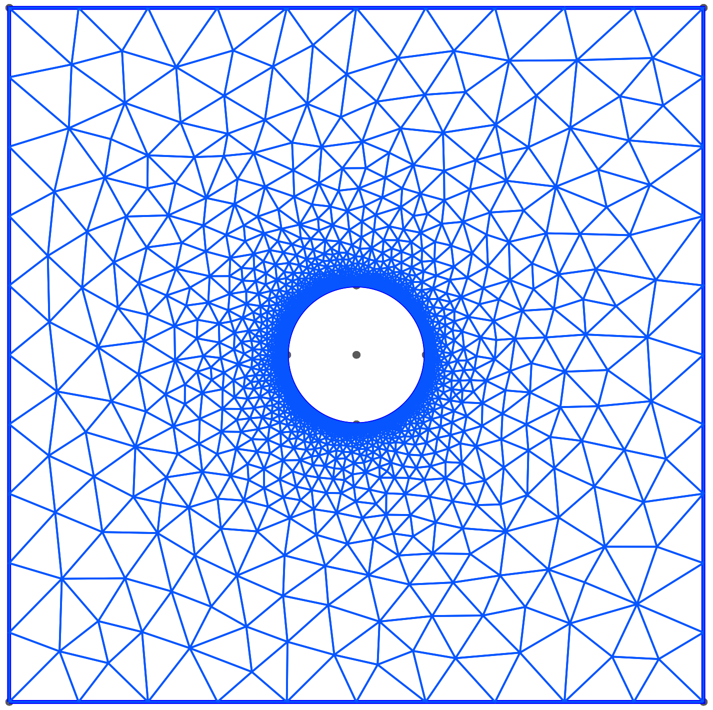
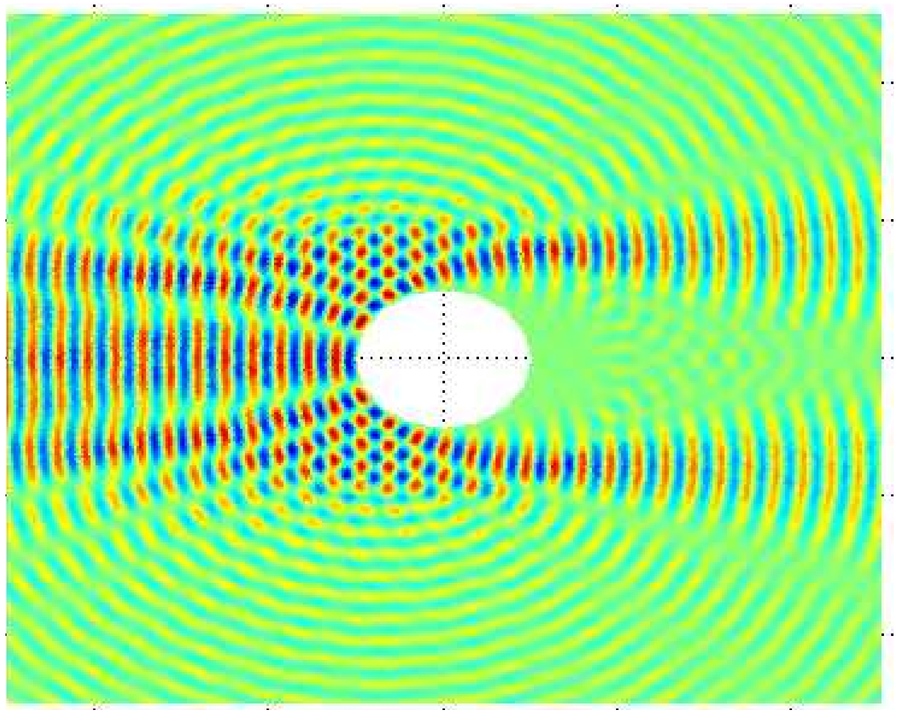
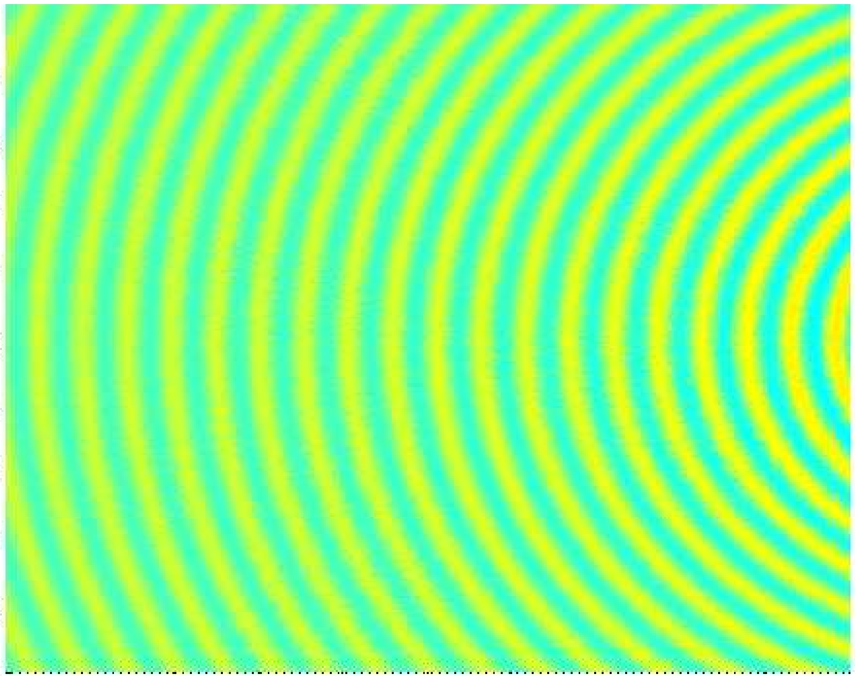
Rapidly Iterating from Prototype to Near-C Performance in Julia: A Finite Element Method Case Study

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.

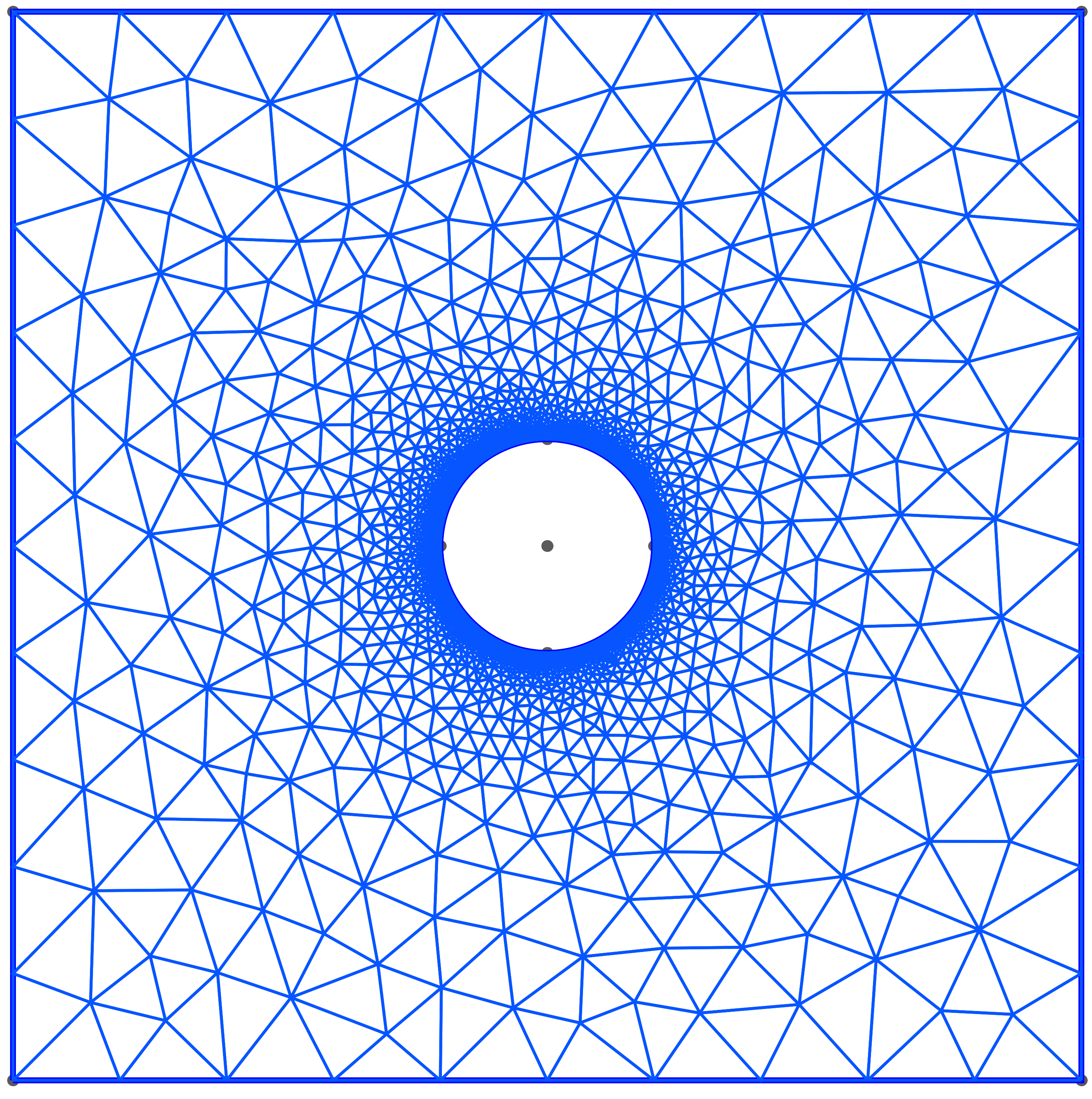
(Potentially) Improved FEM Workflow

- 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

```
    for k = 1 : nElements


        #Map reference operators to general triangle.
        stiffness = mapstiffness(metric,jacobian,refstiff,k);

        #Calculate internal contributions.
        gradp = -gradient(p,stiffness,k);
        divu  = -divergence(u,stiffness,k);


        for f = 1 : nFaces
            (facemassP,facemassM)  = mapface(facejacobian,refface,EToE,EToF,k,f);

            #Compute inter-element jumps.
            du = jump(u,facemassP,facemassM,EToE,EToF,k,f);
            dp = jump(p,facemassP,facemassM,EToE,EToF,k,f);
            ndotdu = ndot(du,normal,k,f);
            ndp =  ntimes(dp,normal,k,f);
            nndotdu = ntimes(ndotdu,normal,k,f);

            #Update internal contributions with approximate boundary conditions.
            divu = divu + 0.5*(ndotdu-dp);
            gradp = gradp + 0.5*(ndp-nndotdu);

        end

        #Fill output.
        outp[:,k] = divu;
        outu[:,:,k] = gradp;
    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

```
    #Preallocate operator memory.
    stiffness = zeros((Np,Np,dim));
    facemassM = zeros((Np,Np));
    facemassP = zeros((Np,Np));

    #Preallocate workspace.
    gradp = zeros((Np,dim));
    bdryu = zeros((Np,));
    bdryp = zeros((Np,dim));
    divu = zeros((Np,));
    du = zeros((Np,dim));
    dp = zeros((Np,));
    ndotdu = zeros((Np,));
    nndotdu = zeros((Np,dim));
    ndp = zeros((Np,dim));
```

```
        #Map reference operators to general triangle.
        stiffness = mapstiffness(metric,jacobian,refstiff,k);

        #Calculate internal contributions.
        gradp = -gradient(p,stiffness,k);
        divu  = -divergence(u,stiffness,k);

```

Becomes

```
        #Map reference operators to general tetrahedron.
        ccall((:mapstiffness,"C/dgacoustic.so"),Void,
        (Ptr{Float64},Ptr{Float64},Int64,Int64,Int64,Ptr{Float64}),
        metric,refstiff,Np,nElements,k,stiffness);

        #Calculate internal contributions.
        #gradp = gradient(p,stiffness,k);
        ccall((:gradient,"C/dgacoustic.so"),Void,
        (Ptr{Float64},Ptr{Float64},Int64,Int64,Ptr{Float64}),
        p,stiffness,k,Np,gradp);


        #divu  = divergence(u,stiffness,k);
        ccall((:divergence,"C/dgacoustic.so"),Void,
        (Ptr{Float64},Ptr{Float64},Int64,Int64,Ptr{Float64}),
        u,stiffness,k,Np,divu);
```

```
        (facemassP,facemassM)  = mapface(facejacobian,refface,EToE,EToF,k,f);

        #Compute inter-element jumps.
        du = jump(u,facemassP,facemassM,EToE,EToF,k,f);
        dp = jump(p,facemassP,facemassM,EToE,EToF,k,f);
```

Becomes

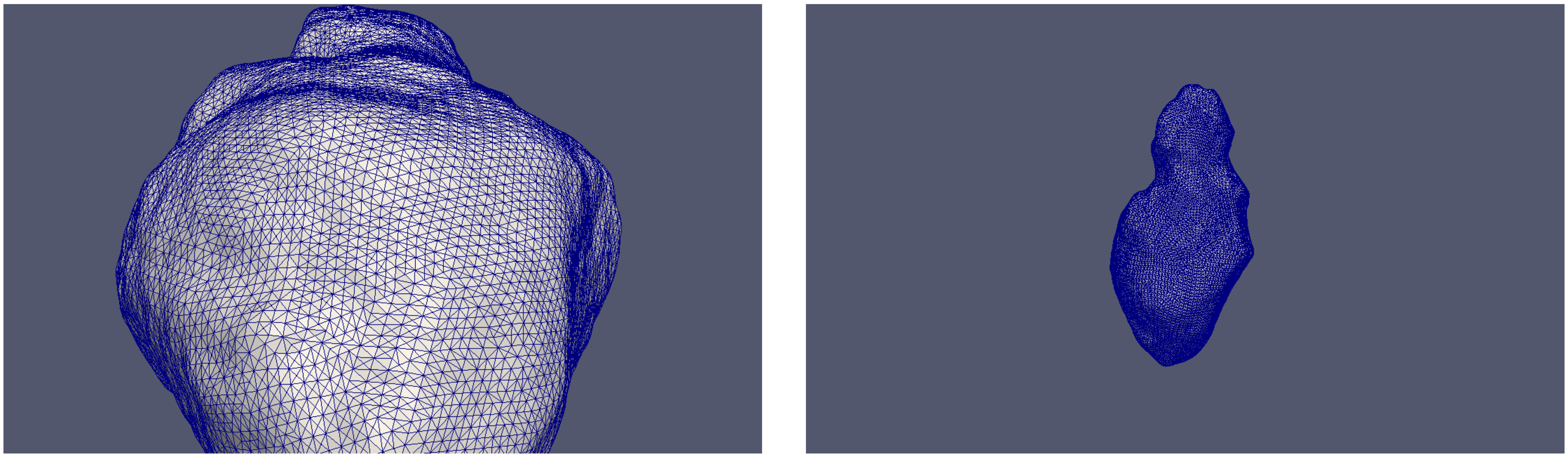
```
        ccall((:mapface,"C/dgacoustic.so"),Void,
        (Ptr{Float64},Ptr{Float64},Ptr{Float64},Ptr{Int64},Ptr{Int64},Int64,Int64,Int64,Int64,Ptr{Float64},Ptr{Float64}),
        facejacobian,jacobian,refface,EToE,EToF,nElements,Np,k,f,facemassP,facemassM);

        #Compute inter-element jumps.
        ccall((:vector_jump,"C/dgacoustic.so"),Void,
        (Ptr{Float64},Ptr{Float64},Ptr{Float64},Ptr{Int64},Int64,Int64,Int64,Int64,Ptr{Float64}),
        u,facemassP,facemassM,EToE,nElements,Np,k,f,du);


        ccall((:scalar_jump,"C/dgacoustic.so"),Void,
        (Ptr{Float64},Ptr{Float64},Ptr{Float64},Ptr{Int64},Int64,Int64,Int64,Int64,Ptr{Float64}),
        p,facemassP,facemassM,EToE,nElements,Np,k,f,dp);
```

```
ndotdu = ndot(du,normal,k,f);
ndp =  ntimes(dp,normal,k,f);
nndotdu = ntimes(ndotdu,normal,k,f);

#Update internal contributions with approximate boundary conditions.
divu = divu + 0.5*(ndotdu-dp);
gradp = gradp + 0.5*(ndp-nndotdu);
```

Becomes

```
 #ndotdu = ndot(du,normal,k,f);
 for i = 1 : dim
     nifk = normal[i,f,k];
     @devec ndotdu = ndotdu + nifk.*du[:,i];
     @devec ndp[:,i] = nifk.*dp;
 end
 for i = 1 : dim
     nifk = normal[i,f,k];
     @devec nndotdu[:,i] = nifk.*ndotdu;
end


 #Update internal contributions with approximate boundary conditions.
 @devec bdryu = bdryu + 0.5.*(ndotdu-dp);
 @devec bdryp = bdryp + 0.5.*(ndp-nndotdu);
```

Test Problem: Toutatis Asteroid

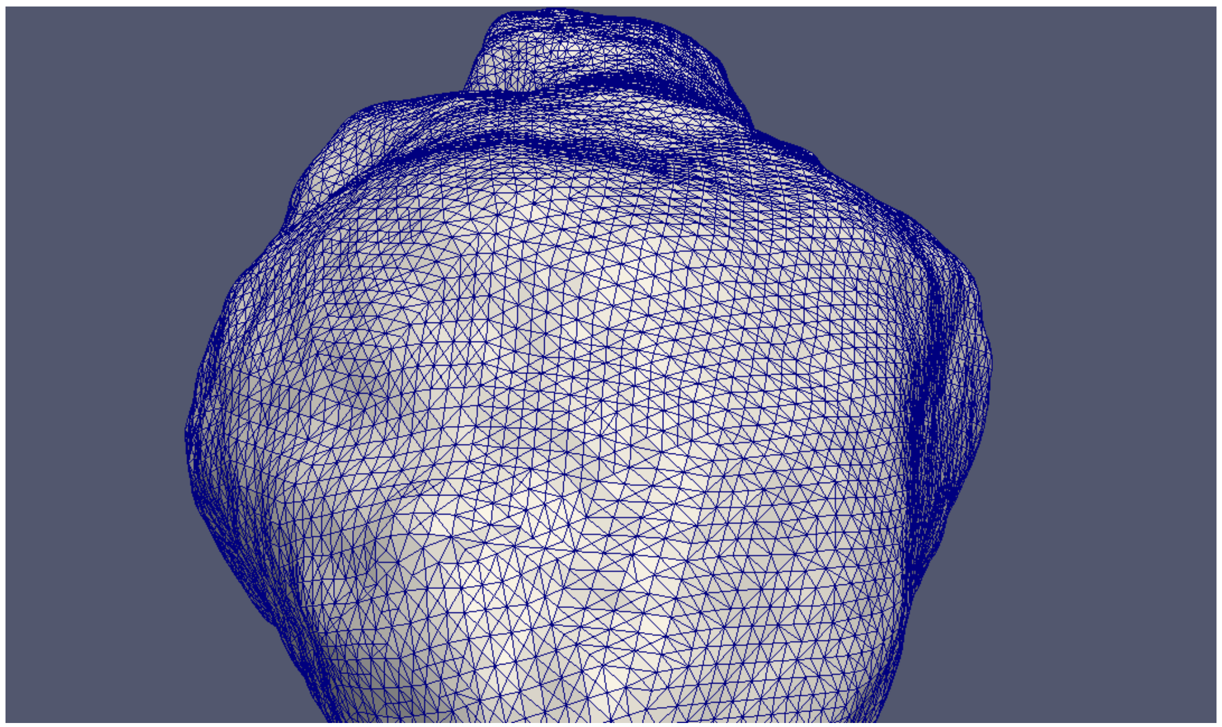


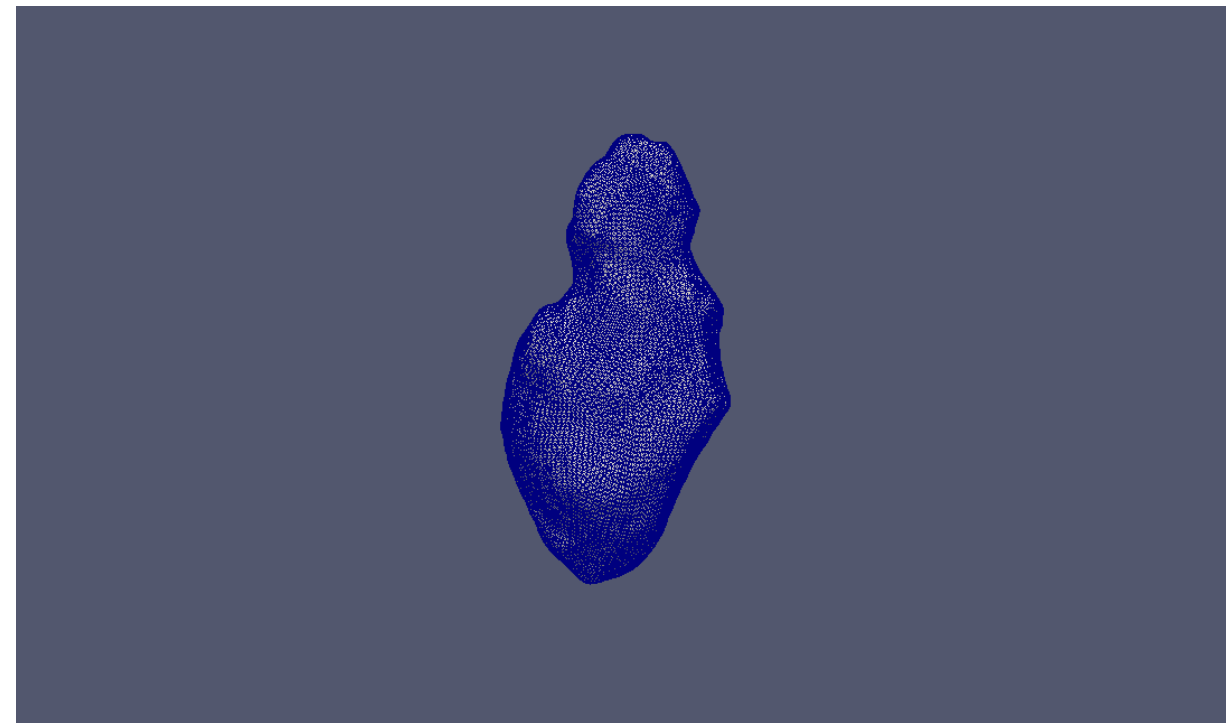
~500,000 Tetrahedra

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