For the online version check:
https://github.com/weria-pezeshkian/FreeDTS/wiki/Workshop-2024

FreeDTS version 2 Tutorials (2024)

Welcome to the FreeDTS version 2 tutorials. This guide is here to introduce you to FreeDTS Version 2 and show you how to simulate membranes at the mesoscale. In a series of tutorials, you'll progressively learn to set up, run, and analyze FreeDTS simulations, beginning with basic examples and advancing to more complex scenarios. The final tutorial even covers modifying the FreeDTS source code for custom simulation needs.

List of the tutorials

- Tutorial I: Vesicle
- Tutorial II: Framed membrane
- Tutorial III: Proteins
- Tutorial IV: Custom simulation

Required Software

For visualisation: preferably ParaView, or VMD

Getting Started

To begin, download the source code, navigate to the `version_2` folder, and compile the executables in one step:

```
git clone git@github.com:weria-pezeshkian/FreeDTS.git
cd FreeDTS/version_2
./compile.sh
```

This command sequence will generate three executables:

`DTS`: For running simulations. `GEN`: For generating triangulated structures. `CNV`: For converting files between formats. With these files, you can proceed with the tutorials below.

Vesicle

The simplest simulation you can run with FreeDTS is a spherical vesicle. To set this up, you first need to generate a triangulated surface (TS) file in the shape of a tetrahedron. This can be done using the `GEN` binary. binary. The following command will create a tetrahedron TS file in `*.q` format in a box with size of $100 \times 100 \times 100$.

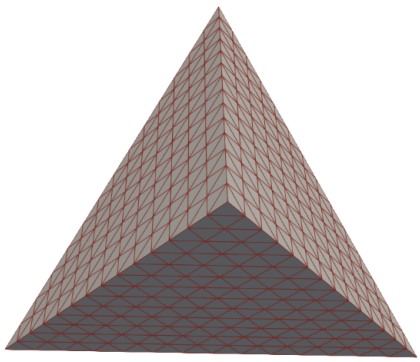
```
$path/GEN -box 100 100 100 -type tetrahedron -N 18 -o tetra.q
```

To visualise the created configuration, you can use CNV to generate **ParaView** or **VMD** compatible files as:

```
$path/CNV -in tetra.q -o conf.vtu
```

or

```
$path/CNV -in tetra.q -o conf.gro
```



Next, this file name should be added to a topology file name with top extension (here it is named top.top) as:

```
echo "tetra.q 2" > top.top
```

The number in front of the file gives an id to the entire mesh (the value is not important but needs to be unique if multiple files are included).

Next provide the simulation input parameters in a file with `*.dts` extension. This file can be empty by default, which would lead to a simulation using the default variables. However, for this specific tutorial, the content of the input file should be as follows:

```
Integrator_Type          = MC_Simulation
Set_Steps                = 1 20000
Temperature              = 1 0
Kappa                    = 10 0 0
VertexArea               = 0 0 0 0
VertexPositionIntegrator = MetropolisAlgorithmOpenMP 1 1 0.05
AlexanderMove            = MetropolisAlgorithmOpenMP 0.2
;so that the volume/area/GC will be printed in the out
VolumeCoupling           = SecondOrder 0 0 0.77
GlobalCurvatureCoupling  = HarmonicPotential 0 0.3
TotalAreaCoupling        = HarmonicPotential 0 0.34
;output managment
VisualizationFormat      = VTUFileFormat VTU_F 1000
```

```
NonbinaryTrajectory      = TSI TrajTSI 1000
TimeSeriesData_Period    = 100
Restart_Period           = 1000
```

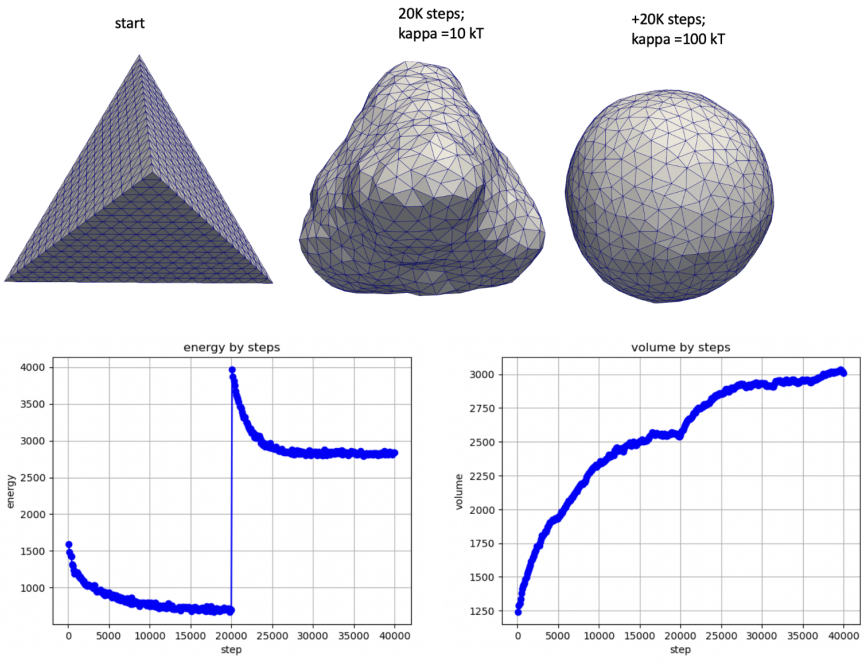
Next run a simulation using

```
$path/DTS -in input.dts -top top.top
```

After the first run, in the input file, increase the kappa to 100 i.e., `Kappa = 100 0 0` and restart the simulations for another 5000 steps using

```
$path/DTS -in input.dts -top top.top -restart dts.res -e 40000
```

Using **ParaView**, you can visualize the files in the `VTU_F` folder and observe the evolution of the system. Also, `dts-en.xvg` contains information about system energy, volume, total area and total mean curvature.



Due to the limited time available during the workshop, we will consider this section of the tutorial as the conclusion. However, there are additional interesting features that you can explore later. For the next tutorial click here. * **One can use the** CNV **script to convert the last frame of the above run into a** TS** file for different runs, which we will use in Tutorial 4. This can be done as follows:

```
$path/CNV -in TrajTSI/dts40.tsi -o vesicle.q
```

One of the interesting features of vesicle shapes is the formation of multi-spherical structures. These structures can be obtained by modulating two macroscopic parameters: vesicle volume (representing the effect of osmotic pressure) and membrane curvature. Curvature can be either global or local. To induce local membrane curvature in FreeDTS, you only need to modify the last number in front of `kappa` keyword in the input file.

`Kappa = 10 0 0` For global curvature, one needs to couple the system to a fixed global curvature algorithm, which is specified as follows in the input file.

```
GlobalCurvatureCoupling  = HarmonicPotential 60 0.3
```

Fixed volume can also be obtained by coupling the system energy to second order potentials.

```
VolumeCoupling           = SecondOrder 0 10000 0.7
```

By tuning these parameters, one can get multitude of different shapes. For examples, it is known that for reduced volume of 0.4 i.e., `VolumeCoupling = SecondOrder 0 10000 0.7`, in absence of membrane curvature, stomatocyte shape will form. Also for below options, dumbbell shape structure will form.

```
Kappa = 20 0 0
GlobalCurvatureCoupling  = HarmonicPotential 60 0.3
VolumeCoupling           = SecondOrder 0 10000 0.7
```

Rapid changes in macroscopic parameters can drive the system into configurations that take many simulation steps to escape. To avoid this, several non-equilibrium command can be used (See the manual for more details).

Framed membrane

The simplest model of membranes is a fluid elastic surface with Periodic Boundary Conditions (PBC). This type of simulation can be performed using FreeDTS. To begin, we first need to generate a flat TS file, which can be done using the command below.

```
$path/GEN -box 30 30 100 -type flat -o flat.q
```

This creates a TS file with a box size of $30 \times 30 \times 100$. Next, this file name should be added to a topology file name with top extension (here it is named top.top), and we should add the below line in the top file.

```
echo "flat.q 2" > top.top
```

To run a simulation, an input file (with dts extension) is required to define the simulation parameters (see below box and the input file format section).

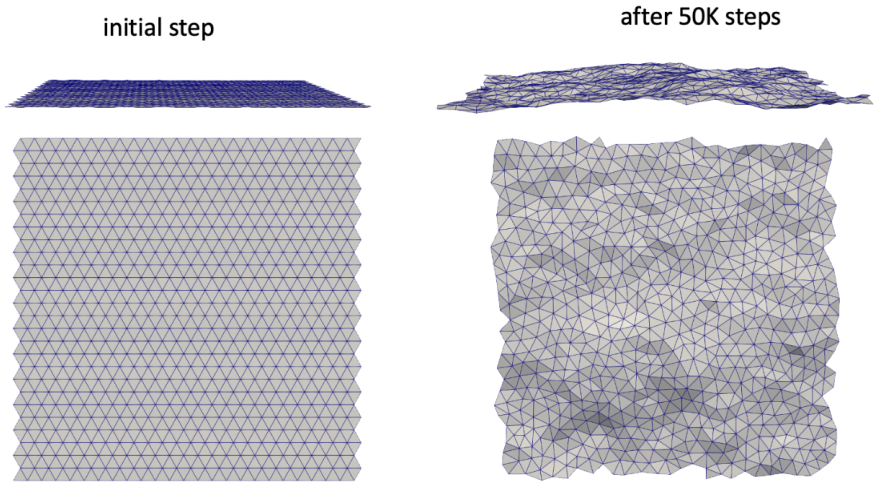
```
Integrator_Type          = MC_Simulation
Set_Steps                = 1 50000
```

```
Temperature               = 1 0
Kappa                     = 10 0 0
VertexArea                = 0 0 0 0
; move methods
VertexPositionIntegrator = MetropolisAlgorithmOpenMP 1 1 0.05
AlexanderMove             = MetropolisAlgorithmOpenMP 0.2
Dynamic_Box               = IsotropicFrameTensionOpenMP  1 0 XY
; to print the total area
TotalAreaCoupling         = HarmonicPotential 0 0.34
;output management
VisualizationFormat       = VTUFileFormat VTU_F 1000
NonbinaryTrajectory       = TSI TrajTSI 1000
TimeSeriesData_Period     = 100
Restart_Period            = 1000
```

Next run a simulation using

```
$path/DTS -in input.dts -top top.top
```

The resulting snapshots will be something as below

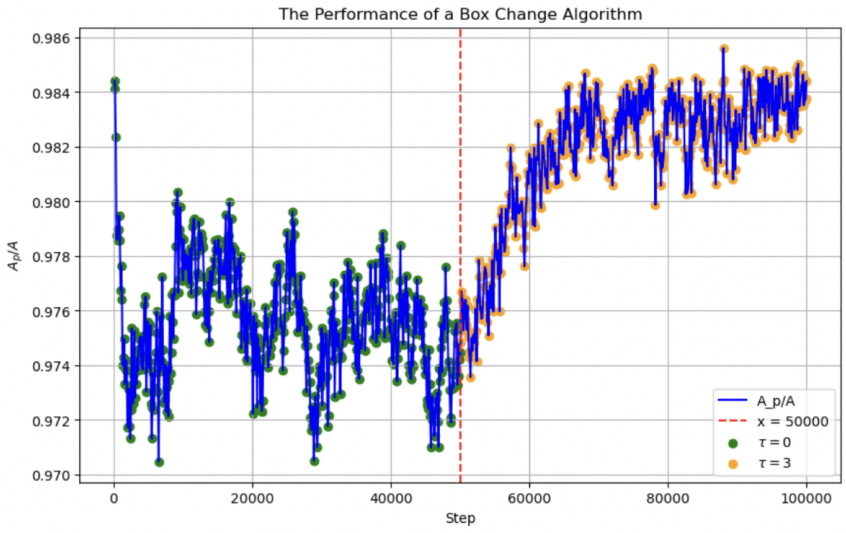


Next, increase the frame tension to 3 $[kT/d^2]$ and restart the simulation. For this, you need to change the frame tension coupling line.

```
Dynamic_Box               = IsotropicFrameTensionOpenMP  1 3 XY
```

```
$path/DTS -in input.dts -top top.top -restart dts.res -e 100000
```

Due to the limited time available during the workshop, we will consider this section of the tutorial as the conclusion. However, there are additional interesting



features that you can explore later. For the next tutorial click here. ***

Many features of fluid membranes can be studied using the framed membranes, among which is universal behaviour of membrane undulation spectrum, this can be done by analysing the trajectories in the *TrajTSI* folder, for some examples see soft membranes and membranes with inclusions. One can also study how the ration of projected area with total area changes with bending rigidity and membrane tension.

Proteins

In this tutorial, we will simulate a flat membrane covered by 10% curvature-inducing proteins. In **FreeDTS**, proteins are modeled as *inclusions*. The type and the model parameters of the corresponding inclusion must be defined in the input file (`input.dts`). We build the input parameters for this tutorial using the outputs from the previous tutorial.

In case you skipped the previous tutorial, you can download the files here.

First, we will take output of the previous simulation from step 50k (since this was the last step for tension = 0) can convert this to a `*.q` file for input.

```
$path/CNV -in $path2previous/TrajTSI/dts50.tsi -o flat.q
```

then

```
echo "flat.q 2" > top.top
```

we also need the input file

```
cp  $path2previous/input.dts .
```

Next we need to include the inclusion information to the input file. This consist of 2-3 steps (step 2 can be omitted in some situations). ### Step 1: defining inclusion type Below is an example of defining two inclusion types. The first seven model parameters belong to inclusions when they are on surface vertices. The last four parameters are only relevant when there are boundary vertices, which can be ignored here.

```
INCLUSION
Define 2 Inclusions
SRotation Type   K    KG  KP  KL  C0    C0P C0L  lambda   lkg   lkn    cn0
0      Pro1      10   0   0   0   0.6    0   0
0      Pro2      20   0   0   0   -0.4   0   0
```

Step 2: Creating inclusions:

The number of the inclusions in a simulation can be defined as below.

```
GenerateInclusions
Selection_Type Random
TypeID     1     2
Density    0.1   0.0
```

This will cover 10% of the vertices with inclusions of type 1 (defined in previous section) and 0% of type 2.

Please Note: A simulation can be started using a *restart* file (a binary file with `*.res` extension) or a *tsi* file (a text file with `*.tsi` extension). In this case, the section of the input file will be ignored by **FreeDTS** as both files contains information about inclusions and FreeDTS uses the amount of inclusions defined in these files.

Step 3: Defining inclusion interactions

We need to include a line to define how to update the inclusions. Although **FreeDTS** uses its default updating schemes, it is always better to include it in the input file for better control. This can be included as:

```
InclusionPoseIntegrator = MetropolisAlgorithmOpenMP   1 1
```

Inclusion-inclusion interactions are also defined in the input file. For interaction type 1 $E_{ij} = -A + B\cos(N\Theta)$, it will be as:

```
Inclusion-Inclusion-Int
i    j    ftype   N     A     B
```

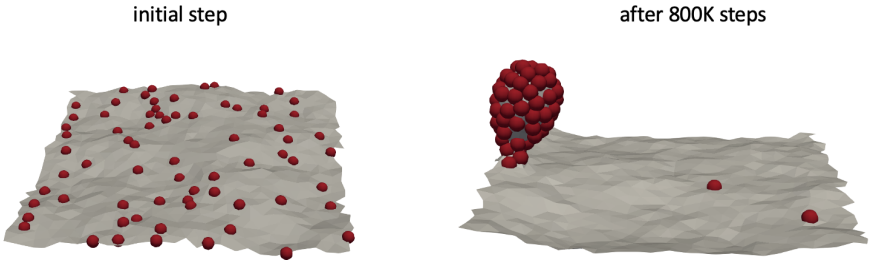
In this dts file, this will be defined as below. Please note: this section MUST always be the last section of the input file (*.dts file).

```
Inclusion-Inclusion-Int
1    1    1   0     2     0.0
1    2    1   0     0     0.0
2    2    1   0     0     0.0
```

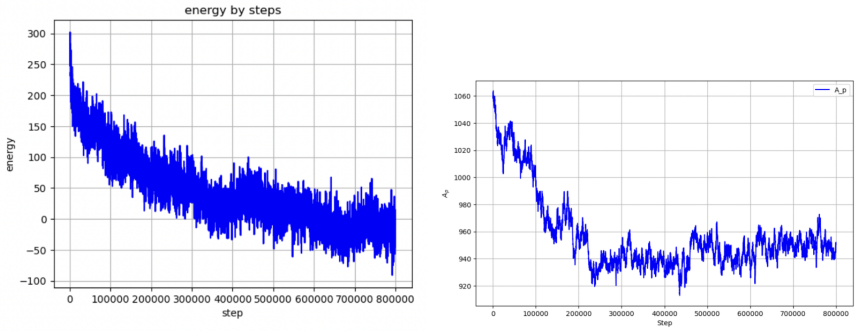
Now, everything is read and you can start the simulation.

```
$path/DTS -in input.dts -top top.top -e 800000
```

The run may take a long time, depending on your computer. However, one should obtain the snapshots shown below. Considering the energy graph it is



clear that the system has not reached equilibrium.



Custom simulation

This tutorial aims to guide you in modifying the FreeDTS source code to create custom functionality. While the process can be applied to various applications,

here we'll focus on pulling three tubes from different sides of a membrane by applying a force to three user-defined vertices. Given three vertices labeled 1, 2, and 3, we define a force on each vertex (in addition to the bending and all the other forces defined within FreeDTS framework) based on the positions of the other two as follows

$$\mathbf{F}_i = K\frac{\mathbf{X}_i - \mathbf{X}_j}{|\mathbf{X}_i - \mathbf{X}_j|} + K\frac{\mathbf{X}_i - \mathbf{X}_k}{|\mathbf{X}_i - \mathbf{X}_k|}$$

This will cause the vertices to repel one another, where K will determine the strength of the force.

To achieve this, we need to modify the source code by updating the UserDefinedForceonVertices class, which is declared in the `UserDefinedForceonVertices.h` file and implemented in the `UserDefinedForceonVertices.cpp` file. Specifically, we will modify and complete the public function `void Initialize();` and the private function `CalculateForce(vertex *pv);`.

The void `Initialize()` is executed in the `State` class when all other required classes are created, and the mesh is initialized. This function is used to initialize member variables that cannot be fully initialized in the constructor. In the `UserDefinedForceonVertices.cpp` file, you can see that the `Initialize()` function is as follows:

```
void UserDefinedForceonVertices::Initialize(){

    std::vector<std::string> data = Nfunction::Split(m_Inputs);
    m_pallV = m_pState->GetMesh()->GetActiveV();
}
```

`m_Inputs` is a member variable of type string that contains all the inputs provided to this class through the input file (`input.dts`). First, we convert `m_Inputs` into a vector of strings and store it for use in data initialization. `m_pallV` is a vector of vertex pointers that contains all vertices.

Our objective is to specify three vertices and a force constant, then apply the repulsive force described in equation (1) to these vertices. To achieve this, we need to define three member variables of type `vertex*` and one member variable of type `double` for fast access in the `CalculateForce(vertex *pv);` function. These member variables can be declared as shown below in the UserDefinedForceonVertices.h file:

```
private:
    vertex *m_pV1;
    vertex *m_pV2;
    vertex *m_pV3;
    double m_K;
```

Then we can modify `Initialize()` function as:

```
void UserDefinedForceonVertices::Initialize(){

    std::vector<std::string> data = Nfunction::Split(m_Inputs);
    m_pallV = m_pState->GetMesh()->GetActiveV();
    m_pV1 = m_pallV[Nfunction::String_to_Int(data[0])];
    m_pV2 = m_pallV[Nfunction::String_to_Int(data[1])];
    m_pV3 = m_pallV[Nfunction::String_to_Int(data[2])];
    m_K = Nfunction::String_to_Double(data[3]);
}
```

Now it is time to complete the CalculateForce(vertex *pv); function. This function takes a pointer to a specific vertex, calculates the force acting on that vertex, and returns the computed force. A straightforward implementation of the force described in equation (1) can be written as follows:

```
Vec3D UserDefinedForceonVertices::CalculateForce(vertex *pv) {

    if(m_pV1 != pv && m_pV2 != pv && m_pV3 != pv){
        return 0;
    }

    Vec3D X0 = pv->GetPos();
    Vec3D X1 = m_pV1->GetPos();
    Vec3D X2 = m_pV2->GetPos();
    Vec3D X3 = m_pV3->GetPos();
    Vec3D f1 = X0 - X1;
    Vec3D f2 = X0 - X2;
    Vec3D f3 = X0 - X3;
    f1.normalize();
    f2.normalize();
    f3.normalize();
    Vec3D f = f1 + f2 + f3;
    f = f*m_K;
    return f;
}
```

The `if` condition ensures that the given vertex is one of the selected vertices; otherwise, the force acting on it will be zero. Next, we calculate the distance differences with three vertices. This is because we do not know which of the `m_pVi` vertices corresponds to `pv`. Therefore, we calculate the distances for all three vertices. However, the self-distance will be zero and will not contribute to the total force.

Now that both functions are completed, close the files and compile the `UserDefinedForceonVertices` class first to make sure there is no error.

```
g++ -c -O3 -std=c++11 UserDefinedForceonVertices.cpp
```

If there are no errors, compile the entire code using the `compile.sh` script.

Everything should now be ready to perform simulations. First, we need a topology file to run. You can use the topology file from tutorial 1 or download a spherical TS file vesicle.q. For the input file, you can use the same one as in tutorial 1, with the addition of the following line: `ForceOnVertices  = User 85 437 256 1000`. This line defines the inputs for the `UserDefinedForceonVertices` class, and everything after the word `User` will be stored in the `m_Inputs` variable of the `Initialize()` function. Therefore, the first three numbers represent the IDs of the three vertices in `vesicle.q` and the last number specifies the force constant (K).

Below you can copy the whole input file for your run.

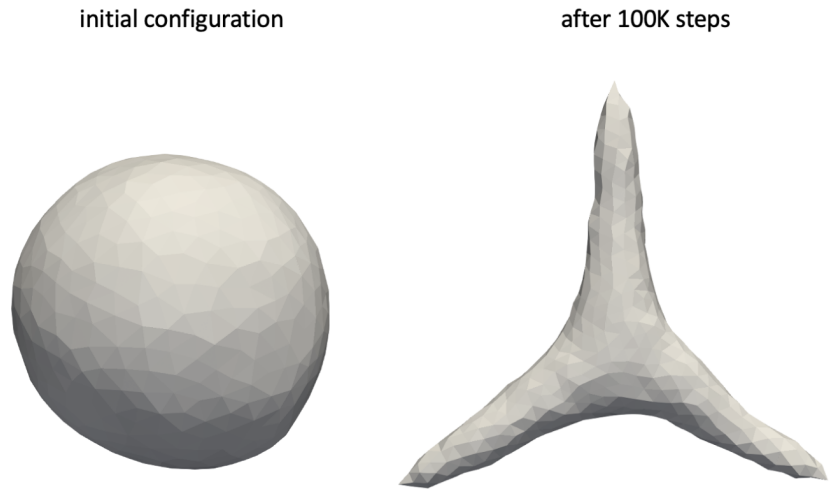
```
Integrator_Type          = MC_Simulation
Set_Steps                = 1 100000
Temperature              = 1 0
Kappa                    = 10 0 0
VertexArea               = 0 0 0 0
VertexPositionIntegrator = MetropolisAlgorithmOpenMP 1 1 0.05
AlexanderMove            = MetropolisAlgorithmOpenMP 0.2
;so that the volume/area/GC will be printed in the out
VolumeCoupling           = SecondOrder 0 100 0.8
GlobalCurvatureCoupling  = HarmonicPotential 0 0.3
TotalAreaCoupling        = HarmonicPotential 0 0.34
;output managment
VisualizationFormat      = VTUFileFormat VTU_F 1000
NonbinaryTrajectory      = TSI TrajTSI 1000
TimeSeriesData_Period    = 100
Restart_Period           = 1000
ForceOnVertices  = User 85 437 256 1000
```

Note, you might want to change the IDs of the vertices to select vertices from different corners of the mesh. You can do this by converting the TS file to `gro` using **CNV** binary file.

```
$path/DTS -in input.dts -top top.top
```

The output should be something like below snapshot

Technical details for interested users. The `UserDefinedForceonVertices` class is derived from the `AbstractForceonVertices` base class (see below for the UserDefinedForceonVertices class declaration). In the `State` class, a pointer to an `AbstractForceonVertices` object is dynamically cast to a `UserDefinedForceonVertices` pointer using polymorphism. The first keyword in the input file, i.e., *ForceOnVertices*, instructs **FreeDTS** to create an instance of the `AbstractForceonVertices` class. The second keyword, i.e., *User*, specifies that this instance should be dynamically cast to a



`UserDefinedForceonVertices` instance. These keywords are defined in the `GetDerivedDefaultReadName()` function. If you want to modify these names, you can easily change them by editing the `UserDefinedForceonVertices` and `AbstractForceonVertices` classes. In **FreeDTS**, most of the functionality follows a polymorphism-based design pattern. Base class names are prefixed with 'Abstract', and these base class pointers are cast to derived class instances within the State class (for each base class, there can be multiple derived classes). This structure allows for extensibility, enabling users to implement new simulation methods or alternative update algorithms by simply creating new derived classes and corresponding files.

In the `UserDefinedForceonVertices`, `Energy_of_Force(vertex *p, Vec3D dx);` is a public function and calculate an energy associated with a vertex move. This function is called by all the vertex position updating schemes.

```
class UserDefinedForceonVertices : public AbstractForceonVertices{
public:
    UserDefinedForceonVertices(State *pState, std::string inputs);
    ~UserDefinedForceonVertices();
    double Energy_of_Force(vertex *p, Vec3D dx);
    std::string CurrentState();
    inline  std::string GetDerivedDefaultReadName()  {return "User";}
    inline static std::string GetDefaultReadName() {return "User";}


    void Initialize();
private:
    Vec3D CalculateForce(vertex *pv);
```

```
    std::string m_Inputs;
    State *m_pState;
    std::vector<vertex*> m_pallV;

};
```
