

***flippy*: User friendly and open source framework for lipid membrane simulations**

George Dadunashvili¹ and Timon Idema¹

¹Department of Bionanoscience, Kavli Institute of Nanoscience, Delft University of Technology, The Netherlands

Corresponding author:

Timon Idema¹

Email address: t.idema@tudelft.nl

ABSTRACT

Animal cells are both encapsulated and subdivided by lipid bilayer membranes. Beyond just acting as boundaries, these membranes' shapes influence the function of cells and their compartments. Physically, membranes are two-dimensional fluids with complex elastic behavior, which makes it impossible, for all but a few simple cases, to predict membrane shapes analytically. Instead, the shape and behavior of biological membranes can be determined by simulations. However, the setup and use of such simulations require a significant programming background. The availability of open-source and user-friendly packages for simulating biological membranes needs improvement.

Here, we present *flippy*, an open-source package for simulating lipid membrane shapes, their interaction with proteins or external particles, and the effect of external forces. Our goal is to provide a tool that is easy to use without sacrificing performance or versatility. *flippy* is an implementation of a dynamically triangulated membrane. We use a precise yet fast algorithm for calculating the geometric properties of membranes and can also account for local spontaneous curvature, a feature not all discretizations allow. Finally, in *flippy* we can also include regions of purely elastic (non-fluid) membranes and thus explore various shapes encountered in living systems.

BACKGROUND

Lipid bilayer membranes form the envelopes of animal cells and of many organelles contained in them. These biological membranes are highly flexible materials, capable of adopting many nontrivial shapes corresponding to specific cell functions and responding to environmental circumstances. Therefore, we can infer which processes occur inside a cell or organelle from the shapes of their membranes [1]. Inversely, in synthetic biology, membranes can be manipulated to mimic growth and division processes of living cells [2, 3, 4, 5]. Achieving symmetric, stable division over many generations is a significant challenge in the bottom-up assembly of living cells. A crucial part of the problem is understanding how external mechanical and chemical cues drive membrane reshaping. Predicting the shapes of membranes analytically is very difficult and therefore limited to cases of membranes with few constraints and high symmetry. Even numeric solutions to analytic equations are usually only possible in highly symmetric cases. In order to predict membrane shapes for generic problems, we need to use simulations. Full atomistic simulations are out of the question due to computational constraints when we are interested in large-scale membrane reshaping. Luckily several types of simulations describe the membrane behavior on a large scale, like self-assembled membranes [6], phase fields based methods [7], and dynamically triangulated membrane Monte Carlo (DTMMC) simulations [8]. The latter method is based on minimizing membrane surface energy, which makes interpreting results easy and leaves an opportunity to connect the findings to an analytical model [9]. Thus, it is not surprising that the method of dynamically triangulated membrane simulations has found broad adoption in the field and has been used to model diverse set of experimental systems from membranes responding to osmotic conditions [9] and shear flow [10], to interactions of membranes with colloidal particles [11, 12, 13, 14, 15] and with proteins [16, 17, 18].

IMPLEMENTATION

While DTMMC simulations are widely used, simulation codes are rarely published. We therefore wrote the *flippy* software as an open-source package. While DTMMC simulations are popular, they are hard to write and even harder to

optimize. Even with only basic functionality, a DTMMC code quickly becomes large and hard to maintain. Therefore, to make further progress in the development of DTMMC simulations, we need an open-source library with a vibrant community and developer base around it. We want our package to help biophysicists to get to implementing the specifics of their system without needing to reinvent the wheel and code the whole dynamically triangulated membrane from scratch. *flippy* is designed with this objective in mind.

An ideal simulation framework for membranes would not involve programming at all on the end user's side. Simulating a membrane under a specific physical constraint would be like conducting an experiment. A fully interactive framework would drastically reduce the barrier to simulations, only require understanding the experimental setup, and enable researchers to directly compare their results to simulations. However, this ideal case of an interactive framework is hard to implement in a vacuum. While keeping it in mind as an end-goal, we decided to start with a more manageable task. Even though using a C++ library requires much more knowledge than just using interactive software, we keep user-friendliness and a high level of abstraction as our primary goals. The implementation of *flippy* as a C++ library instead of a domain-specific scripting language allows users the flexibility to incorporate it in existing code bases and easily extend it, thus contributing to our goals of community-based growth. Having an implementation in a compiled language additionally allows the users to create fast simulations, increasing the range of systems that can be simulated by *flippy* in a reasonable time.

The fact that C++ does not have a centralized package manager usually makes it hard to obtain or use external libraries. In our experience, this is the most significant inconvenience related to using the C++ language. To minimize this friction as much as possible, we opted to implement a header-only library and eliminate almost all external dependencies. Our code only relies on an external JSON parser to easily save simulation data. This parser is itself licensed under the same open source license as *flippy*, which enabled us to bundle it with *flippy* [19]. This independence from external dependencies makes using our software package as easy as it gets for C++ libraries.

Code quality control

Every large code base is prone to hidden bugs and unexpected behaviors in new use cases. To minimize errors, we implemented an extensive unit testing framework, and we are happy to report that our code base has over 95% coverage. Thus, almost every function implemented in our base is covered by at least one test case, and we intend our library of unit tests to grow continuously. We are aware that unit tests cannot guarantee that the code is bug-free, and we intend to use the bug reporting facilities of the GitHub repository to enable our users to report bugs and help improve the package.

Mathematical details of the implementation

Since we aspire to a user-friendly framework, *flippy* must implement commonly needed utilities that almost every DTMMC simulation will require. The triangulation provided by *flippy* needs to do proper bookkeeping of several important geometric quantities, during the update of the triangulation, like the local curvature vector, local area, and local unit bending energy of each node. We also keep track of the global counterparts of these quantities, i.e., the total area and total unit bending energy of the triangulated shape. These quantities are defined on continuous shapes, requiring a mathematically rigorous discretization on a triangulated lattice. By this, we mean that the discretized quantities should converge to their continuous counterparts for finer triangulations, and the simulation should become more precise with an increasing number of triangles. From all the above, the local curvature is most challenging to discretize and can lead to triangulation-dependent curvature energies, as demonstrated by Gompper and Kroll [20]. We use an extension of the method proposed in [20] for calculating local mean curvature. This extension was introduced by Meyer et al. [21] to calculate the local area associated with a node more precisely and sidestep numerical problems that occur for triangulations containing obtuse triangles. Finally, we use the same expression for the node-associated volume as Guegen et al.[9], which is fast to calculate since it only relies on already computed quantities. However, this node-associated volume does not have a physical meaning, it only sums to the correct total volume enveloped by a closed triangulation.

RESULTS

In this section, we want to demonstrate *flippy*'s ability to abstract away the implementation details of a dynamic triangulation and Monte Carlo updating scheme. To this end, we go through the process of simulating a simple experimental system of a deflated giant unilamellar vesicle (GUV) and use *flippy* to predict the equilibrium shape of the vesicle. In the following, we will only present the key elements of the code. The complete version of this simulation is

provided on GitHub; for more details, please see the *Summary and outlook* section. The system of a deflated GUV can be modeled by the following surface energy

$$E_{\text{surf}} = \frac{\kappa}{2} \int dA (2H)^2 + K_A \frac{(A - A_t)^2}{A_t} + K_V \frac{(V - V_t)^2}{V_t}, \quad (1)$$

where κ is the bending rigidity and H is the local mean curvature of the membrane. The integral $\int dA$ ranges over the surface area of the vesicle. This part of the energy describes the tendency of the biological membranes to minimize their local square mean curvature [22, 23]. The Lagrange multipliers K_A and K_V fix the area A and volume V to their target values $A_t = A_0 = 4\pi R_0^2$ and $V_t = 0.6V_0 = 0.6 \frac{4\pi}{3} R_0^3$, where R_0 is the radius of the initial (pre deflation) spherical GUV. Since the deflation of the vesicle does not change its area, we keep it fixed to the initial value. However, the target value of the volume is fixed to 60% of the initial volume to account for deflation. We picked 60% of the initial volume because, for this value, we expect the equilibrium configuration to be a biconcave shape, providing an easy visual way to judge the success of the simulation. However, the prediction of a biconcave shape is only precise for zero temperature, i.e., for $k_B T = 0$. For simplicity, the following example describes a simulation performed at $k_B T = 1$. Thus, the resulting shapes will be noisy and not perfectly biconcave.

All the complexity of creating and maintaining a dynamic triangulation is hidden in three conceptual steps; define the energy, initiate a triangulation, and initiate an updater that will use the energy to update the triangulation. We can start with the definition of the energy function that implements the eq. (1). Since the `MonteCarloUpdater` will use this energy, its signature needs to follow a specific convention that the updater will recognize,

```
double surface_energy(fp::Node<double, unsigned int> const&,
                    fp::Triangulation<double, unsigned int> const& ,
                    EnergyParameters const& )
```

where the first argument needs to be a *flippy* Node type, representing the node that is being updated, and the second argument needs to be a `Triangulation` type representing the triangulation that is being updated. The third argument can be any type and is intended to be a user-defined data struct containing all the parameters of the energy function. The actual function body is then a straightforward implementation of eq. (1):

```
double surface_energy([[maybe_unused]] fp::Node<double, unsigned int> const& node,
                    fp::Triangulation<double, unsigned int> const& trg,
                    EnergyParameters const& prms) {
    double V = trg.global_geometry().volume;
    double A = trg.global_geometry().area;
    double dV = V-prms.V_t;
    double dA = A-prms.A_t;
    double energy = prms.kappa*trg.global_geometry().unit_bending_energy +
                    prms.K_V*dV*dV/prms.V_t + prms.K_A*dA*dA/prms.A_t;
    return energy;
}
```

Here the first variable in the function signature is designated `[[maybe_unused]]` since in this particular implementation of the energy, we are not interested in the local properties of any given node and thus do not use this variable. The second step in the implementation of the model is to declare a triangulation:

```
fp::Triangulation<double, unsigned int> tr(n_triangu, R_0, r_Verlet);
```

where the template parameters `double` and `unsigned int` specify which internal representation of floating point and integer numbers the `Triangulation` class is supposed to use. The first argument of the instantiation `n_triangu` specifies the level of triangulation, which sets the fineness of the mesh. The second argument, `R_0`, sets the initial radius of the triangulated sphere, and the last argument `r_Verlet`, relates to the implementation of membrane self-intersection avoidance. *flippy* implements a Verlet list to check spatial closeness of the nodes efficiently [24].

The third step is to declare a Monte Carlo updater that will use the energy function to update the triangulation according to a Metropolis algorithm [25]:

```
fp::MonteCarloUpdater<double, unsigned int, EnergyParameters,
                    std::mt19937, fp::SPHERICAL_TRIANGULATION>
mc_updater(tr, prms, surface_energy, rng, l_min, l_max);
```

The signature of this class instantiation is quite large since the updater needs to know the energy function, all necessary update parameters, and the triangulation. The first two template parameters specify the internal representation of numbers, just like in the case of the `Triangulation` class. These parameters must be the same in both cases.

`EnergyParameters` specifies the user-defined struct type name that contains the parameters used inside the energy function. `std::mt19937` specifies the type of the random number generator that we will provide to the updater for generating random numbers for the Metropolis algorithm. The last parameter specifies the triangulation type (currently, spherical and planar triangulations are possible). The instance of the updater itself has six arguments. The first four provide the updater with references to the already declared instances of triangulation class, energy parameters struct, energy function, and a random number generator. The last two arguments specify minimum and maximum allowed distances between the triangulation nodes.

All that is left to do is to create an update loop that specifies in what order and how often we want to update the triangulation.

```
for(unsigned int mc_step=0; mc_step<max_mc_steps; ++mc_step){
    for(unsigned int node_id: shuffled_ids) {
        displ = {displ_distr(rng), displ_distr(rng), displ_distr(rng)};
        mc_updater.move_MC_updater(guv[node_id], displ);
    }
    std::shuffle(shuffled_ids.begin(), shuffled_ids.end(), rng);
    for(unsigned int node_id: shuffled_ids) {
        mc_updater.flip_MC_updater(guv[node_id]);
    }
}
```

where in every update step, we loop over each node and use the methods of the `MonteCarloUpdater` class to move nodes and flip bonds. Between the loops where we *move* and *flip* the nodes, we shuffle the `shuffled_ids` vector, which was defined before the loop and contains the ids of the nodes. The shuffling step ensures that we iterate randomly through the nodes at each Monte Carlo step and do not introduce unwanted correlations between node updates. This loop represents the logic of the experiment that we want to model. In this case, the experiment is simple; we are equilibrating a vesicle at a constant temperature (starting from slightly unphysical initial conditions of mismatching volume). This loop will become more complex as the needs of the simulation will grow. However, this corresponds to the true complexity that arises from the system itself and not from the implementation. Some higher stages of complexity might require a more sophisticated updater. The `MonteCarloUpdater` class is provided by *flippy* because a Metropolis updating scheme is a popular one. However, the `Triangulation` class itself is completely agnostic towards the updating scheme that is used on it. The user is free to implement another updating scheme if the Metropolis algorithm is not suitable to their problem and still be able to use the triangulation provided by *flippy*. This also enables us to easily extend *flippy* with new updaters.

Finally, to make saving the state of the simulation easy, *flippy*'s `Triangulation` class has a method that saves the representation of the data as a *JSON* object, which is a text-based human-readable data format [26]. *flippy* comes bundled with an open source *JSON* parser [19]. A single statement is sufficient to create *JSON* data of the current state of the triangulation

```
fp::Json data = tr.make_egg_data();
```

and a utility function in *flippy* allows the saving of this data to a text file as follows:

```
fp::json_dump("test_run_final", data);
```

The `make_egg_data` methods naming refers to the fact that this *JSON* data contains the necessary information to reinitialize the triangulation (like an egg contains all the nutrients for the chicken that will hatch from it), thus allowing the user to continue simulation from a save-file. If we use this simple code [27] (which is comfortably below 100 lines, including all imports, variable definitions, and comments), we will obtain (in a few minutes) the expected biconcave shape (see fig. 1 B and C). This example clearly shows that *flippy* is capable of simulating a simple physical system in few lines of code, where all unnecessary complexity is abstracted away in the library and all the complexity that is still left in the user written code, contains necessary information about specific characteristics of the simulated system in question. Importantly, this abstraction and simplicity don't come at the cost of unreasonable runtime of the simulation.

SUMMARY AND OUTLOOK

The source code of *flippy* is available on GitHub [28], together with a full documentation and further demonstrations. The code used for the simulation in the *Results* section is also part of *flippy*'s GitHub repository, and the most up-to-date version of it can be found in the `demo/biconcave_shapes_MC` folder of the repository [28]. The version of the code that was most up to date at the time of writing this paper, and was used to generate the code snippets in the *Results*

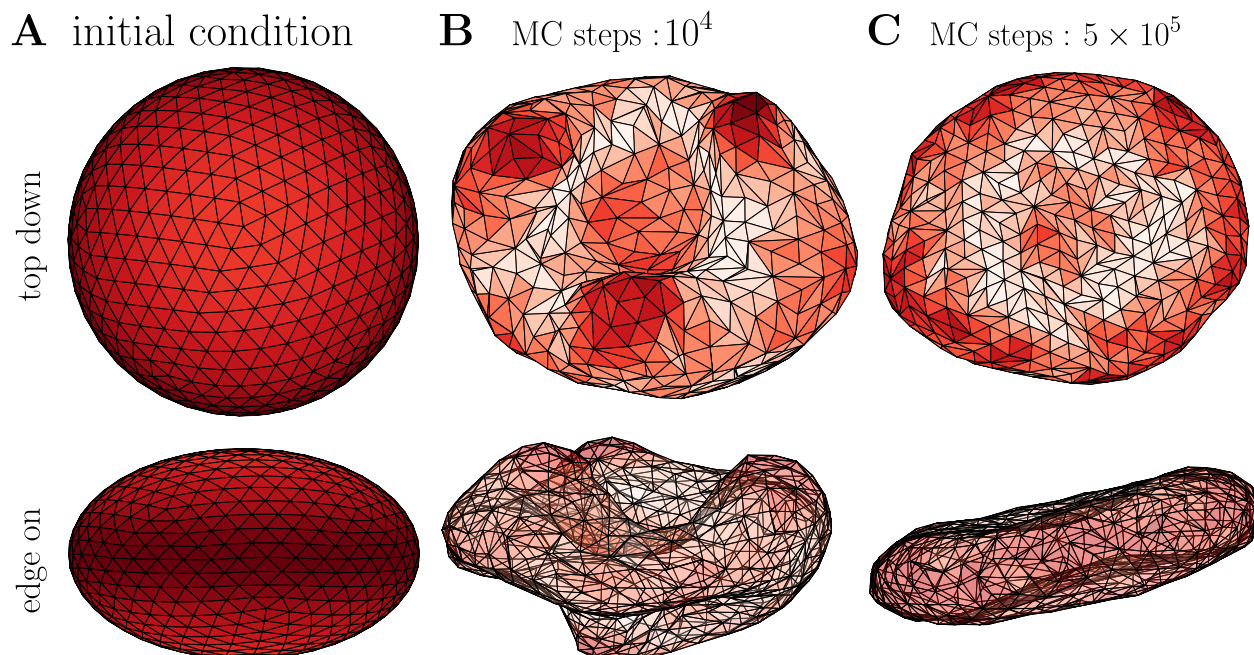


Figure 1. Result of the Monte Carlo Simulation The triangulation consisted of 642 nodes. **A:** Initial condition. A slightly oblate spheroid. **B:** Simulation that ran for 10^4 Monte Carlo steps per node (took 10 seconds). **C:** Simulation that ran for 5×10^5 Monte Carlo steps per node (took approximately 7 minutes). C is not a longer run of B but a new simulation that ran longer than B. All simulations were done on Intel i7-8650U 1.9GHz processor.

section as well as the data for fig. 1, can be found in reference [27] We use this code for several projects and have developed a robust workflow for introducing updates. New features are first used and tested by us before incorporating it into *flippy*, which makes easier to detect and eliminate problems that slip by our unit testing framework. We intend to maintain and develop the code for years to come, while feedback and contributions from users are highly encouraged.

ACKNOWLEDGMENTS

We want to thank Emma Verhulst and Ian van Vliet for their use of early versions of *flippy*. Their feedback was instrumental during the refinement of *flippy*'s interface. We also want to thank Alexander Ziepke for his feedback on the manuscript of this paper. G.D. was supported by the “BaSyC – Building a Synthetic Cell” Gravitation grant (024.003.019) of the Netherlands Ministry of Education, Culture and Science (OCW) and the Netherlands Organisation for Scientific Research (NWO).

REFERENCES

- [1] Felix Frey and Timon Idema. More than just a barrier: using physical models to couple membrane shape to cell function. *Soft Matter*, 17(13):3533–3549, 2021.
- [2] Elisa Godino, Jonás Noguera López, David Foschepoth, Céline Cleij, Anne Doerr, Clara Ferrer Castellà, and Christophe Danelon. De novo synthesized Min proteins drive oscillatory liposome deformation and regulate FtsA-FtsZ cytoskeletal patterns. *Nature Communications*, 10(1):4969, 10 2019.
- [3] Thomas Litschel, Beatrice Ramm, Roel Maas, Michael Heymann, and Petra Schwille. Beating vesicles: encapsulated protein oscillations cause dynamic membrane deformations. *Angewandte Chemie International Edition*, 57(50):16286–16290, 2018.

- [14] Jan Steinkühler, Roland L Knorr, Ziliang Zhao, Tripta Bhatia, Solveig M Bartelt, Seraphine Wegner, Rumiana Dimova, and Reinhard Lipowsky. Controlled division of cell-sized vesicles by low densities of membrane-bound proteins. *Nature communications*, 11(1), 2020.
- [15] Emeline Rideau, Rumiana Dimova, Petra Schwille, Frederik R Wurm, and Katharina Landfester. Liposomes and polymersomes: a comparative review towards cell mimicking. *Chemical society reviews*, 47(23):8572–8610, 2018.
- [16] Hongyan Yuan, Changjin Huang, Ju Li, George Lykotrafitis, and Sulin Zhang. One-particle-thick, solvent-free, coarse-grained model for biological and biomimetic fluid membranes. *Physical review E*, 82(1):011905, 2010.
- [17] Thierry Biben, Klaus Kassner, and Chaouqi Misbah. Phase-field approach to three-dimensional vesicle dynamics. *Physical Review E*, 72(4):041921, 2005.
- [18] Gerhard Gompper and Daniel M Kroll. Network models of fluid, hexatic and polymerized membranes. *Journal of Physics: Condensed Matter*, 9(42):8795, 1997.
- [19] Guillaume Gueguen, Nicolas Destainville, and Manoel Manghi. Fluctuation tension and shape transition of vesicles: Renormalisation calculations and monte carlo simulations. *Soft Matter*, 13(36):6100–6117, 2017.
- [10] Hiroshi Noguchi and Gerhard Gompper. Fluid vesicles with viscous membranes in shear flow. *Physical review letters*, 93(25):258102, 2004.
- [11] Anđela Šarić and Angelo Cacciuto. Mechanism of membrane tube formation induced by adhesive nanocomponents. *Physical review letters*, 109(18):188101, 2012.
- [12] Anđela Šarić and Angelo Cacciuto. Fluid membranes can drive linear aggregation of adsorbed spherical nanoparticles. *Physical Review Letters*, 108(11):118101, 2012.
- [13] Amir Houshang Bahrami, Reinhard Lipowsky, and Thomas R Weikl. Tubulation and aggregation of spherical nanoparticles adsorbed on vesicles. *Physical review letters*, 109(18):188102, 2012.
- [14] Casper Van Der Wel, Afshin Vahid, Anđela Šarić, Timon Idema, Doris Heinrich, and Daniela J Kraft. Lipid membrane-mediated attraction between curvature inducing objects. *Scientific reports*, 6(1), 2016.
- [15] Afshin Vahid, Anđela Šarić, and Timon Idema. Curvature variation controls particle aggregation on fluid vesicles. *Soft Matter*, 13(28):4924–4930, 2017.
- [16] Sebastian Carsten Johannes Helle, Qian Feng, Mathias J Aebbersold, Luca Hirt, Raphael R Grüter, Afshin Vahid, Andrea Sirianni, Serge Mostowy, Jess G Snedeker, Anđela Šarić, et al. Mechanical force induces mitochondrial fission. *Elife*, 6:e30292, 2017.
- [17] Gaurav Kumar, N Ramakrishnan, and Anirban Sain. Tubulation pattern of membrane vesicles coated with biofilaments. *Physical Review E*, 99(2):022414, 2019.
- [18] Miha Fošnarič, Samo Penič, Aleš Iglič, Veronika Kralj-Iglič, Mitja Drab, and Nir S Gov. Theoretical study of vesicle shapes driven by coupling curved proteins and active cytoskeletal forces. *Soft Matter*, 15(26):5319–5330, 2019.
- [19] Niels Lohmann. JSON for Modern C++, 1 2022. URL: <https://github.com/nlohmann>.
- [20] G. Gompper and D. M. Kroll. Random Surface Discretizations and the Renormalization of the Bending Rigidity. *Journal de Physique I*, 6(10):1305–1320, 10 1996. Publisher: EDP Sciences.
- [21] Mark Meyer, Mathieu Desbrun, Peter Schröder, and Alan H Barr. Discrete differential-geometry operators for triangulated 2-manifolds. In *Visualization and mathematics III*, pages 35–57. Springer, 2003.
- [22] Peter B Canham. The minimum energy of bending as a possible explanation of the biconcave shape of the human red blood cell. *Journal of theoretical biology*, 26(1):61–81, 1970.
- [23] Wolfgang Helfrich. Elastic properties of lipid bilayers: theory and possible experiments. *Zeitschrift für Naturforschung c*, 28(11-12):693–703, 1973.
- [24] Loup Verlet. Computer "experiments" on classical fluids. i. thermodynamical properties of lennard-jones molecules. *Physical review*, 159(1):98, 1967.
- [25] Nicholas Metropolis, Arianna W Rosenbluth, Marshall N Rosenbluth, Augusta H Teller, and Edward Teller. Equation of state calculations by fast computing machines. *The journal of chemical physics*, 21(6):1087–1092, 1953.

- ^[26] ECMA. Standard ecma-404 the json data interchange syntax, 2017. URL: <https://www.ecma-international.org/publications-and-standards/standards/ecma-404/>.
- ^[27] George Dadunashvili. Demo code: generating biconcave shapes using flippy, 2023. This URL links to the version of the code that was used in this paper: https://github.com/flippy-software-package/flippy/blob/71625fa4512b07a2cb4c4eeb4b6d99415ef872aa/demo/biconcave_shapes_MC/main.cpp.
- ^[28] George Dadunashvili. flippy: source code, 3 2023. Available at: <https://github.com/flippy-software-package/flippy>. DOI: www.doi.org/10.5281/zenodo.7758178.