alecjacobson / **geometry-processing-mesh-reconstruction**

Mesh Reconstruction assignment for Geometry Processing course

github.com/alecjacobson/geometry-processing

MPL-2.0 License

43 stars **67** forks

Star Watch

Code Issues 5 Pull requests 34 Actions Projects Wiki

master

alecjacobson another bump? this time from local ... 22 hours ago **48**

View code

README.md

Geometry Processing – Mesh Reconstruction

> **To get started:** Clone this repository then issue
>
> ```
> git clone --recursive http://github.com/alecjacobson/geometry-processing-mesh-reconstruction.git
> ```

Installation, Layout, and Compilation

See introduction.

Execution

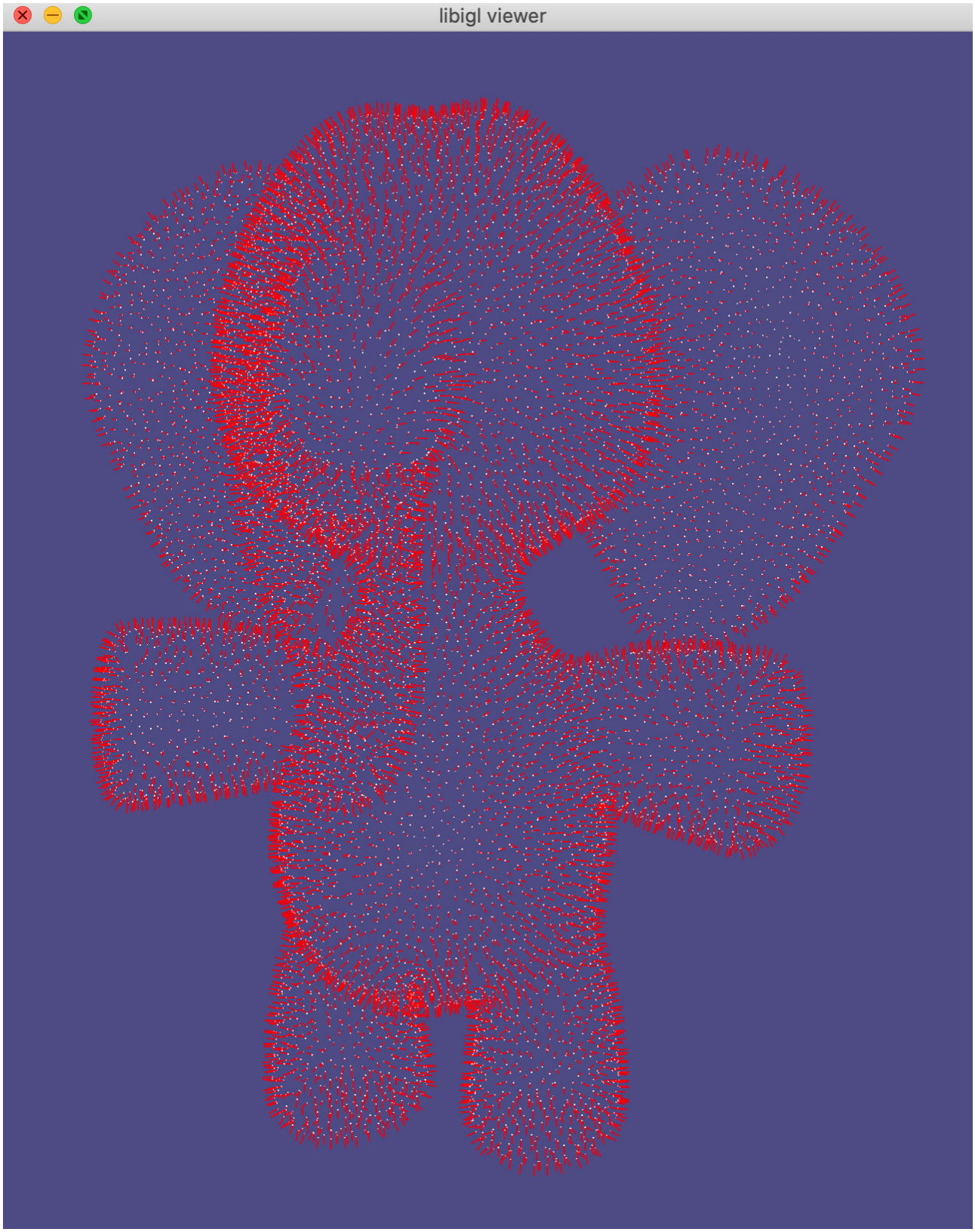
Once built, you can execute the assignment from inside the `build/` using

```
./mesh-reconstruction [path to point cloud]
```

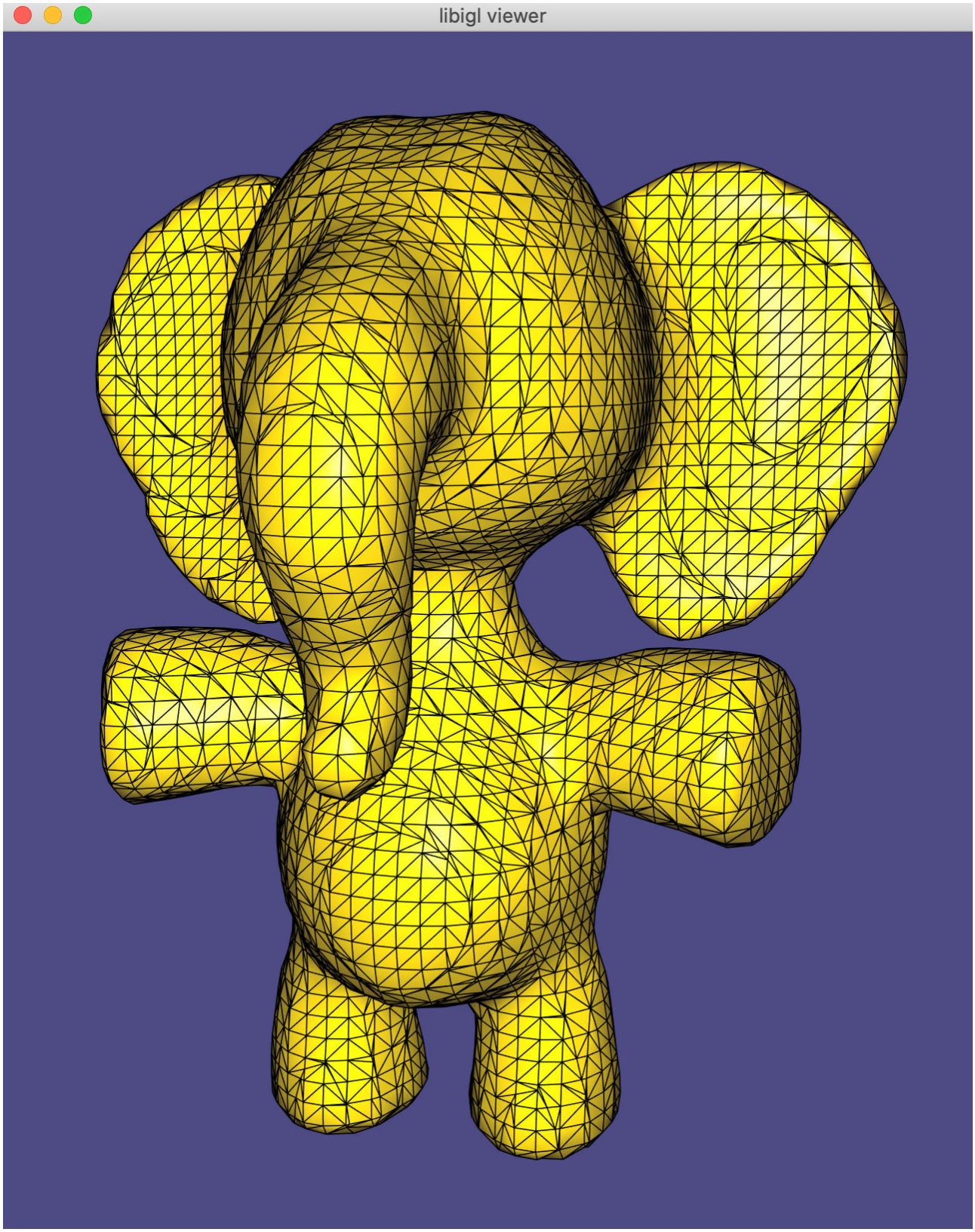
Background

In this assignment, we will be implementing a simplified version of the method in "Poisson Surface Reconstruction" by Kazhdan et al. 2006. (Your first "task" will be to read and understand this paper).

Many scanning technologies output a set of n point samples $\mathbf{P}$ on the surface of the object in question. From these points and perhaps the location of the camera, one can also estimate normals $\mathbf{N}$ to the surface for each point $\mathbf{p} \in \mathbf{P}$. This image shows the `data/elephant.pwn` input data with a white dot for each point and a red line segment pointing outward for each corresponding normal vector.



For shape analysis, visualization and other downstream geometry processing phases, we would like to convert this finitely sampled *point cloud* data into an *explicit continuous surface representation*: i.e., a triangle mesh (a special case of a polygon mesh). This image shows the corresponding output mesh for `data/elephant.pwn` input data above:



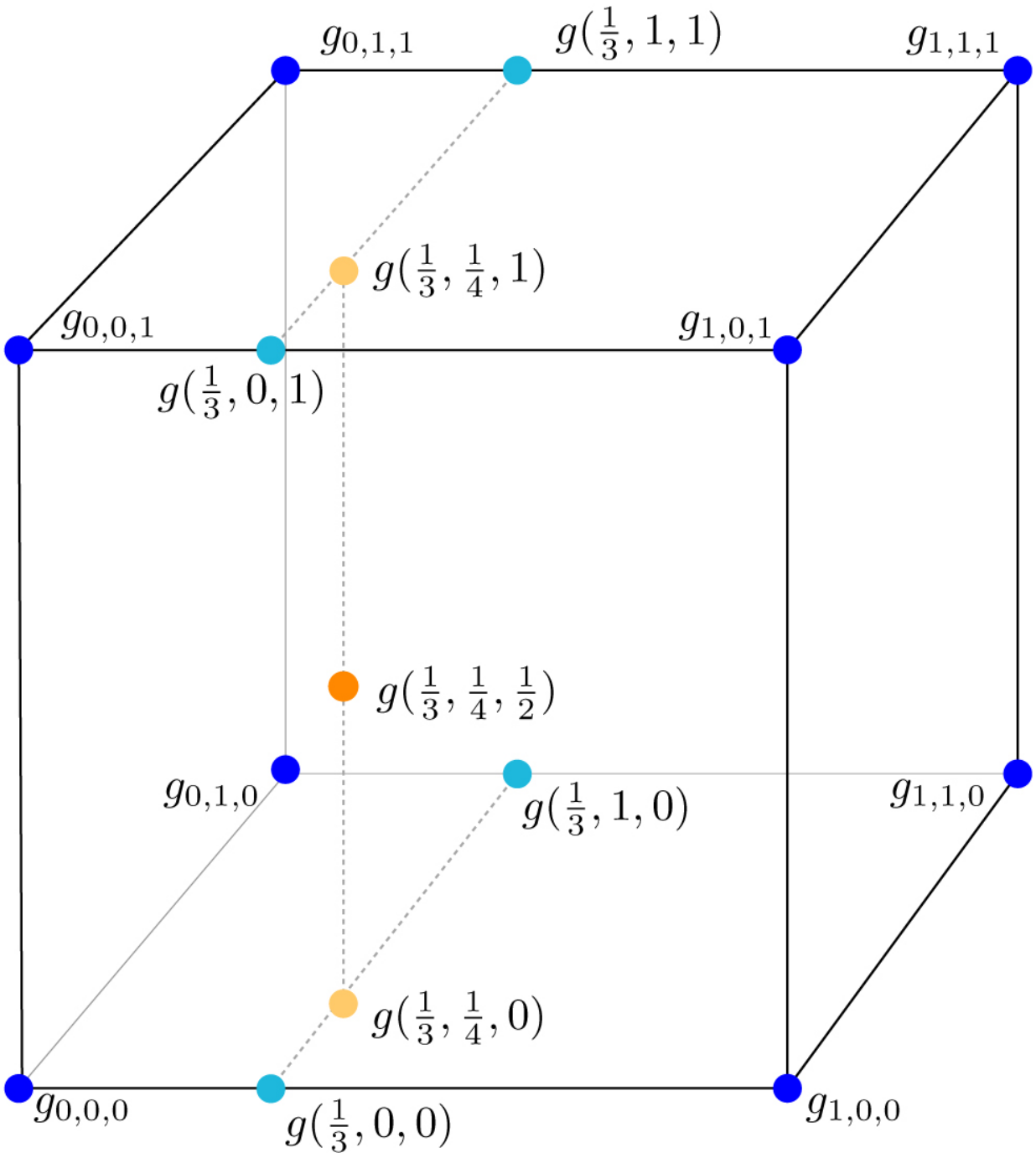
Voxel-based Implicit Surface

Converting the point cloud directly to a triangle mesh makes it very difficult to ensure that the mesh meets certain *topological* postconditions: i.e., that it is manifold, closed, and has a small number of holes.

Instead we will first convert the point cloud *sampling representation* into a an *implicit surface representation*: where the unknown surface is defined as the level-set of some function $g : \mathbf{R}^3 \Rightarrow \mathbf{R}$ mapping all points in space to a scalar value. For example, we may define the surface $\partial\mathbf{S}$ of some solid, volumetric shape $\mathbf{S}$ to be all points $\mathbf{x} \in \mathbf{R}^3$ such that $g(x) = \sigma$, where we may arbitrarily set $\sigma = \frac{1}{2}$.

$$\partial\mathbf{S} = \{\mathbf{x} \in \mathbf{R}^3 | g(\mathbf{x}) = \sigma\}.$$

On the computer, it is straightforward discretize an implicit function. We define a regular 3D grid of voxels containing at least the bounding box of $\mathbf{S}$. At each node in the grid $\mathbf{x}_{i,j,k}$ we store the value of the implicit function $g(\mathbf{x}_{i,j,k})$. This defines g *everywhere* in the grid via trilinear interpolation.



For example, consider a point $\mathbf{x} = (\frac{1}{3}, \frac{1}{4}, \frac{1}{2})$ lying in the middle of the bottom-most, front-most, left-most cell. We know the values at the eight corners. Trilinear interpolation can be understood as linear interpolation in the x-direction by $\frac{1}{3}$ on each x-axis-aligned edge, resulting in four values *living* on the same plane. These can then be linearly interpolated in the y direction by $\frac{1}{4}$ resulting in two points on the same line, and finally in the z direction by $\frac{1}{2}$ to get to our evaluation point $(\frac{1}{3}, \frac{1}{4}, \frac{1}{2})$.

An implicit surface stored as the level-set of a trilinearly interpolated grid can be *contoured* into a triangle mesh via the Marching Cubes Algorithm. For the purposes of this assignment, we will treat this as a black box. Instead, we focus on determining what values for g to store on the grid.

Characteristic functions of solids

We assume that our set of points $\mathbf{P}$ lie on the surface $\partial\mathbf{S}$ of some physical solid object $\mathbf{S}$. This solid object must have some non-trivial volume that we can calculate *abstractly* as the integral of unit density over the solid:

$$\int_{\mathbf{S}} 1 \; dA.$$

We can rewrite this definite integral as an indefinite integral over all of $\mathbf{R}^3$:

$$\int_{\mathbf{R}^3} \chi_{\mathbf{S}}(\mathbf{x}) \; dA,$$

by introducing the characteristic function of $\mathbf{S}$, that is *one* for points inside of the shape and *zero* for points outside of $\mathbf{S}$:

$$\chi_{\mathbf{S}}(\mathbf{x}) = \begin{cases} 1 & \text{if } \mathbf{x} \in \mathbf{S} \\ 0 & \text{otherwise.} \end{cases}$$

Compared to typical implicit surface functions, this function represents the surface $\partial\mathbf{S}$ of the shape $\mathbf{S}$ as the *discontinuity* between the one values and the zero values. Awkwardly, the gradient of the characteristic function $\nabla\chi_{\mathbf{S}}$ is *not defined* along $\partial\mathbf{S}$.

One of the key observations made in [Kazhdan et al. 2006] is that the gradient of a infinitesimally mollified (smoothed) characteristic function:

1. points in the direction of the normal near the surface $\partial\mathbf{S}$, and
2. is zero everywhere else.

Our goal will be to use our points $\mathbf{P}$ and normals $\mathbf{N}$ to *optimize* an implicit function g over a regular grid, so that its gradient ∇g meets these two criteria. In that way, our g will be an approximation of the mollified characteristic function.

Poisson surface reconstruction

Or: how I learned to stop worrying and minimize squared Gradients

Let us start by making two assumptions:

1. we know how to compute ∇g at each node location $\mathbf{x}_{i,j,k}$, and
2. our input points $\mathbf{P}$ all lie perfectly at grid nodes: $\exists\ \mathbf{x}_{i,j,k} = \mathbf{p}_\ell$.

We will find out these assumptions are not realistic and we will have to relax them (i.e., we ***will not*** make these assumptions in the completion of the tasks). However, it will make the following algorithmic description easier on the first pass.

If our points $\mathbf{P}$ lie at grid points, then our corresponding normals $\mathbf{N}$ also *live* at grid points. This leads to a very simple set of linear equations to define a function g with a gradient equal to the surface normal at the surface and zero gradient away from the surface:

$$\nabla g(\mathbf{x}_{i,j,k}) = \mathbf{v}_{i,j,k} := \begin{cases} \mathbf{n}_\ell & \text{if } \exists\ \mathbf{p}_\ell = \mathbf{x}_{i,j,k}, \\ \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} & \text{otherwise.} \end{cases}$$

This is a *vector-valued* equation. The gradients, normals and zero-vectors are three-dimensional (e.g., $\nabla g \in \mathbf{R}^3$). In effect, this is *three equations* for every grid node.

Since we only need a single number at each grid node (the value of g), we have *too many* equations.

Like many geometry processing algorithms confronted with such an over determined, we will *optimize* for the solution that best *minimizes* the error of equation:

$$\|\nabla g(\mathbf{x}_{i,j,k}) - \mathbf{v}_{i,j,k}\|^2.$$

We will treat the error of each grid location equally by minimizing the sum over all grid locations:

$$\min_{\mathbf{g}} \sum_i \sum_j \sum_k \frac{1}{2} \|\nabla g(\mathbf{x}_{i,j,k}) - \mathbf{v}_{i,j,k}\|^2,$$

where $\mathbf{g}$ (written in boldface) is a vector of *unknown* grid-nodes values, where $g_{i,j,k} = g(\mathbf{x}_{i,j,k})$.

Part of the convenience of working on a regular grid is that we can use the finite difference method to approximate the gradient ∇g on the grid.

After revisiting our assumptions, we will be able to compute approximations of the x-, y- and z-components of ∇g via a sparse matrix multiplication of a "gradient matrix" $\mathbf{G}$ and our vector of unknown grid values $\mathbf{g}$. We will be able to write the minimization problem above in matrix form:

$$\min_{\mathbf{g}} \frac{1}{2}\|\mathbf{G}\mathbf{g} - \mathbf{v}\|^2,$$

or equivalently after expanding the norm:

$$\min_{\mathbf{g}} \frac{1}{2}\mathbf{g}^\mathsf{T}\mathbf{G}^\mathsf{T}\mathbf{G}\mathbf{g} - \mathbf{g}^\mathsf{T}\mathbf{G}^\mathsf{T}\mathbf{v} + \text{constant},$$

This is a quadratic "energy" function of the variables of $\mathbf{g}$, its minimum occurs when an infinitesimal change in $\mathbf{g}$ produces no change in the energy:

$$\frac{\partial}{\partial \mathbf{g}} \frac{1}{2}\mathbf{g}^\mathsf{T}\mathbf{G}^\mathsf{T}\mathbf{G}\mathbf{g} - \mathbf{g}^\mathsf{T}\mathbf{G}^\mathsf{T}\mathbf{v} = 0.$$

Applying this derivative gives us a *sparse* system of linear equations

$$\mathbf{G}^\mathsf{T}\mathbf{G}\mathbf{g} = \mathbf{G}^\mathsf{T}\mathbf{v}.$$

We will assume that we can solve this using a black box sparse solver.

Now, let's revisit our assumptions.

Gradients on a regular grid

The gradient of a function g in 3D is nothing more than a vector containing partial derivatives in each coordinate direction:

$$\nabla g(\mathbf{x}) = \begin{pmatrix} \frac{\partial g(\mathbf{x})}{\partial x} \\ \frac{\partial g(\mathbf{x})}{\partial y} \\ \frac{\partial g(\mathbf{x})}{\partial z} \end{pmatrix}.$$

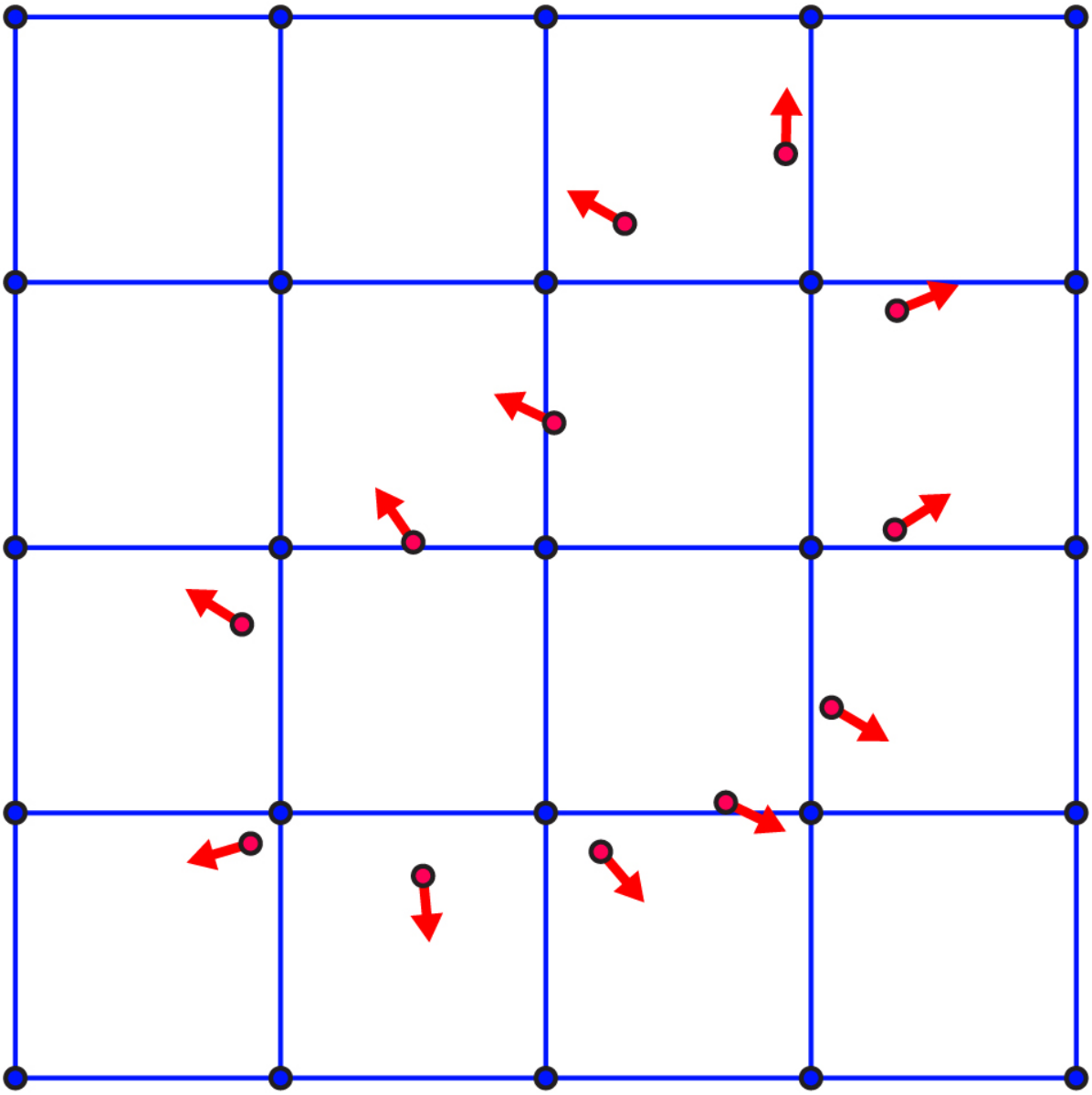
We will approximate each partial derivative individually. Let's consider the partial derivative in the x direction, $\partial g(\mathbf{x})/\partial x$, and we will assume without loss of generality that what we derive applies *symmetrically* for y and z.

The partial derivative in the x-direction is a one-dimensional derivative. This couldn't be easier to do with finite differences. We approximate the derivative of the function g with respect to the x direction is the difference between the function evaluated at one grid node and at the grid node *before* it in the x-direction then divided by the spatial distance between adjacent nodes h (i.e., the grid step size):

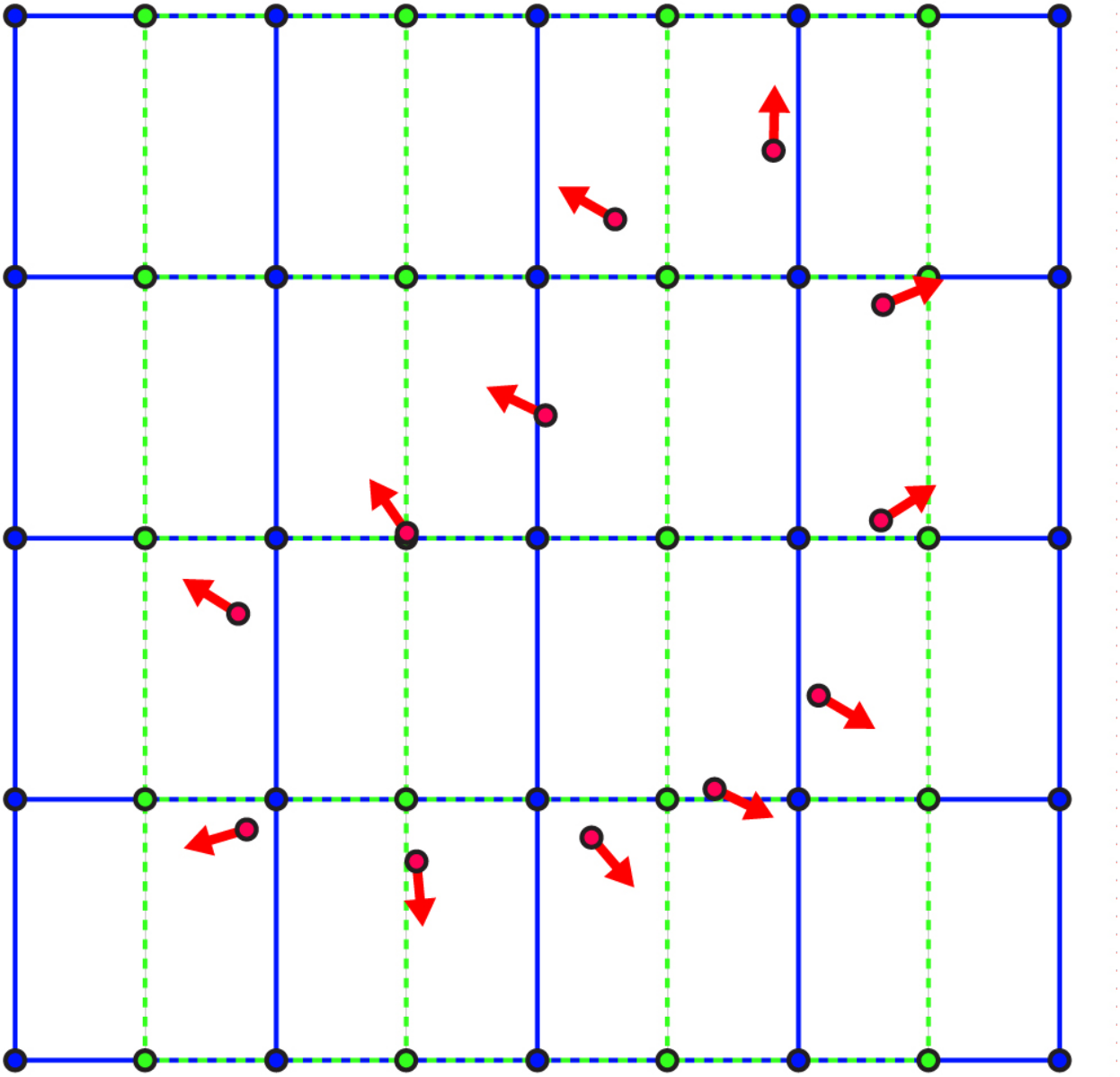
$$\frac{\partial g(\mathbf{x}_{i-\frac{1}{2},j,k})}{\partial x} = \frac{g_{i,j,k} - g_{i-1,j,k}}{h},$$

where we use the index $i - \frac{1}{2}$ to indicate that this derivative in the x-direction lives on a staggered grid *in between* the grid nodes where the function values for g.

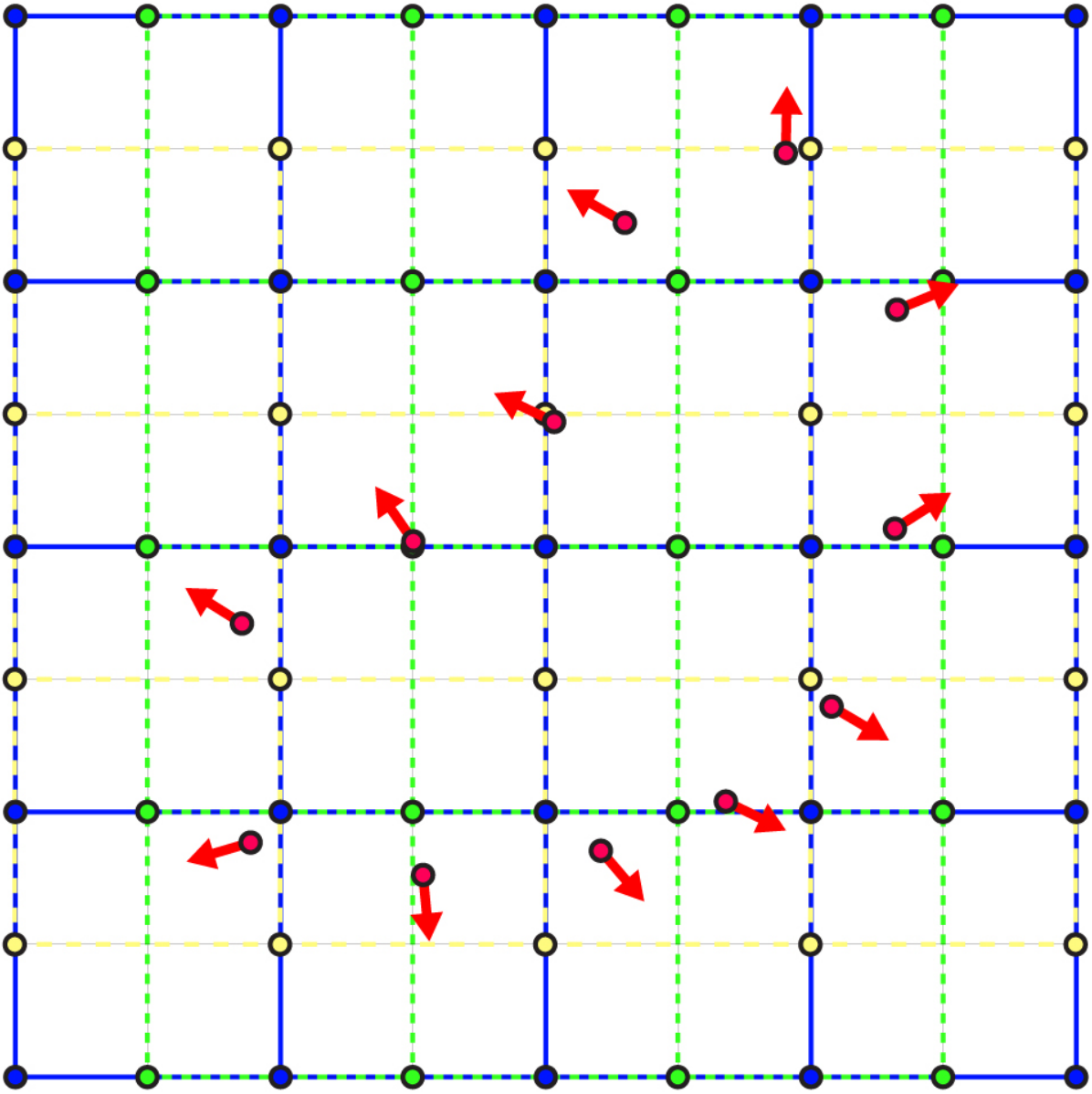
The following pictures show a 2D example, where g lives on the nodes of a 5×5 blue grid:



The partial derivatives of g with respect to the x-direction $\partial g(\mathbf{x})/\partial x$ live on a 4×5 green, staggered grid:



The partial derivatives of g with respect to the y-direction $\partial g(\mathbf{x})/\partial y$ live on a 5×4 yellow, staggered grid:



Letting $\mathbf{g} \in \mathbf{R}^{n_x n_y n_z \times 1}$ be column vector of function values on the *primary grid* (blue in the example pictures), we can construct a sparse matrix $\mathbf{D}^x \in \mathbf{R}^{(n_x-1) n_y n_z \times n_x n_y n_z}$ so that each row $\mathbf{D}^x_{i-\frac{1}{2},j,k} \in \mathbf{R}^{1 \times n_x n_y n_z}$ computes the partial derivative at the corresponding staggered grid location $\mathbf{x}_{i-\frac{1}{2},j,k}$. The ℓth entry in that row receives a value only for neighboring primary grid nodes:

$$\mathbf{D}^x_{i-\frac{1}{2},j,k}(\ell) = \begin{cases} -1 & \text{if } \ell = i-1 \\ 1 & \text{if } \ell = i \\ 0 & \text{otherwise} \end{cases}.$$

> **Indexing 3D arrays**

> Now, obviously in our code we cannot *index* the column vector $\mathbf{g}$ by a triplet of numbers $\{i,j,k\}$ or the rows of $\mathbf{D}^x$ by the triplet $i-\frac{1}{2},j,k$. We will assume that $\mathbf{g}_{i,j,k}$ refers to `g(i+j*n_x+k*n_y*n_x)`. Similarly, for the staggered grid subscripts $i-\frac{1}{2},j,k$ we will assume that $\mathbf{D}^x_{i-\frac{1}{2},j,k}(\ell)$ refers to the matrix entry `Dx(i+j*n_x+k*n_y*n_x,l)`, where the $i-\frac{1}{2}$ has been *rounded down*.

We can similarly build matrices $\mathbf{D}^y$ and $\mathbf{D}^z$ and *stack* these matrices vertically to create a gradient matrix $\mathbf{G}$:

$$\mathbf{G} = \begin{pmatrix} \mathbf{D}^x \\ \mathbf{D}^y \\ \mathbf{D}^z \end{pmatrix} \in \mathbf{R}^{((n_x-1)n_yn_z+n_x(n_y-1)n_z+n_xn_y(n_z-1))\times n_xn_yn_z}$$

This implies that our vector $\mathbf{v}$ of zeros and normals in our minimization problem should not *live* on the primary, but rather it, too, should be broken into x-, y- and z-components that live of their resepctive staggered grids:

$$\mathbf{v} = \begin{pmatrix} \mathbf{v}^x \\ \mathbf{v}^y \\ \mathbf{v}^z \end{pmatrix} \in \mathbf{R}^{((n_x-1)n_yn_z+n_x(n_y-1)n_z+n_xn_y(n_z-1))\times 1}.$$

This leads to addressing our second assumption.

B-b-b-b-but the input normals might not be at grid node locations?

At this point, we would *actually* liked to have had that our input normals were given component-wise on the staggered grid. Then we could immediate stick them into $\mathbf{v}$. But this doesn't make much sense as each normal $\mathbf{n}_\ell$ *lives* at its associated point $\mathbf{p}_\ell$, regardless of any grids.

To remedy this, we will distribute each component of each input normal $\mathbf{n}_\ell$ to $\mathbf{v}$ at the corresponding staggered grid node location.

For example, consider the normal $\mathbf{n}$ at some point $\mathbf{x}_{1,\frac{1}{4},\frac{1}{2}}$. Conceptually, we'll think of the x-component of the normal n_x as floating in the staggered grid corresponding to $\mathbf{D}^x$, in between the eight staggered grid locations:

$$\mathbf{x}_{\frac{1}{2},0,0}, \mathbf{x}_{1\frac{1}{2},0,0}, \mathbf{x}_{\frac{1}{2},1,0}, \mathbf{x}_{1\frac{1}{2},1,0}, \mathbf{x}_{\frac{1}{2},0,1}, \mathbf{x}_{1\frac{1}{2},0,1}, \mathbf{x}_{\frac{1}{2},1,1}, \text{ and } \mathbf{x}_{1\frac{1}{2},1,1}$$

Each of these staggered grid nodes has a corresponding x value in the vector $\mathbf{v}^x$.

We will distribute n_x to these entries in $\mathbf{v}^x$ by *adding* a partial amount of n_x to each. I.e.,

$$v^x_{\frac{1}{2},0,0} = w_{\frac{1}{2},0,0}\left(\mathbf{x}_{1,\frac{1}{4},\frac{1}{2}}\right)\ n_x, v^x_{1\frac{1}{2},0,0} = w_{1\frac{1}{2},0,0}\left(\mathbf{x}_{1,\frac{1}{4},\frac{1}{2}}\right)\ n_x, \vdots v^x_{1\frac{1}{2},1,1} = w_{1\frac{1}{2},1,1}\left(\mathbf{x}_{1,\frac{1}{4},\frac{1}{2}}\right)\ n_x.$$

where $w_{i+\frac{1}{2},j,k}(\mathbf{p})$ is the trilinear interpolation *weight* associate with staggered grid node $\mathbf{x}_{i+\frac{1}{2},j,k}$ to interpolate a value at the point $\mathbf{p}$. The trilinear interpolation weights so that:

$$n_x = w_{\frac{1}{2},0,0}(\mathbf{x}_{1,\frac{1}{4},\frac{1}{2}})\ v^x_{\frac{1}{2},0,0} + w_{1\frac{1}{2},0,0}(\mathbf{x}_{1,\frac{1}{4},\frac{1}{2}})\ v^x_{1\frac{1}{2},0,0} + \vdots w_{1\frac{1}{2},1,1}(\mathbf{x}_{1,\frac{1}{4},\frac{1}{2}})\ v^x_{1\frac{1}{2},1,1}.$$

Since we need to do these for the x-component of each input normal, we will assemble a sparse matrix $\mathbf{W}^x \in n \times (n_x-1)n_y n_z$ that *interpolates* $\mathbf{v}^x$ at each point $\mathbf{p}$:

$$(\mathbf{W}^x\mathbf{v}^x) \in \mathbf{R}^{n\times 1}$$

the transpose of $\mathbf{W}^x$ is not quite its *inverse*, but instead can be interpreted as *distributing* values onto staggered grid locations where $\mathbf{v}^x$ lives:

$$\mathbf{v}^x = (\mathbf{W}^x)^\mathsf{T}\mathbf{N}^x.$$

Using this definition of $\mathbf{v}^x$ and analogously for $\mathbf{v}^y$ and $\mathbf{v}^z$ we can construct the vector $\mathbf{v}$ in our energy minimization problem above.

BTW, what's Poisson got to do with it?

The discrete energy minimization problem we've written looks like the squared norm of some gradients. An analogous energy in the smooth world is the Dirichlet energy:

$$E(g) = \int_\Omega \|\nabla g\|^2 dA$$

to *minimize* this energy with respect to g as an unknown *function*, we need to invoke Calculus of Variations and Green's First Identity. In doing so we find that minimizers will satisfy:

$$\nabla\cdot\nabla g = 0 \text{ on } \Omega,$$

known as Laplaces' Equation.

If we instead start with a slightly different energy:

$$E(g) = \int_\Omega \|\nabla g - V\|^2 dA,$$

where V is a vector-valued function. Then applying the same machinery we find that minimizers will satisfy:

$$\nabla\cdot\nabla g = \nabla\cdot V \text{ on } \Omega,$$

known as Poisson's equation.

> Notice that if we interpret the transpose of our gradient matrix $\mathbf{G}^\mathsf{T}$ as a *divergence matrix* (we can and we should), then the structure of these smooth energies and equations are directly preserved in our discrete energies and equations.
>
> This kind of *structure preservation* is a major criterion for judging discrete methods.

Choosing a good iso-value

Constant functions have no gradient. This means that we can add a constant function to our implicit function g without changing its gradient:

$$\nabla g = \nabla(g + c) = \nabla g + \nabla c = \nabla g + 0.$$

The same is true for our discrete gradient matrix $\mathbf{G}$: if the vector of grid values $\mathbf{g}$ is constant then $\mathbf{G}\mathbf{g}$ will be a vector zeros.

This is potentially problematic for our least squares solve: there are many solutions, since we can just add a constant. Fortunately, we *don't really care*. It's elegant to say that our surface is defined at $g = 0$, but we'd be just as happy declaring that our surface is defined at $g = c$.

To this end we just need to find *a solution* $\mathbf{g}$, and then to pick a good iso-value σ.

As suggested in [Kazhdan et al. 2006], we can pick a good iso-value by interpolating our solution $\mathbf{g}$ at each of the input points (since we know they're on the surface) and averaging their values. For an appropriate interpolation matrix $\mathbf{W}$ on the *primary (non-staggered) grid* this can be written as:

$$\sigma = \frac{1}{n}\mathbf{1}^\mathsf{T}\mathbf{W}\mathbf{g},$$

where $\mathbf{1} \in \mathbf{R}^{n \times 1}$ is a vector of ones.

Just how much does this assignment simplify [Kazhdan et al. 2006]?

Besides the insights above, a major contribution of [Kazhdan et al. 2006] was to setup and solve this problem on an adaptive grid rather than a regular grid. They also track "confidence" of their input data effecting how they smooth and interpolate values. As a result, their method is one of the most highly used surface reconstruction techniques to this day.

> Consider adding your own insights to the wikipedia entry for this method.

Tasks

Read [Kazhdan et al. 2006]

This reading task is not directly graded, but it's expected that you read and understand this paper before moving on to the other tasks.

`src/fd_interpolate.cpp`

Given a regular finite-difference grid described by the number of nodes on each side (`nx` , `ny` and `nz`), the grid spacing (`h`), and the location of the bottom-left-front-most corner node (`corner`), and a list of points (`P`), construct a sparse matrix `W` of trilinear interpolation weights so that `P = W * x` .

`src/fd_partial_derivative.cpp`

Given a regular finite-difference grid described by the number of nodes on each side (`nx` , `ny` and `nz`), the grid spacing (`h`), and a desired direction, construct a sparse matrix `D` to compute first partial derivatives in the given direction onto the *staggered grid* in that direction.

`src/fd_grad.cpp`

Given a regular finite-difference grid described by the number of nodes on each side (`nx` , `ny` and `nz`), and the grid spacing (`h`), construct a sparse matrix `G` to compute gradients with each component on its respective staggered grid.

> **Hint**
>
> Use `fd_partial_derivative` to compute `Dx` , `Dy` , and `Dz` and then simply concatenate these together to make `G` .

`src/poisson_surface_reconstruction.cpp`

Given a list of points `P` and the list of corresponding normals `N` , construct a implicit function `g` over a regular grid (*built for you*) using approach described above.

You will need to *distribute* the given normals `N` onto the staggered grid values in `v` via sparse trilinear interpolation matrices `Wx` , `Wy` and `Wz` for each staggered grid.

Then you will need to construct and solve the linear system $\mathbf{G}^\mathsf{T}\mathbf{G}\mathbf{g} = \mathbf{G}^\mathsf{T}\mathbf{v}$.

Determine the iso-level `sigma` to extract from the `g` .

Feed this implicit function `g` to `igl::copyleft::marching_cubes` to contour this function into a triangle mesh `V` and `F`.

Make use of `fd_interpolate` and `fd_grad`.

> **Hint**
>
> Eigen has many different sparse matrix solvers. For these *very regular* matrices, it seems that the conjugate gradient method will outperform direct methods such as Cholesky factorization. Try `Eigen::BiCGSTAB`.

> **Hint**
>
> Debug in debug mode with assertions enabled. For Unix users on the command line use:
>
> ```
> cmake -DCMAKE_BUILD_TYPE=Debug ../
> ```
>
> but then try out your code in *release mode* for much better performance
>
> ```
> cmake -DCMAKE_BUILD_TYPE=Release ../
> ```

Releases

No releases published

Packages

No packages published

Contributors 3


alecjacobson Alec Jacobson


psarahdactyl Sarah Kushner

Languages


Pawn 99.8%
Other 0.2%