ThickerSphereLab

November 15, 2025

1 A sphere with Dleto

This notebook will walk you through the creation of a tensor with an embedded surface. We will use a sphere but you can change the parameters to get a look at the effects.

1.1 Step 1: Installing necessary packages

This program is written in Julia and so first we need to load Julia and IJulia if that is not already part of your system. This my involve upgrading your Python tools and restarting your Jupyter notebook, VS code or your favorite notebook shell. While in principle this is compatible with Julia 1.7, we have successfully tested it with later versions as well.

Run the following cell. A successful run will report something like

```
Julia kernel is active!
Julia version: 1.10.3
```

If an error occurs, uncomment the first 5 lines and rerun the cell. This may require you to restart your notebook. If this does not fix it, consider putting the error message into an LLM to get supporting installation instructions relevant to your setting.

[64]:
```
## Uncomment if you do not have iJulia installed
# using Pkg
# Pkg.add("IJulia")
# This installs Julia's Jupyter kernel without Python dependencies
# println("IJulia installed! Restart VS Code and select Julia kernel.")

# Ensure Julia kernel is properly recognized
# This notebook requires Julia kernel for execution and export
using IJulia
println("Julia kernel is active!")
println("Julia version: ", VERSION) # Fix Jupyter/Julia setup - Install IJulia␣
 ↪for Julia notebooks
```

```
Julia kernel is active!
Julia version: 1.10.3
```

1.2 Step 2: Installing Dleto

To load `OpenDelto` run the next cell. This assumes that you have cloned the full Git Repository and that you have launched this notebook from the subfolder `./examples/`

[65]:
```
include("../Dleto.jl")
```

```
plotTensor (generic function with 2 methods)
```

1.3 Step 3: Plotting a sphere in a tensor

Now lets approximate the surface of a sphere as an array and add some randomization surrounding it.

The command `randomSurfaceTensor(u,v,w,t)` builds a tensor with nonzero values clustered near `(i,j,k)` whenever $u[i] + v[j] + w[k] \approx 0$.

Let us try this with a sphere with equation

$$r^2 = (x-a)^2 + (y-b)^2 + (z-c)^2$$

To make this discrete we select values `u[i]`, `v[j]` and `w[k]` by the following rule.

$$\begin{aligned} u[i] &\approx (i-c)^2 - r^2/3 \\ v[j] &\approx (j-c)^2 - r^2/3 \\ w[k] &\approx (k-c)^2 - r^2/3 \end{aligned}$$

Notice that from this choice we get the following.

$$\begin{aligned} 0 &\approx u[i] + v[j] + w[k] \\ &\approx \left((i-a)^2 - \frac{r^2}{3}\right) + \left((j-b)^2 - \frac{r^2}{3}\right) + \left((k-c)^2 - \frac{r^2}{3}\right) \end{aligned}$$

Hence,

$$0 \approx u[i] + v[j] + w[k] \quad \Rightarrow \quad (i-a)^2 + (j-b)^2 + (k-c)^2 \approx r^2.$$

This is what we need to store a discretized approximation of a sphere of radius r and center (a, b, c).

First let us make a sphere with 11 pixels, i.e. indices $i, j, k \in \{0, \dots, 10\}$ centered at $(5, 5, 5)$ radius 5.

Fiddle with the parameters until you see their effect. For example you might not wish to fit the full sphere, you can edit the range form (0:1:10) for one of the axes to make is shorter. It will cut through the sphere.

[66]:
```
a = 5.0; b = 5.0; c = 5.0; r = 5.0;
Ues = [(0:1:10)...] .|> i-> ((i-a)*(i-a)- (r*r)/3.0)
Ves = [(0:1:10)...] .|> j-> ((j-b)*(j-b)- (r*r)/3.0)
Wes = [(0:1:10)...] .|> k-> ((k-c)*(k-c)- (r*r)/3.0)


sphere5 = randomSurfaceTensor( Ues, Ves, Wes, 1.5)
```

```
11×11×11 Array{Float64, 3}:
[:, :, 1] =
 0.0  0.0  0.0  0.0   0.0        0.0      0.0      0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0   0.0        0.0      0.0      0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0   0.0        0.0      0.0      0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0   0.0        0.0      0.0      0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0   0.0        2.16386  0.0      0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  -0.653097  -0.60631  1.17153  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0   0.0       -2.01326  0.0      0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0   0.0        0.0      0.0      0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0   0.0        0.0      0.0      0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0   0.0        0.0      0.0      0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0   0.0        0.0      0.0      0.0  0.0  0.0  0.0

[:, :, 2] =
 0.0  0.0   0.0        0.0        0.0       …  0.0        0.0       0.0  0.0
 0.0  0.0   0.0        0.0        0.0          0.0        0.0       0.0  0.0
 0.0  0.0   0.0        0.0       -0.372592     0.0        0.0       0.0  0.0
 0.0  0.0   0.0        0.542003   0.0          0.524457   0.0       0.0  0.0
 0.0  0.0  -1.47712    0.0        0.0          0.0        0.404314  0.0  0.0
 0.0  0.0  -0.959915   0.0        0.0       …  0.0       -0.33735   0.0  0.0
 0.0  0.0  -1.4714     0.0        0.0          0.0       -0.586524  0.0  0.0
 0.0  0.0   0.0       -0.15164    0.0          0.102015   0.0       0.0  0.0
 0.0  0.0   0.0        0.0       -1.03013      0.0        0.0       0.0  0.0
 0.0  0.0   0.0        0.0        0.0          0.0        0.0       0.0  0.0
 0.0  0.0   0.0        0.0        0.0       …  0.0        0.0       0.0  0.0

[:, :, 3] =
 0.0   0.0       0.0  0.0   0.0       …   0.0       0.0  0.0  0.0       0.0
 0.0   0.0       0.0  0.0  -0.763474      -0.845893  0.0  0.0  0.0       0.0
 0.0   0.0       0.0  0.0   0.0            0.0       0.0  0.0  0.0       0.0
 0.0   0.0       0.0  0.0   0.0            0.0       0.0  0.0  0.0       0.0
 0.0   1.21816   0.0  0.0   0.0            0.0       0.0  0.0  0.460773  0.0
 0.0  -0.666956  0.0  0.0   0.0       …   0.0       0.0  0.0  0.961866  0.0
 0.0   1.48292   0.0  0.0   0.0            0.0       0.0  0.0  0.371347  0.0
 0.0   0.0       0.0  0.0   0.0            0.0       0.0  0.0  0.0       0.0
 0.0   0.0       0.0  0.0   0.0            0.0       0.0  0.0  0.0       0.0
 0.0   0.0       0.0  0.0   0.408632      -0.978451  0.0  0.0  0.0       0.0
 0.0   0.0       0.0  0.0   0.0       …   0.0       0.0  0.0  0.0       0.0

;;; …

[:, :, 9] =
 0.0   0.0       0.0  0.0   0.0       …   0.0       0.0  0.0   0.0       0.0
 0.0   0.0       0.0  0.0   1.92213       0.325594  0.0  0.0   0.0       0.0
 0.0   0.0       0.0  0.0   0.0           0.0       0.0  0.0   0.0       0.0
 0.0   0.0       0.0  0.0   0.0           0.0       0.0  0.0   0.0       0.0
 0.0   0.638806  0.0  0.0   0.0           0.0       0.0  0.0  -0.849546  0.0
```

```
 0.0  -1.05361   0.0  0.0   0.0       …   0.0       0.0  0.0   0.953026  0.0
 0.0  -0.371075  0.0  0.0   0.0           0.0       0.0  0.0   0.72648   0.0
 0.0   0.0       0.0  0.0   0.0           0.0       0.0  0.0   0.0       0.0
 0.0   0.0       0.0  0.0   0.0           0.0       0.0  0.0   0.0       0.0
 0.0   0.0       0.0  0.0  -0.997958     -0.581801  0.0  0.0   0.0       0.0
 0.0   0.0       0.0  0.0   0.0       …   0.0       0.0  0.0   0.0       0.0

[:, :, 10] =
 0.0  0.0   0.0        0.0        0.0       …   0.0       0.0       0.0  0.0
 0.0  0.0   0.0        0.0        0.0           0.0       0.0       0.0  0.0
 0.0  0.0   0.0        0.0        1.05475       0.0       0.0       0.0  0.0
 0.0  0.0   0.0       -0.924488   0.0          -1.15473   0.0       0.0  0.0
 0.0  0.0  -0.425933   0.0        0.0           0.0       0.758502  0.0  0.0
 0.0  0.0   0.823997   0.0        0.0       …   0.0      -0.851371  0.0  0.0
 0.0  0.0  -1.16555    0.0        0.0           0.0       0.591758  0.0  0.0
 0.0  0.0   0.0       -0.51261    0.0           1.66252   0.0       0.0  0.0
 0.0  0.0   0.0        0.0       -0.499808      0.0       0.0       0.0  0.0
 0.0  0.0   0.0        0.0        0.0           0.0       0.0       0.0  0.0
 0.0  0.0   0.0        0.0        0.0       …   0.0       0.0       0.0  0.0

[:, :, 11] =
 0.0  0.0  0.0  0.0  0.0         0.0       0.0       0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0         0.0       0.0       0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0         0.0       0.0       0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0         0.0       0.0       0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0         0.208567  0.0       0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0657088  -0.267042  0.499611  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0        -0.307842  0.0       0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0         0.0       0.0       0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0         0.0       0.0       0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0         0.0       0.0       0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0         0.0       0.0       0.0  0.0  0.0  0.0
```

1.4 Step 4: Visualization

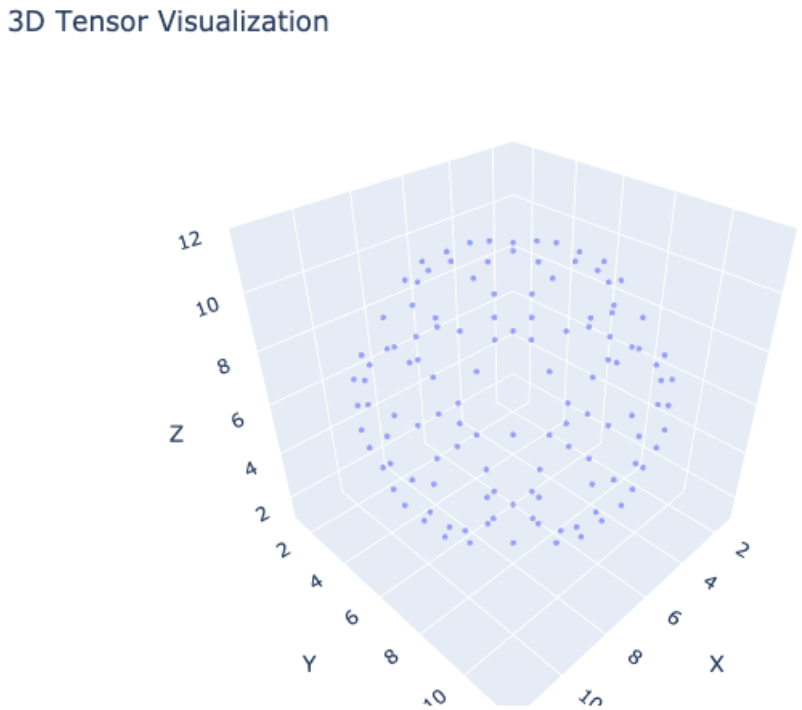
It will help to see the tensor as a graphic we can visualize as 3D. There are many visualization tools in Jupyter, but to pick just one we can use Plotly. You may need to install PlotlyJS if you do not yet have it installed. If so uncomment the top two lines.

`plotTensor` returns a 3D scatter plot that you can manipulate with a mouse, rotate, zoom, and export as `png` format. **Caution** With large dense tensors Plotly may take a long time to load or may fail to rotate or even display content. For sparse tensors you may need to work with ranges around 100 x 100 x 100, and for dense tensors reduce that to 50 x 50 x 50, though specific ranges depend on your hardware and installations.

`plotTensor` also supports an optional second parameter with a threshold to drop points of a tensor that are too small to be seen. This can speed up displays but may suppress some points.

[67]:
```
# Pkg.add( "PlotlyJS" )
# using PlotlyJS

plotTensor(sphere5)
```



1.5 Step 5: Scaling up.

Now lets jump to something larger. We can repeat the method above but this time use a radius 50 sphere, centered at (50, 50, 50). It will make a 101 x 101 x 101 tensor. Once more you can play with the parameters and see the effects.

[82]:
```
Ues = [(0:1:50)...] .|> i-> ((i-25.0)*(i-25.0)- (25.0*25.0)/3.0)
Ves = [(0:1:50)...] .|> j-> ((j-25.0)*(j-25.0)- (25.0*25.0)/3.0)
Wes = [(0:1:50)...] .|> k-> ((k-25.0)*(k-25.0)- (25.0*25.0)/3.0)

sphere51 = randomSurfaceTensor( Ues, Ves, Wes, 4); # No need to print.
```

For a tensor this large we may want an option other than printing to the screen or visualizing. We can for example save the tensor to a file. Use the `saveTensorToFile` command. We will be making 3 versions of this tensor by the end of this notebook so we start by labeling this one as `source` and also list its dimensions.

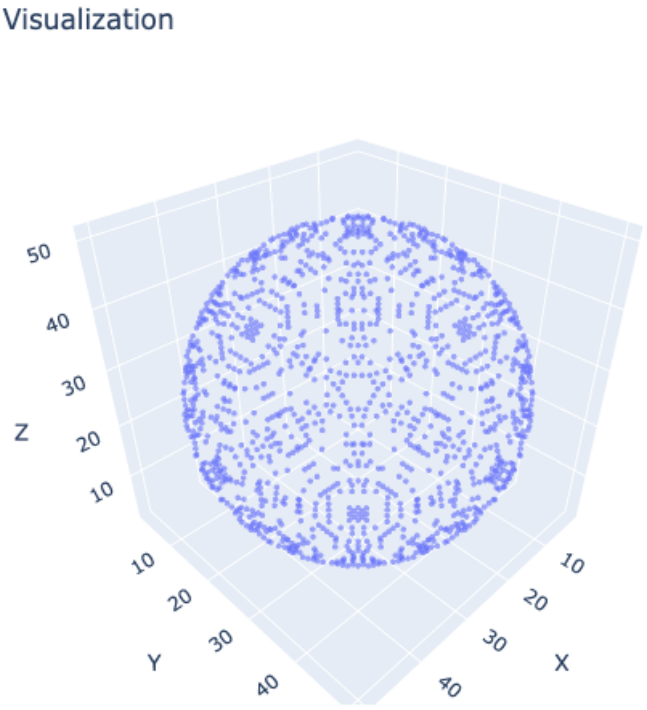
Similar to the visualization `plotTensor` you can provide an optional final number like `0.001` to

drop any coordinates whose absolute value is smaller than `0.001` and save on storage. Of course this creates a coarser approximation to your original tensor so you will need to work with the tolerances you deem appropriate.

For sparse tensors in the range of 100 x 100 x 100, plotting tends to still be robust, but it may load slower and render less smoothly depending on the parameters choosen.

[90]:
```
saveTensorToFile(sphere51, "thick-sphere-51x51x51-source.txt"); # Save a copy.

plotTensor(sphere51, 0.001) # try taking a picture with the camera icon
```



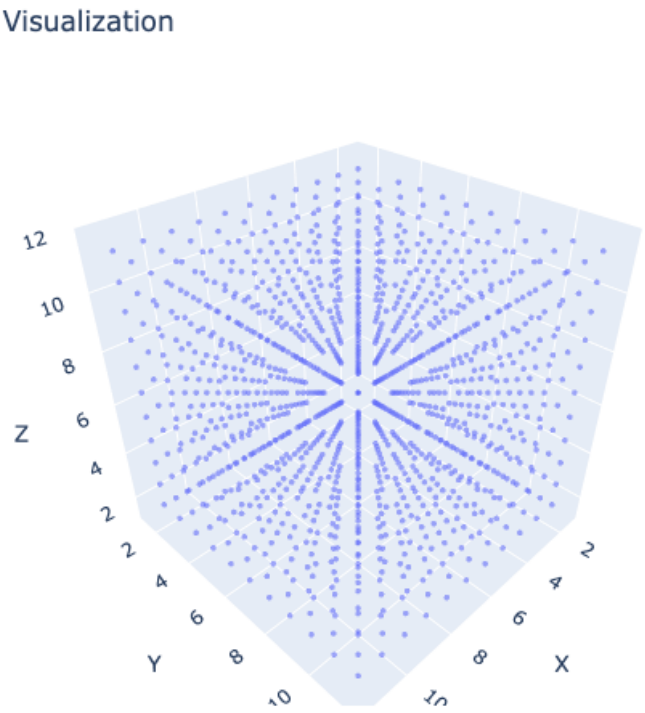
1.6 Step 6: Randomizing the source basis

Dleto works by detecting a hidden constraint equation in a given tensor. So far the tensors we created are obviously equations of a sphere. So what we do next is randomize the bases of the x, y, and z axes. In the case of our original sphere with radius 5 the entire tensor fit inside $11 \times 11 \times 11$-array. This means that are operating in 3 different 11-dimensional vector spaces. So we apply a random 11×11 invertible matrix X to the slices `tensor[i,:,:]`, an invertible matrix Y to each slice `tensor[:,j,:]` and finally an invertible matrix Z to the each `tensor[:,:,k]`. If you changed the dimensions then the relevant matrix sizes will change accordingly. The command `randomTensor` uses the arrays internally stored dimensions to calculate the appropriate changes. The return includes not only the resulting tensor but also the matrices X, Y, Z used to affect the change. To access the resulting tensor use the record name `.tensor`. To access the coordinate

changes use `.Xes`, `.Yes` and `.Zes`.

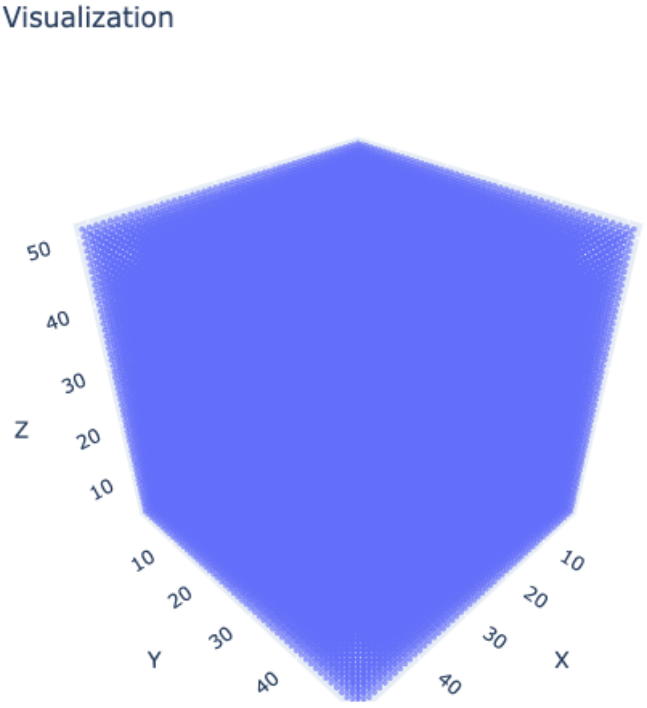
Here is our small sphere randomized, it likely looks completely filled in like dense tensor.

[84]:
```
hidden_sphere5 = randomizeTensor(sphere5)
plotTensor(hidden_sphere5.tensor)
```



And here is our larger sphere randomized. Since visualization can be slow if not impossible for large dense matrices, this might be a step more suitable to save as a file and do statistical comparisons. For the default dimensions uses here it may take up to a minute to render and the result may appear like a solid cube.

[91]:
```
hidden_sphere51 = randomizeTensor(sphere51)
saveTensorToFile(hidden_sphere51.tensor, "thick-sphere-51x51x51-rand.txt"); #␣
 ↪Save instead.
plotTensor(hidden_sphere51.tensor, 0.0001) # If too dense then visualization␣
 ↪may fail.
```



Another way to inspect the results is to print some random slice of the original and compare it to to the same slice in the randomized tensor. Since the original is a sphere, a random slice is mostly 0's. When we plot the randomized tensor it will appear dense.

[92]:
```
sphere51[:,:,10]
```

```
51×51 Matrix{Float64}:
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  …  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  …  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0

 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  …  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
```

```
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  …  0.0  0.0  0.0  0.0  0.0  0.0  0.0
```

[93]:
```
hidden_sphere51.tensor[:,:,10]
```

```
51×51 Matrix{Float64}:
 -0.151834    0.0104677  -0.116059    …  -0.128804    -0.0803133   0.0584485
  0.15697    -0.106945    0.119821       0.0444357   -0.0821587   0.170323
 -0.0530711   0.0978948  -0.0229972      0.213655    -0.115025   -0.125001
 -0.0655883  -0.0552283  -0.0469252     -0.0490826    0.0529651   0.118339
 -0.165265   -0.0628175   0.0587496      0.107752     0.0246479   0.0605908
 -0.0643546   0.0559815  -0.0891111    …  -0.0290695    0.0290753  -0.0526336
 -0.0441771  -0.0446952   0.0210896     -0.195081    -0.117437    0.0181872
  0.0451112   0.0740932   0.125207      -0.122612    -0.0837833  -0.0543817
  0.116936   -0.0188135  -0.00508499    -0.020298     0.118799    0.0491921
  0.0511337  -0.0552849  -0.0611129      0.0625001    0.108241    0.0108006

  0.118882    0.0617917   0.0997053     -0.0680663   -0.0717714   0.0239496
  0.0565968   0.0759301   0.190935       0.00676218  -0.0406056  -0.113074
  0.247503   -0.054862    0.0681257     -0.0731472   -0.0908827  -0.0841633
 -0.096877   -0.045929   -0.00991971    …  0.170761    -0.0935063  -0.0611398
 -0.206822    0.0172311  -0.166531       0.0684995    0.173961   -0.0221481
  0.18687     0.0989623   0.125941       0.00542189   0.0173467  -0.0190548
 -0.0445017  -0.0263617  -0.088978      -0.0858664    0.0680845   0.0770267
  0.0784367   0.140805   -0.0484114     -0.140896     0.176677    0.138711
  0.152761   -0.0459955   0.086765    …  -0.00981338   0.0515494   0.0653086
```

1.7 Step 7: Stratification

Now we finally get to applying the Dleto(chisel) algorithms. We will be using the universal derivation chisel of valence 3 which is select by command `toSurfaceTensor`. You just give it a tensor and you get back a tensor along with a new change of basis.

Let us begin with the radius 5 sphere. You will notice that we did not reconstruct a sphere. Instead you will see thick boxes approximating one octant of a sphere. This is because a sphere is not the type of surface that is stable under Dleto chiselling. In particular Dleto does not distinguish between antipodal points along each axis. So the effect is that the sphere has been folded in half along the x-axis to make a hemisphere twice as thick, then along the y-axis to make a quarter sphere 4 times as thick, and then along the z-axis to make an octant 8 times as thick. For a discreted sphere there may be fewer than 8 points in alignment which means the blocks may be thinner.

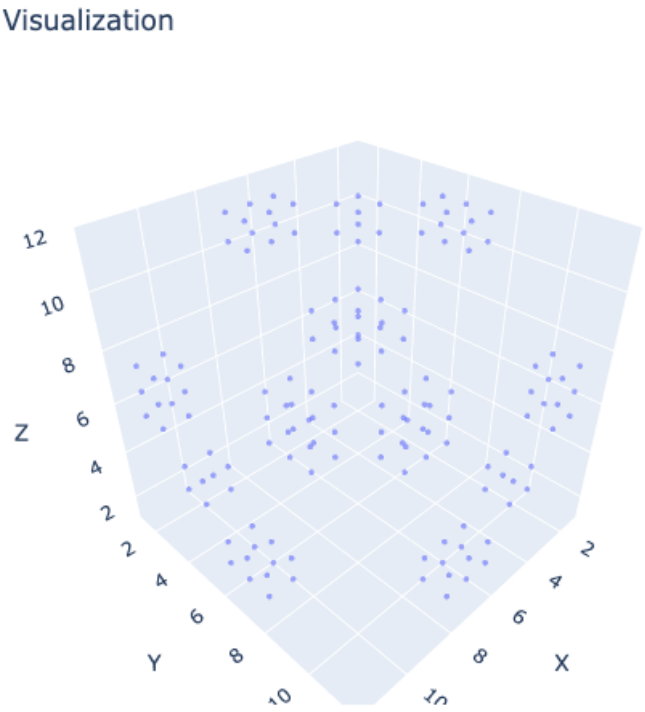
A further artifact of is that the sphere was discrete, and with small radius 5 those discrete effects fold up into more noticeable block shapes.

Finally, the algorithm uses a small amount of randomized methods to improve performance. Try re-running the cell a few times and watch what happens. If you notice a few blocks being reordered, this is an effect cause by the randomized selection of solution. The random choices make subtle rounding differences which can in close cases permute adjecent blocks. For a highly discrete surface

like the radius 5 sphere these permutation may be more noticeable but they still cluster along a discrete 8-think octant of a sphere. With higher resolutions these permutations become less perceptible.

[94]:
```
recovered_sphere5=toSurfaceTensor(hidden_sphere5.tensor)
plotTensor(recovered_sphere5.tensor, 0.000001)
```

```
eigens[1] = [-2.25559984033171e-14, -2.6161764819064343e-15,
3.826114515338398e-15, 5.634175604011219e-15, 0.3636844555464981,
1.026200969664649, 1.4878521707627756, 1.6314630414776559, 1.8116246575523902,
2.3128365313651016, 2.517422873566984, 2.6615725292917642, 3.1451699805361213,
3.479461751827864, 3.5769579279849286, 3.7718406652880647, 3.9644751277484915,
4.100753529958154, 4.120660084482994, 4.405922327069531]
```

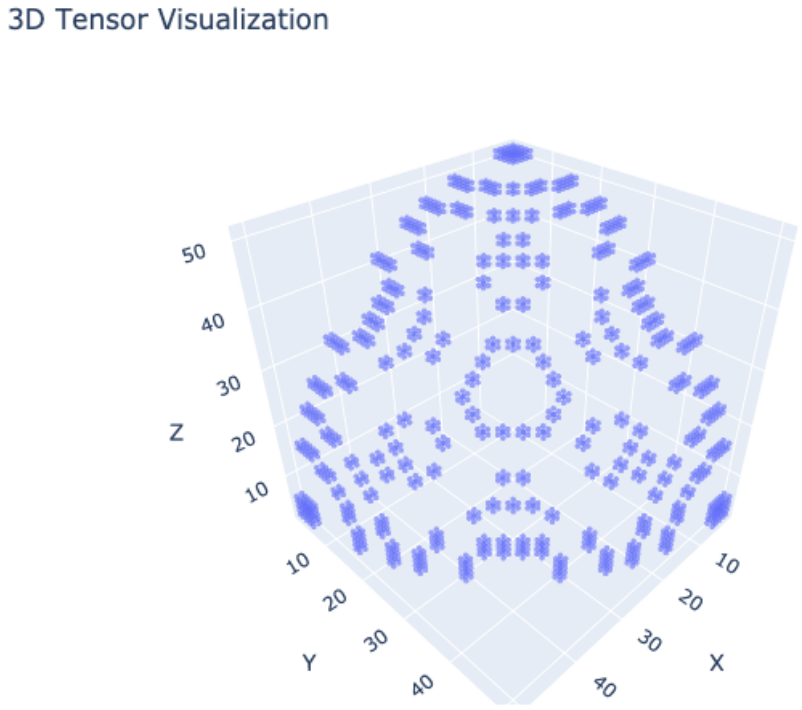


Now we shift to he larger sphere. This make take a minute to compute and longer if you have changed parameter sets. The result is typically far more sparse than the input tensor and thus if you can visualize the input tensor then you should have no problem plotting the result.

If we begin with a sphere of radius 51 the result will again be an octant folding of that sphere, but the curvature should be more apparent than our smaller radius 5 sphere. Once more there are a number of effects to consider, from altering parameters, to choosing a tolerance for how many digits of precision to include. Those concepts are best analyzed using statistics and metrics on the actual tensors and not the visualization.

[95]:
```
recovered_sphere51=toSurfaceTensor(hidden_sphere51.tensor)
saveTensorToFile(recovered_sphere51.tensor, "thick-sphere-51x51x51-recov.txt");␣
 ↪# Save .
plotTensor(recovered_sphere51.tensor, 0.000001) ## try to plot
```

```
eigens[1] = [-4.938352879601905e-15, 1.4092880518136187e-14,
4.425943781751478e-14, 1.4394332654719835, 1.7225774956898279,
1.871588937843783, 4.221778214794144, 4.836001868028761, 5.096662573505474,
5.565435743561636, 6.050453621628038, 6.5595543273413695, 6.824791074018187,
7.196699001286864, 7.420746819655228, 7.808052890183717, 8.490880027145776,
8.707419224926577, 8.731300510813398, 9.184908655405582]
```



1.8 Step 8: Analyzing the output

For the case of a sphere it is clear that the the Dleto chiseling has altered the input source. This is because Dleto is not designed to recover arbitrary surfaces but rather the resulting "convex-contour" surface. As such, a direct comparison of the source to the recovered output maybe be quit large, but falls outside of the design of the algorithm. A more intentional comparison is to first stratify the original tensor before randomization, produces statistics on that surface, than the compare those statistics to the one we get from the recovered surface. Here are some steps to achieve this.

First use `toSurfaceTensor` on the source tensors. Then use `testSurfaceTensor` along with all the

output of the recovery. This simply applies the change of basis back to reconstruct a new tensor that can be compared to the original and a difference is computed as single floating point number. A number close to 0 indicates that the recovered form is a very close approximation to the original.

[96]:
```
# Chisel the source without randomization
chiseled_sphere5=toSurfaceTensor(sphere5)

# Calculate a metric of how close this approximates the surface.
s_orig = testSurfaceTensor(chiseled_sphere5.tensor, chiseled_sphere5.Xes,␣
 ↪chiseled_sphere5.Yes, chiseled_sphere5.Zes)
# Compare to recovered from randomization.
s_recovered = testSurfaceTensor(recovered_sphere5.tensor, recovered_sphere5.
 ↪Xes, recovered_sphere5.Yes, recovered_sphere5.Zes)

println("Surface metric for original sphere5: ", s_orig)
println("Surface metric for recovered sphere5: ", s_recovered)
```

```
eigens[1] = [-9.211400490765218e-15, -5.0597298615084345e-15,
-3.058266548521835e-15, 2.4932908695347087e-15, 0.20485684781993155,
0.6743298973548767, 0.8487577674133221, 1.115237450073399, 1.4769726304947193,
1.591993447153828, 1.8351358174449306, 2.2589931427163656, 2.420741496251763,
2.787558519233028, 3.1663593004967705, 3.1833953906312824, 3.488394497398522,
3.8098098831260816, 3.868821259409805, 4.374520570237638]
Surface metric for original sphere5: 2.5462275892643833e-30
Surface metric for recovered sphere5: 3.143681685258852e-30
```

[97]:
```
# Chisel the source without randomization
chiseled_sphere51=toSurfaceTensor(sphere51)

# Calculate a metric of how close this approximates the surface.
s_orig = testSurfaceTensor(chiseled_sphere51.tensor, chiseled_sphere51.Xes,␣
 ↪chiseled_sphere51.Yes, chiseled_sphere51.Zes)

# Compare to recovered from randomization.
s_recovered = testSurfaceTensor(recovered_sphere51.tensor, recovered_sphere51.
 ↪Xes, recovered_sphere51.Yes, recovered_sphere51.Zes)

println("Surface metric for original sphere51: ", s_orig)
println("Surface metric for recovered sphere51: ", s_recovered)
```

```
eigens[1] = [6.7759039785392075e-15, 1.1716298178799941e-14,
1.1777878163599922e-14, 0.775219710614402, 0.9232374955138101,
0.9746394496278862, 2.312647774222762, 2.641320143097221, 3.0864599208491375,
3.1498427775603806, 3.549598699623959, 3.722260365893612, 4.006169388728461,
4.215090365576912, 4.3889021968146995, 4.83016077074168, 5.037880439377791,
5.134226202604904, 5.474307383094354, 5.666389593300634]
Surface metric for original sphere51: 2.5865657864798635e-29
Surface metric for recovered sphere51: 8.728267780631695e-28
```

Another metric is to simply look at the file sizes of our stored tensors as these are more direct measure of the complexity and "sparsification" achieved by the algorithm. Note that these are stored as text files of coordinate `i j k value`. They compress roughly equally well in different formats so the comparison here appears to be format independent. We also have the option to drop more or less precision and this too can effect storage.

[99]:
```
# Get file sizes for the three saved tensors
source_size = stat("thick-sphere-51x51x51-source.txt").size
rand_size = stat("thick-sphere-51x51x51-rand.txt").size
recov_size = stat("thick-sphere-51x51x51-recov.txt").size

println("File size for source tensor: ", source_size, " bytes")
println("File size for randomized tensor: ", rand_size, " bytes: ratio ",␣
 ↪round(rand_size / source_size))
println("File size for recovered tensor: ", recov_size, " bytes: ratio ",␣
 ↪round(recov_size / source_size))
```

```
File size for source tensor: 37306 bytes
File size for randomized tensor: 3829895 bytes: ratio 103.0
File size for recovered tensor: 48522 bytes: ratio 1.0
```

[100]:
```
# Count nonzeros in the three sphere51 tensors
source_nnz = count(x -> x != 0, sphere51)
rand_nnz = count(x -> x != 0, hidden_sphere51.tensor)
recov_nnz = count(x -> x != 0, recovered_sphere51.tensor)

println("Number of nonzeros in source tensor: ", source_nnz)
println("Number of nonzeros in randomized tensor: ", rand_nnz)
println("Number of nonzeros in recovered tensor: ", recov_nnz)
println("Sparsification ratio (recovered/source): ", round(recov_nnz /␣
 ↪source_nnz, digits=4))
```

```
Number of nonzeros in source tensor: 1326
Number of nonzeros in randomized tensor: 132651
Number of nonzeros in recovered tensor: 132651
Sparsification ratio (recovered/source): 100.0385
```

[101]:
```
dropSmall(x) = abs(x) < 0.0001 ? 0 : x
source_nnz = count(x -> dropSmall(x) != 0, sphere51)
rand_nnz = count(x -> dropSmall(x) != 0, hidden_sphere51.tensor)
recov_nnz = count(x -> dropSmall(x) != 0, recovered_sphere51.tensor)

println("After dropping small values (<0.0001):")
println("Number of nonzeros in source tensor: ", source_nnz)
println("Number of nonzeros in randomized tensor: ", rand_nnz)
println("Number of nonzeros in recovered tensor: ", recov_nnz)
println("Sparsification ratio (recovered/source): ", round(recov_nnz /␣
 ↪source_nnz, digits=4))
```

```
After dropping small values (<0.0001):
Number of nonzeros in source tensor: 1326
Number of nonzeros in randomized tensor: 132542
Number of nonzeros in recovered tensor: 1734
Sparsification ratio (recovered/source): 1.3077
```

1.8.1 Compare to linear algebra.

Finally you might want to compare the timing to the cost of some linear algebra problems like finding singular values of a matrix. The complexity of our program is on the order of $O(n^6)$ for an $n \times n \times n$-tensor. That's not great but we are finding something that is new so it wont get any slower than we have now.

So here is a random matrix of about the size of the operations we expect and the command compute its singular values. You will see it is exceptionally faster. So there is much room for improvement in our strategy.

[102]:
```julia
# Create a random matrix of size (51^3) x (3*51) and compute singular values
using LinearAlgebra

n = 51
matrix_size = (n^3, 3*n)
random_matrix = randn(matrix_size...)

println("Computing SVD for a ", matrix_size, " matrix...")
@time svd_result = svd(random_matrix)
println("Number of singular values: ", length(svd_result.S))
```

```
Computing SVD for a (132651, 153) matrix…
  1.175064 seconds (64.60 k allocations: 314.909 MiB, 5.24% gc time, 3.58%
compilation time)
Number of singular values: 153
Number of singular values: 153
```