

Documentation for the Example Implementations for Finding a Spherical Clebsch Map

Albert Chern
Caltech

Felix Knöppel
TU Berlin

Ulrich Pinkall
TU Berlin

Peter Schröder
Caltech

This document explains the example implementations for finding a spherical Clebsch map¹ with **Houdini** 15.0 or later versions. It includes the instruction for running example files, the basic usage for the digital assets [Volume DEC](#) and [Volume Clebsch](#), and the core codes for the algorithm.

Contents

1	Running Example Files	1
1.1	Making Sure SciPy is Installed	2
2	Description for the Example Files	2
2.1	Flow Definition	2
2.2	Clebsch Solver	3
2.3	Clebsch Visualization	4
3	Digital Assets Volume DEC and Volume Clebsch	5
3.1	Implementation in Volume Laplacian	6
3.2	Implementation for Clebsch Ginzburg-Landau Step	8

1 Running Example Files

Our example Houdini files (`.hip` files) run on **Houdini** 15.0 or later for both commercial and non-commercial versions. A free non-commercial version of **Houdini** is available on www.sidefx.com for Windows, Mac and Linux platforms. The Houdini files also require installation of our

¹See our paper: *Inside Fluids: Clebsch Maps for Visualization and Processing*

digital assets `volume_DEC.hda` and `volume_clebsch.hda` in the `./otl/` folder. These examples also require the SciPy library for Python (See Section 1.1).

To run an example program, open the application **Houdini FX** (with non-commercial license installed). Warnings are expected opening our hip files using non-commercial license. `File` → `Open` our example `.hip` files. Then, to import our [Volume DEC](#) and [Volume Clebsch](#) tools, click `File` → `Import` → `Houdini digital asset` and choose both `volume_DEC.hda` and `volume_clebsch.hda` in the `./otl/` folder.

Click the “play animation” button found on the bottom-left to start iteration.

1.1 Making Sure SciPy is Installed

We import the SciPy library for solving sparse linear systems. If you are running **Houdini** on

- **Linux:** Install SciPy from <http://www.scipy.org/install.html>. In particular it is recommended to install the [Anaconda](#) pack version 2.7. On Linux, **Houdini**’s internal Python will read the Python library installed on your machine.
- **MacOSX:** You don’t need to do anything. Mac already has SciPy.
- **Windows:** **Houdini** on Windows installs its own Python inside the Houdini folder in Program Files. So the standardly installed Python packages on your machine are not visible by Houdini. Here is a simple trick. Download and install the Python from the [Anaconda](#) pack version 2.7 (which comes with SciPy). Go to the Houdini folder in Program Files; inside the Houdini folder there is a “python27” folder. Replace (perhaps with a backup) Houdini’s python27 folder by the Python folder installed from Anaconda, and rename it to “python27”. After that your Houdini is able to use SciPy.

2 Description for the Example Files

We explain the functionality of each node in the example file `example_classicHill.hip`. The scene includes three objects: `flow_definition`, `clebsch_solver` and `clebsch_visualization`.

2.1 Flow Definition

In the geometry node `flow_definition` (double click to dive in), we create a velocity field in a 64^3 volume primitive by creating an empty vector volume `vel.x`, `vel.y`, `vel.z`, and use a `volumewrangle` node to specify the vector field using Houdini’s shading language VEX. With the convention where the *y*-axis points up, the velocity in this example is Hill’s spherical vortex





given by

$$\text{vel} = \begin{cases} \left(\frac{3}{2}yx, \frac{3}{2}(1-2|\mathbf{r}|^2) + 1, \frac{3}{2}yz \right)^\top, & |\mathbf{r}| \leq 1 \\ \left(\frac{3yx}{2|\mathbf{r}|^5}, \frac{3y^2-|\mathbf{r}|^2}{2|\mathbf{r}|^5}, \frac{3yz}{2|\mathbf{r}|^5} \right)^\top, & |\mathbf{r}| > 1 \end{cases}$$



where $\mathbf{r} = (x, y, z)$.


The node with label Velocity_Field is a [Null](#) node, which is just a placeholder for the data.


Volume_slice and Volume_trail nodes are for visualizing the vector field by trailing integral curves.

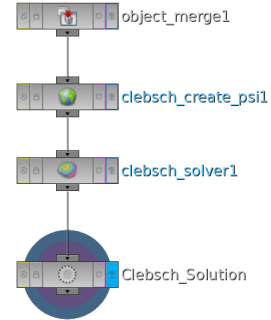
Notes on Geometry Spreadsheet In one of the panels of the Houdini window, choose “Geometry Spreadsheet” to view the data of the selected node of the network. In Geometry Spreadsheet one can view the data on  points,  vertices,  primitives, or  detail (geometry as a whole). For instance, at the nodes where the velocity field is defined, one finds `vel.x`, `vel.y`, `vel.z` in the primitives entry of the Geometry Spreadsheet. For another example, the geometry data on the volume_trail becomes a set of polylines.

2.2 Clebsch Solver

The geometry node of the in-file directory “ obj/ clebsch_solver” includes a network of four nodes: object_merge1, clebsch_create_psi1, clebsch_solver1, and Clebsch_Solution.

The network first reads the velocity field (by object_merge1) by referencing to the flow defined in Section 2.1. The next node (clebsch_create_psi1) creates 7 more volume primitives. In Geometry Spreadsheet,  primitives entry, one observes volumes named `psilre`, `psilim`, `psi2re`, `psi2im`, `s.x`, `s.y`, `s.z` in addition to `vel`’s. They represent $\psi = (\psi_1, \psi_2)^\top \in \mathbb{C}^2$ and $s \in \mathbb{S}^2 \subset \mathbb{R}^3$ stored on each voxel². The ψ variables are initialized randomly.

Subsequently, clebsch_solver1 is the main solver finding the Clebsch map ψ . By default, the overall iteration for the algorithm is driven by the frame number. (Otherwise check “Use For Loop” and specify iteration number.) The parameter interface for clebsch_solver1 is shown in Fig. 1. The parameters include \hbar , the time step Δt for gradient descent, the schedule for ϵ as a function of the iteration number. In addition, one may specify the residual tolerance for the conjugate gradient called by each iteration. Lastly, in the “Error Report” tab in the parameter interface, one may specify the attribute names for the L^2 and L^∞ norm for $\|\eta - \eta_0\|/\|\eta_0\|$. The values for these relative errors can be found in the  Detail Attributes of the Geometry



²Houdini’s voxel corresponds to the vertices \mathcal{V} in our paper.

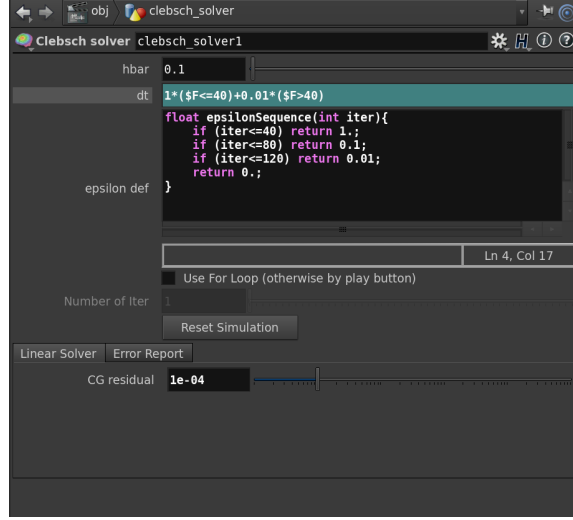




Figure 1: The parameter interface for *clebsch solver*

Spreadsheet. The error $\|\eta - \eta_0\|/\|\eta_0\|$ can be measured disregarding a few layers of voxels away from the boundary; the number of those voxels are specified also in the “Error Report” tab in the parameter interface.

2.3 Clebsch Visualization

In “ obj/ clebsch_visualization”, the network first reads the solution obtained from Section 2.2, and it gives two different schemes to visualize the Clebsch map.

The *clebsch_tubes* node takes pre-images of two regions on \mathbb{S}^2 through $s: M \rightarrow \mathbb{S}^2$. More precisely, what are drawn are the isosurfaces of $\{s.x = \pm a\}$, where $s.x$ is the x -component of s , and a is given by the parameter *Isovalue*. The colors for the two regions can be specified. UV frequency is the frequency for the \mathbb{S}^1 -valued texture map providing vortex-line-aligning brushed textures (a material needs to be assigned to see the result). Parameters for Volume resample controls whether, and by what scheme, a volume upscaling is applied to the volume before iso-surfacing. Enabling Vortex lines supplies a few thickened vortex lines (as pre-images of a few points in the regions on \mathbb{S}^2 for the vortex tubes) in the scene. *Caution: creating the thickened vortex lines geometry is very expensive at the initial frame where ψ and s are random.*

The other node, *clebsch_vortex_lines*, creates the pre-images of a few points on \mathbb{S}^2 given from its 2nd input. Upstream of the 2nd input, a few points were created using the builtin Scatter node. These points were colored according to their positions. We adopt the *Normal Mapping* color scheme $(R, G, B) = \left(\frac{x+1}{2}, \frac{y+1}{2}, \frac{z+1}{2}\right) \in [0, 1]^3$ for each $(x, y, z) \in \mathbb{S}^2$. The space curves are

created as polylines without thickening; *i.e.* it is not overwhelming to view 20 points' pre-images on a 64^3 volume even at the initial frame when s is random.

3 Digital Assets **Volume DEC** and **Volume Clebsch**

We developed the digital assets **Volume DEC** and **Volume Clebsch** for **Houdini**. We give a brief overview of the functionality and some of the implementations of the available nodes included in these packages.

The **Volume DEC** tool includes the following SOP nodes

- **Volume Laplacian** — Caches the sparse matrices for each d and $*$ operator in DEC (discrete exterior calculus) on a regular grid represented by a Houdini volume primitive. The $\nabla = (d - i\alpha)$ operator for complex covariant derivatives is included.
- **Volume Musical Isomorphism** — Implements the \flat and \sharp operators. \flat turns vector fields (vectors on vertices) into circulations on edges (stored on vertices in a staggered grid fashion) whereas \sharp turns circulation on edges into vectors on vertices.
- **Curl** — Takes the curl of a vector field. It calls the builtin VDB analysis tool with boundary artifact removed.
- **Curl Inverse** — Finds a divergence-free vector field whose curl is the given vector field. This is done by solving the Helmholtz-Hodge Decomposition using DEC.
- **Volume Codimension2 Levelset** — Finds the zero set of a complex-valued scalar function represented by two scalar volume primitives (real and imaginary part). The resulting zero set is a set of polylines.

and the **Volume Clebsch** tool includes the following SOP nodes

- **Create Psi** — Creates four scalar volumes for ψ and a 3-vector field for s with the same size as the input volume.
- **Clebsch Solver** — Takes a set of volumes with names `vel.x`, `vel.y`, `vel.z`, `psilre`, `psilim`, `psi2re`, `psi2im`, `s.x`, `s.y`, `s.z`, and compute the Clebsch maps ψ and s (stored in `psi`'s and `s`'s) that fits the given velocity `vel`.
- **Clebsch Vortex Lines** — Takes the volumes `s.x`, `s.y`, `s.z` as a function $s: M \rightarrow \mathbb{S}^2$ from its left input, takes the set of points $\{p_i\}_i \subset \mathbb{S}^2$ from the right input, and produces the space curves $s^{-1}(\{p_i\})$ for each i .
- **Clebsch Tubes** — Takes the volumes `s.x`, `s.y`, `s.z` as a function $s: M \rightarrow \mathbb{S}^2$ and creates the iso-surfaces $\{s_x = \pm a\}$ for some $0 < a < 1$.
- **Clebsch Normalize** — Takes an input ψ and returns $\psi/|\psi|$.
- **Clebsch Compute s** — Takes an input ψ and computes $s = \pi \circ \psi$ where π is the Hopf map.
- **Clebsch Compute vel 1form** — Takes ψ and computes $\eta = \hbar \langle d\psi, i\psi \rangle$.

- **Clebsch Ginzburg-Landau Step** — Computes one gradient descent step for minimizing $\int_M |\nabla^{\eta_0} \psi|_\epsilon^2$.

3.1 Implementation in Volume Laplacian

The following Python script reads the size and the resolution of an input volume (the variable `vol`). These information defines a regular grid. The script finally caches the ∇^{η_0} operator (`d0`, where η_0 are read as the variables `vol_vel`'s), the mass matrix $M = *_0(M)$, the mass matrix for 1-forms $*_1(star1)$ and the magnetic Laplacian $L = -\bar{\nabla}^T *_1 \nabla$.

```
import numpy as np
import numpy.matlib
from scipy.sparse import csr_matrix
from scipy.sparse import diags

node = hou.pwd()          # see http://www.sidefx.com/docs/houdini/hom/hou/
geo = node.geometry()

prims = geo.prims()
vol = prims[0]

res = vol.resolution()
size = vol.indexToPos(res) - vol.indexToPos((0,0,0))
dx = size[0]/res[0]
dy = size[1]/res[1]
dz = size[2]/res[2]

Npoint = res[0]*res[1]*res[2]
Nedgex = (res[0]-1)*res[1]*res[2]
Nedgex = res[0]*(res[1]-1)*res[2]
Nedgez = res[0]*res[1]*(res[2]-1)
Nedge = Nedgex+Nedgex+Nedgez

# geometry types
point = lambda x,y,z: x+y*res[0]+z*res[0]*res[1]
edgex = lambda x,y,z: x+y*(res[0]-1)+z*(res[0]-1)*res[1]
edgex = lambda x,y,z: x+y*res[0]+z*res[0]*(res[1]-1) + Nedgex
edgez = lambda x,y,z: x+y*res[0]+z*res[0]*res[1] + Nedgex+Nedgex

# master index II.
# II[:,_,_,_] is the 3D slicing which returns 3-component sliced indices.
II = np.array(np.meshgrid(range(res[0]), range(res[1]), range(res[2]), indexing='ij')
              )

# Ind(geotype, IIslice) returns the index of geotype
Ind = lambda geotype, IIslice: geotype(IIslice[0], IIslice[1], IIslice[2])
flatten = lambda mat: np.reshape(mat, -1, order='F')

# ADJACENCY
edgexSrc = flatten(Ind(point, II[:,0:-1, :, :]))
edgexDst = flatten(Ind(point, II[:,1:  , :, :]))
edgeySrc = flatten(Ind(point, II[:, :,0:-1, :]))
edgeyDst = flatten(Ind(point, II[:, :,1:  , :]))
edgezSrc = flatten(Ind(point, II[:, :, :,0:-1]))
```

```

edgezDst = flatten(Ind(point,II[:, :, :, 1: ]))

# BOUNDARY
IsXBdy = np.zeros(np.shape(II[0]), dtype=bool)
IsYBdy = np.zeros(np.shape(II[0]), dtype=bool)
IsZBdy = np.zeros(np.shape(II[0]), dtype=bool)
IsXBdy[[0,-1], :, :] = True
IsYBdy[:, [0,-1], :] = True
IsZBdy[:, :, [0,-1]] = True

# WEIGHT
pointW = dx*dy*dz/(1.+IsXBdy.astype(float)) \
           /(1.+IsYBdy.astype(float)) \
           /(1.+IsZBdy.astype(float))

edgexW = dy*dz/dx/(1.+IsYBdy[0:-1, :, :].astype(float)) \
           /(1.+IsZBdy[0:-1, :, :].astype(float))
edgeyW = dz*dx/dy/(1.+IsZBdy[:, 0:-1, :].astype(float)) \
           /(1.+IsXBdy[:, 0:-1, :].astype(float))
edgezW = dx*dy/dz/(1.+IsXBdy[:, :, 0:-1].astype(float)) \
           /(1.+IsYBdy[:, :, 0:-1].astype(float))

# BUILD MATRICES

# read rotation 1-form
velInd = geo.intAttribValue("velInd") # velInd has stored the index
                                         # for the velocity volume

vol_velx = prims[velInd]
vol_vely = prims[velInd+1]
vol_velz = prims[velInd+2]
velx = np.array(vol_velx.allVoxels())
vely = np.array(vol_vely.allVoxels())
velz = np.array(vol_velz.allVoxels())
edgeeta = np.concatenate((velx[edgexSrc], vely[edgeySrc], velz[edgezSrc]))

#d0
edgeInd = range(Nedge)
d0row = np.append(edgeInd, edgeInd)
d0col = np.concatenate((edgexSrc, edgeySrc, edgezSrc, edgexDst, edgeyDst, edgezDst))
d0val = np.append(-np.ones(Nedge), np.exp(-1j*edgeeta))
d0 = csr_matrix((d0val, (d0row, d0col)), (Nedge, Npoint))

#star0
pointweight = flatten(pointW)
edgeweight = np.concatenate((flatten(edgexW), flatten(edgeyW), flatten(edgezW)))
star0 = diags(pointweight, 0)
star1 = diags(edgeweight, 0)

# laplace and mass
L = -d0.getH().dot(star1.dot(d0))
M = star0

# Cache
cache_node = hou.node("..");

cache_node.setCachedUserData("L", L)
cache_node.setCachedUserData("M", M)

```

```

cache_node.setCachedUserData("d0",d0)
cache_node.setCachedUserData("star1",star1)
# `chs('...')` will evaluate into a string for the names of cached data
# accessible from other Python nodes.

```

3.2 Implementation for Clebsch Ginzburg-Landau Step

The following Python script builds the matrix L of Appendix C in our paper, and solves the gradient descent step $(M_V + \Delta t L)\psi^{(\text{new})} = M_V \psi$ presented in Appendix C of the paper. Formally, the matrix L is built by

$$L = \overline{\nabla^{\eta_0}}^T W \nabla^{\eta_0}$$

where W is a diagonal matrix (with size the number of edges) of 4×4 blocks of $(*)_e(P^\epsilon)_e$, where $(*)_e \in \mathbb{R}$ is the edge weight and $(P^\epsilon)_e$ is the projection operator for each edge $e \in \mathcal{E}$. Note that for the convenience of the coding, we adopt a permutation so that the diagonal of 4×4 blocks is written as 4×4 blocks of diagonal matrix.

```

node = hou.pwd()
geo = node.geometry()

import numpy as np
from scipy.sparse.linalg import gmres
from scipy.sparse.linalg import cg
from scipy.sparse import diags
from scipy.sparse import csr_matrix
import scipy.sparse as sp

DEC = hou.node("../`chs('../pathToVolumeLaplace')`")
prims = geo.prims()
psiInd = geo.intAttribValue("psiInd")

# READ VOXELS
psilre = np.array(prims[psiInd ].allVoxels())
psilim = np.array(prims[psiInd+1].allVoxels())
psi2re = np.array(prims[psiInd+2].allVoxels())
psi2im = np.array(prims[psiInd+3].allVoxels())
sx = np.array(prims[psiInd+4].allVoxels())
sy = np.array(prims[psiInd+5].allVoxels())
sz = np.array(prims[psiInd+6].allVoxels())

# READ DEC MATRICES
M = DEC.cachedUserData("M")
d0 = DEC.cachedUserData("d0")
star1 = DEC.cachedUserData("star1")

# READ PARAMETERS
dt = hou.evalParm("../dt") # time step
epsilon = hou.evalParm("../epsilon") # \epsilon
cgres = hou.evalParm("../cgres") # conjugate gradient tolerance
if (cgres<1e-05):

```



```

    cgres=1e-05

# EDGE SPIN
averageOp = np.abs(d0)
sx_edge = averageOp.dot(sx)
sy_edge = averageOp.dot(sy)
sz_edge = averageOp.dot(sz)
tmp_norm = np.sqrt(sx_edge**2 + sy_edge**2 + sz_edge**2)
sx_edge/=tmp_norm
sy_edge/=tmp_norm
sz_edge/=tmp_norm

# PAULI SPIN & PROJECTION
Zx = sz_edge;
Zy = -sy_edge;
Zz = sx_edge;

proj11 = 0.5 + 0.5*Zz;
proj22 = 0.5 - 0.5*Zz;
proj12 = 0.5*Zx -1j*0.5*Zy;

w11 = epsilon + (1-epsilon) * proj11
w12 = (1-epsilon) * proj12
w21 = np.conj(w12)
w22 = epsilon + (1-epsilon) * proj22

# ASSEMBLE BLOCK SYSTEM MATRICES

# edge weight
W11 = diags(w11,0).dot(star1)
W12 = diags(w12,0).dot(star1)
W21 = diags(w21,0).dot(star1)
W22 = diags(w22,0).dot(star1)
W = sp.vstack([
    sp.hstack([W11,W12]),
    sp.hstack([W21,W22])
])

# d operator
Zd = csr_matrix(np.shape(d0),dtype=np.dtype(d0))
D = sp.vstack([
    sp.hstack([d0,Zd]),
    sp.hstack([Zd,d0])
])

# Berger magnetic laplacian (positive definite convention)
L = D.getH().dot(W.dot(D))

# mass matrix
ZM = csr_matrix(np.shape(M),dtype=np.dtype(M))
MM = sp.vstack([
    sp.hstack([M,ZM]),
    sp.hstack([ZM,M])
])

# SOLVE LINEAR SYSTEM
psil = psilre + 1j*psilim

```

```

psi2 = psi2re + 1j*psi2im
Psi = np.append(psi1,psi2)

LeftOp  = MM + dt* L
RightOp = MM

# for recording number of CG iterations
num_iters = 0
def counteriter(xk):
    global num_iters
    num_iters += 1

Psi,tmp = cg(LeftOp,RightOp.dot(Psi),tol=cgres,callback=counteriter)

psi1 = Psi[0:len(psi1)]
psi2 = Psi[len(psi1):]

# SET RESULT
prims[psiInd ].setAllVoxels(psi1.real)
prims[psiInd+1].setAllVoxels(psi1.imag)
prims[psiInd+2].setAllVoxels(psi2.real)
prims[psiInd+3].setAllVoxels(psi2.imag)

geo.setGlobalAttribValue("last_cg_iters",num_iters)

```