Capillary interaction between floating shapes

Gastón Barboza

April 2nd, 2023

Introduction

Floating objects, are, by definition, constrained to move on the surface of a liquid. The force of their weight is compensated by their buoyancy. However, if the surface on which they rest is curved, this upwards-pointing buoyant force has a non-zero projection along the direction tangential to the surface (see Figure 1).

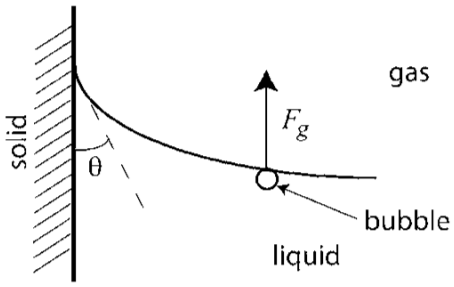

Figure 1: Buoyancy on a curved surface

This effect is called the Cheerios effect, since it causes Cheerios cereal floating in milk to stick to the walls of the bowl. Moreover, floating objects distort the surface of the water as well, and depending on how they distort it (hydrophilically or hydrophobically) this can act as a medium for attractive and repulsive forces between them (see Figure 2).

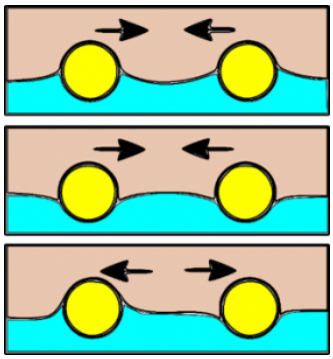

Figure 2: Attractive and repulsive forces

The original aim of this project, as outlined in the flowchart at the end of the document, was to simulate this effect. The final implementation of this goal was not reached, but the basic framework to construct a fuller simulation is laid down. A single object is simulated on a sloped interface to achieve an effect similar to the one shown in Figure 1.

Main program step by step

The program `cheerios` runs using three modules: grid, shapes and forces. The grid module is for manipulating the global grid on which objects would be placed to interact; the shapes module contains the implementation of shapes through user-defined types, and the forces module calculates and integrates the forces acting on shapes. The program performs the following steps to set up the initial conditions:

1. Calling the `read_params` subroutine defined in the `cheerios` program, which opens the *system.param* file and reads from it the size of the box containing the liquid and shapes (*x_length* and *y_length*), the resolution of the grid *grid_res* assigned to the variable n, the maximum number of steps for which to simulate the system *max_steps*, and the *time_step* delta with which to integrate.

2. Allocating an array *history* of size 2 by *max_steps* to store the x and y components of the location of a single object as the simulation evolves in time, and calling the `initialize_grid` subroutine from the grid module, which allocates the global grid array with a size of n by n. This is a rank 2 real array whose values correspond to the z-height of the water surface at a certain x-y point, represented by the array's indices.

3. Calling the `read_shape` subroutine from the shapes module which initializes a sphere from the information in the *sphere1.param* file. This sphere has a defined xy coordinate location, an initial acceleration of 0, a radius, a height at which it is initially floating on the water, and a density. It also contains a local coordinate system of water height levels (see section *Shapes module*).

4. Calling the `balance` subroutine from the forces module, which balances the sphere's weight with the hydrostatic buoyancy force created by the volume of water displaced when the sphere is submerged. This is a static force balancing iteration which produces the equilibrium floating height of the sphere.

5. Calling the `tilt_water` subroutine from the forces module which imposes a slant on the surface of the water, similar to what a cheerios would experience near the edge of a bowl or near another cheerios.

Afterwards, calling the `make_dip` subroutine from the shapes module, which indents the surface of the water due to the presence of the sphere. These water levels are set on the object's local coordinate system.

The program now integrates the trajectory of the sphere. This is done by first calling the forces-module `integrate_normals` subroutine, which calculates the tangential components of the buoyancy force and saves the resulting acceleration as an attribute of the sphere object. As the position and acceleration of the object are known, it can now be integrated using a Verlet algorithm; this is done through the `integrate_time` subroutine from the forces module. This whole process is repeated until the *history* array of length *max_steps* is filled, at which point the array is saved to the *trajectory.dat* file and the program terminates.

1 Grid module

This module is underutilized as its main purpose would be to represent and visualize multiple shapes floating on water and approaching eachother. It contains the `initialize_grid` subroutine, which allocates a grid according to its supplied arguments and sets it to 0. The `save_grid` subroutine writes the array of water heights to a *grid.dat* file, and the x and y axis coordinates to an *x_axis.dat* and *y_axis.dat* file respectively. As the grid array is indexed by integers but these indices must represent x-y coordinates, the `xpos` and `xint` subroutines scale and convert integer indices to real coordinates and vice versa.

2 Shapes module

The shapes module contains various interfaces for manipulating shapes: `read_shape`, `make_dip`, `submerged_volume`, `local_pos`. As only the *sphere* shape type is defined, these interfaces are only implemented for this shape.

Spheres are a user-defined type created through the `read_sphere` function. They have radius and density attributes, from which mass and volume attributes are calculated; initial coordinates; acceleration (initially zero); and an attached reference frame with a hard-coded resolution.

The reason for attaching a reference-frame to the type and hard coding its resolution is to be able to center the sphere at the integer index (0,0) and set array bounds from -n/2 to n/2. This is necessary for the `integrate_normals` procedure to work correctly; applying an offset when converting real indices to integer indices (a necessary step) introduces numerical error which results in non-zero forces appearing. Allocatable arrays on the other hand are by construction always indexed from 1 to n. A more elegant solution may be achievable using pointers. In any case, the effect of the water surface level is local, so the sphere does not need the information of the global grid for its dynamics, and a finer-grained local array provides better accuracy.

The `make_dip_sphere` subroutine calculates the z-height of the bottom half of the sphere's surface and pushes the water down to that level to create an indentation given by its position.

The `submerged_volume_sphere` subroutine uses the formula for the volume of a spherical cap to calculate the volume of the part of the sphere under the surface of the water.

The `local_pos_sphere` subroutine converts the integer indices of the local reference system array to real coordinate values.

3 Forces module

The forces module various procedures for calculating forces on the objects and integrating them. It also contains various public procedures which are only implemented for the sphere type.

The `balance_sphere` subroutine performs static force equilibration to find the equilibrium height of a partially submerged sphere floating on water.

The `tilt_water` subroutine slopes the surface of the water on the object's local coordinate system so as to simulate the effect of being near a wall curved by the capillary effect.

The `integrate_normals_sphere` sums all of the normal vectors of the sphere where it is in contact with the surface of the water. It normalizes the sum to find the average tangent slope of the water where the object floats. The component of the buoyancy force tangential to the water's surface is given by

$$mg\hat{z} - mg(\hat{z}\cdot\hat{n})\hat{n}.$$

Projecting the tangential force onto the xy axes, this force results in an acceleration given by

$$-g(\hat{z}\cdot\hat{n})(\hat{x}\cdot\hat{n})\hat{x} - g(\hat{z}\cdot\hat{n})(\hat{y}\cdot\hat{n})\hat{y}.$$

This acceleration is calculated and saved to the object's acceleration attribute.

Finally, the `integrate_time` subroutine uses the Verlet algorithm

$$x_{n+1} = 2x_n - x_{n-1} + a_n\Delta t^2$$

to evolve positions in time. Velocities are not calculated as magnitudes that depend on them (e.g. energy) are not needed for this simulation.