NanoShaper 0.3.1 User Guide

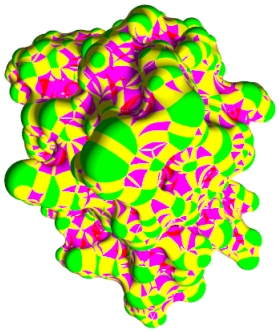
Sergio Decherchi
Fondazione Istituto Italiano di Tecnolgia

November 12, 2012

Analyzing molecular surfaces it is a key issue in biophysical modeling. The aim of NanoShaper is to measure molecular properties such as the volume, the surface area and cavities, build a Finite Difference grid for PDE solution and to triangulate the molecular surface; this is achieved by employing ray-casting [4, 2].
Ray-casting here is used as a tool to inspect, in a grid based world, the inner part of the surface. This allows to build an in/out map by which one can perform floodfill to identify cavities, estimate their volume, and fill them if requested. The software can be used also with non molecular surfaces provided that they are manifold. The algorithm can deal with analytical or meshes surfaces. The algorithm is robust to duplicated vertices/faces nad almost degenerate triangles. Volume is estimated by a triple ray-casting process using as starting points of the ray the x,y,z sides of the cube that contains the molecule; the three obtained values of the volume are averaged thus getting an highly accurate result.
In order to estimate the surface area an ad hoc Marching Cubes algorithm has been developed in order both to triangulate the surface and to accurately perform surface area estimation [4]; moreover if voids are removed the final triangulation is consistent with cavities removal. All the framework is developed in portable, expandable C++.
In this version one can load a .off/.ply triangulated mesh, a msms file [5] and build the Skin [3], Blobby [1] and the SES (Connolly) molecular surfaces.

1 What's new

In this version the 2D grid data structure for ray-casting is applied to triangles mesh with improved performance of a factor of 3x in big meshes (more than 100k triangles).
The Skin surface build-up has been optimized with a increased performance from 3x to 12x wrt NanoShaper 0.2; now Skin surface build-up is only slightly slower than Connolly surface. Ray-casting performance for skin has been also slightly improved.
NanoShaper 0.3.1 is shipped with the script setup.py; this script for Linux/Mac try to recognize your distribution install the required packages, download/patch



(surf.u3d)

Figure 1: Example of Skin surface ray-traced by PovRay and triangulated (interactive)

CGAL and compile NanoShaper all automatically.
Supported package managers are: fink (Mac), apt-get (Ubuntu/Debian), yast (Suse) and yum(RedHat/Centos). For Windows user the manual installation is mandatory; however NanoShaper is shipped with an executable for Windows.
Now new keywords are defined which allow to disable epsilon map colouring (grid cube facets), to disable status map (the map used for cavity detection) and to control the settings of the acceleration grids for ray casting and boundary grid projection. By default the dielectric map (for FD, PDE solution) is disabled thus allowing a significant reduction of memory usage.

2 Getting Started

NanoShaper can be automatically configured and compiled with the setup.py script in Linux/Mac or compiled and installed manually. For Windows user an executable is in bin directory. For optimal performance Linux is suggested. To run NanoShaper type the NanoShaper executable followed by the configuration file. Remember that NanoShaper needs at least 4 atoms to work: if you have less atoms put the missing atoms with 0 radi on the same position of the original atoms.
Here the two procedures for compiling NanoShaper.

2.1 Automatic mode for Linux/Mac

In a Linux shell, log as root and run:

```
python setup.py
```

This will try to automatically guess your distribution, download and install the necessary packages accordingly, download CGAL, compiling CGAL, patching

CGAL, configuring and compiling NanoShaper. You will find NanoShaper in the build folder.

2.2 Manual mode for any OS

In order to build the full version of the software the system must have installed CGAL (for the Skin/SES support CGAL 3.9 was used), Boost libraries and cmake; In Linux/Mac GCC is required and in Windows NanoShaper it has been tested on Visual Studio 2008/2010 but it should work also on next versions. Building the software consists in a cmake configuration step and make; additionally patching CGAL is required (for Visual Studio 2010 also Power_test_3.h is required as patch); prior to building, overwrite the files contained in CGALpatch in the CGAL include directory where the same .h are present. For simplicity and to minimize the number of packages to be installed use the following cmake command for CGAL:

```
cmake . -DWITH_examples=false -DWITH_CGAL_Qt4=false
 -DWITH_CGAL_Qt3=false -DWITH_CGAL_ImageIO=false
```

Once CGAL is compiled to build the configuration for the software go to NanoShaper build folder and type cmake ... If you are on Windows and on a 64 bit machine perform cmake by expliciting setting the current compiler (see cmake help).
On a Windows system you will get a Visual Studio project and Linux/Mac a make file. Build either typing make or using Visual Studio.
If CGAL is not present, mesh, msms and Blobby surfaces will be still available and the code should be compilable with any C++ compiler that has boost support. If neither boost is present the same functionalities will be given but using a single execution thread; in this last case the software does not depend on any external library except from STL containers.
NanoShaper can be directly used in Python: to this aim you will need the Swig package installed (Enthough Python distro already has). Rename *CMake-Lists_python.txt* into CMakeLists.txt and run cmake as in the previous case but this time inside build_python folder. This process will generate a project or a make file; by building the project/or performing make you will be end up with the following files: NanoShaper.py and _NanoShaper.pyd. You should put these files in a place where they can be reached by Python such as the same folder where you write your script: then you can use NanoShaper classes as for any other Python package by importing NanoShaper. In the folder python_example see the file example.py where the NanoShaper classes are used and the surface is visualized via MolFX.py script.

3 Surface Configuration

The configuration file has the following flags and options:

3.1 Grid parameters

Grid_scale: Real value. Specify in Angstrom the inverse of the side of the grid cubes. E.g. 2.0

Grid_perfil: Percentage that the surface maximum dimension occupies with respect to the total grid size. E.g. 80.

3.2 Maps settings

Build_epsilon_maps: Bool value. Build the epsilon and salt maps needed for an FD solver (e.g. Delphi) solver of the PB equation. By default these are disabled.

Build_status_map: Bool value. Build the status map needed for cavity detection. Enable this if you need to triangulate without Accurate_Triangulation enabled.

3.3 Surface parameters

XYZR_FileName: String value. This is the file name of the molecule to be loaded in the format xyzr. The format xyzr simply has one atom per line and each line is respectively the x,y,z coordinate and the radius.

Surface_File_Name: String value. This is the name of the surface to be loaded. For .off and .ply write the full file name, for msms file write the file name without the extension (.face,.vert).

Surface: This string specifies the possible surfaces that can be loaded/built. msms means that the surface type is of the type given by the MSMS tool [5].

mesh indicates a triangulated mesh either in .ply or .off format; skin will build the Skin surface, blobby will build the Blobby surface and ses means the Connolly surface. When an msms file is loaded the surface is autmatically converted in a .off file named msms.off. For Skin and Blobby this field is ignored.

Skin_Surface_Parameter: Real value. It is the $s \in (0, 1)$ parameter of the Skin Surface. Its default value is 0.45 that leads to a surface very similar to the SES surface. For $s = 1.0$ the surface is the convex hull of the atoms, if $s = 0.0$ the van der Waals surface is obtained.

Blobbyness: This is the blobbyness value of the Blobby surface. Its default value is -2.5 that makes it not too far from the SES.

Cavity_Detection_Filling: Bool value. If enabled cavity detection/filling is run after surface ray-casting.

Keep_Water_Shaped_Cavities: Bool value. This is an experimental feature. It will try to check the shape of cavity. If spherical the cavity will be maintained, if highly non spherical it will be removed. This flag is usefull when one wants to filter out cavities with volume higher than a water molecule but which shape is non spherical (e.g. tight long tube); this can happen in the skin surface.

Conditional_Volume_Filling_Value: Real value in cubic Angstrom. It indicates the threshold volume for which a void/cavity has to be filled or not. Cavities/voids whose volume is less than this threshold are filled. The triangulation is corrected accordingly

Accurate_Triangulation: Bool value. If enabled the surface is triangulated and it is granted that every vertex is analytically sampled from the surface. If disabled vertices are not analytically sampled but approximated. Disabling accurate triangulation will signficantly decrease both the quality of the surface and the memory requirements. If you set false consider enabling surface smoothing to increase visual quality.

Triangulation: Bool value. If enabled the surface is triangulated and saved in triangulatedSurf.off. For the Blobby surface, triangulation is always performed

and the file is saved in blobby.off; if triangulation is enabled triangulatedSurf.off represents the Blobby surface after cavity detection/removal.

3.4 Mesh settings

Check_duplicated_vertices: Bool value. If enabled both during triangulation writing and reading duplicated vertices are checked. During writing this means that duplicated vertices are removed.

Smooth_Mesh: Bool value. If enabled the surface is triangulated and then smoothed by Laplacian smoothing. Smoothing should be used to increase visual quality of the mesh if Accurate_Triangulation is false or in general when triangles quality matters such as in Boundary Element Solvers.

3.5 Acceleration Grid Settings

Max_mesh_auxiliary_grid_size: Integer value, default 100. Maximum grid size of the acceleration grid for meshes for boundary grid projections.

Max_mesh_patches_per_auxiliary_grid_cell: Integer value, default 250. Maximum number of triangles in a grid cube of the acceleration grid for boundary grid projections.

Max_mesh_auxiliary_grid_2d_size: Integer value, default 100. Maximum grid size of the acceleration grid for meshes for ray casting.

Max_mesh_patches_per_auxiliary_grid_2d_cell: Integer value, default 250. Maximum number of triangles in a grid cube of the acceleration grid for ray casting.

Max_ses_patches_auxiliary_grid_2d_size: Integer value, default 50. Maximum

grid size of the acceleration grid for ses ray casting.

Max_ses_patches_per_auxiliary_grid_2d_cell: Integer value, default 400. Maximum number of patches in a grid cube of the acceleration grid for ray casting.

Max_ses_patches_auxiliary_grid_size: Integer value, default 100. Maximum grid size of the acceleration grid for boundary grid projections.

Max_ses_patches_per_auxiliary_grid_cell: Integer value, default 400. Maximum number of patches in a grid cube of the acceleration grid for boundary grid projections.

Max_skin_patches_auxiliary_grid_size: Integer value, default 100. Maximum grid size of the acceleration grid for skin boundary grid projections.

Max_skin_patches_per_auxiliary_grid_cell: Integer value, default 400. Maximum number of patches in a grid cube of the acceleration grid for boundary grid projections.

Max_skin_patches_auxiliary_grid_2d_size: Integer value, default 50. Maximum grid size of the acceleration grid for skin ray casting.

Max_skin_patches_per_auxiliary_grid_2d_cell: Integer value, default 400. Maximum number of patches in a grid cube of the acceleration grid for ray casting.

3.6 File storage and others

Save_Status_map: Bool value. If enabled grid info is saved together with cavities info. It must be true in order to visually inspect grid points in cavities.

Save_eps_maps: Bool value. Save the epsmap needed by a Finite Difference PDE solver.

Save_PovRay: Bool value. If enabled the Skin surface is converted into a file named skin.pov ready to be ray-traced by PovRay program. The analytical Skin surface is saved in the PovRay file and not its triangulation; this is done to get the best visual effect. Conservatively the camera position is quite far from the molecule, so it may need manual adjustments.

Number_Thread: Integer value. Here you explicitly set the number of execution threads to optimize the execution time or to use only a subset of your cores. As a rule thumb optimal performances can be obtained by setting as number of threads the number of cores multiplied by 4; this should grant full CPU usage. If this keyword is not present NanoShaper will use a number of threads equal to the number of physical cores, remember that this choice is not optimal.

4 Extending the framework

The architecture of NanoShaper is in plain C++ and allows easy extensibility. Most of the algorithms are in the Surface class in which also the parallelization is performed. In order to expand the framework one has to extend the Surface class; to this aim two strategies are possible: in the first case one re-implements the method getSurf and the Surface class facilities are not used, in a second scenario one wants to use Surface class facilities.
In this latter case the following pure virtuals must be implemented:

- bool build()
- bool save(char* fileName)
- bool load(char* fileName)
- void printSummary()
- bool getProjection(double p[3],double* proj1,double* proj2,double* proj3,double* normal1,double* normal2,double* normal3)
- void getRayIntersection(double p1[3],double p2[3],vector<pair<double,double*>>& intersections,int thdID)

The functions save, printSummary and getProjection are only formally mandatory and can be safely implemented as empty functions; in particular getProjection is used when NanoShaper is interfaced with DelPhi and it is not necessary for triangulation. Instead the only core routine is the getRayIntersection routine. If this routine is implemented then, from Surface inheritance, one will

get: volume estimation, surface area estimation, cavity detection, cavities removal, cavities-removal-consistent surface triangulation. The ray casting will be automatically parallelized by Surface class; one has to assure that getRayIntersection routine is thread safe, that is rays can be casted in parallel without compromising consistency; note that no concurrent writing should be requested. If you need concurrent writing (such as concurrent cout) a boost mutex in Surface class is available.
After the surface is implemented the user has to modify the main routine in order to properly recognize the newly introduced surface. For more information about each single function consult the doxygen functions documentation; for any /bug fix/help/improvement/ contact the Author at sergio.decherchi@iit.it .

5 Examples

In the examples folder some examples molecules can be run; moreover for convenience a precompiled win32 NanoShaper is given.
The first one is given by the configuration file conf.prm; this file loads the atoms file barstar.xyzr and compute the Skin surface. Playing with this file you can change the shape of the surface by changing the Skin surface parameter or using the Blobby surface instead of the Skin.
The file bunny_conf.prm shows how a standard mesh can be processed. In order to load the surface two *virtual* atoms are defined which represents the two opposite points of the bounding cube of the mesh; this is done in order to build a grid consistent with the mesh. Other two dummy atoms are added on the same positions because NanoShaper needs at least four atoms to work.
In the folder python_example the file example.py shows how to interact with NanoShaper classes from Python.

Acknowledgments

The Author would like to thank Nico Kruithof for useful discussions and code snippets about the CGAL implementation of the Skin surface and Marco Attene from IMATI for providing the .ply file reader.
This work is supported by NIGMS, NIH, grant number 1R01GM093937-01.

References

[1] Y. Zhang G. Xu C. Bajaj. Quality meshing of implicit solvation models of biomolecular structures. *Computer Aided Geometric Design*, 23:510530, 2006.

[2] Sergio Decherchi and Walter Rocchia. A general and robust ray casting based algorithm for triangulating surfaces at the nanoscale. *submitted*.

[3] H. Edelsbrunner. Deformable smooth surface design. *Discrete and Computational Geometry*, 21(1):87–115, 1999.

[4] M. Phillips, I. Georgiev, A.K. Dehof, S. Nickels, L. Marsalek, H.-P. Lenhof, A. Hildebrandt, and P. Slusallek. Measuring properties of molecular surfaces using ray casting. In *Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, pages 1 –7, april 2010.

[5] M. F. Sanner, A. J. Olson, and J.C. Spehner. Reduced surface: An efficient way to compute molecular surfaces. *Biopolymers*, 38:305320, 1996.