Institute for Microelectronics

Simulation of Semiconductor Device Fabrication

Report for Exercise 1

Author:
Mario Hiti, 01327428

Submission: April 26, 2021

Contents

0 Project structure

Workspace

The whole project is coded in C++11 all relevant source files are located in */src*. To build and run the workspace use: [1]

```
cd Ex1
make
./out/build/grid [parameters]
```

Generated files such as binaries and grid data is located in */out*.
Raw data of all plots and images presented in this report is available in */results*

Furthermore a set of unit tests have been written during development which are located in */test*. To start all tests run:

```
make test
```

However code concerning tests is not relevant for the grading of the assignment.

System requirements:

- Linux Platform
- Compiler supporting C++11 or higher
- ParaView (for viewing results)

Software

The project consists of 3 important classes:

- **Vector3D:** A custom datastructure that represents 3D vectors and supports various operations that are commonly used in graphical computation (min, max, norm, etc.)
- **SDF:** is an abstract class that represents the signed distance function and supports vector calculus operations (gradient, curvature, etc). To create an SDF object this class has to be extended with shape specific variables and the SDF::eval() method has to be overloaded with the actual SDF algorithm. Two classes are derived for this assignment: SDF_Sphere and SDF_Rectangle.
- **Grid:** is the representation of the 2D simulation domain. It holds all simulation data which can be exported to ParaView. Furthermore it contains a reference to 1 SDF object

Combining these classes results in the following API:

Algorithm 1: Example usage

```
SDF * sdf = new SDF_Sphere(argv); // create sphere
Grid grid; // create grid
grid.add_surface(sdf); // Add Sphere to grid
grid.update(); // calculate signed distance values for the grid
grid.write(out/sphere_t0.vtk);    // create file for paraview
grid.advance_simple(1,1); // advance time by calculating the volocity field
grid.update();  // calculate values with the updated velocity field
grid.write(out/sphere_t1.vtk);
```

[1]Make also calls the program once to check if it runs

1 Dense Grid Signed Distance Function (SDF)

1.1 Example usage

```
cd Ex1
./out/build/grid 20 10 0.01 Rectangle 5 5 8 8 rprp out/grid.vtk
>> Adding Rectangle to grid...
>> Done
```

Expected result: The program creates a grid of size 20x10 and an axis parallel rectangle with one corner at (5,5) and another corner at (8,8). Points are uniformly spread with a spacing of 0.01. Boundaries are reflective along the horizontal and periodic along the vertical axis. The final result is then written to *out/grid.vtk*

1.2 Basic algorithm structure

1.2.1 Simulation Domain and Data structures

Although the simulation domain is two-dimensional all components (except the grid data structure) are implemented using 3D coordinates. This is because the assignment specifies a "Sphere" as one of the SDF-objects. After further clarification a 2D circle would have been enough but by that time too many changes would have been necessary. This means that many operations require a redundant 0 as a z-coordinate but the underlying calculations are the same. Furthermore, taking the cross-section of a 3D sphere is essentially the same as just drawing a 2D circle.

The grid is created according to the first three command line arguments. While x and y size is variable the spacing is uniform in both directions. The total number of points created is:

$$n = \frac{size\_x}{spacing + 1} \cdot \frac{size\_y}{spacing + 1}$$

Grid points are stored in as `std::vector<std::vector<double>>`. STL vectors are chosen because the number of grid points is constant and we require fast random access (O(1) complexity). While the underlining points are stored sequentially in memory the syntax of STL-vectors allows for quasi-2D access of grid points (i.e. data[x][y]). Note that the indices do not correspond to real coordinates but the index within the data structure. To calculate the real-world position the index has to be multiplied by the spacing

$$\begin{pmatrix} coord_x \\ coord_y \end{pmatrix} = \begin{pmatrix} index_x \cdot spacing \\ index_y \cdot spacing \end{pmatrix}$$

For Task 3 a velocity field is required which uses the same data structure.

1.2.2 Signed distance functions

SDF-Objects do not hold any data except parameters essential to describing the surface (type, radius, position, etc.). SDF is an abstract base class that supports vector calculus operations (gradient, surface normal, curvature) since they are indifferent to the type of object. To implement a specific object this class has to be extended with necessary parameters and an eval() function has to be defined.

Algorithm 2: Declaration of an SDF object

```
class SDF_Sphere : public SDF
{
public:
    SDF_Sphere(char **argv);

    // object specific variables
    Vector3D m_center;
    double m_radius;

    // overloaded evaluation function
    double eval(const Vector3D &);
};
```

Two different SDFs are implemented which can be configured via the command line arguments. The type is specified using a case sensitive string (Sphere or Rectangle) followed by 4 arguments. Both integer and floating points are valid.

- **Sphere:** The first 3 arguments correspond to the x/y/z coordinates of the center. The 4th argument specifies the radius.

  The SDF for a sphere is rather simple due to its rotational symmetry. To calculate the SDF value of a given point first we need to calculate the distance to the center. Next we have to subtract the radius to get the distance to the surface of the sphere.

  Algorithm 3: evaluation function for the sphere

  ```
      double SDF_Sphere::eval(const Vector3D & pos)
      {
          Vector3D v = pos - m_center;    // Distance to center
          return v.norm_l2() - m_radius;
      }
  ```

- **Axis-parallel rectangle:** The SDF for a rectangle (or quad) is rather complex. However for this assignment the quad lies on the x-y plane and all edges are parallel or orthogonal to the axis. This results in a much simpler evaluation fucntion (Alg. 4)

  Algorithm 4: evaluation function for the axis-parallel rectangle

  ```
  double SDF_Rectangle::eval(const Vector3D & p)
  {
      double dx =  max(m_minCorner.m_x - p.m_x, p.m_x - m_maxCorner.m_x);
      double dy =  max(m_minCorner.m_y - p.m_y, p.m_y - m_maxCorner.m_y);

      if((dx < 0) && (dy < 0))
      {
          return min(0, max(dx, dy));
      }

      if((dx > 0) && (dy > 0))
      {
          return sqrt(dx * dx + dy * dy + p.m_z * p.m_z);
      }
      return max(dx, dy);
  }
  ```

  First the normal distance to all edges is calculated. Taking the max of the distance to x and y borders respectively gives us 2 distances to the closest edge. Now there are 3 different cases:

1. If both values are negative then the point is inside of the surface. $\Longrightarrow$ take the distance to the closest border
2. If both values are positive the closest point on the surface is a corner $\Longrightarrow$ calculate the euclidian distance to the corner
3. If one is positive and one is negative the point is outside of the surface but closest point is not a corner.

Note that both SDF evaluation functions require real world coordinates as arguemnts.

1.2.3 Boundary Conditions

By invoking Grid.update() the program iterates over all grid points and calculates the respective signed distance values. Depending on the boundary conditions additional points outside of the grid are calculated.

Both reflective and periodic boundary conditions have been implemented and can be configured using a 4 letter string as a command line argument. Every letter corresponds to a edge of the grid. The mapping is:

$$[BOTTOM|RIGHT|TOP|LEFT]$$

Fore example the string "rprp" corresponds to reflective BC in the vertical and periodic BC in the horizontal direction.

Grid.update() initially calculates the singed distance for a given point and then repeats that calculation with shifted coordinates depending on the chosed border types. The final result is the smallest of all calculated values.

Algorithm 5: Calculating the signed distance value for a grid point with respect to boundary conditions. In order to check all cases a total of 8 if-conditions are necessary

```
double center_val = m_sdf->eval({x * m_delta, y * m_delta, 0});
vector<double> values(9, center_val);

// Shifting a point from the bottom
if(m_border_types[BOTTOM] == BORDER_PERIODIC)
    values[BOTTOM] = m_sdf->eval({
        (x * m_delta),
        (y * m_delta) - m_y_size,
        0
    });

// Shifting a point from the bottom left
if( (m_border_types[BOTTOM] == BORDER_PERIODIC) &&
    (m_border_types[LEFT] == BORDER_PERIODIC))
        values[BOTTOM_LEFT] =  m_sdf->eval({
            (x * m_delta) + m_x_size,
            (y * m_delta) - m_y_size,
            0
        });

//  keep only the smallest value
m_data[x][y] = minElement(values);
```

Alg. 5 shows the calculation for a point that is shifted from the bottom and from the bottom left. Values written in caps-lock correspond to magic numbers and are globally defined using preprocessor directives which can be found in *common.h*.

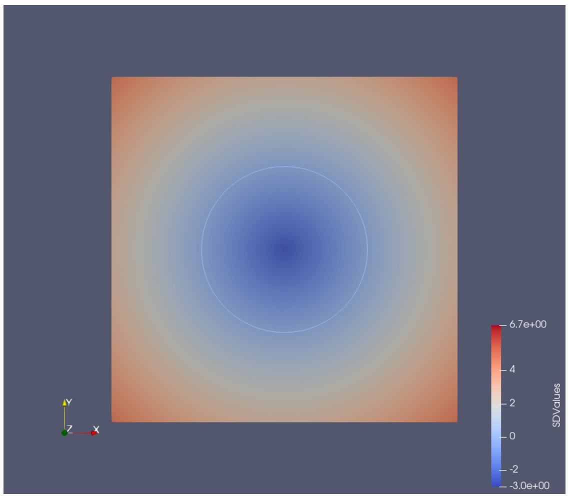
The second if-condition is required if the Surface touches a corner of the domain with boundary conditions on all sides. Furthermore the check has to be repeated for all other directional combinations thus up to 9 different values are calculated. However we only need to keep the shortest distance to any surface including projected surfaces. So we keep only the smallest of all distances.

Computationally there is no difference between reflective boundary conditions and no boundary conditions. Therefore we only need to check for periodic ones. However this is no longer true once we require values from outside the grid. For example when calculating the derivative of the SDF on a boundary point.

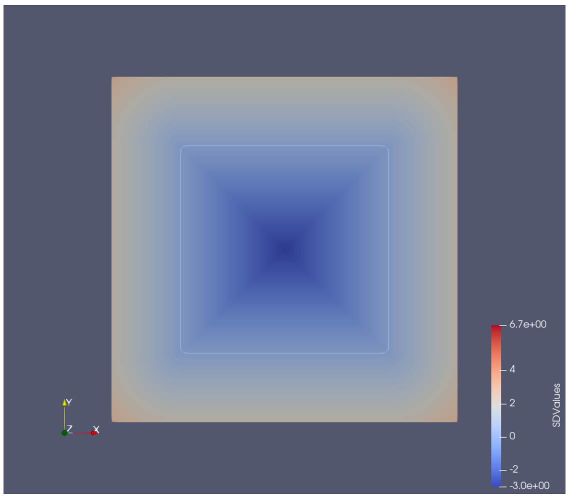
1.3 Results

The following section provides a gallery of possible results which are rendered with ParaView. The colour of the grid represents the value of the SDF for a certain grid point. Red corresponds to positive values (outside of the surface) while blue represents negative values (inside the surface).

The thin line is the contour. Unfortunately I could not find any setting to adjust the line thickness but since this is a PDF one can simply zoom in or use the parameters below the image to recreate the results. Alternatively all .vtk and .png files are available in the */results* folder.

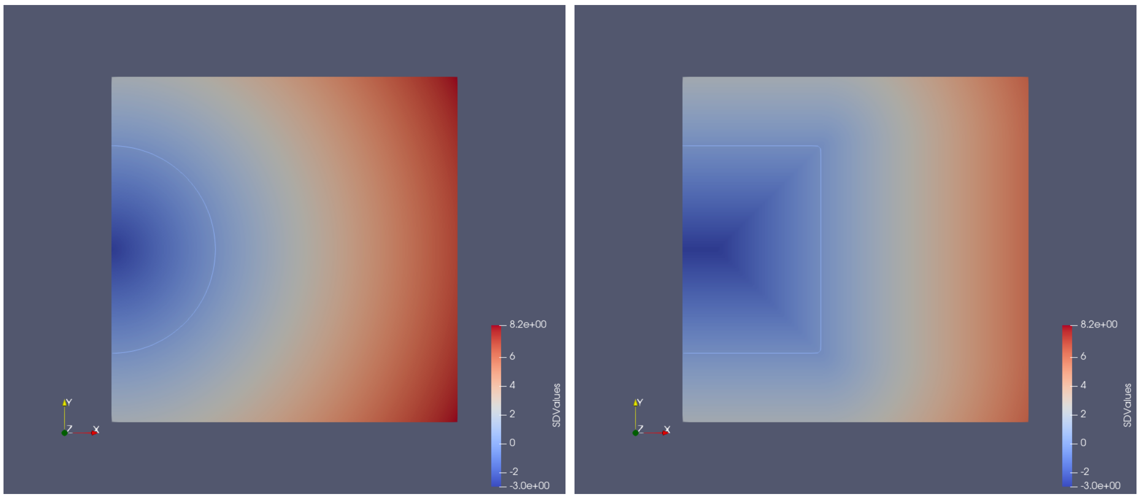


(a) 10 10 0.1 Sphere 5 5 0 3 rrrr



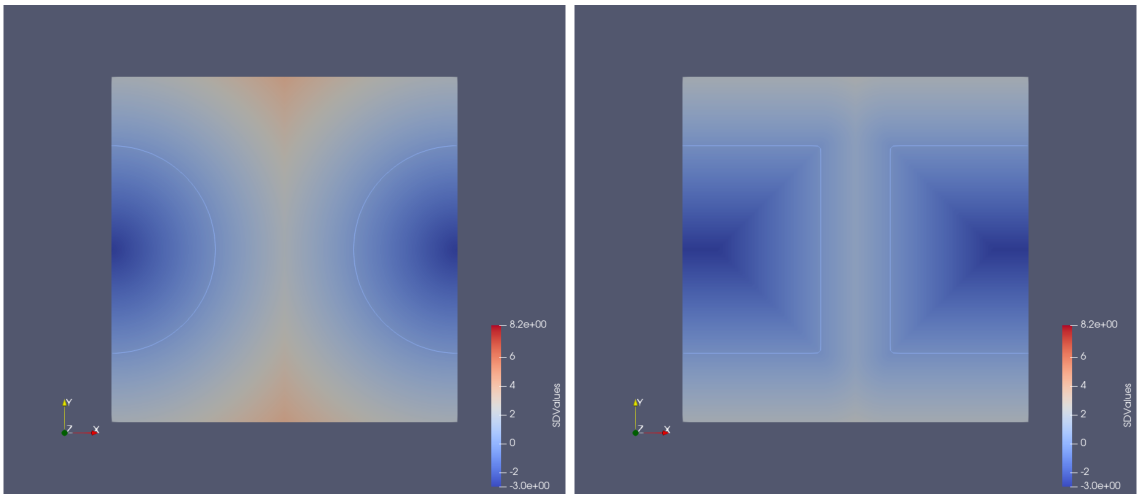
(b) 10 10 0.1 Rectangle 2 2 8 8 rrrr

Fig. 1: A centered sphere and rectangle. The edges of the rectangle appear slightly cut off. This is because ParaView is not correctly representing the contour and omits corner points for some reason. Looking at the numerical data one can see the program correctly calculates 0 as the signed distance on corner points.



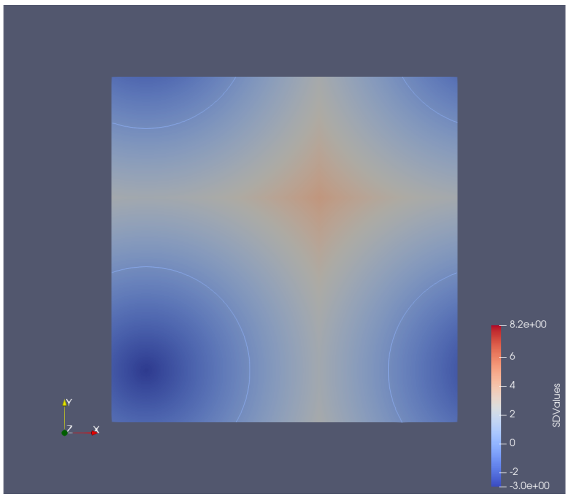
(a) 10 10 0.1 Sphere 0 5 0 3 rrrr (b) 10 10 0.1 Rectangle -4 2 4 8 rrrr

Fig. 2: A shifted sphere and rectangle with reflective boundary conditions. The surface simply vanishes behind the border since any grid point is always closer to the surface within the grid than a reflected point outside of the grid.

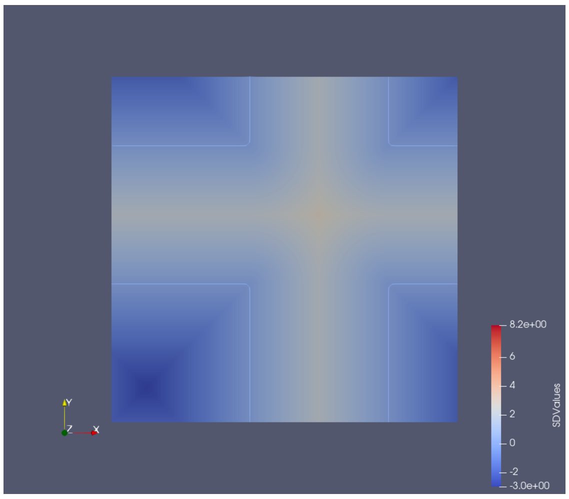


(a) 10 10 0.1 Sphere 0 5 0 3 pppp (b) 10 10 0.1 Rectangle -4 2 4 8 pppp

Fig. 3: A shifted sphere and rectangle with periodic boundary conditions. The surface correctly wraps around the domain. The left surface is the original SDF. The one on the right is projected by the boundary

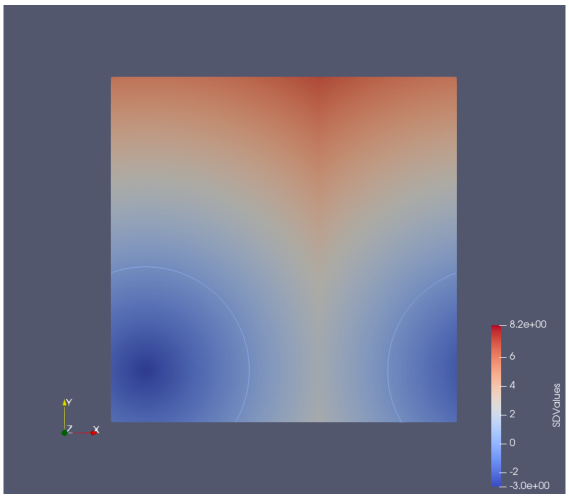


(a) 10 10 0.1 Sphere 1 1.5 0 3 pppp

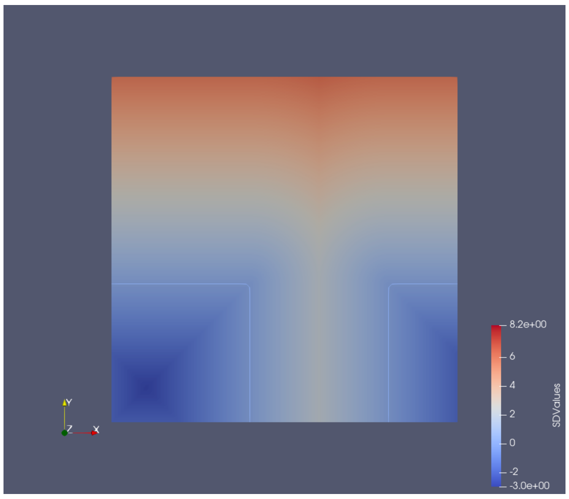


(b) 10 10 0.1 Rectangle -2 -2 4 4 pppp

Fig. 4: A fancier example of periodic boundary conditions. The total area of all 4 sub surfaces is the same as the original object which extends beyond the grid. X and Y edges of the rectangle are collinear.

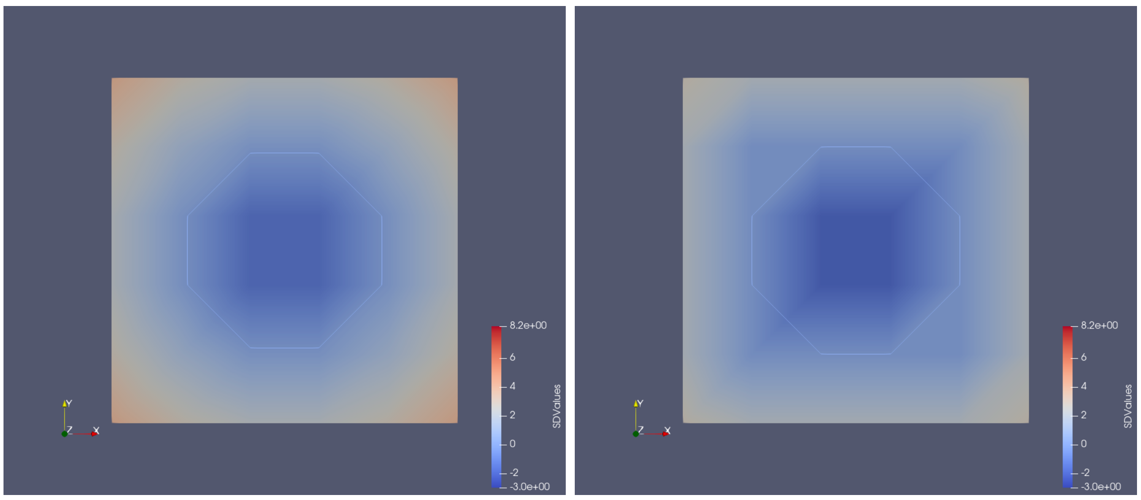


(a) 10 10 0.1 Sphere 1 1.5 0 3 rprp



(b) 10 10 0.1 Rectangle -2 -2 4 4 rprp

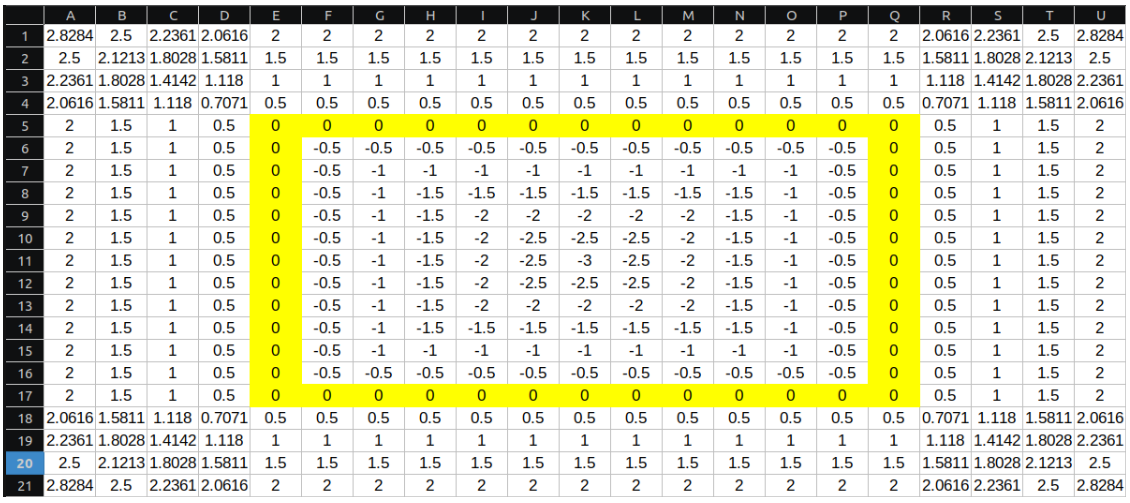
Fig. 5: Same as above but this time with mixed boundary conditions. The surface correctly wraps around horizontally but is reflected vertically



(a) 10 10 2 Sphere 5 5 0 3 rrrr

(b) 10 10 2 Rectangle 2 2 8 8 rrrr

Fig. 6: A centered sphere and grid with a very big spacing (6x6 points). Because Paraview cuts off edges the contours of rectangle and sphere are no longer distinguishable. However one can see the difference in the shading of the grid.



	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
1	2.8284	2.5	2.2361	2.0616	2	2	2	2	2	2	2	2	2	2	2	2	2	2.0616	2.2361	2.5	2.8284
2	2.5	2.1213	1.8028	1.5811	1.5	1.5	1.5	1.5	1.5	1.5	1.5	1.5	1.5	1.5	1.5	1.5	1.5	1.5811	1.8028	2.1213	2.5
3	2.2361	1.8028	1.4142	1.118	1	1	1	1	1	1	1	1	1	1	1	1	1	1.118	1.4142	1.8028	2.2361
4	2.0616	1.5811	1.118	0.7071	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.7071	1.118	1.5811	2.0616
5	2	1.5	1	0.5	0	0	0	0	0	0	0	0	0	0	0	0	0	0.5	1	1.5	2
6	2	1.5	1	0.5	0	-0.5	-0.5	-0.5	-0.5	-0.5	-0.5	-0.5	-0.5	-0.5	-0.5	-0.5	0	0.5	1	1.5	2
7	2	1.5	1	0.5	0	-0.5	-1	-1	-1	-1	-1	-1	-1	-1	-1	-0.5	0	0.5	1	1.5	2
8	2	1.5	1	0.5	0	-0.5	-1	-1.5	-1.5	-1.5	-1.5	-1.5	-1.5	-1.5	-1	-0.5	0	0.5	1	1.5	2
9	2	1.5	1	0.5	0	-0.5	-1	-1.5	-2	-2	-2	-2	-2	-1.5	-1	-0.5	0	0.5	1	1.5	2
10	2	1.5	1	0.5	0	-0.5	-1	-1.5	-2	-2.5	-2.5	-2.5	-2	-1.5	-1	-0.5	0	0.5	1	1.5	2
11	2	1.5	1	0.5	0	-0.5	-1	-1.5	-2	-2.5	-3	-2.5	-2	-1.5	-1	-0.5	0	0.5	1	1.5	2
12	2	1.5	1	0.5	0	-0.5	-1	-1.5	-2	-2.5	-2.5	-2.5	-2	-1.5	-1	-0.5	0	0.5	1	1.5	2
13	2	1.5	1	0.5	0	-0.5	-1	-1.5	-2	-2	-2	-2	-2	-1.5	-1	-0.5	0	0.5	1	1.5	2
14	2	1.5	1	0.5	0	-0.5	-1	-1.5	-1.5	-1.5	-1.5	-1.5	-1.5	-1.5	-1	-0.5	0	0.5	1	1.5	2
15	2	1.5	1	0.5	0	-0.5	-1	-1	-1	-1	-1	-1	-1	-1	-1	-0.5	0	0.5	1	1.5	2
16	2	1.5	1	0.5	0	-0.5	-0.5	-0.5	-0.5	-0.5	-0.5	-0.5	-0.5	-0.5	-0.5	-0.5	0	0.5	1	1.5	2
17	2	1.5	1	0.5	0	0	0	0	0	0	0	0	0	0	0	0	0	0.5	1	1.5	2
18	2.0616	1.5811	1.118	0.7071	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.7071	1.118	1.5811	2.0616
19	2.2361	1.8028	1.4142	1.118	1	1	1	1	1	1	1	1	1	1	1	1	1	1.118	1.4142	1.8028	2.2361
20	2.5	2.1213	1.8028	1.5811	1.5	1.5	1.5	1.5	1.5	1.5	1.5	1.5	1.5	1.5	1.5	1.5	1.5	1.5811	1.8028	2.1213	2.5
21	2.8284	2.5	2.2361	2.0616	2	2	2	2	2	2	2	2	2	2	2	2	2	2.0616	2.2361	2.5	2.8284

Fig. 7: A visual representation of grid data in the most literal way. Edges are correctly calculated but ParaView draws the contour incorrectly. However once spacing is sufficiently small the error ParaView makes is negligible. Parameters: 10 10 0.5 Rectangle 2 2 8 8 rrrr

2 Geometric Variables of a Signed Distance Function

2.1 Examnple usage:

```
./out/build/grid 4 4 0.1 Sphere 2 2 0 1 rrrr out/grid.vtk 3 2
>> Adding Sphere to grid...
>> Parsed point (3|2)
>> Gradient=[1, 0, 0]
>> Surface Normal=[1, 0, 0]
>> Curvature=-1.34164
>> Done
```

Expected result: A sphere with radius 1 with its center at (2, 2, 0) is created. The Gradient, surface normal and curvature at (3, 2) with respect to the SDF object is calculated.

The assignment probably intended us to calculate the geometric variables with respect to the grid. This leads to different results for some special cases such as points on the boundary or periodic boundary conditions. These implications will be discussed in the following section.

2.2 Numerical Derivatives and gradient

The numerical derivative is the basis for all other functions in this section and many other applications. Mathematically the derivative relies on infinitesimal values which cannot be represented using regular floating point operations.

One way of resolving this issue is to use forward and backward differences of points that are sufficiently close.

$$D_i(\Phi(\vec{x})) = \frac{\Phi\left(\vec{x} + \epsilon \cdot \vec{e}_i\right) - \Phi\left(\vec{x} - \epsilon \cdot \vec{e}_i\right)}{2\Delta x}$$

By choosing a sufficiently small ϵ one can calculate a good approximation of the first derivative. Since the SDF is independent of the grid the derivative is not necessarily limited by the resolution of the grid. However it is possible to set ϵ to the spacing of the grid to get results have approximately the same error as the grid.

Algorithm 6: Calculating the gradient for a given point of the SDF

```
Vector3D SDF::gradient(const Vector3D & p, double eps)
{
    return Vector3D(
        eval(p + Vector3D(eps, 0, 0)) - eval(p - Vector3D(eps, 0, 0)),
        eval(p + Vector3D(0, eps, 0)) - eval(p - Vector3D(0, eps, 0)),
        eval(p + Vector3D(0, 0, eps)) - eval(p - Vector3D(0, 0, eps))
    ) / (2 * eps);
}
```

Alternatively one could calculate the gradient by using neighbouring points of the grid at the cost of limited accuracy. The advantage of this approach is that it is quite easy to handle points on the edge of the simulation or surfaces that are projected by periodic boundary conditions. As a reflected point outside of the grid can easily be calculated using a mapping function.

I have chosen to implement the gradient as a method of SDF (Alg. 6) due to the design decision of separating the grid from the SDF. Unfortunately I have not realized the implications of this decision for boundary conditions until the very end of the project and I have run out of time to fix this issue. Therefore I will only provide pseudo-code on how such a function could be implemented within the Grid-Class:

Algorithm 7: Pseudo Code for calculating the gradient at a given point using grid data

```
Vector3D Grid::gradient(int x, int y)
{
    return Vector3D(
        get(x + 1, y) - get(x - 1, y),
        get(x , y + 1) - get(x, y - 1),
        0
    ) / (2 * spacing);
}
```

Where get(int, int) is a method of Grid that returns the value at grid coordinates x and y. If the coordinates are outside of the grid than the indices are mapped to points inside the grid depending on the type of boundary.

Algorithm 8: Pseudo Code for get The function has to be extended with more cases for all other directions

```
Vector3D Grid::get(int x, int y)
{
    if(x < x_size && y < y_size && x >= 0 && y >= 0)
        return data[x][y];

    // recursively call get with shifted coordinates until we are in the grid
    if(x < 0 && y >= 0)
        return m_border_types[LEFT] == BOUNDARY_PERIODIC ?
            get(x + x_size, y) :    // periodic case
            get(-x, y); // reflective case

    ...
}
```

Retrospectively I think that this is the way I was intended to implement the gradient but unfortunately I ran out of time. The task description is also a bit confusing as it uses the SDF $\Phi(x)$ in the definition of the central difference (Eq. 2) whereas the explicit usage of grid indices would have been more appropriate. This problem also has implications for the Enquist-Osher scheme but I will discuss that in the next chapter.

2.3 Surface Normal

Using the derivative from Alg. 6 the surface normal is implemented the following way:

Algorithm 9: Implementation of the surface normal

```
Vector3D SDF::surfaceNormal(const Vector3D & v, double eps)
{
    Vector3D grad = gradient(v, eps);
    return grad / grad.norm_l2();
}
```

Again this calculates the surface normal of the SDF while omitting projected surfaces and boundary conditions. However replacing SDF::gradient() with Grid::gradient() from Alg. 7 should resolve the issue.

2.3.1 Curvature

Algorithm 10: calculation of the curvature

```
double SDF::curvature(const Vector3D & v, double eps)
{
    Vector3D n = gradient(surfaceNormal(v, eps));
    return (n.m_x + n.m_y + n.m_z);
}
```

The curvature is essentially the second derivative and can be calculated by taking the derivative of the surface normal and summing its components.

3 Moving the Zero Level Set to Advance a Surface

3.1 Example Usage

For Task 3 no dedicated API has been implemented. In order to recreate the results of the following section please uncomment the respective lines in *main.cpp* and recompile the project.

Algorithm 11: Advancing the surface by subtracting a constant

```
Grid grid(argv);
grid.add_surface(sdf);
grid.update();
grid.advance_simple(10, 1); // Advance surface with speed 10 for 1 unit of time
grid.update();
```

3.2 simple advancement

The simplest way of advancing the surface is to add a constant term to the SDF. For a sphere that would correspond to a change in radius. The case for a rectangle (or quad) is a bit more complicated. In 3D subtracting a constant form the SDF of a box results in rounded edges and corners. If we now take a 2D cross section the expected result should be a rectangle with rounded corners.

The class Grid contains a second data structure that is identical to m_data[ ][ ] but describes change in distance. When calling Grid::update() the velocity field simply substracted from each data point. For advance_simple() the velocity field is constant at every point.

Algorithm 12: Advancing the surface by substracting a constant

```
void Grid::advance_simple(double velocity, double delta_t)
{
    double distance = delta_t*velocity;
    for(size_t y = 0; y < m_velocity_field[0].size(); y++)
    {
        for(size_t x = 0; x < m_velocity_field.size(); x++)
        {
            m_velocity_field[x][y] += distance;
        }
    }
}

void Grid::update()
{
    for(size_t y = 0; y < m_data[0].size(); y++)
    {
        for(size_t x = 0; x < m_data.size(); x++)
        {
        // calculate SDF values with respect to the borders
        ...
        m_data[x][y] = minElement(values) - m_velocity_field[x][y];
        }
    }
}
```

Note that the name velocity field is not entirely appropriate in that context since the SDF returns a distance. Adding a velocity to that is physically not correct. However Alg. 12 adds the product of $time \cdot velocity$ which is actually a distance. The term velocity field is still used in the code because the data structured served a different purpose initially.

3.3 Advancement using the Engquist Osher scheme

The EO-scheme is another method to advance the surface and has been implemented as described in the assignment. Alg. 13 shows the algorithm for a single point. If the CFL condition is violated the algorithm is recursively repeated twice with half the distance until the CFL condition is fullfilled.

Algorithm 13: Engquist osher implementation

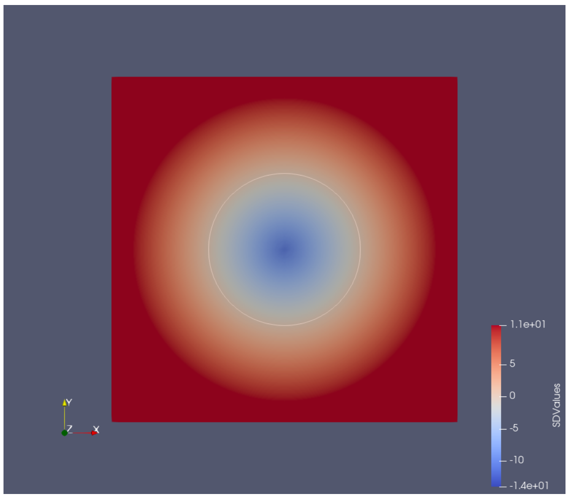
```
double Grid::advance_engquist_osher_single(int x, int y, double velocity, double
    delta_t)
{
    double distance = delta_t*velocity;
    Vector3D pos = {x * m_delta, y * m_delta, 0};
    double V =  distance * l2_norm(m_sdf->gradient(pos));

    Vector3D forward_diff = m_sdf->diff(pos, FORWARD);
    Vector3D backward_diff = m_sdf->diff(pos, BACKWARD);
    double enquist_osher_factor;
    if(V <= 0)
    {
        enquist_osher_factor = sqrt(
            pow(max((-1.0) * backward_diff.m_x, 0), 2) + pow(min((-1.0) *
                forward_diff.m_x, 0), 2) +
            pow(max((-1.0) * backward_diff.m_y, 0), 2) + pow(min((-1.0) *
                forward_diff.m_y, 0), 2)
        );
    }
    else
    {
        enquist_osher_factor = sqrt(
            pow(max(backward_diff.m_x, 0), 2) + pow(min(forward_diff.m_x, 0), 2) +
            pow(max(backward_diff.m_y, 0), 2) + pow(min(forward_diff.m_y, 0), 2)
        );
    }


    if(abs(V) <= m_delta / delta_t)
    {
        return distance * enquist_osher_factor;
    }
    else
    {
        return advance_engquist_osher_single(x, y, velocity, delta_t/2.0)*2;
    }
```

3.4 Results

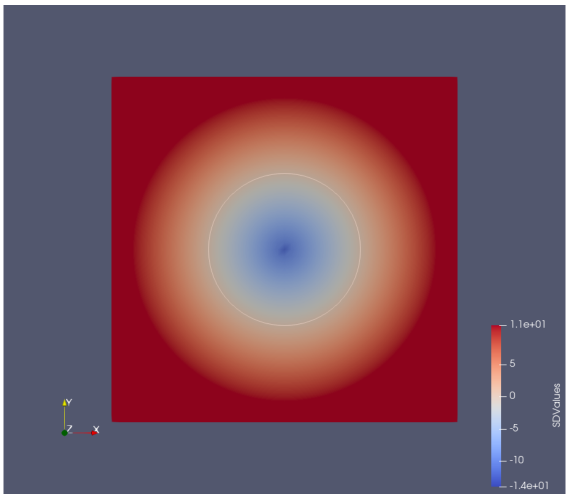
The following section shows the results when using different configurations of grid spacing as stated in the task. The results can be summed up as follows:

Both algorithms, simple advance and Engquist-osher, produce the same contour The SDF values inside of the Surface sligthly different. The difference is most noticable for high velocities and fine grids advancing the surface is the same as continuously depositing material on the outside. Therefore sharp edges become continuously more rounded

3.4.1 simple vs. Engquist osher

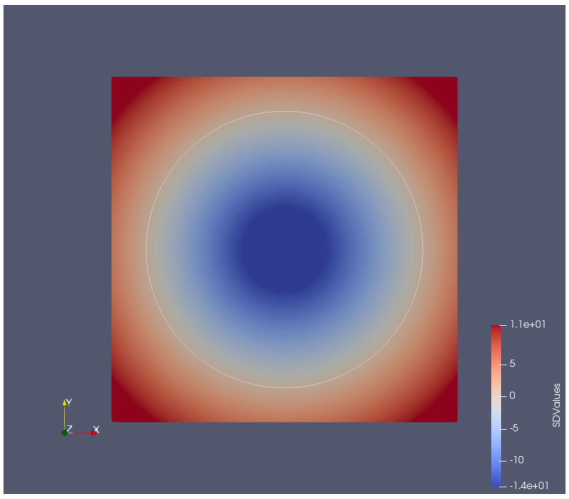


(a) simple

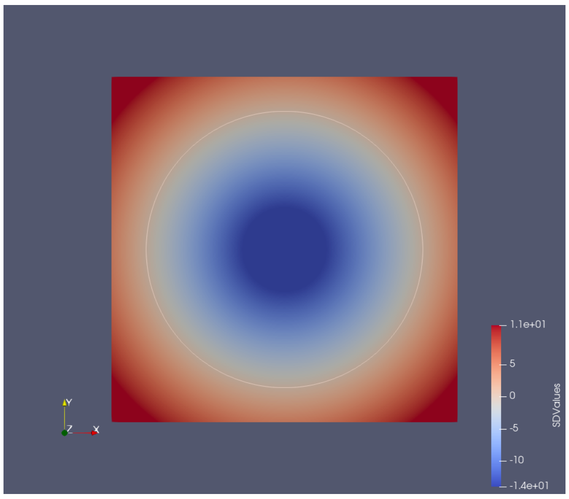


(b) Enquist Osher

Fig. 8: Circle, V=10, $\Delta x = 1$, t=0.1

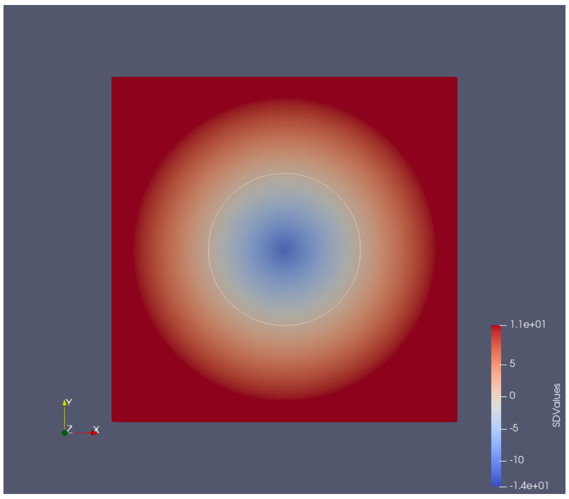


(a) simple

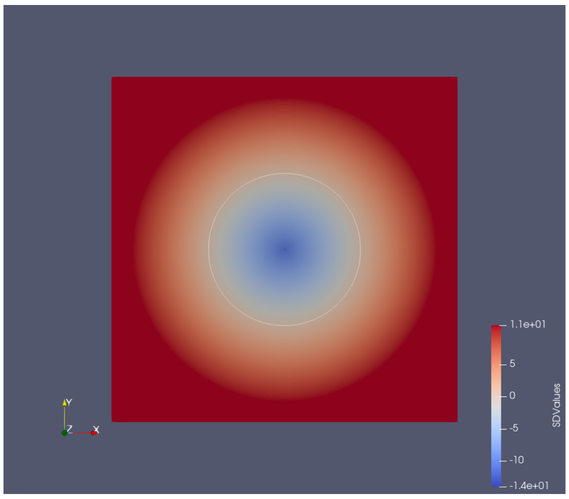


(b) Enquist Osher

Fig. 9: Circle, V=10, $\Delta x = 1$, t=1

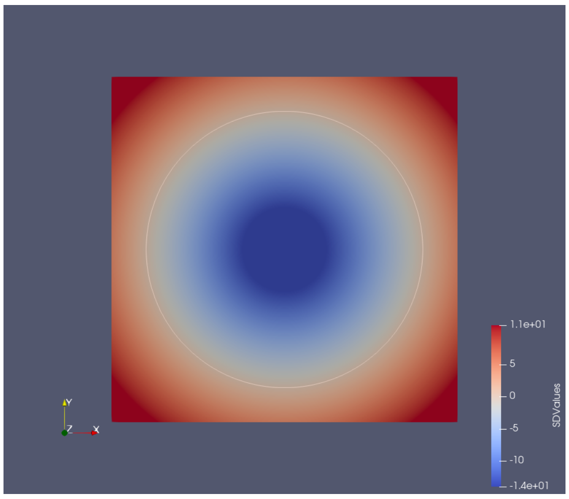


(a) simple

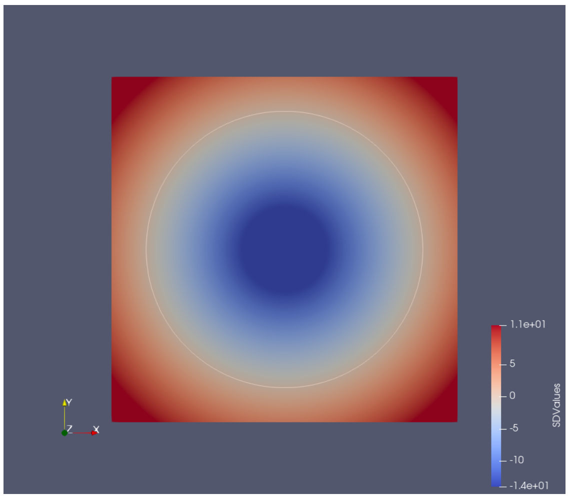


(b) Enquist Osher

Fig. 10: Circle, V=10, $\Delta x = 0.25$, t=0.1

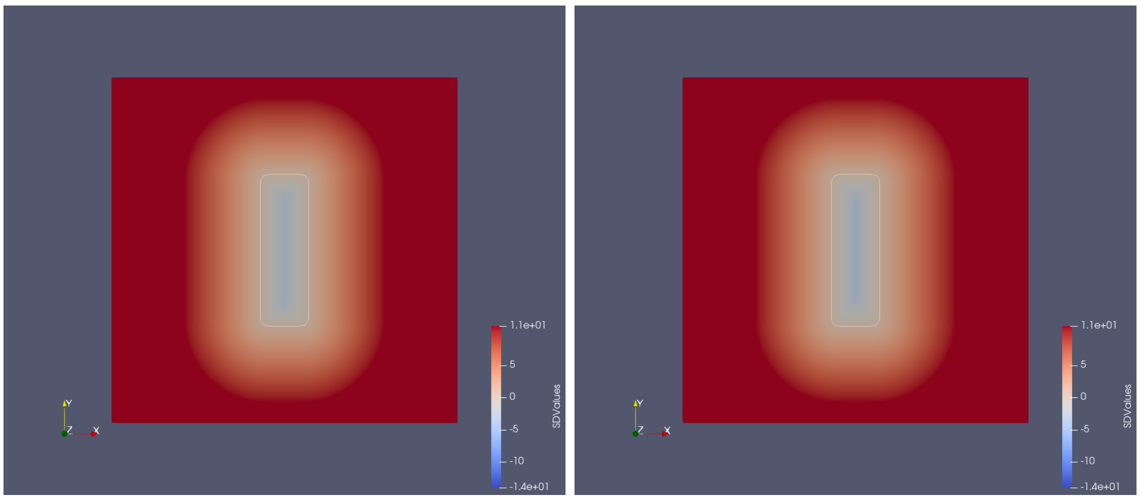


(a) simple



(b) Enquist Osher

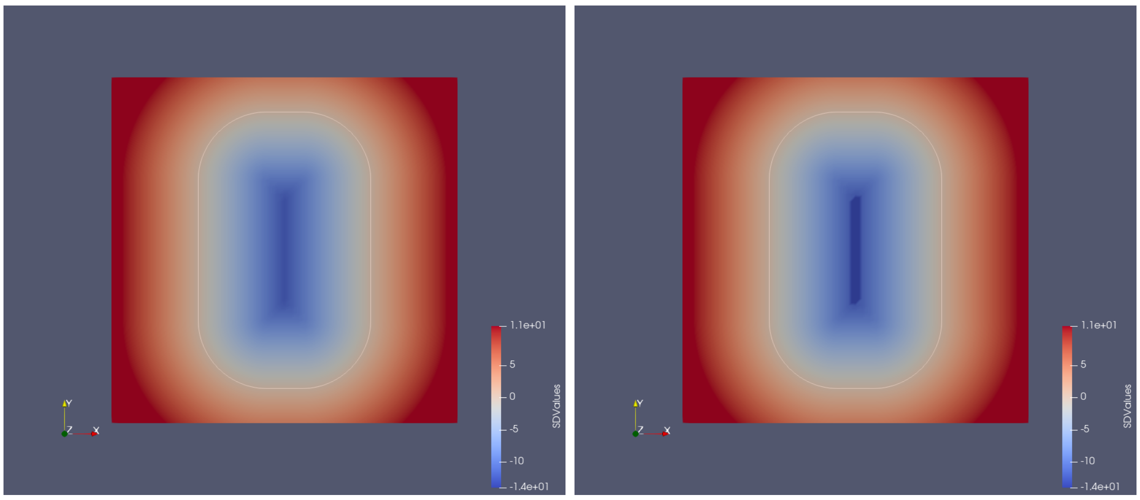
Fig. 11: Circle, V=10, $\Delta x = 0.25$, t=1



(a) simple

(b) Enquist Osher

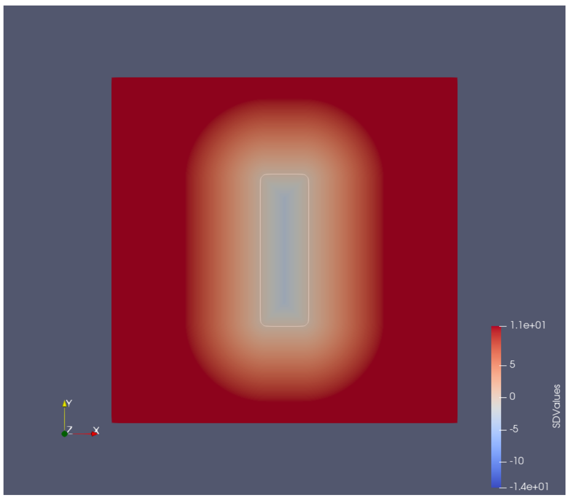
Fig. 12: Circle, V=10, $\Delta x = 1$, t=0.1



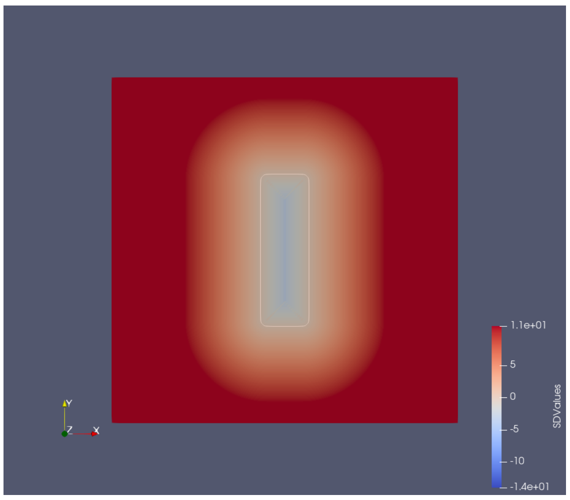
(a) simple

(b) Enquist Osher

Fig. 13: Rectangle, V=10, $\Delta x = 1$, t=1

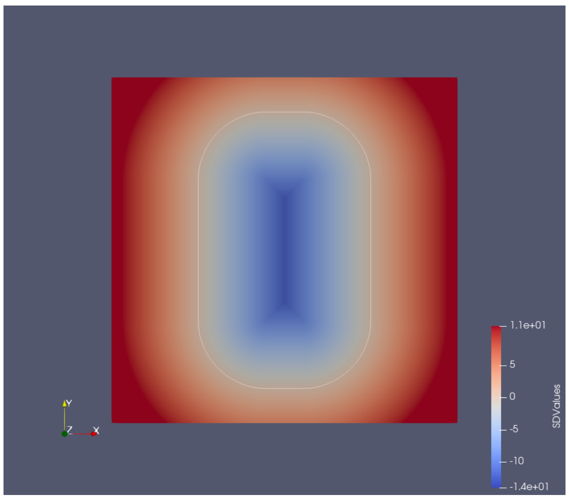


(a) simple

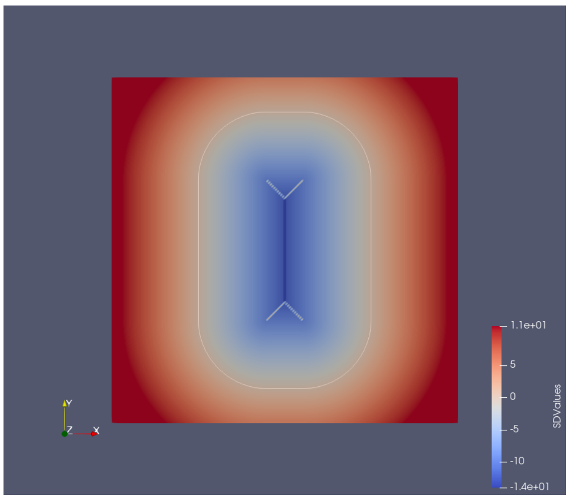


(b) Enquist Osher

Fig. 14: Rectangle, V=10, $\Delta x = 0.25$, t=0.1



(a) simple



(b) Enquist Osher

Fig. 15: Rectangle, V=10, $\Delta x = 0.25$, t=1

3.4.2 vector velocity function

The results of the last two points are definetly wrong but I have left them in there because I think they are fun to look at. Task 3 took me much longer than expected which is why I have run out of time to fix the issue. My guess is that the Engquist-Osher scheme is not working as intended.

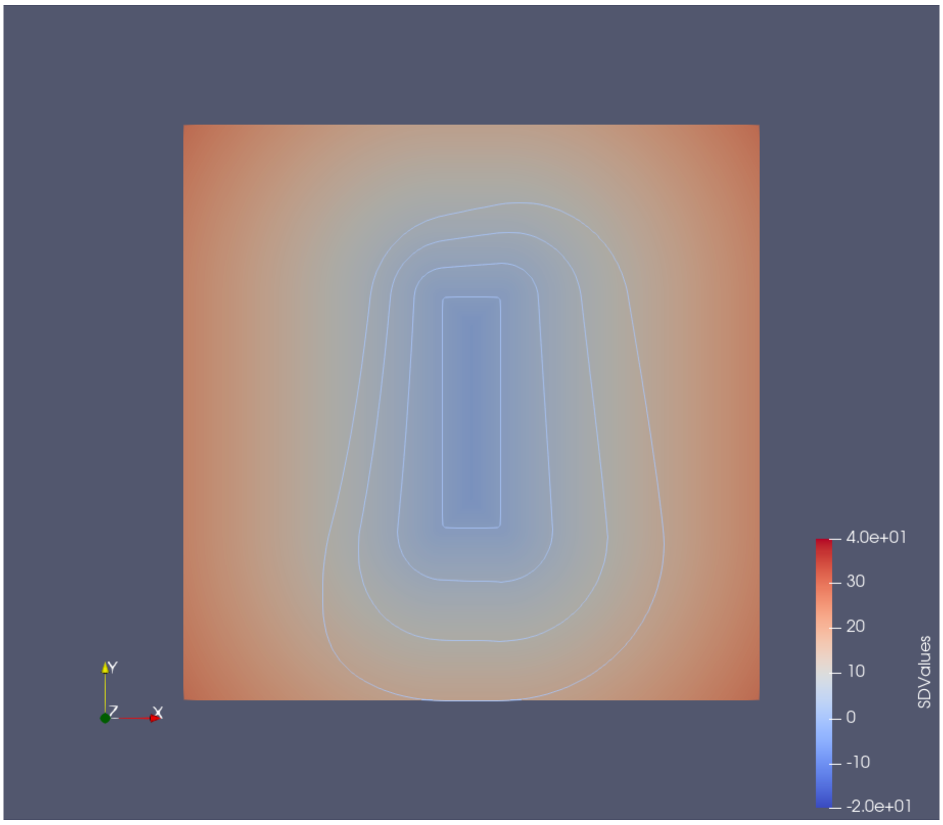


Fig. 16: Behaviour of a rectangle over multiple time steps when moved by the vector velocity function

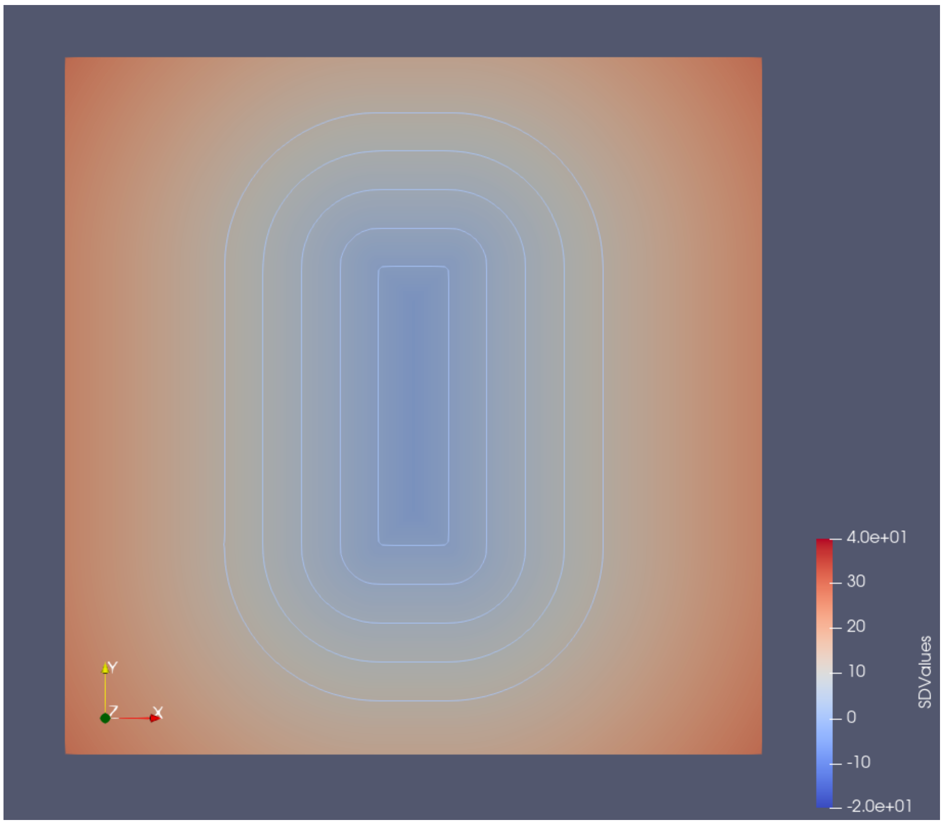


Fig. 17: Advancing the surface using the curvature with multiple time steps. Since this is essentially the same as advancing the surface with a constant I doubt that this is correct.