symbolizing the particle) is drawn in the axes of the particle system. After its creation, the color (or any other property) of the graphical particle representation can be modified by means of its graphics handle

```
set (Particle.graphics_handle, 'color', 'blue');
```

1.3 Spring Creation

A default spring between two already existing particles `Particle_1` and `Particle_2` is created and incorporated into the particle system via

```
>> Spring = spring (Particle_System, Particle_1, Particle_2)

          particle_1: [1x1 particle]
          particle_2: [1x1 particle]
                rest: 1
            strength: 1
             damping: 0
      graphics_handle: 195.0077
```

Again, the long version directly sets the rest length and the strength of the spring and the damping coefficient of an additional damper parallel to the spring

```
>> Spring = spring ...
(Particle_System, Particle_1, Particle_2, 2, 1, 0)

          particle_1: [1x1 particle]
          particle_2: [1x1 particle]
                rest: 2
            strength: 1
             damping: 0
      graphics_handle: 196.0077
```

The default graphical representation of a spring is a solid blue line between the corresponding particles.

1.4 Attraction Creation

Just like a spring, an attraction (force) connects two particles

```
>> Attraction = attraction ...
(Particle_System, Particle_1, Particle_2)

          particle_1: [1x1 particle]
          particle_2: [1x1 particle]
            strength: 1
    minimum_distance: 0
     graphics_handle: 198.0077
```

Its properties are the gravitational attraction strength (which can be made negative for a repulsion) and the minimum distance, below which the attraction force will not grow any further. Analogous to the particle and the spring, the following attraction uses are valid syntax

```
>> Attraction = attraction ...
(Particle_System, Particle_1, Particle_2, 1, 0);

>> Attraction.minimum_distance = 0.1;

>> strength = Attraction.strength;

>> set (Attraction.graphics_handle, 'visible', 'off');
```

An attraction is graphically represented by a dotted blue line.

1.5 Simulation

After at least one particle has been thrown into life, the simulation can be started

```
step_time = 0.01;

for i = 1 : inf

  Particle_System.advance_time (step_time);

end
```

Since `inf` has been used as the upper bound of the `for`-loop, the simulation can only be stopped by `Ctrl-C` or by closing the corresponding figure window. In the loop, every call to `advance_time` integrates the underlying nonlinear differential equation system

one single time step further, the length of which is defined in the variable `step_time`. Decreasing `step_time` makes the simulation smoother and more exact, but also slower; increase `step_time` to make the particles move faster. If you get too greedy speed-wise, the simulation becomes bumpier and finally unstable. In the simulation loop the current position of the mouse pointer can be determined

```
current_point = mean (get (gca, 'currentpoint'));
```

and e. g. used to move[2] certain particles "by hand"

```
Particle.position = current_point;
```

[2]Actually, the command does not directly move the graphical particle representation, but only sets the position property of the particle object. The corresponding graphics object is then automatically updated during the next simulation cycle.

2 Demos

It might be a good idea to read through the demos in chronological order, since the grade of general parameter explanation detailedness is higher in the first demo descriptions.

2.1 Demo 1: Free Fall

The first – pretty boring – demo simulates the free fall of a single mass in a gravity field. After the creation of a default particle system (no drag, unit limits) with "earthy" gravity

```
Particle_System = particle_system;

gravity = [0 0 -9.81];

Particle_System.gravity = gravity
```

the default 3-D view is reduced to two dimensions

```
view (0, 0);
```

A single particle with default properties (unit mass, initial position at the origin, no initial velocity, not fixed, infinite lifespan) is created and incorporated into the particle system

```
Particle = particle (Particle_System);
```

A step time of 1 millisecond seems to produce a smooth and visually traceable particle motion on a 3 GHz computer[3]:

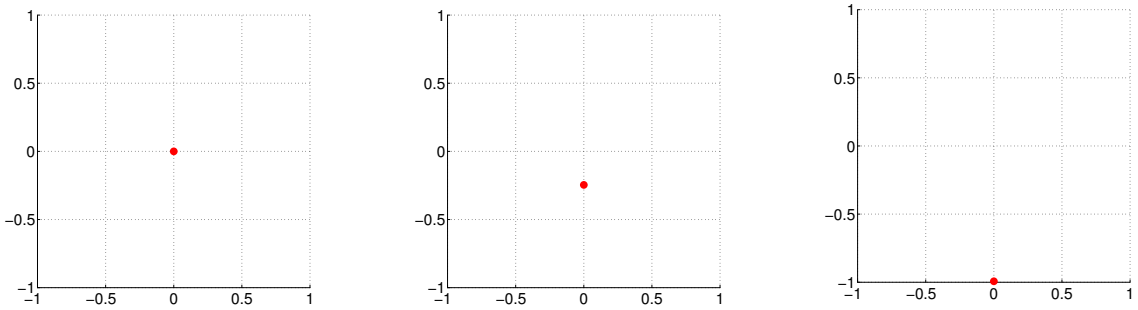
```
step_time = 0.001;
```

Finally, the most simple form of a simulation loop can be used

```
for i = 1 : 451

  Particle_System.advance_time (step_time);

end
```

Choosing `451` as the number of steps makes the simulation stop when the particle reaches the lower axes limit[4].

[3] If you use a faster machine, you might want to reduce the step time (and v. v.).
[4] The junior high proof is up to the reader ...



(a) $t = 0$ (b) $t = 0.225$ (c) $t = 0.451$

Figure 8: Free fall

As you might have expected, the unspectacular simulation (Figure 8) shows a red dot falling down with (linearly) increasing velocity.

2.2 Demo 2: Bullet Time

The second demo simulates the motion of a bullet in zero gravity with aerodynamic resistance[5]. The few lines of source code are very similar to the first demo. Create a particle system with no gravity but a drag coefficient of `10` and set the view to 2D

```
Particle_System = particle_system ([0, 0, 0], 10, 1);

view (0, 0);
```

Position a particle on the left edge (`-1`) of the axes and give it an initial velocity of `20` to the right

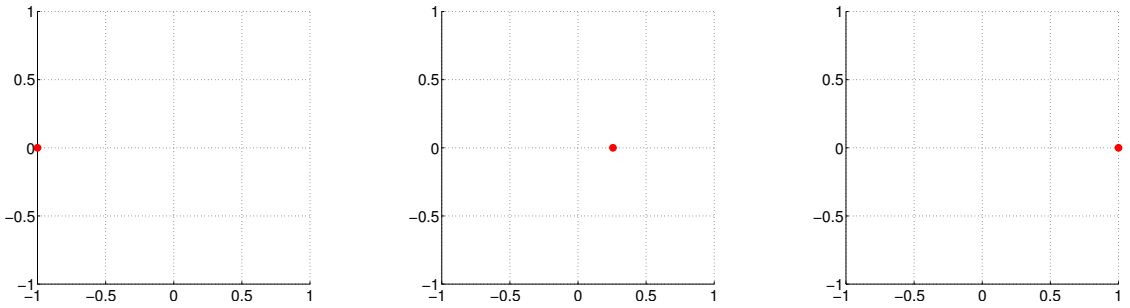
```
Particle = particle ...
(Particle_System, 1, [-1, 0, 0], [20, 0, 0], false, inf);
```

Simulate infinitely long with a step time of 1 millisecond

```
step_time = 0.001;

for i = 1 : inf

  Particle_System.advance_time (step_time);

end
```

[5]Can you imagine an environment (or a movie title) where this experiment could take place?

The simulation depicted in Figure 9 produces the same low level of enthusiasm as the previous demo.



(a) $t = 0$ (b) $t = 0.1$ (c) $t = 1$

Figure 9: Bullet time

The "bullet" starts with initial kinetic energy at the left hand side of the axes (Figure 9 a), becomes slower and slower due to the aerodynamic resistance (Figure 9 b) until it comes to a complete[6] stop at the right hand side, when all of its energy has been dissipated (Figure 9 c).

2.3 Demo 3: Bungee Jumping

Wikipedia defines bungee jumping as

> an activity in which a person jumps off from a high place (generally of several hundred feet/meters) with one end of an elastic cord attached to his/her body or ankles and the other end tied to the jumping-off point.

Well, what do you need for a half-decent bungee jumping simulation? The jumping person is just a free particle with a certain mass and the elastic cord can be modeled by a spring between the jumper and another (hopefully fixed) "particle". Add some aerodynamic resistance and a bit of spring damping and the jump will look quite realistic. In order to open up the second dimension, the jumping-off point should not equal the fixed chord end point[7].

$< Genesis\_mode\_on >$

[6] Actually, the bullet will not really stop before Judgment Day, but its velocity will rapidly become too small to be visually detected.

[7] Besides, this overcomes the problem that a spring (but not a bungee chord) produces a pressing force if its length is less than its rest length. A similar real-world effect that cannot easily be modeled by a simple spring is the fact that the first part of the jump is actually a free fall where the chord does not produce any force at all.

In the beginning The User created the particle system and the particles. And The User said, Let there be gravity and aerodynamic resistance, and there was gravity and aerodynamic resistance

```
Particle_System = particle_system ([0, 0, -9.81], 5, 200);
```

The User saw all that he had made, and it was very good (even in 2D)

```
view (0, 0);
```

$< Scientific\_mode\_on >$

The first particle acts as the anchor point the chord is "tied to". Its initial position is fixed (`true`) somewhere in the upper half of the axes (`[0, 0, 150]`) and it has an infinite life span[8]. Since the particle is fixed and does not move, its mass is completely irrelevant[9]. The default graphical representation of a fixed particle is a red asterisk

```
Particle_1 = particle ...
(Particle_System, 1, [0, 0, 150], [0, 0, 0], true, inf);
```

The second particle represents the jumper. It should have a reasonable mass[10] and an initial position at the height of the first particle, with a horizontal offset equalling the rest length of the spring

```
Particle_2 = particle ...
(Particle_System, 70, [100, 0, 150], [0, 0, 0], false, inf);
```

The two particles are now used to define both ends of the spring to be created

```
Spring = spring ...
(Particle_System, Particle_1, Particle_2, 100, 6, 0.1);
```

The spring has a rest length[11] of `100`, a strength[12] of `6`, and a damping factor[13] of `0.1`. To make the graphical representation (which is a solid blue line by default) look a bit more like a real spring, you can increase its line width and change its line style to dotted

```
set (Spring.graphics_handle, 'linewidth', 10, 'linestyle', ':')
```

The particle system is now ready to be pushed into life

[8] If you feel a strong urge to act out your sadistic touch, you could reduce the life span of the first particle to e. g. 42 and see what happens to the jumper...

[9] Nevertheless, in order to avoid "Divide by zero" warnings, the mass of a particle should never be zero.

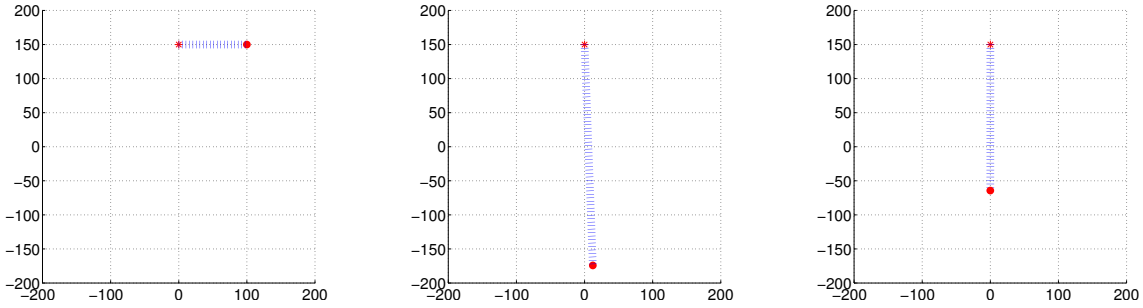
[10] Yes, 70 is a reasonable mass for an adult in the SI metric system.

[11] Rest length: Length of a spring producing no force

[12] Strength: Deflection-dependent force coefficient

[13] Damping: Velocity-dependent force coefficient

```
step_time = 0.1;

for i = 1 : inf

  Particle_System.advance_time (step_time);

end
```



(a) $t = 0$ (b) $t = 12$ (c) $t = 200$

Figure 10: Bungee jumping

Departing from Figure 10 a with a relaxed spring, the free particle is accelerated downwards in the gravity field and immediately to the left by the horizontal component of the spring force. At about 12 sec, it reaches its lowest point (Figure 10 b) and is driven back up by the extended spring. After a few more vertical and horizontal oscillations the atmospherical and the spring damping have eaten up all kinetic energy. Figure 10 c shows the "final" steady state, where the weight of the particle and the spring force are in equilibrium.

2.4 Demo 4: Gimme ya Energy!

This demo does not offer too many new insights, compared to the previous one; it is just a nice two-masses-two-springs experiment, where a big particle sucks energy from a smaller one (just like in real life...). There is no gravity nor drag in this environment

```
Particle_System = particle_system ([0, 0, 0], 0, 3);
```

The first particle is just a nail in the middle of a wall, where you can later on fix the first spring

```
Particle_1 = particle ...
(Particle_System, 1, [0, 0, 0], [0, 0, 0], true, inf);
```

The second particle is free and has a bigger mass (10)

```
Particle_2 = particle ...
(Particle_System, 10, [1, 0, 0], [0, 0, 0], false, inf);
```

which can also be graphically represented

```
set (Particle_2.graphics_handle, 'markersize', 60);
```

The third particle is smaller (1) than the second one, but has an initial velocity (5) driving it upwards

```
Particle_3 = particle ...
(Particle_System, 1, [2, 0, 0], [0, 0, 5], false, inf);
```

Finally we create two identical springs with a strength of 10 and a damping factor of 1; one spring between the (fixed) first particle and the second particle

```
Spring_1 = spring ...
(Particle_System, Particle_1, Particle_2, 1, 10, 1);
```

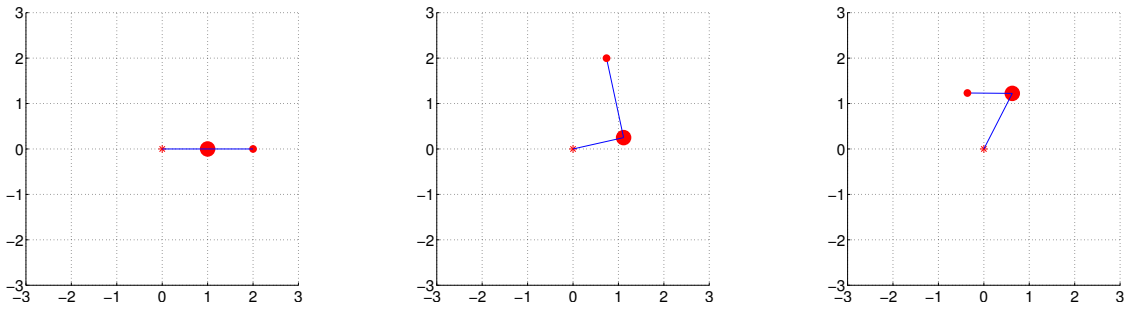
and another spring between the second and the third particle

```
Spring_2 = spring ...
(Particle_System, Particle_2, Particle_3, 1, 10, 1);
```

Ready to rumble

```
step_time = 0.1;

for i = 1 : inf

  Particle_System.advance_time (step_time);

end
```

The snapshot in Figure 11 a shows the initial positions of both particles and springs. The initial velocity of the small particle that will move it upwards in the next simulation steps is not directly visible. On its way up, the small particle exerts a force through the spring on the big particle; but due to the inertia of the big particle, at first, the small particle rotates about the big one (Figure 11 b). During its rotations, the small particle transfers more and more energy to the big particle, until finally the latter is rotating so fast about the fixed particle that the small particle is not able to do full rotations about the big one any more (Figure 11 c).



(a) $t = 0$ (b) $t = 1$ (c) $t = 100$

Figure 11: Gimme ya energy!

2.5 Demo 5: Magic Chain Cable

If you take a bunch of particles, arrange them in one long row and connect them with springs, you end up with something like a worm, a rope, a rubber band, a pearl necklace, or a chain cable – depending on the number of particles and the parameters you choose for mass, spring strength, damping, ... Define some gravity (`-1`), a bit of drag (`0.1`) and an ample axes (`10`)

```
Particle_System = particle_system ([0, 0, -1], 0.1, 10);
```

The number of particles (and the corresponding number of springs) directly determine the speed of the simulation. A single core 3 GHz machine running MATLAB object oriented programming would be hopelessly overextended even with a hundred particles. Therefore

```
n_particles = 20;
```

Next, the 20 particles are created and incorporated into the particle system

```
for i = 1 : n_particles

  Particles{i} = particle ...
  (Particle_System, 0.2, [0, 0, 0], [0, 0, 0], false, inf);

  if i == 1 || i == n_particles

    Particles{i}.fixed = true;

  end

end
```

Additionally, the first and the last particle of the chain are declared as "fixed". This is useful, because their motion should not be influenced by the other particles, but will be externally modified in the simulation loop. The particles are connected by pretty stiff (`100`) springs with unit length (`1`) and a reasonable damping factor (`4`)

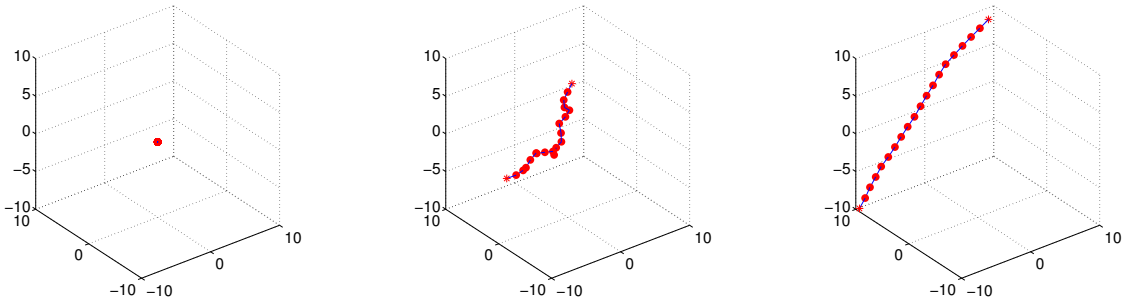
```
for i = 1 : n_particles - 1

  Spring{i} = spring ...
  (Particle_System, Particles{i}, Particles{i + 1}, 1, 100, 4);

end
```

Let the games begin

```
step_time = 0.05;

for i = 1 : inf

  Particles{1}.position = ...
  [10*sin(0.01*i), 10*sin(0.011*i), 10*sin(0.012*i)];

  Particles{n_particles}.position = ...
  [-10*sin(0.013*i), 10*sin(0.014*i), -10*sin(0.015*i)];

  Particle_System.advance_time (step_time);

end
```

In the simulation loop the first (`Particles{1}`) and last (`Particles{n_particles}`) particles are permanently moved by altering their `position` property. Both end particles are driven by slow, slightly distorted, harmonic oscillators with an amplitude of `10` in all three spacial dimensions. As a result, the particles seem to float randomly through the axes cube.

Since all particles have their initial positions at the origin, all you can see in Figure 12 a is one red dot. The springs are not visible at all; they are all compressed to zero length and therefore exert expanding forces on the particles immediately. In Figure 12 b the sine generators have moved the end particles apart and the springs have reached their rest length. Since the distance between the end particles is much smaller than the number of (unit) springs, the chain is not tense but relaxed in a random pattern. After the end particles have been forced into different vertices of the axes cube (Figure 12 c), the length of the chain is greater than the sum of the spring rest lengths, straightening the chain and extending every single spring.



(a) $t = 0$ (b) $t = 2$ (c) $t = 50$

Figure 12: Magic chain cable

2.6 Demo 6: Heavy Chain Mail

This demo is an attempt[14] to simulate the motion of a medieval knight's heavy chain mail. With adapted particle and spring parameters, other types of cloth can be simulated as well. To make things look realistic, there has to be some gravity (`-0.4`) and a little bit of drag (`0.1`)

```
Particle_System = particle_system ([0, 0, -0.4], 0.1, 5);

view (0, 0);
```

For a more detailed view the axes limits have to be readjusted

```
axis ([0, 6, -1, 1, 0, 6]);
```

Again, we are talking about MATLAB object oriented programming; so let's not overstrain the CPU and restrict ourselves to 25 particles, arranged in a square matrix-like structure of 5 rows by 5 columns

```
for col = 1 : 5

  for row = 1 : 5

    Particles{row, col} = particle ...
    (Particle_System, 1, [col, 0, row], [0, 0, 0], false, inf);

    if row == 5 && col == 1 || row == 5 && col == 5

      Particles{row, col}.fixed = true;
```

[14] Traer Bernstein [2] can do that much better...

```
    end

  end

end
```

After their creation, the upper left (5^{th} row, 1^{st} column) and upper right (5^{th} row, 5^{th} column) particles are nailed (`fixed`) to the wall. Next, you have to create the two-dimensional mesh by connecting every particle to its nearest neighbors with unit length springs. This can be done in two similar steps; one double loop for the horizontal springs

```
for col = 1 : 4

  for row = 1 : 5

    Springs_horizontal{row, col} = spring (Particle_System, ...
    Particles{row, col}, Particles{row, col + 1}, 1, 100, 1);

  end

end
```

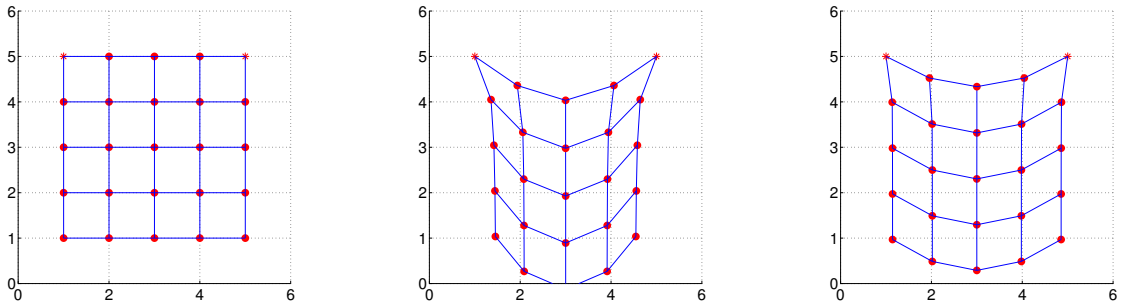
and another one for the vertical springs

```
for col = 1 : 5

  for row = 1 : 4

    Springs_vertical{row, col} = spring (Particle_System, ...
    Particles{row, col}, Particles{row + 1, col}, 1, 100, 1);

  end

end
```

The simulation loop looks pretty unspectacular

```
step_time = 0.1;

for i = 1 : inf

  Particle_System.advance_time (step_time);

end
```

Figure 13 a shows the initial state of the fabric; right before gravity starts sucking.



(a) $t = 0$ (b) $t = 3$ (c) $t = 50$

Figure 13: Heavy chain mail

After 3 sec (Figure 13 b) most of the particles have reached their maximum deflection, narrowing down the mean width of the texture. After a few oscillations (both horizontally and vertically), the "final steady" state of the web can be seen in Figure 13 c.

2.7 Demo 7: Don't Touch me!

The aim of this "outer space game" is to establish a circular orbit[15] of the free particle (planet) about the origin by moving the "fixed" particle (sun) to the right position at the right time. Problem: If the planet gets too close to the sun, the gravity force accelerates it quickly outside the visible square (the major axis of the elliptic orbit becomes very long). The particle system itself produces no gravity and no drag

```
Particle_System = particle_system ([0, 0, 0], 0, 10);

view (0, 0);
```

The first particle represents the "fixed" sun that will be moved by the mouse during the simulation

```
Particle_1 = particle ...
(Particle_System, 1, [0, 0, 0], [0, 0, 0], true, inf);
```

while the free planet is modeled by the second particle

```
Particle_2 = particle ...
(Particle_System, 1, [0, 0, 0], [0, 0, 0], false, inf);
```

[15] In an orbit, gravitational attraction and centrifugal forces are in equilibrium.

The gravitational attraction force between two masses is proportional to both masses and reciprocally proportional to the square of their distance. The attraction strength[16] is set to `10` and the minimum distance (below which the attraction force will not grow any further) is virtually set to zero (`eps = 2.2204e-016`)

```
Attraction = attraction ...
(Particle_System, Particle_1, Particle_2, 10, eps);
```

In the simulation loop, the current position of the mouse pointer is determined by the `currentpoint` property of the axes which returns a matrix of two points. The two points lie on the line that is perpendicular to the plane of the screen and passes through the pointer. The 3-D coordinates are the points where this line intersects the front and back surfaces of the axes volume. Therefore, even in 3-D the mean of both points can be used to intuitively move the first particle (sun) to a new position

```
step_time = 0.02;

for i = 1 : inf

  current_point = mean (get (gca, 'currentpoint'));

  current_point(2) = 0;

  Particle_1.position = current_point;

  Particle_System.advance_time (step_time);

end
```

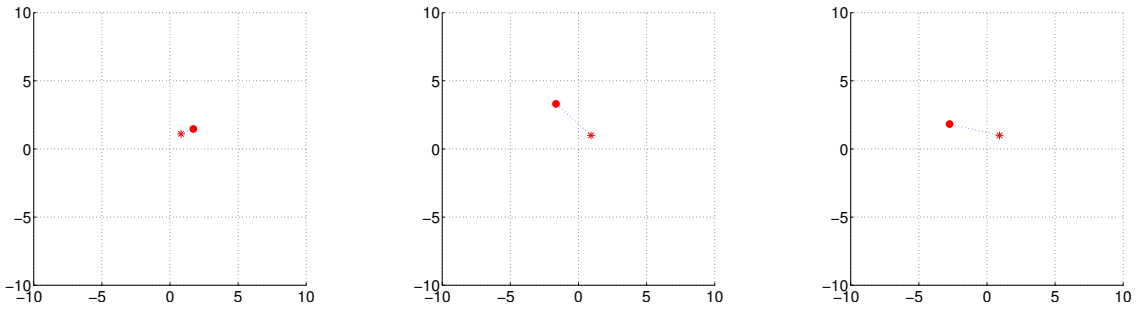
Keeping the second component of `current_point` at zero restricts the motion of the particle to the visible plane (thank you, Wulf!). Since this is an interactive demo, the situations depicted in Figure 14 depend on your own mouse pointer input and can therefore not easily be reproduced.

Nevertheless, playfully learning how a close encounter with a star gives you fresh impetus, can really be fun ...

2.8 Demo 8: Keep it up

If you switch on gravity and create a negative attraction between two particles (one of which is glued to your mouse) you end up with something like a more-dimensional

[16] From the view of real-world physics, the "attraction strength" between two masses is defined by big G (the gravitational constant: $G = 6.6742 \cdot 10^{-11}\,\mathrm{N\,m^2\,kg^{-2}}$), but it seems convenient to have an additional attraction strength twiddle factor in a particle system.



(a) $t = 2$ (b) $t = 4$ (c) $t = 6$

Figure 14: Don't touch me!

variable-length inverted pendulum. The 2-D particle system provides gravity (`-20`) but no drag (`0`)

```
Particle_System = particle_system ([0, 0, -20], 0, 10);

view (0, 0)
```

Both particles are created at the origin; the first one will later be moved by the mouse and should therefore be "fixed" (`true`)

```
Particle_1 = particle ...
(Particle_System, 1, [0, 0, 0], [0, 0, 0], true, inf);

Particle_2 = particle ...
(Particle_System, 1, [0, 0, 0], [0, 0, 0], false, inf);
```

An attraction with a negative attraction strength is a repulsion (like the force between two equal electrical charges)

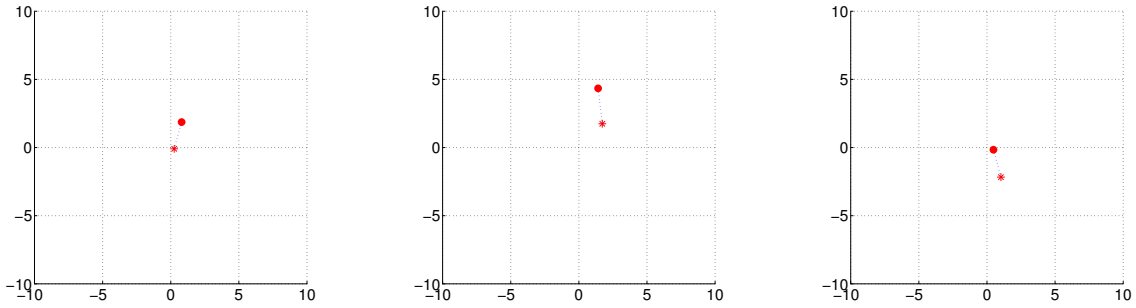
```
Attraction = attraction ...
(Particle_System, Particle_1, Particle_2, -100, eps);
```

In the simulation loop, the first particle follows the mouse pointer

```
step_time = 0.01;

for i = 1 : inf

  current_point = mean (get (gca, 'currentpoint'));

  current_point(2) = 0;
```

```
    Particle_1.position = current_point;

    Particle_System.advance_time (step_time);

end
```

As always, the speed of the simulation is directly proportional to the step time. If the default value of `0.01` does not really challenge your hand eye coordination and control capabilities (visual comprehension, finger dexterity, and small motor competence), just increase it ... [4]



(a) $t = 1$ (b) $t = 2$ (c) $t = 3$

Figure 15: Keep it up

Figure 15 shows some – not really stunning – intermediate states of the author's balancing act.

2.9 Demo 9: Catch me if you Can (There's a Hole in the Bucket)

While the inverted pendulum demands constant control activity to keep the free particle in the air, a "potential bucket" of three particles – arranged in the form of an equilateral triangle (V-configuration, s. Figure 16) – should give the user a chance to "catch" a fourth particle (the ball) and keep it in a steady state equilibrium; even in the presence of proper gravity (`-30`) and with the help of some drag (`1`)

```
Particle_System = particle_system ([0, 0, -30], 1, 10);

view (0, 0)
```

The three bucket particles and the ball particle are initially[17] located at

$$P_{left} = \begin{bmatrix} -2 \\ 0 \\ 2 \end{bmatrix} \qquad P_{right} = \begin{bmatrix} 2 \\ 0 \\ 2 \end{bmatrix} \qquad P_{bottom} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \qquad P_{ball} = \begin{bmatrix} 0 \\ 0 \\ 5 \end{bmatrix} \tag{1}$$

```
Particle_left = particle ...
(Particle_System, 1, [-2, 0, 2], [0, 0, 0], true, inf);

Particle_right = particle ...
(Particle_System, 1, [2, 0, 2], [0, 0, 0], true, inf);

Particle_bottom = particle ...
(Particle_System, 1, [0, 0, 0], [0, 0, 0], true, inf);

Particle_ball = particle ...
(Particle_System, 1, [0, 0, 5], [0, 0, 0], false, inf);
```

In order to keep the ball in the air, there are strong repulsions (`-100`) from every bucket particle to the ball

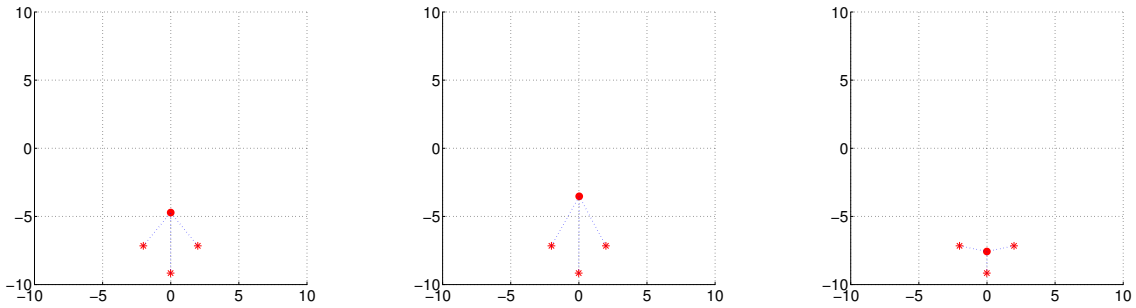
```
Attraction_1 = attraction ...
(Particle_System, Particle_left, Particle_ball, -100, eps);

Attraction_2 = attraction ...
(Particle_System, Particle_right, Particle_ball, -100, eps);

Attraction_3 = attraction ...
(Particle_System, Particle_bottom, Particle_ball, -100, eps);
```

In the simulation loop all three bucket particles are attached to the mouse pointer, preserving their relative positions towards each other

```
step_time = 0.01;

for i = 1 : inf

  current_point = mean (get (gca, 'currentpoint'));

  current_point(2) = 0;

  Particle_left.position = current_point + [-2 0 2];
```

[17] As a matter of fact, the initial position of the bucket particles is absolutely irrelevant, since the whole bucket is glued to the mouse pointer in the very first simulation step.

```
    Particle_right.position = current_point + [2 0 2];

    Particle_bottom.position = current_point + [0 0 0];

    Particle_System.advance_time (step_time);

end
```



(a) $t = 1$ (b) $t = 2$ (c) $t = 10$

Figure 16: Catch me if you can

Catching a slow ball (reaching a steady state in Figure 16c after some bouncing oscillations in Figure 16b) is not too difficult, but if you do not decelerate a fast ball carefully and *symmetrically* with the bottom particle, it will slip through the potential walls of the bucket and get lost. The reason for this "hole in the bucket" is illustrated (Figure 17) in the potential field of the initial particle configuration according to Equation 1.

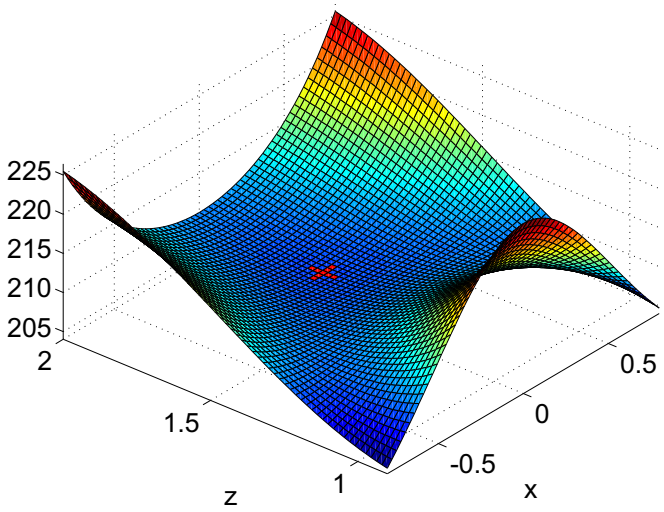


Figure 17: Potential field of the three-particle "bucket"

You can clearly see the build-up of the potential walls towards the three particles, you can also imagine a very shallow minimum at $x = 0$ and $z = 1.5872$ (marked by a red cross), and you can definitely understand how the ball could easily decide to travel through one of the lower side potential tunnels instead of rolling into the center equilibrium.

2.10 Demo 10: The Hose

How would you simulate water flowing out of a hosepipe? Well – what about an endless stream of water particles (droplets), the direction of which you can control via the mouse pointer? Sounds easy, but how can you solve the problem of the steadily increasing number of living particles the simulation environment has to maintain? Just let'em die! Every particle can be given a limited span of life. After that time span the particle system automatically deletes the particle and all springs and attractions connected to it. Since all particles are created during the simulation, the initialization block only has to create a particle system with gravity (`-10`) and drag (`1`)

```
Particle_System = particle_system ([0, 0, -10], 1, 1);
```

Right off the bat, the simulation loop can be started

```
step_time = 0.02;

for i = 1 : inf
```

Now, all you have to do in every single time step is to create a new particle at the origin (`[0, 0, 0]`), with a limited lifespan of `0.5` and an initial velocity (direction) that depends on the position of the current mouse pointer position (`4*current_point`). And yes, add a little bit of jitter to the initial velocity (`0.2*rand (1, 3)`); it makes the stream look more realistic

```
    current_point = mean (get (gca, 'currentpoint'));

    Particles{i} = particle (Particle_System, 1, [0, 0, 0], ...
    4*current_point + 0.2*rand (1, 3), false, 0.5);
```

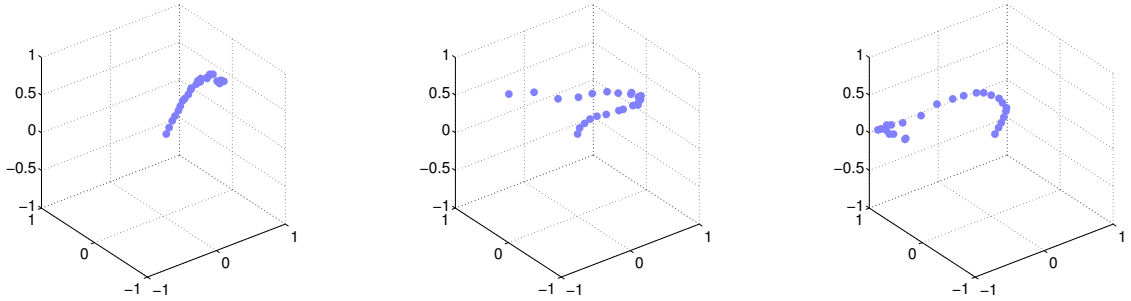
Transmogrify blood to water

```
    set (Particles{i}.graphics_handle, 'color' , [0.5, 0.5, 1])
```

and keep going

```
    Particle_System.advance_time (step_time);

end
```

Since the particle lifespan is limited to `0.5` and the simulation step time has a value of `0.02` all three snapshots in Figure 18 show some 25 particles.

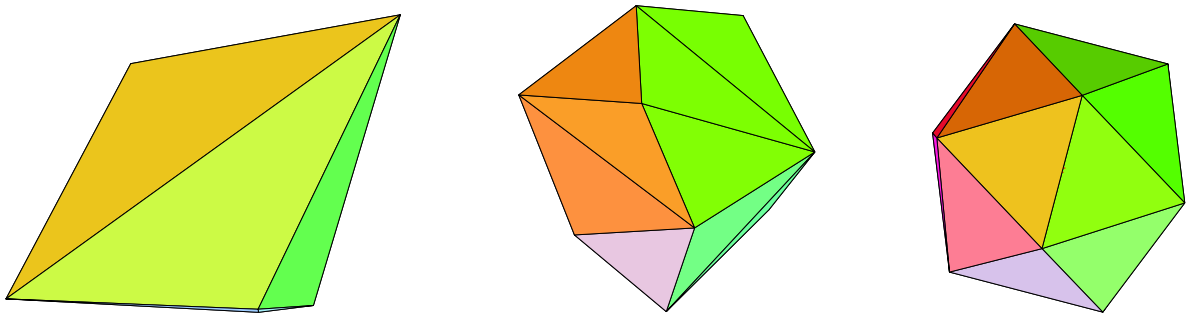


(a) $t = 2$ (b) $t = 4$ (c) $t = 6$

Figure 18: The hose

2.11 Demo 11: Polyhedrons

The classical polyhedron is a three-dimensional shape made up of planar polygons. The five Platonic solids (tetrahedron, cube, octahedron, dodecahedron, and icosahedron) are the only convex regular polyhedrons. Is there a chance to persuade particles in a particle system to arrange themselves as the vertices of Platonic solids? How would you have to define springs and attractions to simulate the self-inflation of the icosahedron depicted in Figure 19?



(a) $t = 0$ (b) $t = 2$ (c) $t = 20$

Figure 19: Self-configuration of an icosahedron

All 12 vertices of an icosahedron have an equal distance (the radius of the circumscribed sphere) to its center; this could be modeled by a fixed center particle and 12 springs (with equal rest lengths) between the center particle and the 12 free vertex particles. Next,

all vertices have to find their place on the sphere, equally distributed, with maximum mean distance to all neighbors. Therefore, repulsions between every single particle and all other particles could force every particle into its own minimum energy position. The particle system provides drag (1), but no gravity ([0, 0, 0]).

```
Particle_System = particle_system ([0, 0, 0], 1, 2);
```

Turn off all axis labeling, tick marks, and background

```
axis off;
```

The center particle should be fixed, to keep the polyhedron in the center of the axes

```
Particle_center = particle ...
(Particle_System, 1, [0, 0, 0], [0, 0, 0], true, inf);
```

Define the number of vertex particles

```
n_particles = 12;
```

and start a loop over all vertices to be created

```
for i = 1 : n_particles
```

All vertex particles have a random initial position (randn (1, 3)) and a unit mass

```
  Particle{i} = particle ...
  (Particle_System, 1, randn (1, 3), [0, 0, 0], false, inf);
```

Every vertex is connected to the center particle via a unit spring with a strength of 10

```
  Spring{i} = spring ...
  (Particle_System, Particle_center, Particle{i}, 1, 10, 1);

end
```

Repulsions (-1) with a little bit of damping (0.01) are defined between every vertex and every other vertex

```
for i = 1 : n_particles

  for j = 1 : n_particles

    if i < j

      Attraction{i, j} = attraction ...
      (Particle_System, Particle{i}, Particle{j}, -10, 0.01);

    end
```

```
    end

  end
```

The if-clause (`if i < j`) makes sure that there is only one repulsion between every vertex pair. Since the number of repulsions grows quadratic with the number of particles

$$n_{repulsions} = \frac{n_{particles}\,(n_{particles} - 1)}{2}$$

the step time should not be chosen too small

```
  step_time = 0.2;

  for i = 1 : inf
```

The next few lines of code produce the colored faces of the polyhedron. The basic idea is to chose the color depending on the orientation of the normal vector of the particular face. Thus, only adjacent parallel faces will have the same color. First, the current particle position vector has to be rearranged into a vertex matrix with three columns (x-, y-, and z-coordinates)

```
    particles_positions = ...
    Particle_System.get_particles_positions;

    positions = reshape (particle_positions, 3, n_particles + 1);

    vertices = positions';
```

The next – a little bit more difficult – task is to find that minimum number of "outer" faces that make up the convex hull surface of the polyhedron. Fortunately, MATLAB provides the `convhulln` command that can directly find the indices of the involved hull vertices

```
    faces = convhulln (vertices);

    n_faces = size (faces, 1);
```

Before the faces of the hull can be created, the handle vector, holding the corresponding patches, has to be initialized

```
    patch_handle = [];
```

This is necessary, because the number of patches can decrease during the simulation and in that case the leftover from the last simulation step would crash the `delete` command. The hull faces are created in a loop over all index rows the `convhulln` command has found

```
for i_faces = 1 : n_faces
```

Every row of the `faces` matrix contains the indices of three vertices that make up the triangular facet. Using these values as indices into the `vertices` matrix gives you the position vectors of all three facet vertices

```
vertex_1 = vertices(faces(i_faces, 1), :);
vertex_2 = vertices(faces(i_faces, 2), :);
vertex_3 = vertices(faces(i_faces, 3), :);
```

The difference between two vertex position vectors describes an edge vector of the facet

```
edge_1 = vertex_1 - vertex_2;
edge_2 = vertex_1 - vertex_3;
```

and the cross product of two edge vectors defines the vector standing perpendicular on the plane spanned by the edge vectors

```
normal = cross (edge_1, edge_2);
```

In order to use the normal vector as an RGB color vector, it has to be scaled, until its components are in the $0 \ldots 1$ domain

```
face_color = 0.5*(normal/norm (normal) + 1);
```

Now the patch can be created using the recently computed vertices, color, and a transparency (`facealpha`) of 90%

```
patch_handle(i_faces) = patch ...
('vertices', vertices(faces(i_faces, :), :), ...
'faces', [1 2 3], ...
'facecolor', face_color, ...
'facealpha', 0.9);

end
```

The drawnow command makes sure that the patch is drawn immediately after its creation

```
drawnow
```

After the computation of the next simulation step and before the creation of a new polyhedron, the "old one" has to be deleted

```
    Particle_System.advance_time (step_time);

    delete (patch_handle)

  end
```

2.11.1 Number of Particles: 3

You cannot really create a true regular convex 3-dimensional polyhedron from just three vertex particles. If you try it

```
n_particles = 3;
```

the first few simulation steps actually show a 3-D object, since the initial positions of the particles are chosen at random; but soon the springs and repulsions force the particles to form an equilateral triangle. Unfortunately, triangles do not have a volume; thus the `convhulln` command runs into a problem, as the particles arrange themselves more and more in one single plane. After a few warnings about a "`narrow hull`", `convhulln` stops the simulation with the understandable error message

```
Qhull could not construct a clearly convex simplex
```

2.11.2 Number of Particles: 4

The Platonic solid with four vertices is called a regular tetrahedron. Obviously, the particles are forced into a configuration of the four (Greek: tetra-) equivalent equilateral triangular faces depicted in Figure 20.

2.11.3 Number of Particles: 5

There is no Platonic solid with five vertices. Nevertheless, the corresponding simulation creates the highly symmetrical polyhedron in Figure 21. It is called a triangular dipyramid, because it can be "constructed" by "gluing" two (Greek: di-) triangular pyramids (tetrahedrons) base-to-base.

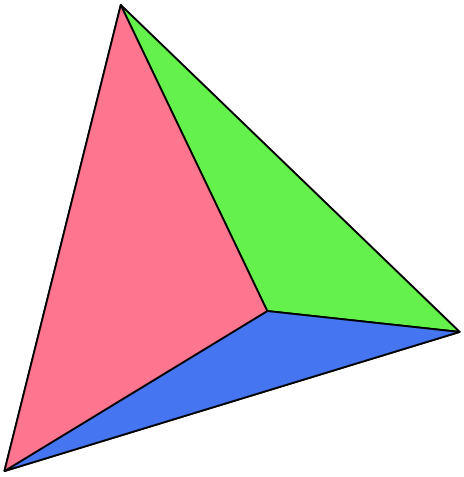


Figure 20: Tetrahedron ($n_{particles} = 4$)

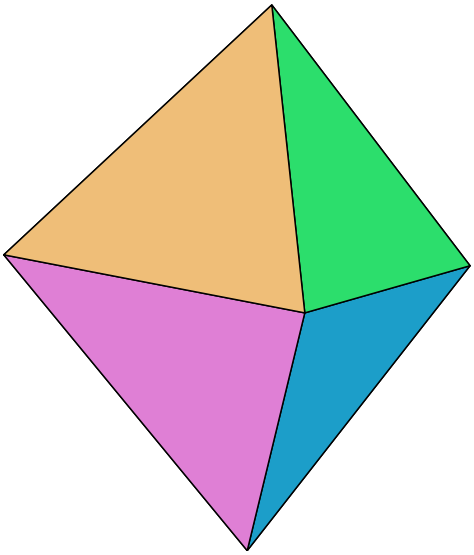


Figure 21: Triangular Dipyramid ($n_{particles} = 5$)

2.11.4 Number of Particles: 6

The six free particles of this simulation inflate into a regular octahedron (Figure 22), which is one of the Platonic solids. The octahedron (Greek for "eight faces") is also called a square dipyramid because it can be dissected into two pyramids with square bases. Seen from another angle of view, the octahedron also is a triangular antiprism, because there are (four) pairs of opposing parallel triangles, of which one has its vertices where the other one has its edges.

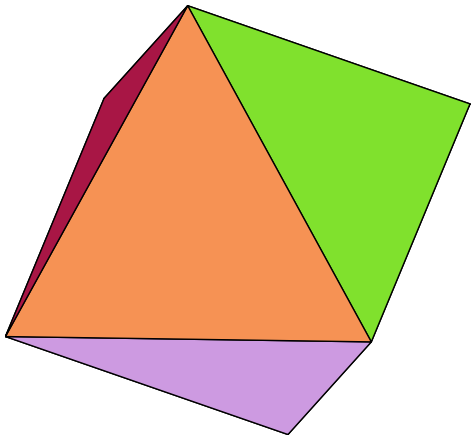


Figure 22: Octahedron, square dipyramid, triangular antiprism ($n_{particles} = 6$)

2.11.5 Number of Particles: 8

Yes – there is a Platonic solid with eight vertices; the regular hexahedron, aka the cube. But no – interestingly, this polyhedron simulation does not create a cube, if you ask it to arrange eight particles in a minimum energy state. The astonishing result is the square antiprism shown in Figure 23, which consists of two squares[18] (a "bottom square" and a "top square") that have been rotated (45°) into paraphase and eight triangles connecting the edges of the bottom square with the vertices of the top square and vice versa[19]. Obviously, this "twisting" of the squares leads to a stable equilibrium of the contradicting forces of the springs and repulsions, while the cube, where the vertices of the squares "face each other directly", produces an indifferent equilibrium that will never be reached in a simulation with random initial positions.

2.11.6 Number of Particles: 12

In a simulation with 12 particles, all of them find their positions at the vertices of the regular icosahedron (Greek for "twenty faces") shown in Figure 24 (and Figure 19), which is another one of the Platonic solids.

[18] You can easily identify the squares as two adjacent triangles of the same color (same plane normal).
[19] You have to try out this simulation! It is really cute to watch the particles swarm around, until they have come to an agreement on which of them makes up the bottom and top squares.

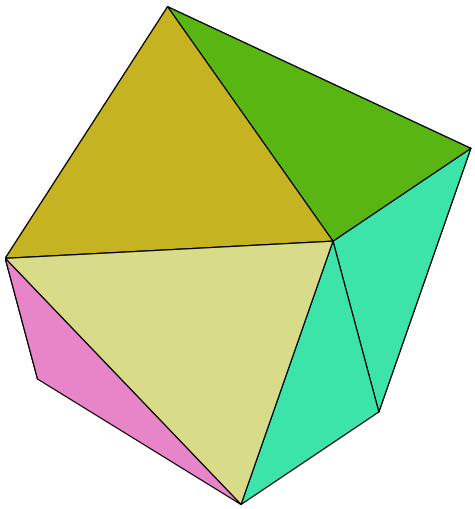


Figure 23: Square antiprism (not a cube!) ($n_{particles} = 8$)

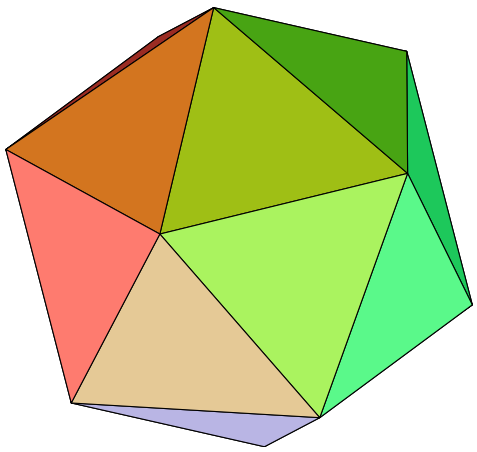


Figure 24: Icosahedron ($n_{particles} = 12$)

2.11.7 Number of Particles: 20

The Platonic solid with 20 vertices and 12 (Greek: dodeca-) faces is called a regular dodecahedron. But again (as with the square antiprism) the 20 particles minimum energy configuration depicted in Figure 25 is not a dodecahedron. The author has not even found a proper name for that irregular polyhedron. At first glance it seems to consist of triangles and three squares (adjacent triangles of the same color). But if you look more carefully at the "squares" you can discover that the colors of the adjacent triangles do not have the exact same color. Strange . . . !

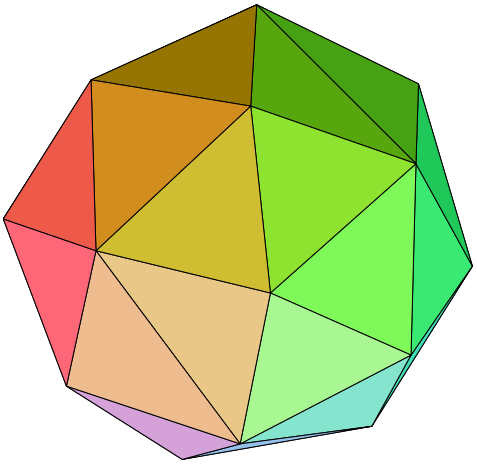
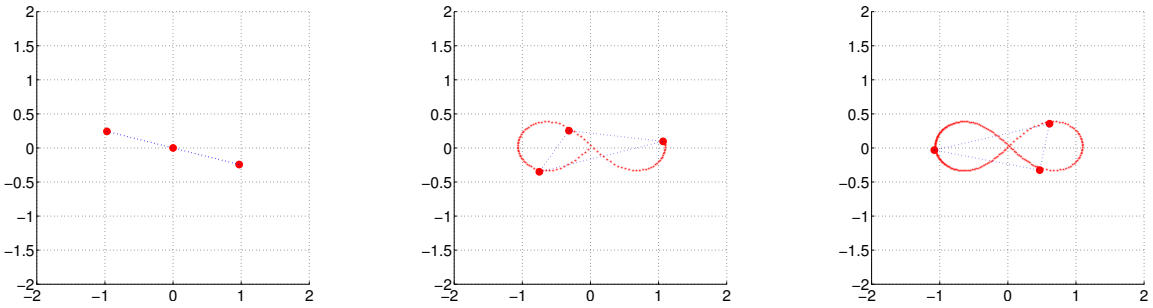


Figure 25: Not a dodecahedron! ($n_{particles} = 20$)

2.12 Demo 12: Three-Body Eight

In the year 2000, Alain Chenciner and Richard Montgomery published *A Remarkable Periodic Solution of the Three-Body Problem in the Case of Equal Masses* [14]. They showed that three particles – initialized with proper positions and velocities – chase each other around the fixed eight-shaped curve depicted in Figure 26.



(a) $t = 0$ (b) $t = 5$ (c) $t = 10$

Figure 26: Three-body eight

The particle system is 2-D, no-gravity, no-drag, but king-size axes (2)

```
Particle_System = particle_system ([0, 0, 0], 0, 2);

view (0, 0);
```

The initial positions and velocities are chosen according to [14]

```
initial_position = [0.97000436, 0, -0.2430875];
initial_velocity = [-0.93240737, 0, -0.86473146];
```

Three particles

```
Particle_1 = particle ...
(Particle_System, 1, initial_position , ...
-initial_velocity/2, false, inf);

Particle_2 = particle ...
(Particle_System, 1, -initial_position, ...
-initial_velocity/2, false, inf);

Particle_3 = particle ...
(Particle_System, 1, [0, 0, 0], ...
initial_velocity, false, inf);
```

and standard attractions between all particles

```
Attraction_1 = attraction ...
(Particle_System, Particle_1, Particle_2, 1, eps);

Attraction_2 = attraction ...
(Particle_System, Particle_1, Particle_3, 1, eps);

Attraction_3 = attraction ...
(Particle_System, Particle_2, Particle_3, 1, eps);
```

are all we need. During the simulation

```
step_time = 0.05;

for i = 1 : inf

  Particle_System.advance_time (step_time);
```

a quick-and-dirty path trace is drawn by plotting an asterisk character at the current position of one of the particles. Since every new asterisk has to be managed as an object in the axes, the simulation might slow down after a while and you might want to comment out the following two lines of code on slow machines

```
  pp = num2cell (Particle_3.position);
  text (pp{:}, '*', 'color', 'r')

end
```

3 Mathematical Background

The motion of a particle in Figure 27 can be described by three motion vectors

- $\boldsymbol{p}$: Position vector of the particle
- $\boldsymbol{v}$: Velocity vector of the particle
- $\boldsymbol{a}$: Acceleration vector of the particle

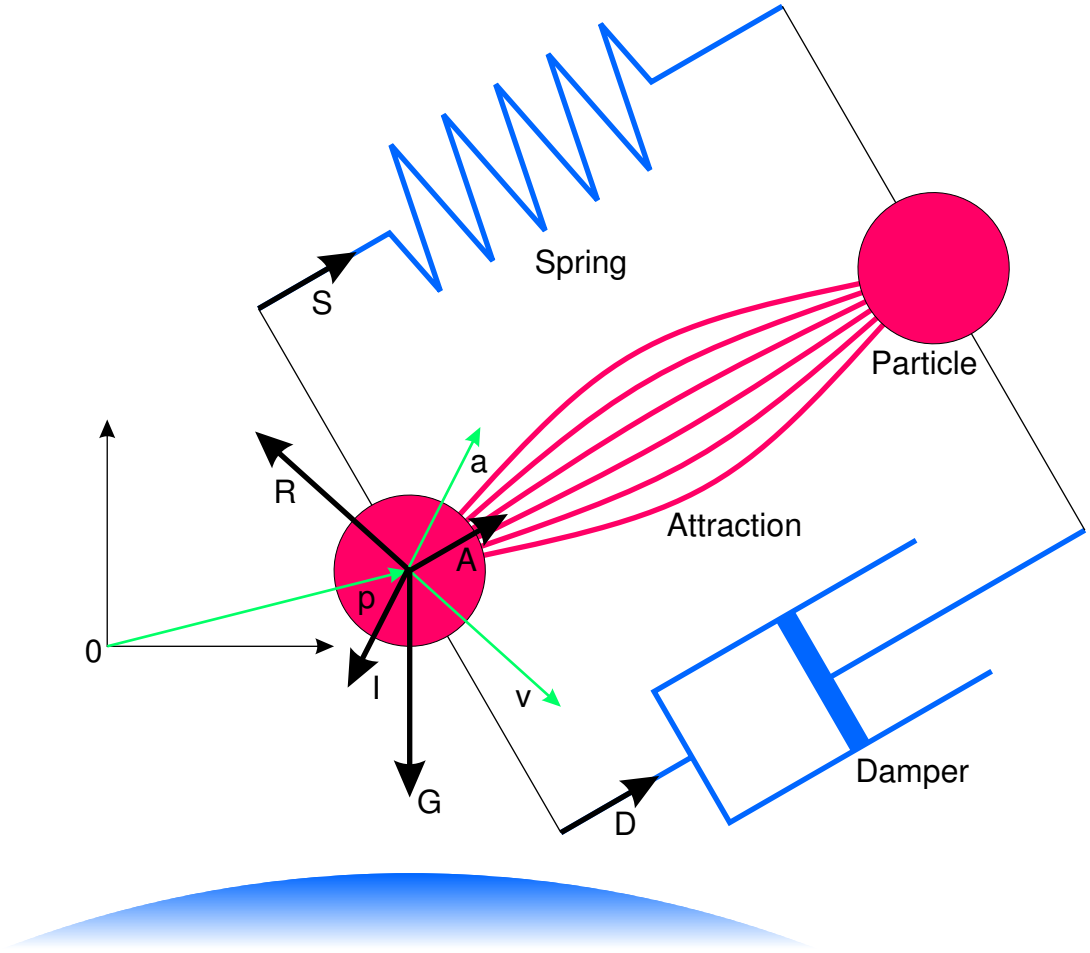


Figure 27: Force and motion vectors

While the position and the velocity vectors describe the (energetic) state of the particle (these two vectors build up the particle state vector), the acceleration vector results from the vectorial sum of all external force vectors acting on the particle (Newton's second law of motion). Seen from the view of d'Alembert's principle, the forces in Figure 27 are

- $\boldsymbol{G}$: Gravity (force) vector, modeling the mass attracting force in a gravitational field
- $\boldsymbol{R}$: Resistance force vector (drag), modeling the effect that surface-tainted objects "lose" energy while moving under atmospheric conditions
- $\boldsymbol{I}$: Inertial force vector, d'Alembert's fictitious acceleration defying force
- $\boldsymbol{A}$: Attraction force vector, resulting from an attraction or repulsion between two particles
- $\boldsymbol{S}$: Spring force vector, resulting from a spring between two particles, that is not at its rest length
- $\boldsymbol{D}$: Damping force vector,[20] resulting from a (realistic) spring, that dissipates energy while being stretched or compressed

Utilizing d'Alembert's Inertial force, the sum of all forces produces a dynamic equilibrium

$$\boldsymbol{G} + \boldsymbol{R} + \boldsymbol{I} + \boldsymbol{A} + \boldsymbol{S} + \boldsymbol{D} = \boldsymbol{0} \tag{2}$$

3.1 May the Force be with you...

This section describes every single force in more detail.

3.1.1 Gravity

In general, gravity refers to an attractive force that all massive objects exert on each other. In the simplified context of this toolbox all particles are attracted by a huge mass (the earth or another planet) that produces a field with parallel lines of flux and a force vector $\boldsymbol{G}$, that does not depend on the position, but only on the mass m of the particle and the gravitational acceleration vector $\boldsymbol{g}$

$$\boldsymbol{G} = m\boldsymbol{g} \tag{3}$$

The acceleration due to gravity $\boldsymbol{g}$ is a constant property of the particle system object, that cannot only be used to model conventional "all particles fall *down*" environments $\boldsymbol{g}_1$

$$\boldsymbol{g}_1 = \begin{bmatrix} 0 \\ 0 \\ -9.81 \end{bmatrix} \qquad \boldsymbol{g}_2 = \begin{bmatrix} 0 \\ 0 \\ 42 \end{bmatrix} \qquad \boldsymbol{g}_3 = \begin{bmatrix} 12.34 \\ 5.67 \\ 89.0 \end{bmatrix} \qquad \boldsymbol{g}_4 = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

but also those with "negative"[21] gravity $\boldsymbol{g}_2$, arbitrary gravity $\boldsymbol{g}_3$, and zero gravity $\boldsymbol{g}_4$.

[20] Please note, that the spring and the damping force vectors have been displaced for graphical reasons only; in reality, both of them act on the center of gravity and do not produce a torque.

[21] Negative gravity moves objects up, i. e. in positive z-direction.

3.1.2 Drag

This is all about fluid friction, viscous resistance, and air drag. In general, the force of drag $\boldsymbol{R}$ experienced by an object moving through a fluid can be expressed by

$$\boldsymbol{R} = \sum_{i=0}^{n} -r_i \left|\boldsymbol{v}\right|^i \boldsymbol{v} = \underbrace{-r_0 \boldsymbol{v}}_{\text{Viscous resistance}} \quad \underbrace{-r_1 \left|\boldsymbol{v}\right| \boldsymbol{v}}_{\text{Air drag}} \quad -r_2 \left|\boldsymbol{v}\right|^2 \boldsymbol{v} \quad - \ldots \tag{4}$$

where r_i are drag coefficients depending on the size of the particle, the fluid properties, and the velocity domain (slow... fast), $\boldsymbol{v}$ denotes the velocity vector of the particle and $|\boldsymbol{v}|$ its scalar norm or magnitude.

If a small particle moves in a fluid at low speed, only the first term in Equation 4 is significant;[22] the drag linearly depends on the particle velocity

$$\boldsymbol{R} = -r\boldsymbol{v} \tag{5}$$

The negative sign indicates, that the drag vector acts in the opposite direction of the particle velocity vector (s. Figure 27), slowing down the particle. The constant scalar drag coefficient r is a property of the particle system object.

3.1.3 Inertial Force

Utilizing d'Alembert's principle, an inertial force is introduced, that could be interpreted as a "resistance of a mass against acceleration"

$$\boldsymbol{I} = -m\boldsymbol{a} \tag{6}$$

Similar to the drag equation (Equation 5), the opposing directions of inertial force $\boldsymbol{I}$ and acceleration $\boldsymbol{a}$ are expressed in Figure 27, as well as by the negative sign in Equation 6.

3.1.4 Attraction Force

As indicated in Figure 28, an attraction force $\boldsymbol{A}$ between two particles could e. g. represent the gravitational force between two masses, the electrostatic force between two electric charges, or the magnetic force between two dipoles.

[22] In this toolbox, drag is modeled according to Equation 5. Nevertheless, it is quite straightforward to implement further terms of Equation 4 (e. g. air drag) in the corresponding `aggregate_drag_forces` method.

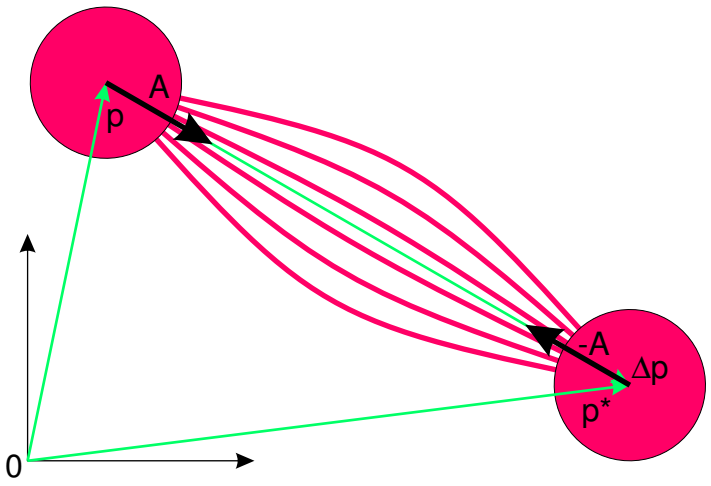


Figure 28: Attraction force $\boldsymbol{A}$ between two particles

For the computation of the attraction force vector, we first have to express the particle distance vector $\Delta\boldsymbol{p}$ as the difference between the position vectors $\boldsymbol{p}$ and $\boldsymbol{p}^*$ of both particles

$$\Delta\boldsymbol{p} = \boldsymbol{p}^* - \boldsymbol{p} \tag{7}$$

Next – e. g. in the gravitational case – the norm $|\boldsymbol{A}|$ of the attraction force vector linearly depends on each of the particle masses (m and m^*) and quadratically on the reciprocal of the scalar distance between the particles

$$|\boldsymbol{A}| = \gamma \frac{m \cdot m^*}{|\Delta\boldsymbol{p}|^2} \tag{8}$$

Additionally, the attraction strength factor γ – which would be the gravitational constant $G = 6.6742 \cdot 10^{-11}\,\mathrm{N\,m^2\,kg^{-2}}$ in a real-world gravity environment – can be chosen arbitrarily as a property of the attraction object. Finally, the attraction vector results from the product of its magnitude (Equation 8) and its normalized (unit) direction vector (Equation 7)

$$\boldsymbol{A} = |\boldsymbol{A}| \cdot \frac{\Delta\boldsymbol{p}}{|\Delta\boldsymbol{p}|} = \gamma \frac{mm^*}{|\Delta\boldsymbol{p}|^2} \frac{\Delta\boldsymbol{p}}{|\Delta\boldsymbol{p}|} = \gamma \frac{mm^*}{|\Delta\boldsymbol{p}|^3} \Delta\boldsymbol{p} \tag{9}$$

While $\boldsymbol{A}$ is the force vector accelerating the first particle towards the second one in Figure 28, according to Newton's third law ("action equals reaction"), an equal force vector with opposing direction ($-\boldsymbol{A}$) accelerates the second particle towards the first one. Attractions can easily be turned into repulsions by utilizing a negative strength factor γ.

3.1.5 Spring Force

According to Figure 29 a spring between two particles causes a force depending on the particle distance.

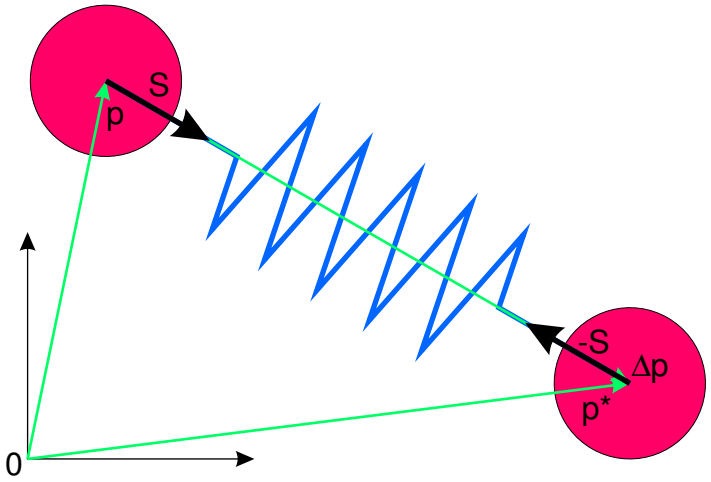


Figure 29: Spring force $\boldsymbol{S}$ between two particles

With the particle distance vector being calculated according to Equation 7, the magnitude[23] of the spring force is proportional to the deviation of the scalar distance from the spring rest length l_r

$$\dagger \boldsymbol{S} \dagger = c\left(|\Delta \boldsymbol{p}| - l_r\right) \tag{10}$$

where c is the spring (strength) constant, which defines the stiffness of the spring and which – just like the rest length – is a property of the spring object. Again, combining the magnitude (and orientation) of Equation 10 with the normalized vector direction of Equation 7 results in the spring force vector

$$\boldsymbol{S} = \dagger \boldsymbol{S} \dagger \cdot \frac{\Delta \boldsymbol{p}}{|\Delta \boldsymbol{p}|} = c\left(|\Delta \boldsymbol{p}| - l_r\right) \frac{\Delta \boldsymbol{p}}{|\Delta \boldsymbol{p}|} \tag{11}$$

acting on both particles.

3.1.6 Damping Force

An ideal spring force does only depend on the length of the spring. A more realistic spring model might also take into account the velocity depending damping forces, that

[23] Strictly spoken, $\dagger \boldsymbol{S} \dagger$ does not only include the magnitude (norm) of the spring force vector but also its orientation (i.e., its sign). $\dagger \boldsymbol{S} \dagger$ is positive (drawing the particles towards each other), if the particle distance is greater than the rest length of the spring; but it can also become negative, if the spring is compressed beyond its rest length.

transform kinetic energy into dissipation energy. Therefore, in this toolbox, every spring object is accompanied by a damper (Figure 30).

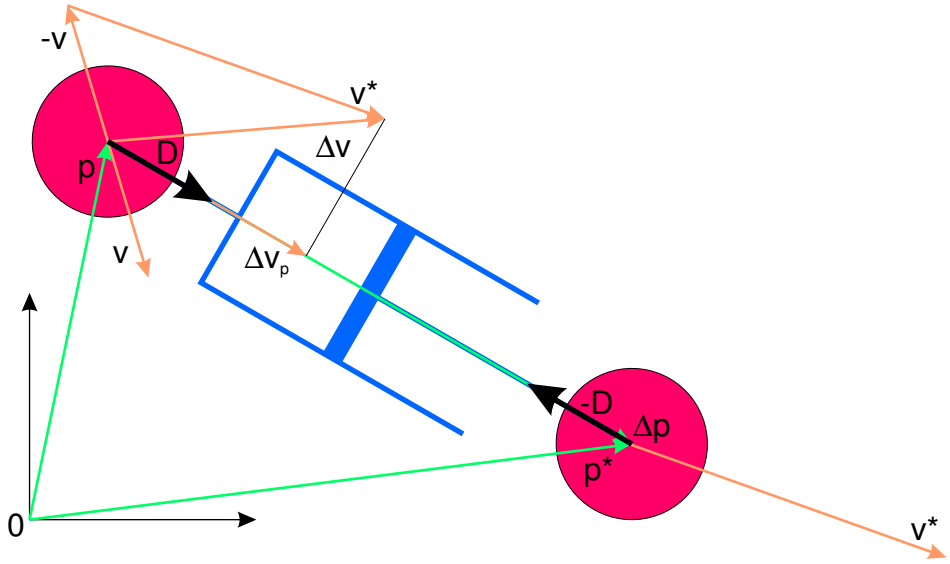


Figure 30: Damping force $\boldsymbol{D}$ between two particles

As a first approximation, the damping force linearly depends on the velocity the damper is elongated or compressed with. If $\boldsymbol{v}$ and $\boldsymbol{v}^*$ denote the velocities vectors of the first and second particle, respectively, $\Delta\boldsymbol{v}$ is the relative[24] velocity between both particles

$$\Delta\boldsymbol{v} = \boldsymbol{v}^* - \boldsymbol{v} \tag{12}$$

Since the damping force does only[25] depend on the velocity component along the moving axis of the damper, we have to compute the projection[26] vector $\Delta\boldsymbol{v}_p$ of the relative velocity vector $\Delta\boldsymbol{v}$ along the damper axis direction, which is defined by the distance vector $\Delta\boldsymbol{p}$ between the particles

$$\Delta\boldsymbol{v}_p = \left(\frac{\Delta\boldsymbol{p}}{|\Delta\boldsymbol{p}|}\Delta\boldsymbol{v}\right)\frac{\Delta\boldsymbol{p}}{|\Delta\boldsymbol{p}|} \tag{13}$$

Finally, the damping force vector $\boldsymbol{D}$ can be computed via

$$\boldsymbol{D} = d \cdot \Delta\boldsymbol{v}_p = d\left(\frac{\Delta\boldsymbol{p}}{|\Delta\boldsymbol{p}|}\Delta\boldsymbol{v}\right)\frac{\Delta\boldsymbol{p}}{|\Delta\boldsymbol{p}|} \tag{14}$$

where d is the constant damping property of the spring object.

[24] To be precise, $\Delta\boldsymbol{v}$ is the velocity of the second particle with respect to the first particle.

[25] Imagine the first particle being fixed and the other particle moving about the first one on a circular orbit. There would be a relative velocity vector $\Delta\boldsymbol{v}$ but no damping force.

[26] The projection of one vector along another vector is derived in Appendix A.

3.2 Differential Equations

Using all final force expressions, Equation 2 can now be expanded

$$m\boldsymbol{g} - r\boldsymbol{v} - m\boldsymbol{a} + \gamma\frac{mm^*}{|\Delta\boldsymbol{p}|^3}\Delta\boldsymbol{p} + c\left(|\Delta\boldsymbol{p}| - l_r\right)\frac{\Delta\boldsymbol{p}}{|\Delta\boldsymbol{p}|} + d\left(\frac{\Delta\boldsymbol{p}}{|\Delta\boldsymbol{p}|}\Delta\boldsymbol{v}\right)\frac{\Delta\boldsymbol{p}}{|\Delta\boldsymbol{p}|} = \boldsymbol{0} \tag{15}$$

Equation 15 describes a vectorial nonlinear second order system with three (translatory) degrees of freedom. This becomes obvious, if we recall that the current state of a single particle is defined by two three-dimensional state vectors: the position vector $\boldsymbol{p}$ and the velocity vector $\boldsymbol{v}$. In order to simulate the motion of a particle, we now have to lay down the differential equation system of that particle. Every state vector has its own (vectorial) first order differential equation. The position differential equation is merely the definition of the velocity, being the time derivative of the position

$$\dot{\boldsymbol{p}} = \boldsymbol{v} \tag{16}$$

To come up with the velocity differential equation, we use the fact that the acceleration is the time derivative of the velocity ($\boldsymbol{a} = \dot{\boldsymbol{v}}$) for the substitution of the acceleration in Equation 15

$$m\boldsymbol{g} - r\boldsymbol{v} - m\dot{\boldsymbol{v}} + \gamma\frac{mm^*}{|\Delta\boldsymbol{p}|^3}\Delta\boldsymbol{p} + c\left(|\Delta\boldsymbol{p}| - l_r\right)\frac{\Delta\boldsymbol{p}}{|\Delta\boldsymbol{p}|} + d\left(\frac{\Delta\boldsymbol{p}}{|\Delta\boldsymbol{p}|}\Delta\boldsymbol{v}\right)\frac{\Delta\boldsymbol{p}}{|\Delta\boldsymbol{p}|} = \boldsymbol{0} \tag{17}$$

Integration algorithms want first order differential equations with the derivative on the left hand side, while the right hand side of the equation has to be programmed in a function. Therefore we have to solve Equation 17 for the derivative of the velocity

$$\dot{\boldsymbol{v}} = \boldsymbol{g} - \frac{r}{m}\boldsymbol{v} + \gamma\frac{m^*}{|\Delta\boldsymbol{p}|^3}\Delta\boldsymbol{p} + \frac{c}{m}\left(|\Delta\boldsymbol{p}| - l_r\right)\frac{\Delta\boldsymbol{p}}{|\Delta\boldsymbol{p}|} + \frac{d}{m}\left(\frac{\Delta\boldsymbol{p}}{|\Delta\boldsymbol{p}|}\Delta\boldsymbol{v}\right)\frac{\Delta\boldsymbol{p}}{|\Delta\boldsymbol{p}|} \tag{18}$$

The last step is to assemble both states ($\boldsymbol{p}$ and $\boldsymbol{v}$) into the phase space state vector $\boldsymbol{x}$

$$\boldsymbol{x} = \begin{bmatrix}\boldsymbol{p}\\ \boldsymbol{v}\end{bmatrix} \tag{19}$$

and to use Equation 16 and Equation 18 to depict the general nonlinear differential equation system $\dot{\boldsymbol{x}} = \boldsymbol{f}\left(\boldsymbol{x}\right)$

$$\begin{bmatrix}\dot{\boldsymbol{p}}\\ \dot{\boldsymbol{v}}\end{bmatrix} = \begin{bmatrix}\boldsymbol{v}\\ \boldsymbol{g} - \frac{r}{m}\boldsymbol{v} + \gamma\frac{m^*}{|\Delta\boldsymbol{p}|^3}\Delta\boldsymbol{p} + \frac{c}{m}\left(|\Delta\boldsymbol{p}| - l_r\right)\frac{\Delta\boldsymbol{p}}{|\Delta\boldsymbol{p}|} + \frac{d}{m}\left(\frac{\Delta\boldsymbol{p}}{|\Delta\boldsymbol{p}|}\Delta\boldsymbol{v}\right)\frac{\Delta\boldsymbol{p}}{|\Delta\boldsymbol{p}|}\end{bmatrix} \tag{20}$$

ready to be simulated by an appropriate integration algorithm.

4 Particle System Object

Just like the mathematical background, the 7 public (user accessible) and 13 private (internal) methods – programmatically described in the following sections – give you detailed information on how the objects of this toolbox interact; this knowledge is not essential for the use of the toolbox, but might help you to adapt it to your own needs.

4.1 Class Definition particle_system

As of R2008a, MATLAB classes can subclass (i. e. inherit from) a new abstract class called *handle class*, finally allowing true aggregation, where only a reference (a handle or a pointer) to the object is aggregated in another object and the original object can still be addressed and modified. The MATLAB documentation elaborates on this concept:

> Objects of handle classes use a handle to reference objects of the class. A handle is a variable that identifies a particular instance of a class. When a handle object is copied, the handle is copied but not the data stored in the object's properties. The copy refers to the same data as the original - if you change a property value on the original object, the copied object reflects the same change.

All objects in the version 2.0 of this toolbox are derived from the handle class, significantly increasing the ease of aggregation and administration of particles, springs, and attractions in the particle system. The corresponding definition of the particle system class encapsulates the properties and methods described in the following sections

```
classdef particle_system < handle
  properties
    ...
  end
  methods
    ...
  end
end
```

4.2 Properties particle_system

The property definition block numerates all properties of the particle system object, including the environmental attributes `gravity` and `drag`, the arrays `particles`, `springs`,

and `attractions` that will later on host the homonymous objects, and the graphics handle of the dedicated figure. The `time` property holds the current simulation time; it is updated in every simulation time step and is directly initialized in the property block

```
properties

  gravity
  drag
  particles
  springs
  attractions
  time = 0
  graphics_handle

end
```

4.3 Constructor particle_system

With a call to `particle_system` you create a particle system with gravity, drag, and axes limits

```
function Particle_System = particle_system ...
(gravity, drag, limits)
```

If you do not supply environmental parameters with your function call, the function makes up its own default parameters

```
  if nargin == 0

    gravity = [0, 0, 0];
    drag = 0;
    limits = 1;

  end
```

Properties of MATLAB objects are stored in structure array fields

```
  Particle_System.gravity = gravity;
  Particle_System.drag = drag;
```

The `limits` parameter is not saved as a particle system property; it is only used once to initialize the axes. If you subsequently want to alter the limits (or any other parameter) of the axes, you can directly address the axes via `gca` or as a child of `graphics_handle`.

Open a new empty figure window and save the figure handle

```
    Particle_System.graphics_handle = figure;
```

The next few lines of code are not really necessary, but set some useful graphical default options. You might e. g. want to render the graphical object representations with OpenGL, in order to see semi-transparent surfaces

```
    set (Particle_System.graphics_handle, 'renderer', 'opengl')
```

You can always switch over to other renderers, but presently OpenGL seems to give the best "cost/performance ratio". Unfortunately, MATLAB does not provide a native way to maximize a window and all tools found on [11] have their own problems and restrictions. One more or less elegant way to "maximize" the figure window could be

```
    screen_size = get (0, 'screensize');

    set (Particle_System.graphics_handle, ...
    'position', screen_size + [20 40 -40 -120]);
```

Next, you might want to create an axes, initialize its limits to the user supplied values, make circles and spheres appear as circles and spheres and display a grid

```
    axis ([-limits limits -limits limits -limits limits]);
    axis equal

    grid on
```

Using the mouse (with a pressed left mouse button) to rotate the axes might be a good idea for most 3D applications. If not – just switch it off in your application or comment this line per default

```
    rotate3d

end
```

4.4 Method get_particles_positions

The `get_particles_positions` method of the particle system provides a convenient user accessible shortcut for the retrieval of all current particle positions

```
function particles_positions = get_particles_positions ...
(Particle_System)
```

The positions of all particles are returned in one long row vector that should be initialized for performance reasons

```
    n_particles = length (Particle_System.particles);

    particles_positions = zeros (1, 3*n_particles);
```

In a loop over all particles in the particle system

```
    for i_particle = 1 : n_particles
```

the current particle ist retrieved and its position property vector is collected in the `particles_positions` vector.

```
      Particle = Particle_System.particles(i_particle);

      particles_positions (3*i_particle - 2 : 3*i_particle) = ...
      Particle.position;

    end

  end
```

4.5 Method get_particles_velocities

The `get_particles_velocities` method retrieves the current velocities of all particles

```
  function particles_velocities = get_particles_velocities ...
  (Particle_System)
```

It is very similar to the `get_particles_positions` method; After the initialization of the `particles_velocities` vector, a single particle is retrieved in a loop over all particles

```
    n_particles = length (Particle_System.particles);

    particles_velocities = zeros (1, 3*n_particles);

    for i_particle = 1 : n_particles

      Particle = Particle_System.particles(i_particle);
```

If the particle has been flagged as fixed, its velocity is explicitly set to zero

```
      if Particle.fixed

        velocity = [0, 0, 0];

      else

        velocity = Particle.velocity;

      end
```

and finally the velocity vector of the current particle is stored in the velocity vector of all particles

```
      particles_velocities (3*i_particle - 2 : 3*i_particle) = ...
      velocity;

    end

  end
```

4.6 Methods kill_spring, kill_attraction

Sometimes you want to delete objects (particles, springs, and attractions) during simulation. Think e. g. of a particle that hits a wall and is absorbed by that wall. As soon as it touches the wall, the particle itself and all springs and attractions attached to that particle have to be deleted. The user addresses a spring to be deleted in the particle system method `kill_spring` directly via its name

```
  function kill_spring (Particle_System, Spring)
```

In order to delete the spring object embedded in the particle system we first have to find its array index

```
    index = Particle_System.springs == Spring;
```

Hereupon the spring object can be removed from the particle system

```
    Particle_System.springs(index) = [];
```

Additionally, the delete method of the spring object is called in order to delete the graphical representation of the spring

```
    Spring.delete;

  end
```

The "killing of an attraction" looks like an identical twin - no need to go into details ...

4.7 Method kill_particle

The complete deletion of a particle

```
function kill_particle (Particle_System, Particle)
```

is quite a bit more complicated because we do not only have to delete the particle itself but also all springs and attractions attached to the particle. At first, we want to delete all attractions attached to the particle. After initializing a kill buffer[27]

```
attractions_to_be_killed = [];
```

we start a loop over all attractions in the particle system

```
for i_attraction = ...
1 : length (Particle_System.attractions)
```

In that loop we retrieve a copy of the current attraction

```
Attraction = Particle_System.attractions(i_attraction);
```

and determine the particles at both ends of that attraction

```
Particle_1 = Attraction.particle_1;
Particle_2 = Attraction.particle_2;
```

If the particle to be deleted is found at either end of the current attraction

```
if Particle == Particle_1 || Particle == Particle_2
```

the attraction is appended to the kill buffer, marking it for future deletion

```
  attractions_to_be_killed = ...
  [attractions_to_be_killed, Attraction];

end

end
```

Now it is time to actually delete the attractions accumulated in the kill buffer. In a loop over all attractions to be deleted

[27] The kill buffer is necessary because you do not want to delete an attraction immediately, while you are still going through all attractions in a `for` loop. Deleting an attraction causes a reindexing of all successors of that attraction, thus leading to potential index mismatch problems in the course of the loop.

```
    for i_attraction = attractions_to_be_killed
```

the already discussed kill_attraction method of the particle system is used to delete every single attraction marked for demolition

```
      Particle_System.kill_attraction (i_attraction);

    end
```

Again, the analogous program section performing the deletion of all springs attached to the particle to be deleted is not discussed here.

Finally, the destruction of the particle itself utilizes the same steps of "find the index", "delete the object", and "delete the graphics object" already described in detail in kill_spring

```
    index = Particle_System.particles == Particle;

    Particle_System.particles(index) = [];

    Particle.delete;

  end
```

4.8 Method advance_time

An interactive real-time simulation environment should provide full user control over every single simulation step. Therefore, in the described particle system toolbox the user calls the `advance_time` method

```
  function advance_time (Particle_System, step_time)
```

in every cycle of the simulation loop, defining the step time of the current simulation step. In order to avoid unnecessary computations, the first task of `advance_time` is to delete all particles that have reached the end of their lifetime

```
    Particle_System.kill_old_particles;
```

The current simulation time can be found as a property of the particle system

```
    time_start = Particle_System.time;
```

and is used to determine the time at the end the simulation step to be carried out

```
    time_end = time_start + step_time;
```

The phase space state vector describes the (potential and kinetic) energy state of the particle system. Therefore, the state vector consists of the positions (p_i) and velocities (v_i) of all n particles arranged in one long row vector

$$\begin{bmatrix} p_1 & p_2 & \cdots & p_n & v_1 & v_2 & \cdots & v_n \end{bmatrix}$$

```
phase_space_state = Particle_System.get_phase_space_state;
```

Now we have to integrate the differential equation system. Since all of MATLAB's build-in differential equation solvers are highly sophisticated variable-step solvers that cannot easily and efficiently be downgraded to meet real-time requirements, a simple fixed-step fourth-order Runge-Kutta solver from [12] is used[28]

```
phase_space_states = ode4 (...
@compute_state_derivative, ...
[time_start, time_end], ...
phase_space_state, ...
Particle_System);
```

Since `ode4` is not a method of the particle system, it has been implemented as a subfunction of the `advance_time` method. The call to the `ode4` solver allows four parameters

- `@compute_state_derivative` is the handle to the function that computes the current state vector derivative (the right hand side of the differential equation system) from the current state vector.
- `[time_start, time_end]` defines the beginning and the end of the time interval the algorithm has to integrate through.[29]
- `phase_space_state` is the (initial condition of the) state vector at the beginning of the current time step.
- `Particle_System` is an additional parameter that is directly channeled through by `ode4` to `compute_state_derivative`.

After a successful integration step, `ode4` returns a state matrix, each row representing the state vector at one of the specified points of time. Since we already know the state vector at `time_start` (corresponding to the first row), we are only interested in the second row, featuring the state vector at the end of the time step

```
phase_space_state = phase_space_states(2,:);
```

[28] Runge-Kutta integration is so common use (see e. g. [8]), that it seems not to be necessary to describe the algorithm in detail.

[29] Even though `ode4` could also compute the state vector for more than one time step at once (using a time span vector with more than two elements), this real-time application needs to do one single integration step after the other.

The new current state vector

```
Particle_System.set_phase_space_state (phase_space_state);
```

and the new current time can now be rewritten into the particle system object

```
Particle_System.time = time_end;
```

Furthermore, the fact that all particles have grown older one time step has to be documented

```
Particle_System.advance_particles_ages (step_time);
```

and the positions of the graphical representations of all particles, springs, and attractions have to be updated

```
Particle_System.update_graphics_positions;

end
```

5 Particle System Object (Private Methods)

5.1 Private Method kill_old_particles

The private method

```
function kill_old_particles (Particle_System)
```

is called by advance_time in order to delete all particles that have reached the end of their life span. The following proceeding is very similar to the deleting of all attractions connected to a particle described in the second half of kill_particle. After initializing a kill buffer

```
    particles_to_be_killed = [];
```

a loop over all particles begins

```
    for i_particle = 1 : length (Particle_System.particles)
```

in which the current particle is retrieved

```
      Particle = Particle_System.particles(i_particle);
```

If the current particle is too old

```
      if Particle.age > Particle.lifespan
```

it is appended to the kill buffer

```
        particles_to_be_killed = ...
        [particles_to_be_killed, Particle];

      end

    end
```

Finally, in a loop over all particles to be deleted

```
    for i_particle = particles_to_be_killed
```

the particle is actually deleted by a call to the public kill_particle method

```
      Particle_System.kill_particle (i_particle);

    end

  end
```

5.2 Private Method get_phase_space_state

The user-called advance_time method needs the particle system state (positions and velocities of all particles) in form of one long state vector. This is achieved by a call to

```
function phase_space_state = ...
get_phase_space_state (Particle_System)
```

that simply retrieves the positions and the velocities of the particles

```
    positions = get_particles_positions (Particle_System);
    velocities = get_particles_velocities (Particle_System);
```

and combines them into the desired vector

```
    phase_space_state = [positions, velocities];

end
```

5.3 Private Method compute_state_derivative

The private method

```
function state_derivative = compute_state_derivative ...
(time, phase_space_state, Particle_System)
```

is called (four times per Runge-Kutta integration step) by the solver routine `ode4`, that provides the current state vector (`phase_space_state`) and awaits the method to return the derivative (`state_derivative`) of the state vector by evaluating the right hand side of the differential equation system:

$$\dot{\boldsymbol{x}} = \boldsymbol{f}\left(\boldsymbol{x}, t\right)$$

The evaluation is initialized by transposing the state vector into a row[30] vector

```
    phase_space_state = phase_space_state(:)';
```

and inserting it (back) into the particle system

```
    Particle_System.set_phase_space_state (phase_space_state);
```

[30] All physical vectors (positions, velocities, forces, ...) in this toolbox are row vectors; with the one exception that the solver `ode4` expects (and returns) the state vector to be a column vector. Since `ode4` is a third party routine, its parameter handling has not been altered.

Next, all forces (springs, attractions, drag, and gravity) acting on the particles are computed and aggregated in the corresponding particle force accumulators

```
Particle_System.aggregate_forces;
```

Now the state derivatives have to be retrieved. While the derivatives of the particle positions are just the particle velocities[31]

```
velocities = Particle_System.get_particles_velocities;
```

the derivatives of the velocities are the accelerations, which have to be computed from the aggregated forces and the masses of the particles

```
accelerations = Particle_System.get_particles_accelerations;
```

Finally, velocities and accelerations are combined into the current state derivative (column) vector to be returned by `compute_state_derivative` to `ode4`

```
state_derivative = [velocities, accelerations]';
```

```
end
```

5.4 Private Method set_phase_space_state

This method

```
function set_phase_space_state ...
(Particle_System, phase_space_state)
```

is the counterpart to `get_phase_space_state` and is used to take the state vector apart and insert its components back into the position and velocity properties of the corresponding particles. After retrieval of the number of particles

```
n_particles = length (Particle_System.particles);
```

a loop over all particles begins

```
for i_particle = 1 : n_particles
```

in which the current particle is retrieved

```
  Particle = Particle_System.particles(i_particle);
```

[31] The particle velocities could also directly be retrieved from the (second half) of the state vector that has just been inserted into the particle system. The method chosen here (retrieval from the particle system; just like the retrieval of the accelerations) is slightly slower but seems to be a little bit more concise.

The positions of the particles can be found (and updated) in groups of three in the first half of the state vector

```
      Particle.position = ...
      phase_space_state(3*i_particle - 2 : 3*i_particle);
```

while the velocities can be extracted from the second half of the state vector

```
      Particle.velocity = phase_space_state...
      (3*(i_particle + n_particles) - 2 : ...
       3*(i_particle + n_particles));

    end

  end
```

5.5 Private Method aggregate_forces

Every particle has a force accumulator in which all forces affecting the particle are aggregated in every time step

```
  function aggregate_forces (Particle_System)
```

Before the new forces can be accumulated, the force accumulator has to be zeroed

```
    Particle_System.clear_particles_forces;
```

Now the forces from springs, attractions, drag, and gravity can be calculated and summed up in the particle accumulator

```
    Particle_System.aggregate_springs_forces;
    Particle_System.aggregate_attractions_forces;
    Particle_System.aggregate_drag_forces;
    Particle_System.aggregate_gravity_forces;

  end
```

5.6 Private Method clear_particle_forces

The method to zero all particle force accumulators

```
  function clear_particles_forces (Particle_System)
```

simply goes through all particles

```
    for i_particle = 1 : length (Particle_System.particles)
```

retrieves the current particle and calls its `clear_force` method

```
      Particle = Particle_System.particles(i_particle);

      Particle.clear_force;

    end

  end
```

5.7 Private Method aggregate_springs_forces

The method

```
  function aggregate_springs_forces (Particle_System)
```

implements the computation and aggregation of the forces the springs exert on the particles. In a loop over all springs

```
    for i_spring = 1 : length(Particle_System.springs)
```

we first retrieve the current spring object

```
      Spring = Particle_System.springs(i_spring);
```

and the particles at both ends of the spring

```
      Particle_1 = Spring.particle_1;
      Particle_2 = Spring.particle_2;
```

Next, we compute the distance vector `position_delta` between the particles (which equals the spring vector)

```
      position_delta = Particle_2.position - Particle_1.position;
```

The spring force is proportional to the spring length `position_delta_norm`, which is the magnitude of the particle distance vector

```
      position_delta_norm = norm (position_delta);
```

If the spring has a very small length (e.g. because you have initialized all particles at the origin)

```
      if position_delta_norm < eps
```

the computation of the unit distance vector could lead to a divide-by-zero exception. Therefore, we restrict the minimum spring length to MATLAB's very small floating point spacing constant `eps`

```
    position_delta_norm = eps;

end
```

Now we can compute the spring unit vector `position_delta_unit`, which has the direction of the spring and a magnitude of one

```
position_delta_unit = position_delta/position_delta_norm;
```

According to Equation 11 the spring force is then proportional – with the spring stiffness as the gain – to the deviation of the actual spring length `position_delta_norm` from the spring rest length

```
spring_force = Spring.strength* ...
position_delta_unit*(position_delta_norm - Spring.rest);
```

Thereby, the direction of the spring force vector is defined by the spring unit vector. Finally, aggregate the current spring force vector in the force accumulator of the particles at both[32] ends of the spring

```
Particle_1.add_force (spring_force);
Particle_2.add_force (-spring_force);
```

As described in subsubsection 3.1.6, a realistic spring does also show some damping force depending on the velocity the spring is elongated or compressed with. Therefore, we compute (according to Equation 12) the relative velocity vector between the particles

```
velocity_delta = Particle_2.velocity - Particle_1.velocity;
```

The reason why we have to project the relative particle velocity vector along the particle distance vector

```
projection_velocity_delta_on_position_delta = dot ...
(position_delta_unit, velocity_delta)*position_delta_unit;
```

is explained in detail in the section before Equation 13. With the velocity projection vector we obtain the damping force vector of the spring via Equation 14:

```
damping_force = Spring.damping* ...
projection_velocity_delta_on_position_delta;
```

[32]Please note, that (unisono with Figure 29) the spring force vector accumulated in the "second" particle has a negative sign (indicating the opposite direction of the force)

The final step is to aggregate the damping force vector with the correct sign in the force accumulators of the end particles

```
      Particle_1.add_force (damping_force);
      Particle_2.add_force (-damping_force);

    end

  end
```

5.8 Private Method aggregate_attractions_forces

The method to aggregate the forces an attraction object exerts on its end particles

```
  function aggregate_attractions_forces (Particle_System)
```

is very similar to the first part of the aggregate_springs_forces method. In a loop over all attractions

```
    for i_attraction = 1 : length(Particle_System.attractions)
```

the current attraction object and its end particles are retrieved

```
      Attraction = Particle_System.attractions(i_attraction);

      Particle_1 = Attraction.particle_1;
      Particle_2 = Attraction.particle_2;
```

and the particle distance vector and its magnitude are computed

```
      position_delta = Particle_2.position - Particle_1.position;

      position_delta_norm = norm (position_delta);
```

The `minimum_distance` property of the attraction object causes a limitation of the maximum attraction force; if the particles are closer to each other than that minimum distance, the force is calculated as if the particles were `minimum_distance` apart[33]

```
      if position_delta_norm < Attraction.minimum_distance

        position_delta_norm = Attraction.minimum_distance;

      end
```

[33] If you really want the attraction force to increase "ad infinitum" (if the particles get closer and closer), you should use MATLAB's minimum floating point constant `eps` as `minimum_distance` instead of zero, which would lead to a divide-by-zero exception.

Now the attraction force vector can be computed according to Equation 9

```
      attraction_force = ...
      Attraction.strength* ...
      Particle_1.mass* ...
      Particle_2.mass* ...
      position_delta/ ...
      position_delta_norm/ ...
      position_delta_norm/ ...
      position_delta_norm;
```

Finally, the attraction force vectors are summed up – appropriately signed – in the force accumulators of the end particles

```
      Particle_1.add_force (attraction_force);
      Particle_2.add_force (-attraction_force);

    end

  end
```

5.9 Private Method aggregate_drag_forces

Compared to the aggregate_springs_forces and aggregate_attractions_forces methods, the method to compute and aggregate the drag forces slowing down the particles

```
  function aggregate_drag_forces (Particle_System)
```

is pretty short. Just start a loop over all particles

```
    for i_particle = 1 : length(Particle_System.particles)
```

and retrieve the current particle

```
      Particle = Particle_System.particles(i_particle);
```

Compute the drag force vector according to Equation 5

```
      drag_force = - Particle_System.drag*Particle.velocity;
```

and add the force to the content of the particle force accumulator

```
      Particle.add_force (drag_force);

    end

  end
```

5.10 Private Method aggregate_gravity_forces

The accumulation of the gravitational forces

```
function aggregate_gravity_forces (Particle_System)
```

is very similar to the aggregation of the drag. After the retrieval of the current particle in a loop over all particles, the g-force is computed in accordance with Equation 3 and fed into the corresponding accumulator.

```
  for i_particle = 1 : length (Particle_System.particles)

    Particle = Particle_System.particles(i_particle);

    gravity_force = Particle.mass*Particle_System.gravity;

    Particle.add_force (gravity_force);

  end

end
```

5.11 Private Method get_particles_accelerations

According to Equation 20, the `compute_state_derivative` method has to retrieve the velocity and the acceleration vectors of all particles and return them as the phase state derivative vector. While the retrieval of the velocities vector is a public method that has already been described, the computation of the particles accelerations vector

```
function accelerations = ...
get_particles_accelerations (Particle_System)
```

is a private method. It begins with the determination of the number of particles and the initialization of the acceleration vector

```
  n_particles = length (Particle_System.particles);

  accelerations = zeros (1, 3*n_particles);
```

and then starts a loop over all particles

```
  for i_particle = 1 : length(Particle_System.particles)
```

After the retrieval of the current particle

```
    Particle = Particle_System.particles(i_particle);
```

we have to distinguish if the particle is a free moving particle or if it is glued to a certain predefined position (which could also e. g. be the current mouse position). If we are talking about a fixed particle

```
      if Particle.fixed
```

there is no outer force that would otherwise lead to an acceleration of the particle

```
        force = [0 0 0];
```

If the particle is free

```
      else
```

we retrieve the current force vector of the particle, that has already been accumulated in aggregate_forces

```
        force = Particle.force;

      end
```

Since Newton we know that the effect (acceleration) resulting from a cause (force) acting on a medium (mass) is simply the ratio of cause and medium. Using the index of the current particle its acceleration vector is inserted at the correct place in the acceleration vector

```
      accelerations (3*i_particle - 2 : 3*i_particle) = ...
      force/Particle.mass;

    end

  end
```

5.12 Private Method advance_particles_ages

The private method

```
  function advance_particles_ages (Particle_System, step_time)
```

is called by advance_time in every simulation time step, in which it updates the ages of all particles. Therefore, in a loop over all particles

```
    for i_particle = 1 : length(Particle_System.particles)
```

the current particle is retrieved

```
      Particle = Particle_System.particles(i_particle);
```

Its age is incremented by the step time of the current integration step and written back into the age property of the particle

```
      Particle.age = Particle.age + step_time;

    end

  end
```

5.13 Private Method update_graphics_positions

The public method advance_time calls

```
  function update_graphics_positions (Particle_System)
```

in every simulation time step, in order to move the graphical representations of all particles, springs, and attractions to their new positions in the axes of the particle system. The first loop moves the particles by a call to the corresponding update method of the particle

```
    for i_particle = 1 : length (Particle_System.particles)

      Particle_System.particles(i_particle).update_graphics_position;

    end
```

Similar loops update the graphical representations of the springs and attractions

```
    for i_spring = 1 : length (Particle_System.springs)

      Particle_System.springs(i_spring).update_graphics_position;

    end

    for i_attraction = 1 : length (Particle_System.attractions)

      Particle_System.attractions(i_attraction).update_graphics_position;

    end
```

Finally, the `drawnow` command makes sure that the moved objects are actually displayed at their new locations

```
    drawnow

  end
```

For performance reasons, drawnow should only be called once per time step.

6 Particle Object

6.1 Class Definition and Properties particle

Just as the particle system, the particle class is derived from the handle class

```
classdef particle < handle
```

and defines the particle properties

```
properties

    mass
    position
    velocity
    fixed
    lifespan
    age = 0
    force = [0 0 0]
    graphics_handle

end
```

The mass, the initial position and velocity vectors, the flag that defines whether the particle is fixed or free, and the life span of the particle can be defined in the call to the particle constructor. The current age and the force accumulator are directly initialized in the property definition block. The graphics handle that references the graphical representation of the particle becomes defined in the particle constructor.

6.2 Constructor particle

The particle constructor

```
function Particle = particle ...
(Particle_System, mass, position, velocity, fixed, lifespan)
```

can be called with only one arguments – the particle system into which the particle will be incorporated. In that case, the properties are defaulted to reasonable initial values (unit mass, position at the origin, no velocity, free particle, and infinite life span)

```
if nargin == 1

  mass = 1;
  position = [0, 0, 0];
```

```
      velocity = [0, 0, 0];
      fixed = false;
      lifespan = inf;

    end
```

All properties (user supplied or default) are saved in the corresponding particle object structure fields

```
    Particle.mass = mass;
    Particle.position = position;
    Particle.velocity = velocity;
    Particle.fixed = fixed;
    Particle.lifespan = lifespan;
```

Finally, the particle is incorporated into the particle system by the private `append` method of the particle class

```
    Particle.append (Particle_System);

  end
```

6.3 Private Method append

The private method

```
  function append (Particle, Particle_System)
```

appends the particle to the particle array of the particle system. It might be a good idea to force the particle system figure window to become visible, and raise it above all other figures on the screen[34]

```
    figure (Particle_System.graphics_handle)
```

Now the graphical representation of the particle can be displayed in the axes as a "big" red dot and its graphics handle can be saved in the corresponding property of the particle

```
    Particle.graphics_handle = ...
    line ( ...
    Particle.position(1), ...
    Particle.position(2), ...
```

[34]Remember, it is very easy to adapt object behavior to your own taste in MATLAB source code. If you don't like it, change it.

```
    Particle.position(3), ...
    'color', [1 0 0], ...
    'markersize', 30, ...
    'marker', '.');
```

If the particle is supposed to be fixed, the red dot is substituted by a star of the same color

```
    if Particle.fixed

      set (Particle.graphics_handle, ...
      'markersize', 10, 'marker', '*')

    end
```

Finally, the particle can be copied into its new home

```
    Particle_System.particles = ...
    [Particle_System.particles, Particle];

  end
```

6.4 Methods add_force, clear_force

The `add_force` method adds (*nomen est omen*) a force to the particle force accumulator

```
  function add_force (Particle, force)

    Particle.force = Particle.force + force;

  end
```

The `clear_force` method can be used to zero the current force accumulator (e. g. at the beginning of a new simulation time step)

```
  function clear_force (Particle)

    Particle.force = [0 0 0];

  end
```

6.5 Method delete

When you delete a particle object – e. g. by the use of `kill_particle` – its graphical representation has to be deleted too. This is done by a call to

```
function delete (Particle)
```

which determines if the particle's graphics handle is still valid

```
  if ishandle (Particle.graphics_handle)
```

and deletes the "red dot" if necessary

```
    delete (Particle.graphics_handle)

  end

end
```

6.6 Method set.fixed

The `set.fixed` method is an example of MATLAB's property access methods that automatically execute whenever object properties are queried or set. Here

```
function set.fixed (Particle, fixed)
```

is called whenever you modify the `fixed` property of a particle after its instantiation. This method is necessary because in that case we do not only want to save the demanded state in the corresponding property

```
  Particle.fixed = fixed;
```

but we also have to adjust the graphical representation of the particle

```
  if Particle.fixed

    set (Particle.graphics_handle, ...
    'markersize', 10, 'marker', '*')

  else

    set (Particle.graphics_handle, ...
    'markersize', 30, 'marker', '.')

  end

end
```

6.7 Method set.position

If you modify the `position` property of a particle MATLAB automatically calls the corresponding property access method

```
function set.position (Particle, position)
```

The new position is saved in the position property of the particle

```
    Particle.fixed = fixed;
```

and graphical representation of the particle is updated

```
    Particle.update_graphics_position;

end
```

6.8 Method update_graphics_position

As its name might already imply, the `update_graphics_position` method

```
function update_graphics_position (Particle)
```

updates the graphics position of a particle

```
    set (Particle.graphics_handle, ...
    'xdata', Particle.position(1), ...
    'ydata', Particle.position(2), ...
    'zdata', Particle.position(3));

end
```

7 Spring Object

7.1 Class Definition and Properties spring

The spring class is derived from the handle class too

```
classdef spring < handle
```

and hosts the following properties

```
properties

    particle_1
    particle_2
    rest
    strength
    damping
    graphics_handle

end
```

which are the particles at the ends of the spring, the rest length (i. e. the length at which the spring does not exert any force), the strength or stiffness, the coefficient of the associated damper, and the graphics handle of the graphical spring representation.

7.2 Constructor spring

The spring constructor

```
function Spring = spring (Particle_System, ...
particle_1, particle_2, rest, strength, damping)
```

can be called with all six or only three arguments; as a minimum you have to supply the particle system the spring has to appear in and the two particles the spring connects. The other spring properties are then defaulted by the constructor to unit rest length, unit strength, and no damping

```
if nargin == 3
    rest = 1;
    strength = 1;
    damping = 0;
end
```

The remaining spring parameters are stored as spring properties too

```
    Spring.particle_1 = particle_1;
    Spring.particle_2 = particle_2;
    Spring.rest = rest;
    Spring.strength = strength;
    Spring.damping = damping;
```

and the spring is copied into the particle system

```
    Spring.append (Particle_System);

  end
```

7.3 Private Method append

The private append method of a spring

```
  function append (Spring, Particle_System)
```

closely resembles the append method of a particle. It forces the particle system figure window to become visible

```
    figure (Particle_System.graphics_handle)
```

computes the spring ends positions, which are the positions of the particles at both ends of the spring

```
    spring_position(1, 1:3) = Spring.particle_1.position;
    spring_position(2, 1:3) = Spring.particle_2.position;
```

draws a blue line symbolizing the spring

```
    Spring.graphics_handle = ...
    line (...
    spring_position(:, 1), ...
    spring_position(:, 2), ...
    spring_position(:, 3), ...
    'linewidth', 1, ...
    'linestyle', '-', ...
    'color', [0 0 1]);
```

and appends the spring to the spring array of the particle system

```
    Particle_System.springs = [Particle_System.springs, Spring];

  end
```

7.4 Method delete

Just like the particle `delete` method, the spring `delete` method

```
function delete (Spring)
```

makes sure that the spring's graphics handle is still valid

```
    if ishandle (Spring.graphics_handle)
```

and deletes the blue line if possible

```
      delete (Spring.graphics_handle)

    end

end
```

7.5 Method update_graphics_position

The method to update the graphical representation of the spring

```
function update_graphics_position (Spring)
```

buffers the start and end positions of the spring as row vectors in a matrix

```
    spring_position(1, 1:3) = Spring.particle_1.position;
    spring_position(2, 1:3) = Spring.particle_2.position;
```

in order to use the corresponding columns as spring coordinates vectors

```
    set (Spring.graphics_handle, ...
    'xdata', spring_position(:, 1), ...
    'ydata', spring_position(:, 2), ...
    'zdata', spring_position(:, 3));

end
```

8 Attraction Object

The attraction class definition, properties and methods (`attraction`, `append`, `delete`, and `update_graphics_position`) are so very similar to their spring counterparts, that it would be quite a waste of disk space to repeat and explain "identical" code fragments here.

A Vector Projection

If $\boldsymbol{a}$ and $\boldsymbol{b}$ are two arbitrary vectors, we are looking for the vector $\boldsymbol{b}_a$ as the projection of vector $\boldsymbol{b}$ along vector $\boldsymbol{a}$ (Figure 31).

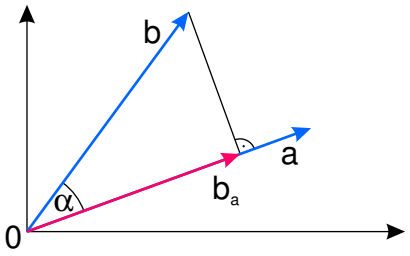


Figure 31: Projection vector $\boldsymbol{b}_a$ of vector $\boldsymbol{b}$ along vector $\boldsymbol{a}$

The dot product (aka scalar product or inner product) of two vectors $\boldsymbol{a}$ and $\boldsymbol{b}$ is usually expressed as

$$\boldsymbol{a}\,\boldsymbol{b} = |\boldsymbol{a}|\,|\boldsymbol{b}|\cos\alpha \tag{21}$$

where $|\boldsymbol{a}|$ and $|\boldsymbol{b}|$ denote the magnitudes or norms of the vectors and α is the angle between $\boldsymbol{a}$ and $\boldsymbol{b}$. On the other hand, the cosine of an angle in a rectangled triangle can be computed as the ratio of its adjacent leg and the hypotenuse

$$\cos\alpha = \frac{|\boldsymbol{b}_a|}{|\boldsymbol{b}|} \tag{22}$$

Insertation of Equation 22 into Equation 21 and the cancelation of $|\boldsymbol{b}|$ leads to

$$\boldsymbol{a}\,\boldsymbol{b} = |\boldsymbol{a}|\,|\boldsymbol{b}|\,\frac{|\boldsymbol{b}_a|}{|\boldsymbol{b}|} = |\boldsymbol{a}|\,|\boldsymbol{b}_a| \tag{23}$$

which states that the dot product can also bee seen as the product of the norms of the first vector and the projection of the second along the first. Equation 23 can easily be solved for the norm $|\boldsymbol{b}_a|$ of the projection vector

$$|\boldsymbol{b}_a| = \frac{\boldsymbol{a}\,\boldsymbol{b}}{|\boldsymbol{a}|} = \frac{\boldsymbol{a}}{|\boldsymbol{a}|}\boldsymbol{b} \tag{24}$$

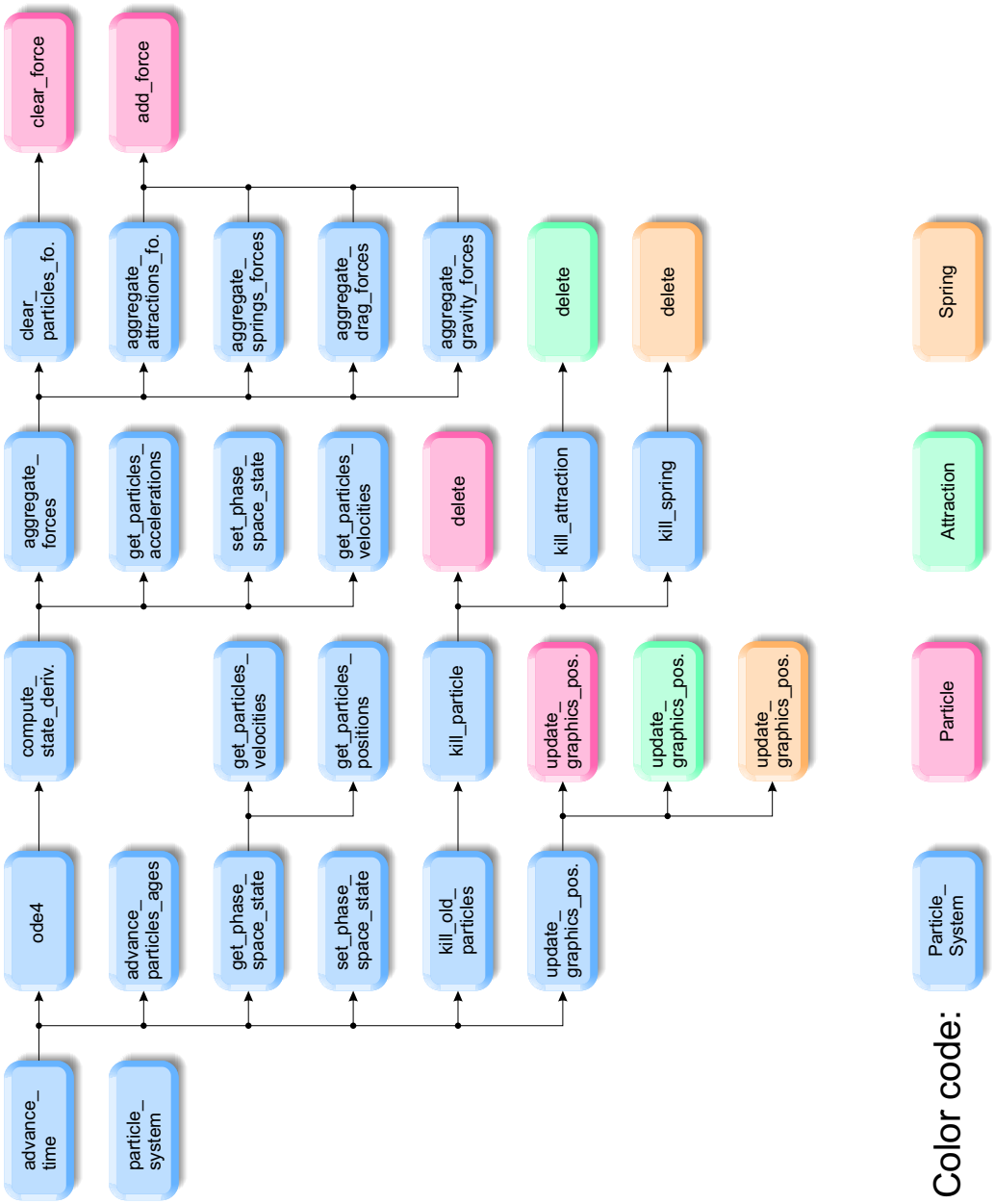
Every vector can be expressed as the product of its norm and its unit vector

$$\boldsymbol{b}_a = |\boldsymbol{b}_a|\,\frac{\boldsymbol{b}_a}{|\boldsymbol{b}_a|} = |\boldsymbol{b}_a|\,\frac{\boldsymbol{a}}{|\boldsymbol{a}|} \tag{25}$$

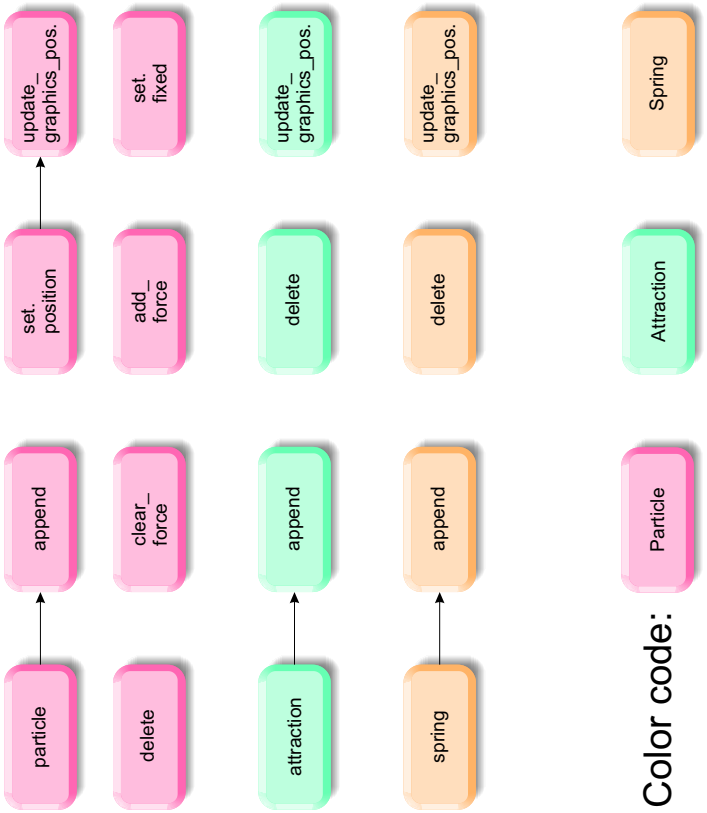
Since $\boldsymbol{b}_a$ and $\boldsymbol{a}$ are collinear (parallel), the unit vector ${}^{\boldsymbol{b}_a}/_{|\boldsymbol{b}_a|}$ is identical to the unit vector ${}^{\boldsymbol{a}}/_{|\boldsymbol{a}|}$. Finally, Equation 24 can be used in Equation 25, leading to the general equation of the projection vector

$$\boldsymbol{b}_a = \left(\frac{\boldsymbol{a}}{|\boldsymbol{a}|}\boldsymbol{b}\right)\frac{\boldsymbol{a}}{|\boldsymbol{a}|} \tag{26}$$

B Particle System Dependency



C Particle, Attraction, and Spring Dependency



References

[1] **Artman, T.**: *Flash 9 Particle System.*
http://www.valveblog.com/2006/07/flash_9_particl.html

[2] **Bernstein, J. T.**: *A Particle Physics Library for Processing.*
http://www.cs.princeton.edu/~traer/physics/

[3] **Frey, B.; Reas, C.**: *Processing 1.0 (BETA).* Open source programming language.
http://processing.org

[4] **Johnson & Johansson**: *Hand-Eye Coordination Test.*
http://jj.thebuchis.de/jj/hand-eye_coordination.swf

[5] **McAllister, D. K.**: *Particle System API.*
http://particlesystems.org

[6] **Mehta, C.**: *Falling Sand Game.* Game using Processing.
http://chir.ag/stuff/sand

[7] **Nimoy, J.**: *Balldroppings.*
http://balldroppings.com

[8] **Press, W. H., et al.**: *Numerical Recipes: The Art of Scientific Computing.* Chapter 16.1: Runge-Kutta Method, pp. 710-714, Cambridge University Press (2002).
http://www.nrbook.com/a/bookcpdf/c16-1.pdf

[9] **Sims, K.**: *Particle Dreams.* Computer Animation (1988).
http://www.archive.org/details/sims_particle_dreams_1988

[10] **Soda Creative Ltd.**: *sodaplay.* The home of creative play.
http://sodaplay.com

[11] **The Mathworks**: *Matlab Central File Exchange.*
http://www.mathworks.com/matlabcentral/fileexchange

[12] **The Mathworks**: *Tech Note 1510: Differential Equations in MATLAB.*
http://www.mathworks.com/support/tech-notes/1500/1510.html#fixed

[13] **The Soupboys**: *Souptoys.*
http://souptoys.com

[14] **Chenciner, A.; Montgomery, R.**: *A Remarkable Periodic Solution of the Three-Body Problem in the Case of Equal Masses.* The Annals of Mathematics, Second Series, Vol. 152, No. 3 (Nov., 2000), pp. 881-901.
http://www.jstor.org/pss/2661357