cornelius — user's guide

Pasi Huovinen
Institut für Theoretische Physik, J. W. Goethe-Universität

Hannu Holopainen
Frankfurt Institute for Advanced Studies (FIAS)

August 7, 2012

Cornelius is a subroutine to find the normal vector and size of a 2- or 3-dimensional isosurface element in a 3- or 4-dimensional discrete grid. This user's guide applies to the following versions of cornelius:

cornelius.f90 Fortran90 subroutine to find a 3-dimensional (hyper)surface element in a 4-dimensional (hyper)volume element.

cornelius2.f90 Fortran90 subroutine to find a 2-dimensional (hyper)surface element in a 3-dimensional (hyper)volume element.

cornelius.cpp C++ subroutine to find either 1-, 2- or 3-dimensional surface in a 2-, 3- or 4-dimensional volume element, respectively.

License

General Information

Cornelius is based on an modified version of the algorithm proposed in Ref. [1]. This "disordered lines" algorithm is explained in detail in Ref. [2]. Cornelius is made for the purpose of evaluating the normal vector, size and the location of the centroid of the surface elements *i.e.* the terms $\Delta\sigma_\mu$ in the discretised Cooper-Frye procedure:

$$E\frac{\mathrm{d}N}{\mathrm{d}p^3} = \int_\sigma \mathrm{d}\sigma_\mu p^\mu f(x,p) \approx \sum_\sigma \Delta\sigma_\mu p^\mu f(x,p). \tag{1}$$

Thus it is not suitable for purposes where one wants to find a mesh describing the surface: The output of the present version of cornelius does not return the corner points of polygons forming the surface required to form the mesh, but only the coordinates of the centroid of the polygons.

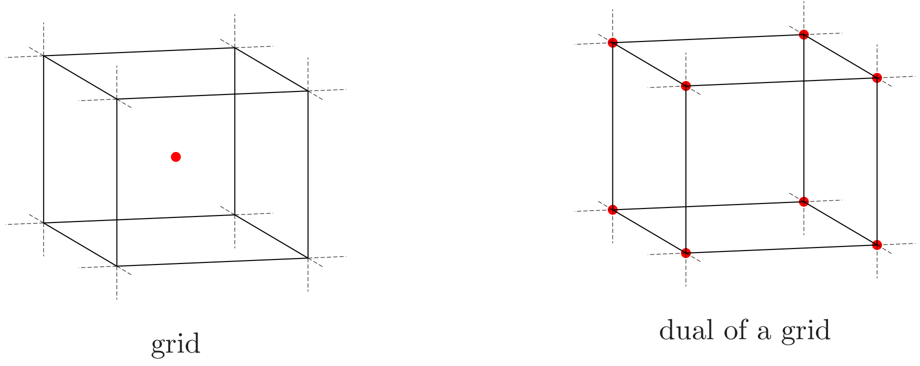

Figure 1: Gridpoint in the middle of a volume element in a grid, and at the corners of a volume element in a dual of a grid.

In algorithms solving the equations of motion of fluid dynamics like SHASTA [3], a grid point is considered to be in the middle of a corresponding volume element. For purposes of surface finding it is useful to consider the dual of the grid, where the grid points are thought to be at the corners of a volume element, see Fig. 1. The values between the grid points are interpolated linearly, and thus an isosurface can be located between two gridpoints, one of which is above and one below the isovalue, for details see Ref. [2]. A surface element is thus within a volume element which has at least one corner above or equal to the isovalue, and at least one corner below the isovalue. Since the grid in different applications of fluid dynamics is very different, cornelius does not search for such elements in a grid, it only evaluates the properties of the surface elements (four vector $\Delta\sigma_\mu$ in Eq. (1)) within volume elements containing them. Note that the grid where the surface is searched for, does not need to be the same than where the hydrodynamical equations are solved. One can use a grid containing, say, every second spatial gridpoint, and every fifth timestep. Other combinations are perfectly possible as well depending on the desired accuracy.

To find the volume elements where the value is equal to the isovalue, it is practical to use so called exhaustive search method, *i.e.* to check every single volume element whether some of its corners are above (or equal) and some below the isovalue. In practice this can be done during the hydrodynamical evolution. After i timesteps ($i \geq 1$ as described) the n-dimensional arrays containing the relevant field now and i timesteps ago are combined to form a $n+1$-dimensional array. All the volume elements are checked whether some of the corners are above (or equal) and some below the isovalue. If this is the case, the volume element contains a surface element, and its properties are evaluated using cornelius. Note that checking the volume elements is a n-dimensional task, since all the volume elements are between the grid points, and there are only two points in time direction in this grid. After the entire grid has been searched, the field at this timestep is stored in the array containing the earlier timestep, the system is evolved i timesteps further, and the procedure is repeated.

Use of cornelius.f90

Once a (hyper)volume element containing a (hyper)surface element, *i.e.* a hypercube with corners above and below the criterion, has been located, and the values in the hypercube corners are stored in an array `HyperCube`, subroutine `Cornelius` is called:

```
CALL Cornelius(E0,HyperCube,dSigma,Nsurf,Vmid,dt,dx,dy,dz,Nambi,Ndisc)
```

The arguments are

E0: isovalue, *i.e.* the criterion for the surface, *e.g.* freeze-out value for a freeze-out surface (`REAL(KIND(0D0))`).

HyperCube(t,i,j,k): A 4D array (2x2x2x2) (`REAL(KIND(0D0))`) for the values of the field at the corners of the volume element. The first index denotes time, the second x, the third y, and the fourth z:
HyperCube(0,0,0,0) $\leftrightarrow t = 0,\ x = 0,\ y = 0,\ z = 0$
HyperCube(0,0,0,1) $\leftrightarrow t = 0,\ x = 0,\ y = 0,\ z = 1$
HyperCube(0,0,1,0) $\leftrightarrow t = 0,\ x = 0,\ y = 1,\ z = 0$
HyperCube(0,0,1,1) $\leftrightarrow t = 0,\ x = 0,\ y = 1,\ z = 1$
HyperCube(0,1,0,0) $\leftrightarrow t = 0,\ x = 1,\ y = 0,\ z = 0$
HyperCube(0,1,0,1) $\leftrightarrow t = 0,\ x = 1,\ y = 0,\ z = 1$
HyperCube(0,1,1,0) $\leftrightarrow t = 0,\ x = 1,\ y = 1,\ z = 0$
HyperCube(0,1,1,1) $\leftrightarrow t = 0,\ x = 1,\ y = 1,\ z = 1$
HyperCube(1,0,0,0) $\leftrightarrow t = 1,\ x = 0,\ y = 0,\ z = 0$
HyperCube(1,0,0,1) $\leftrightarrow t = 1,\ x = 0,\ y = 0,\ z = 1$

HyperCube(1,0,1,0) $\leftrightarrow t = 1,\ x = 0,\ y = 1,\ z = 0$
HyperCube(1,0,1,1) $\leftrightarrow t = 1,\ x = 0,\ y = 1,\ z = 1$
HyperCube(1,1,0,0) $\leftrightarrow t = 1,\ x = 1,\ y = 0,\ z = 0$
HyperCube(1,1,0,1) $\leftrightarrow t = 1,\ x = 1,\ y = 0,\ z = 1$
HyperCube(1,1,1,0) $\leftrightarrow t = 1,\ x = 1,\ y = 1,\ z = 0$
HyperCube(1,1,1,1) $\leftrightarrow t = 1,\ x = 1,\ y = 1,\ z = 1$

dSigma(i,j) : An array (`REAL(KIND(0D0))`) to store the normal vector(s) of the surface elements. The first index runs from 0 to 3, and it contains the components of a normal vector j. The second index runs from 1 to 8, so there is space for eight vectors in this array. The components are as expected — dSigma(0,j) is the time component, dSigma(1,j) the x-component, dSigma(2,j) the y-component, and dSigma(3,j) the z-component.

Nsurf: Number of separate surface elements within the cube, and thus the number of separate normal vectors in dSigma (`INTEGER`).

Vmid(i,j): coordinates of the approximate centroid(s) of the surface element(s) evaluated placing the origin at (0,0,0,0) corner of the cube. Again, Vmid(i,j) is the i'th coordinate of the position vector j.
(`REAL(KIND(0D0))`)

dt, dx, dy, dz: the lengths of the edges of the cube, which can be multiplets of the grid spacing and timestep of the grid where hydro is solved (`REAL(KIND(0D0))`).

Nambi: number of ambiguous faces on surfaces, this is only to study the properties of the surface (`INTEGER`).

Ndisc: number of disconnected surface-elements so far, also only to produce some statistics of the surface (`INTEGER`).

Use of cornelius2.f90

Once a volume element containing a surface element, *i.e.* a cube with corners above and below the criterion, has been located, and the values in the cube corners are stored in an array `Cube`, subroutine `Cornelius2` is called:

```
CALL Cornelius2(E0,Cube,dSigma,Nsurf,Vmid,dt,dx,dy,Nambi,Ndisc)
```

The arguments are

E0: isovalue, *i.e.* the criterion for the surface, *e.g.* freeze-out value for a freeze-out surface (`REAL(KIND(0D0))`).

Cube(i,j,k): A 3D array (2x2x2) (`REAL(KIND(0D0))`) for the values of the field at the corners of the volume element. The first index denotes time, the second x and third y:
Cube(0,0,0) $\leftrightarrow t = 0,\ x = 0,\ y = 0$
Cube(0,0,1) $\leftrightarrow t = 0,\ x = 0,\ y = 1$
Cube(0,1,0) $\leftrightarrow t = 0,\ x = 1,\ y = 0$
Cube(0,1,1) $\leftrightarrow t = 0,\ x = 1,\ y = 1$
Cube(1,0,0) $\leftrightarrow t = 1,\ x = 0,\ y = 0$
Cube(1,0,1) $\leftrightarrow t = 1,\ x = 0,\ y = 1$
Cube(1,1,0) $\leftrightarrow t = 1,\ x = 1,\ y = 0$
Cube(1,1,1) $\leftrightarrow t = 1,\ x = 1,\ y = 1$

dSigma(i,j): An array (`REAL(KIND(0D0))`) to store the normal vector(s) of the surface elements. The first index runs from 0 to 2, and it contains the components of a normal vector j. The second index runs from 1 to 4, so there is space for four vectors in this array. The components are as expected — dSigma(0,j) is the time component, dSigma(1,j) the x-component, and dSigma(2,j) the y-component.

Nsurf: Number of separate surface elements within the cube, and thus the number of separate normal vectors in dSigma (`INTEGER`).

Vmid(i,j): coordinates of the approximate centroid(s) of the surface element(s) evaluated placing the origin at (0,0,0) corner of the cube. Again, Vmid(i,j) is the i'th coordinate of the position vector j.
(`REAL(KIND(0D0))`)

dt, dx, dy: the lengths of the edges of the cube, which can be multiplets of the grid spacing and timestep of the grid where hydro is solved (`REAL(KIND(0D0))`).

Nambi: number of ambiguous faces on surfaces, this is only to study the properties of the surface (`INTEGER`).

Ndisc: number of disconnected surface-elements so far, also only to produce some statistics of the surface (`INTEGER`).

Use of cornelius.cpp

The C++ version `cornelius.cpp` is constructed slightly differently. First, the subroutine is initialised by giving a command

```
  init(dimension,E0,dx)
```

where the arguments are

dimension: dimension of the cube where the surface elements are determined, *i.e.* this should be 2, 3 or 4 (`int`).

E0: isovalue, *i.e.* the criterion for the surface, *e.g.* freeze-out value for a freeze-out surface (`double`).

dx[i]: An array (length `dimension`) (`double*`) for the lengths of the edges of the cube:

4D	3D	2D
dx[0] $\leftrightarrow dt$	dx[0] $\leftrightarrow dt$	dx[0] $\leftrightarrow dt$
dx[1] $\leftrightarrow dx$	dx[1] $\leftrightarrow dx$	dx[1] $\leftrightarrow dx$
dx[2] $\leftrightarrow dy$	dx[2] $\leftrightarrow dy$	
dx[3] $\leftrightarrow dz$		

Lengths can be multiplets of the grid spacing and timestep of the grid where hydro is solved.

After the object has been initialised and the values in the cube corners are stored in an array `HyperCube`, it can be passed on to the object by the following command corresponding the given `dimension`

```
find_surface_4d(HyperCube)
find_surface_3d(Cube)
find_surface_2d(Square)
```

where

HyperCube[i][j][k][l]: A 4D array (2x2x2x2) (`double****`) for the values of the field at the corners of the volume element. The first index denotes time, the second x, the third y and fourth z:
HyperCube[0][0][0][0] $\leftrightarrow t = 0,\ x = 0,\ y = 0,\ z = 0$
HyperCube[0][0][0][1] $\leftrightarrow t = 0,\ x = 0,\ y = 0,\ z = 1$
HyperCube[0][0][1][0] $\leftrightarrow t = 0,\ x = 0,\ y = 1,\ z = 0$
HyperCube[0][0][1][1] $\leftrightarrow t = 0,\ x = 0,\ y = 1,\ z = 1$
HyperCube[0][1][0][0] $\leftrightarrow t = 0,\ x = 1,\ y = 0,\ z = 0$
HyperCube[0][1][0][1] $\leftrightarrow t = 0,\ x = 1,\ y = 0,\ z = 1$
HyperCube[0][1][1][0] $\leftrightarrow t = 0,\ x = 1,\ y = 1,\ z = 0$
HyperCube[0][1][1][1] $\leftrightarrow t = 0,\ x = 1,\ y = 1,\ z = 1$
HyperCube[1][0][0][0] $\leftrightarrow t = 1,\ x = 0,\ y = 0,\ z = 0$
HyperCube[1][0][0][1] $\leftrightarrow t = 1,\ x = 0,\ y = 0,\ z = 1$
HyperCube[1][0][1][0] $\leftrightarrow t = 1,\ x = 0,\ y = 1,\ z = 0$
HyperCube[1][0][1][1] $\leftrightarrow t = 1,\ x = 0,\ y = 1,\ z = 1$
HyperCube[1][1][0][0] $\leftrightarrow t = 1,\ x = 1,\ y = 0,\ z = 0$
HyperCube[1][1][0][1] $\leftrightarrow t = 1,\ x = 1,\ y = 0,\ z = 1$
HyperCube[1][1][1][0] $\leftrightarrow t = 1,\ x = 1,\ y = 1,\ z = 0$
HyperCube[1][1][1][1] $\leftrightarrow t = 1,\ x = 1,\ y = 1,\ z = 1$

Cube[i][j][k]: A 3D array (2x2x2) (`double***`) for the values of the field at the corners of the volume element. The first index denotes time, the second x and third y:
Cube[0][0][0] $\leftrightarrow t = 0,\ x = 0,\ y = 0$
Cube[0][0][1] $\leftrightarrow t = 0,\ x = 0,\ y = 1$
Cube[0][1][0] $\leftrightarrow t = 0,\ x = 1,\ y = 0$
Cube[0][1][1] $\leftrightarrow t = 0,\ x = 1,\ y = 1$
Cube[1][0][0] $\leftrightarrow t = 1,\ x = 0,\ y = 0$
Cube[1][0][1] $\leftrightarrow t = 1,\ x = 0,\ y = 1$
Cube[1][1][0] $\leftrightarrow t = 1,\ x = 1,\ y = 0$
Cube[1][1][1] $\leftrightarrow t = 1,\ x = 1,\ y = 1$

Square[i][j]: A 2D array (2x2) (`double**`) for the values of the field at the corners of the volume element. The first index denotes time and second x:
Square[0][0] $\leftrightarrow t = 0,\ x = 0$
Square[0][1] $\leftrightarrow t = 0,\ x = 1$
Square[1][0] $\leftrightarrow t = 1,\ x = 0$
Square[1][1] $\leftrightarrow t = 1,\ x = 1$

The number of separate surface elements (`int`) within the volume element is subsequently obtained via a command

```
get_Nelements()
```

the normal vector components (`double`) of an element i are obtained using a command

```
get_normal_elem(element,component)
```

and the coordinates of the centroid (`double`) of an element i are obtained by

```
get_centroid_elem(element,component)
```

where the arguments are

element: number of the surface element of interest in studied cube (`int`). Must be in range [0,`get_Nelements()-1`].

component: number of the component of interest (`int`):
$0 \leftrightarrow t$-component
$1 \leftrightarrow x$-component
$2 \leftrightarrow y$-component
$3 \leftrightarrow z$-component
Must be in range [0,`dimension-1`]

Note that initialisation has to be done only once during the run of the program, unless the size of the volume element or the isovalue changes. Same object can also be used for surface finding in a case with a different dimensionality just by initializing it again with a different number of dimensions.

General remarks

Jacobians

Note that if one uses non-Cartesian coordinates, cornelius does *not* express the normal vector in the covariant components $\Delta\sigma_\mu$ required in the Eq. (1). Instead, if q_μ are the general coordinates, cornelius returns the normal vector components $\Delta\tilde{\sigma}_\mu = (\mathrm{d}q_1\mathrm{d}q_2\mathrm{d}q_3, -\mathrm{d}q_0\mathrm{d}q_2\mathrm{d}q_3, -\mathrm{d}q_0\mathrm{d}q_1\mathrm{d}q_3, -\mathrm{d}q_0\mathrm{d}q_1\mathrm{d}q_2)$, but the covariant components of the same normal vector are

$$\begin{aligned}\Delta\sigma_\mu \;=\; & \left(g_{00}\frac{\sqrt{|g_{11}g_{22}g_{33}|}}{\sqrt{|g_{00}|}}\mathrm{d}q_1\mathrm{d}q_2\mathrm{d}q_3,\; g_{11}\frac{\sqrt{|g_{00}g_{22}g_{33}|}}{\sqrt{|g_{11}|}}\mathrm{d}q_0\mathrm{d}q_2\mathrm{d}q_3, \right. \\ & \left. g_{22}\frac{\sqrt{|g_{00}g_{11}g_{33}|}}{\sqrt{|g_{22}|}}\mathrm{d}q_0\mathrm{d}q_1\mathrm{d}q_3,\; g_{33}\frac{\sqrt{|g_{00}g_{11}g_{22}|}}{\sqrt{|g_{33}|}}\mathrm{d}q_0\mathrm{d}q_1\mathrm{d}q_2 \right),\end{aligned}$$

where $g_{\mu\nu}$ is the corresponding metric tensor. Thus to obtain the covariant components $\Delta\sigma_\mu$, users have to multiply the components $\Delta\tilde{\sigma}_\mu$ by the appropriate factors.

Boost invariant calculation

As an example we show in detail how to use cornelius in a boost invariant calculation. In such a case the problem is how to find a two-dimensional surface element in three dimensional grid, and one has to use either `cornelius2.f90` or initialise `cornelius.cpp` for a three dimensional search by a command `init(3,E0,dx)`. Since the calculation in a boost-invariant case is carried out at midrapidity, the time coordinate τ of Milne coordinates is equal to Cartesian time coordinate t, and the timestep `dt` is equal to the multiple of calculational timesteps. As mentioned, if the coordinates are not Cartesian, cornelius does not provide the covariant components of the normal vector. In boost-invariant case all the components of the normal vector must be multiplied by τ to obtain the covariant components, $\Delta\sigma_i = \tau$`Vmid(i,j)`, or $\Delta\sigma_i = \tau$`get_normal_elem(j,i)`, where the time τ is the sum of time at the earlier time step of the volume element, and the time coordinate of the centroid of the surface element, `Vmid(0,j)`, or `get_centroid_elem(j,0)`.

Centroid

For triangles and tetrahedra the center of mass can be easily evaluated as an average of the coordinates of the corners, but more complicated polygons/polyhedra require more complicated evaluation. For polygons we approximate the centroid by first calculating the average of the coordinates of the corners, and use this point to divide the polygon into triangles. The areas and centers of mass of these triangles are evaluated, and the areas of these triangles are used as masses concentrated in these centers of mass. Thus we've got a system of points of mass, which center of mass is easy to evaluate. We take this center of mass as an approximate centroid of the polygon. Note that if the corners of the polygon are all in the same plane, this procedure gives the actual centroid of the polygon, but if they are not in the same plane, the result is only an approximation. In the latter case the correct centroid depends on how one constructs the polygon out of the cornerpoints.

For polyhedra the approximation of the centroid is again more difficult. As explained in Ref. [2], the procedure of tetrahedronising the polyhedron proceeds via identifying the faces of the polygon and triangularising them. The triangularisation proceeds again via looking for the centroid of these faces (which are polygons). In `cornelius.f90` the centroids of the faces are approximated as an average of the coordinates of their corners, whereas `cornelius.cpp` employs the same procedure than described above to evaluate the centroid of 2D surface elements. If the corners of the face are all in the same plane, this difference does not change the final result, but if they are not, some difference arises. How large, depends on how complicated the structure is. The average of the coordinates of the corners of the polyhedron is then used with the corners of the polyhedron and the approximative centroids of the faces to divide the polyhedron into tetrahedra. The volume and centers of mass of these tetrahedra are evaluated, and the volumes are used as masses centered to the centers of mass. Thus we've again got a set of masspoints, which center of mass is easy to evaluate. We take this center of mass as the approximative centroid of the polyhedron. Analogously to the case of a 2D surface element, how good approximation this is to the actual centroid, depends on whether all the corners of the polyhedron are in the same 3D subspace.

It must be remembered that this procedure for evaluating the centroid of the (hyper)surface elements does not guarantee that the centroid, `Vmid(i,j)` or `get_centroid_elem(j,i)`, is on the tri-/quadrilinearly interpolated isosurface within the volume element. Thus we recommend interpolating also the value of the field which isosurface has been searched for to the value at the approximate centroid. The occasions where the interpolated value significantly deviates from the isovalue are fortunately rare. We have studied the possibilities to constrain the centroid to the tri-/quadrilinearly interpolated isosurface, but so far our attempts have required iterative solution of a nonlinear set of equations, and thus are far too slow.

References

[1] G. Wyvill, C. McPheeters and B. Wyvill, The Visual Computer **2** (1986) 227, `http://dx.doi.org/10.1007/BF01900346` or `www.cpsc.ucalgary.ca/~blob/papers/softobjects.pdf`

[2] P. Huovinen and H. Petersen, arXiv:1206.3371 [nucl-th].

[3] J.P. Boris and D.L. Book, J. Comp. Phys. **11** (1973) 38; D.L. Book, J.P. Boris and K. Hain, J. Comp. Phys. **18** (1975) 248; J.P. Boris and D.L. Book, J. Comp. Phys. **20** (1976) 397.