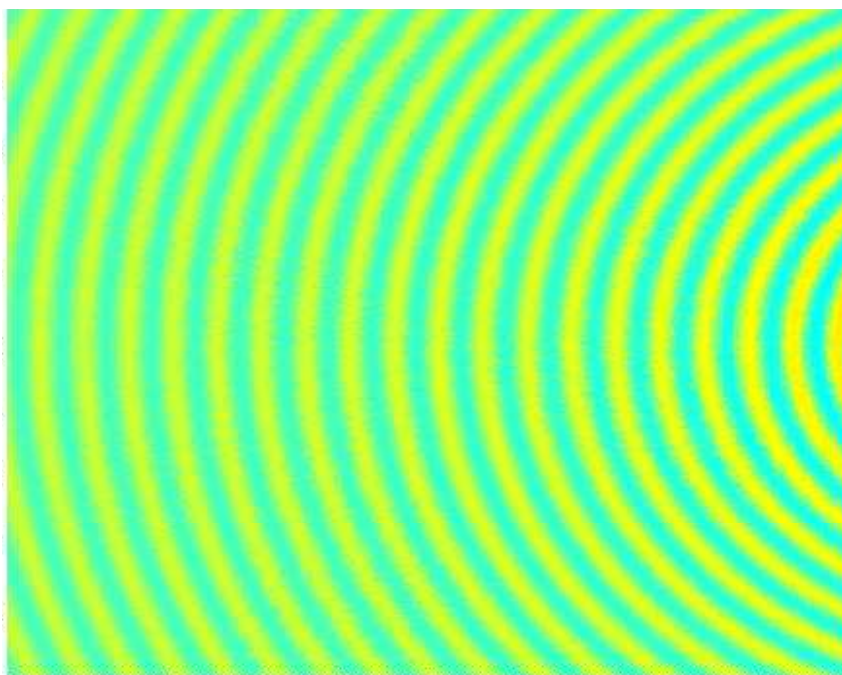
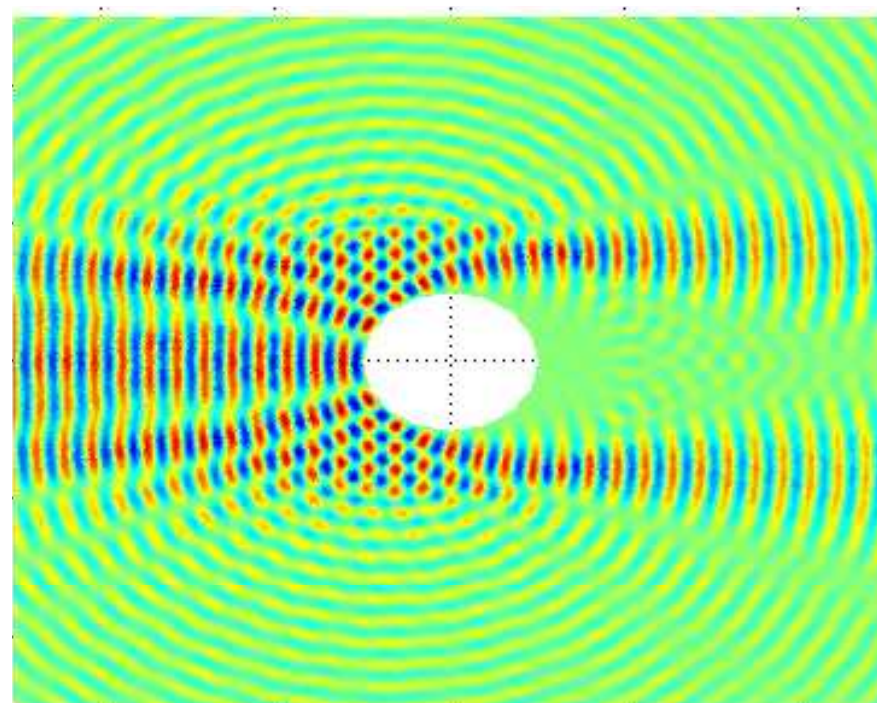
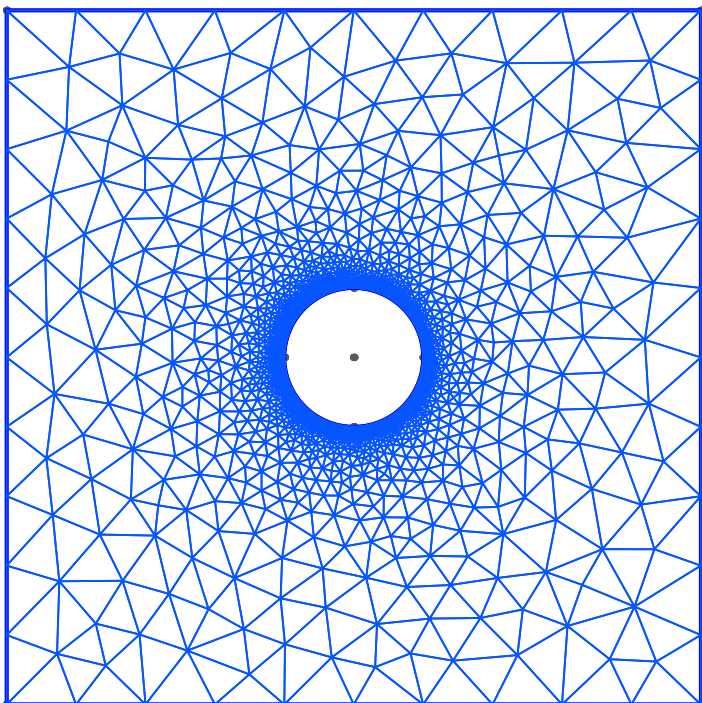


Rapidly Iterating from Prototype to Near-C Performance in Julia: A Finite Element Method Case Study

Reid Atcheson
June 26, 2014



Traditional FEM Workflow

- Mathematical formulation.
- Translate into sparse linear system.
- Program sparse matrix assembly.
- Send to external solver package.

Traditional FEM Workflow

- Excellent tools already exist which specifically target this workflow:
 - Petsc, Deal.ii, DUNE, Trillinos, libMesh, FEniCS...
(many, many more).
- I'm going to talk about a different workflow, and why Julia is uniquely qualified to targeting it.

Fix what isn't broken?

- Sparse matrix assembly can store excessive amounts of redundant information.
- Sparse linear solvers are “black box,” they do not (typically) use domain information to speed up solution.

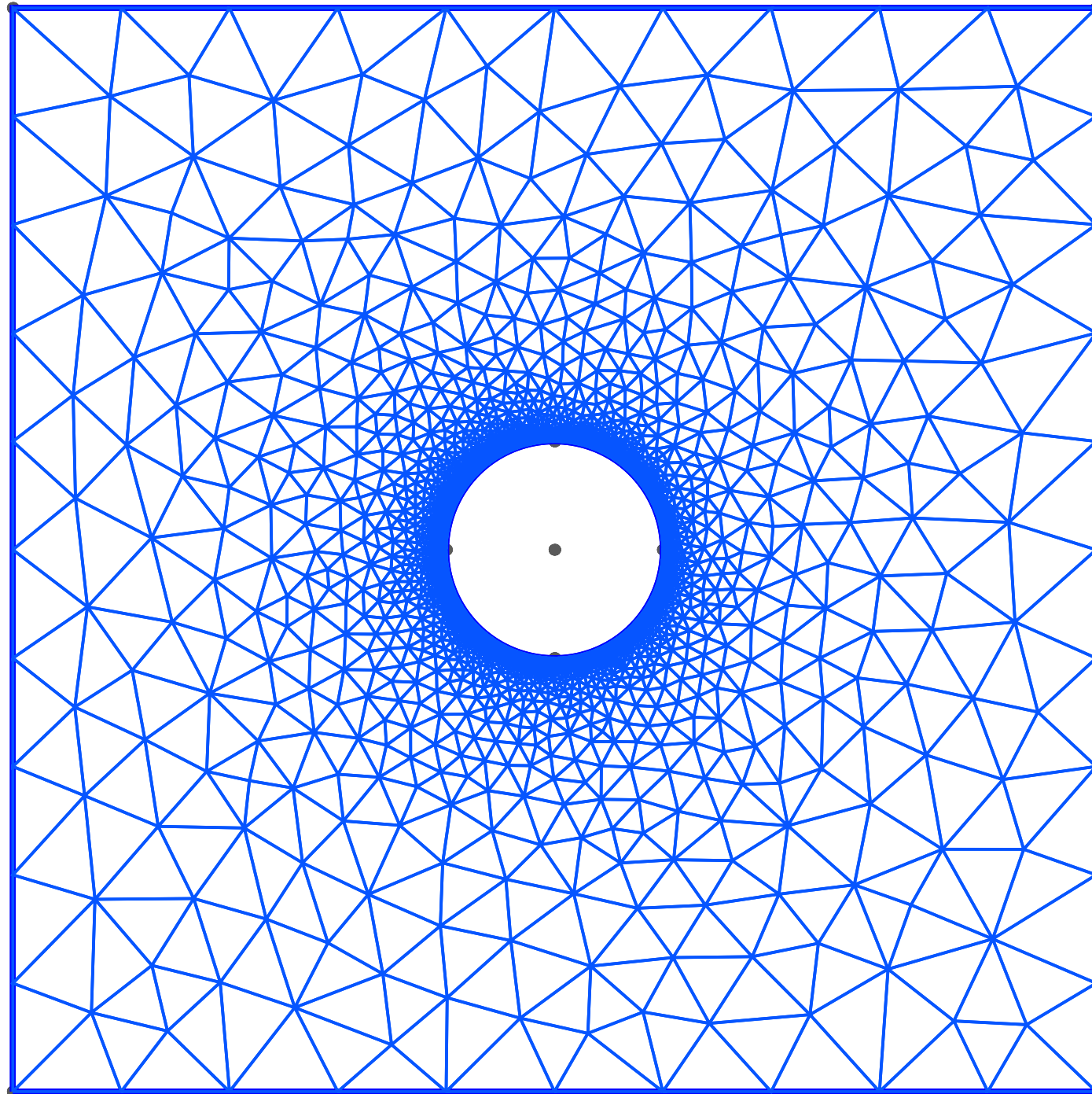
(Potentially) Improved FEM Workflow

- Mathematical formulation.
- Program operator evaluation
 - Sparse matrix-vector without assembling sparse matrix.
- Custom physics based solver.
 - e.g. Multigrid, Additive/Multiplicative Schwarz.

(Potentially) Improved FEM Workflow

- This style is mathematically and algorithmically challenging.
- But by using more domain knowledge, potentially large payoff.
- Need to rapidly generate prototypes to ensure correctness.
- But if a bug happens, need to be able to say: Is my math wrong, or is my code wrong?

“Mathematical Formulation”



Operator Evaluation

```
21 for k = 1 : nElements
22
23
24     #Map reference operators to general triangle.
25     stiffness = mapstiffness(metric,jacobian,refstiff,k);
26
27     #Calculate internal contributions.
28     gradp = -gradient(p,stiffness,k);
29     divu = -divergence(u,stiffness,k);
30
31
32     for f = 1 : nFaces
33         (facemassP,facemassM) = mapface(facejacobian,refface,EToE,EToF,k,f);
34
35         #Compute inter-element jumps.
36         du = jump(u,facemassP,facemassM,EToE,EToF,k,f);
37         dp = jump(p,facemassP,facemassM,EToE,EToF,k,f);
38         ndotdu = ndot(du,normal,k,f);
39         ndp = ntimes(dp,normal,k,f);
40         nndotdu = ntimes(ndotdu,normal,k,f);
41
42         #Update internal contributions with approximate boundary conditions.
43         divu = divu + 0.5*(ndotdu-dp);
44         gradp = gradp + 0.5*(ndp-nndotdu);
45
46     end
47
48     #Fill output.
49     outp[:,k] = divu;
50     outu[:, :,k] = gradp;
51 end
```

Operator Evaluation

- Very slow.
 - Inner loop generates many temporaries.
- Plan of attack: @profile to identify bottlenecks.
 - Devectorize anything amenable to @devec.
 - Replace rest with C.

Eliminating Temporaries

```
23     #Preallocate operator memory.  
24     stiffness = zeros((Np,Np,dim));  
25     facemassM = zeros((Np,Np));  
26     facemassP = zeros((Np,Np));  
27  
28     #Preallocate workspace.  
29     gradp = zeros((Np,dim));  
30     bdryu = zeros((Np,));  
31     bdryp = zeros((Np,dim));  
32     divu = zeros((Np,));  
33     du = zeros((Np,dim));  
34     dp = zeros((Np,));  
35     ndotdu = zeros((Np,));  
36     nndotdu = zeros((Np,dim));  
37     ndp = zeros((Np,dim));
```

```

23
24     #Map reference operators to general triangle.
25     stiffness = mapstiffness(metric,jacobian,refstiff,k);
26
27     #Calculate internal contributions.
28     gradp = -gradient(p,stiffness,k);
29     divu   = -divergence(u,stiffness,k);
30

```

Becomes

```

49     #Map reference operators to general tetrahedron.
50     ccall((:mapstiffness,"C/dgacoustic.so"),Void,
51     (Ptr{Float64},Ptr{Float64},Int64,Int64,Int64,Ptr{Float64})),
52     metric,refstiff,Np,nElements,k,stiffness);
53
54     #Calculate internal contributions.
55     #gradp = gradient(p,stiffness,k);
56     ccall((:gradient,"C/dgacoustic.so"),Void,
57     (Ptr{Float64},Ptr{Float64},Int64,Int64,Ptr{Float64})),
58     p,stiffness,k,Np,gradp);
59
60
61     #divu   = divergence(u,stiffness,k);
62     ccall((:divergence,"C/dgacoustic.so"),Void,
63     (Ptr{Float64},Ptr{Float64},Int64,Int64,Ptr{Float64})),
64     u,stiffness,k,Np,divu);

```

```

32         for i = 1:nElements
33             (facemassP, facemassM) = mapface(facejacobian, refface, EToE, EToF, k, f);
34
35             #Compute inter-element jumps.
36             du = jump(u, facemassP, facemassM, EToE, EToF, k, f);
37             dp = jump(p, facemassP, facemassM, EToE, EToF, k, f);

```

Becomes

```

79     ccall(:mapface, "C/dgacoustic.so"), Void,
80     (Ptr{Float64}, Ptr{Float64}, Ptr{Float64}, Ptr{Int64}, Ptr{Int64}, Int64, Int64, Int64, Int64, Ptr{Float64}, Ptr{Float64}),
81     facejacobian, jacobian, refface, EToE, EToF, nElements, Np, k, f, facemassP, facemassM);
82
83     #Compute inter-element jumps.
84     ccall(:vector_jump, "C/dgacoustic.so"), Void,
85     (Ptr{Float64}, Ptr{Float64}, Ptr{Float64}, Ptr{Int64}, Int64, Int64, Int64, Int64, Ptr{Float64}),
86     u, facemassP, facemassM, EToE, nElements, Np, k, f, du);
87
88
89     ccall(:scalar_jump, "C/dgacoustic.so"), Void,
90     (Ptr{Float64}, Ptr{Float64}, Ptr{Float64}, Ptr{Int64}, Int64, Int64, Int64, Int64, Ptr{Float64}),
91     p, facemassP, facemassM, EToE, nElements, Np, k, f, dp);

```

```

39     ndotdu = ndot(du,normal,k,f);
40     ndp = ntimes(dp,normal,k,f);
41     nndotdu = ntimes(ndotdu,normal,k,f);
42
43     #Update internal contributions with approximate boundary conditions.
44     divu = divu + 0.5*(ndotdu-dp);
45     gradp = gradp + 0.5*(ndp-nndotdu);

```

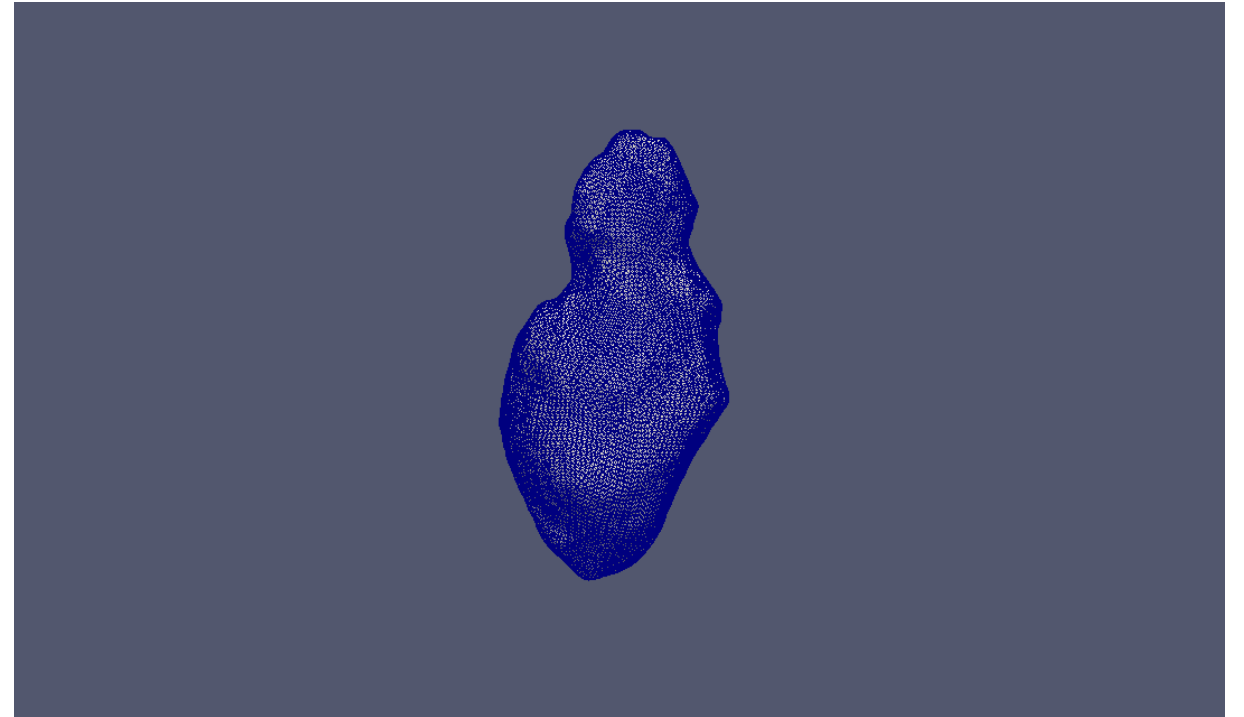
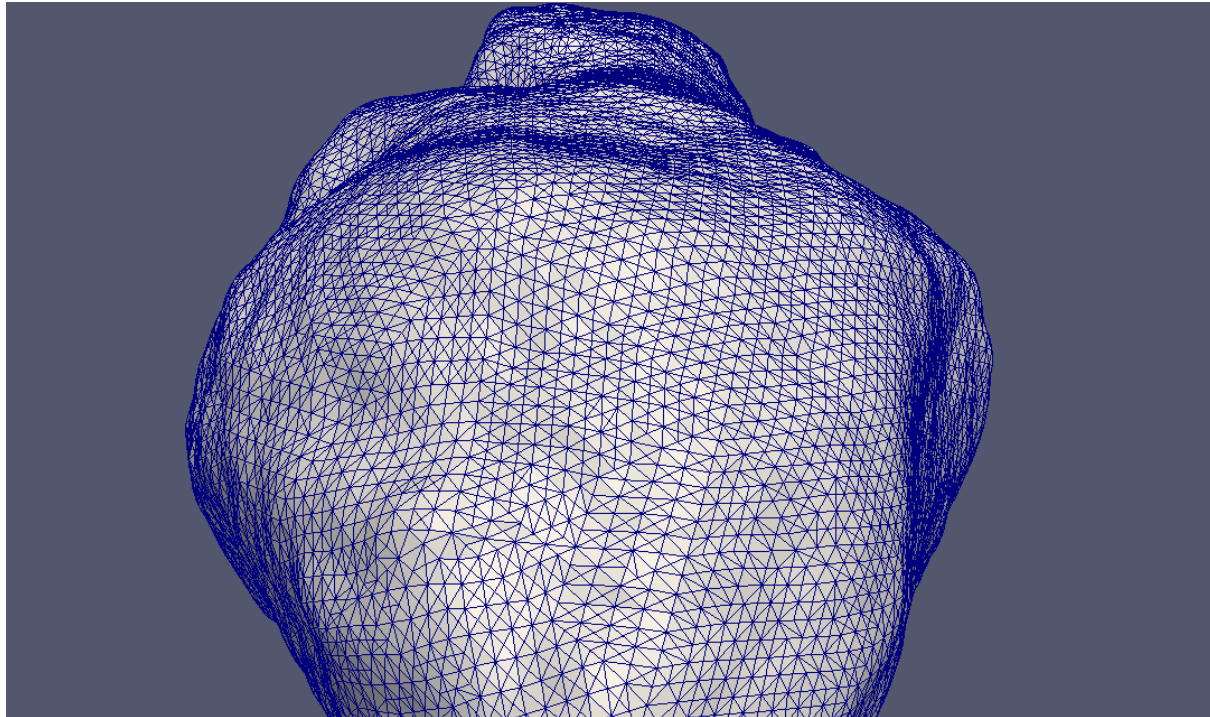
Becomes

```

94     #ndotdu = ndot(du,normal,k,f);
95     for i = 1 : dim
96         nifk = normal[i,f,k];
97         @devec ndotdu = ndotdu + nifk.*du[:,i];
98         @devec ndp[:,i] = nifk.*dp;
99     end
100    for i = 1 : dim
101        nifk = normal[i,f,k];
102        @devec nndotdu[:,i] = nifk.*ndotdu;
103    end
104
105
106    #Update internal contributions with approximate boundary conditions.
107    @devec bdryu = bdryu + 0.5.*(ndotdu-dp);
108    @devec bdryp = bdryp + 0.5.*(ndp-nndotdu);

```

Test Problem: Toutatis Asteroid

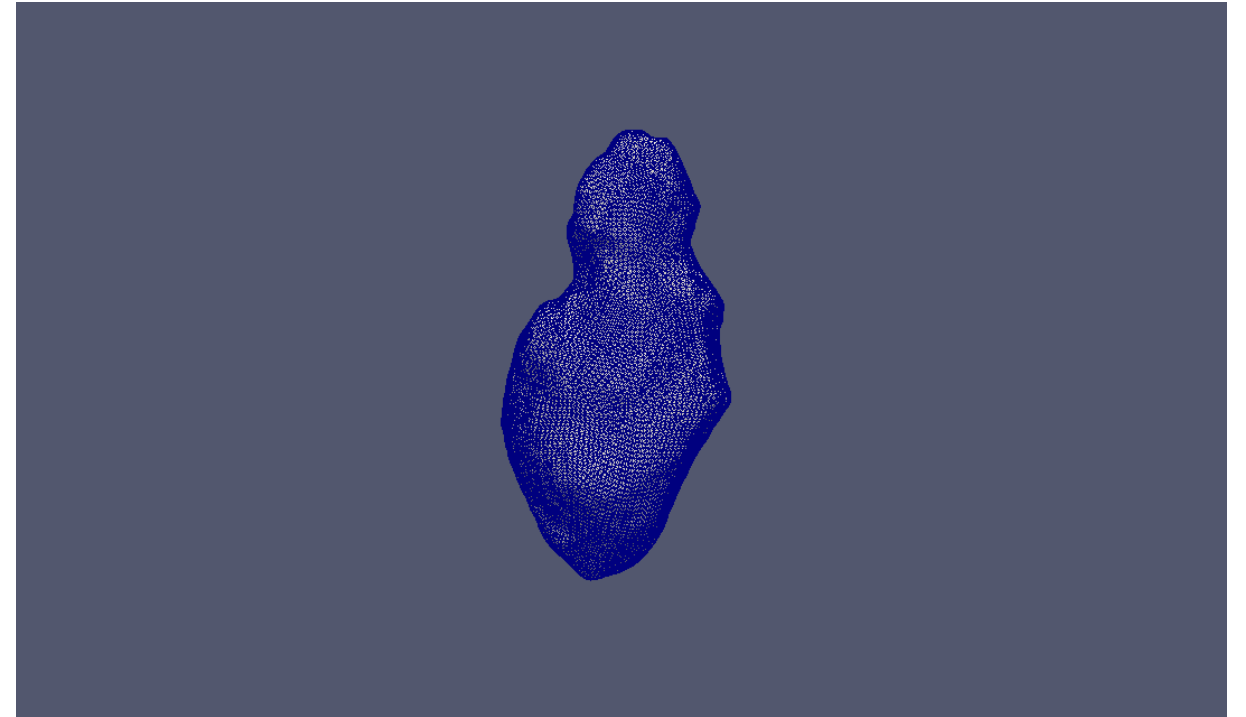
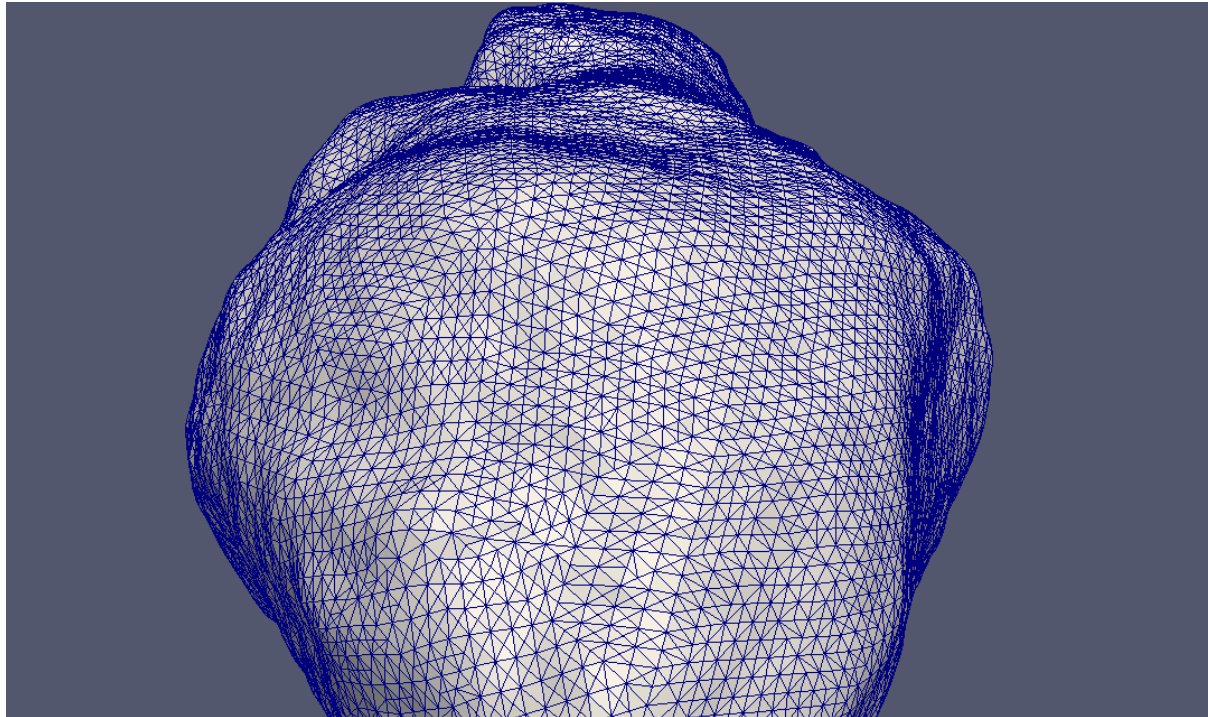


~500,000 Tetrahedra

One Prototype Operator Eval: 223s

One Optimized Operator Eval: 43s

Test Problem: Toutatis Asteroid



~500,000 Tetrahedra

One Prototype Operator Eval: 223s

One Optimized Operator Eval: 43s

Speedup ~ 5x

Conclusions

- Prototype reads very straightforwardly from math.
- Easy to separate math bugs from programming bugs.
- Rapidly produce correct prototype.
- Devectorization and C interface provides straightforward optimization opportunities.
- Start with slow correct code and use it as unit test for optimization iterations.

Acknowledgements

- Jeff Bezanson, Stefan Karpinski, Viral Shah, Alan Edelman et al. **Julia**
- Dahua Lin et al. **Devectorize.jl**
- Toutatis mesh: **gmsh**
 - <http://geuz.org/gmsh/>
- Visualization: **Paraview**
 - <http://www.paraview.org/>
- Toutatis STL definition:
 - <http://www.thingiverse.com/thing:4092>
- VTK format I/O: My code + LightXML.jl
 - <https://github.com/lindahua/LightXML.jl>