Robustness Prediction for Rapid Manufacturing

HPC Project: Julian Panetta
Research: James Zhou, Denis Zorin

18 December 2012

NEW YORK UNIVERSITY

Rapid Manufacturing

- 3D printers are becoming affordable!

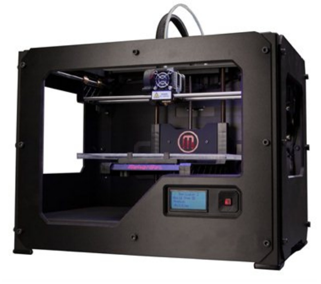
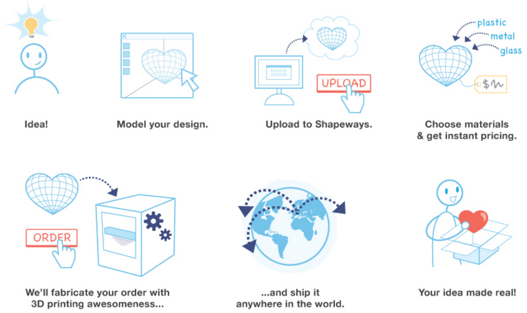


Figure : "MakerBot Replicator 2" $2199

Services

- Several companies do 3D printing/sales for you:
    - Shapeways (NYC-based, James's internship)


Idea!
Model your design.
UPLOAD
Upload to Shapeways.
plastic
metal
glass
Choose materials & get instant pricing.
ORDER
We'll fabricate your order with 3D printing awesomeness...
...and ship it anywhere in the world.
Your idea made real!

    - i.materialise
    - NYU

Image Source: shapeways.com

The Problem

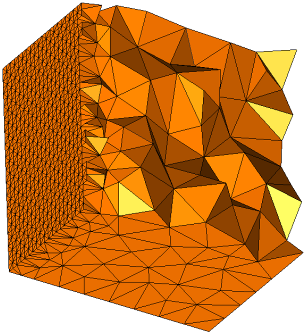
- The model might not be sturdy enough...

- It would be nice to know this beforehand.
- Even better: interactive feedback while designing.

Image Source: http://www.turbosquid.com/3d-models/heart-broken-broke-3d-model/648084

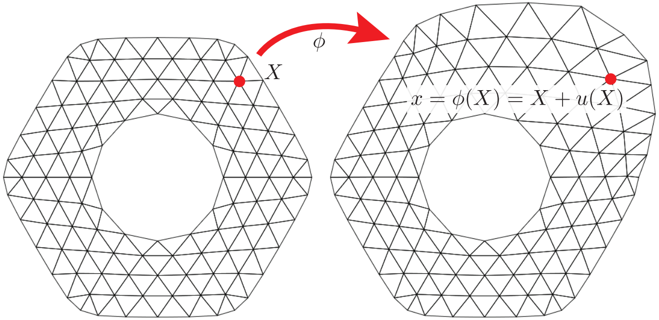
Robustness Prediction

- Predict amount of force required to break the object and where it will break.
- Challenge: where is this force going to be applied?
- Input: 3D Volume Meshes (tetrahedra)



Approach: Linear Elasticity

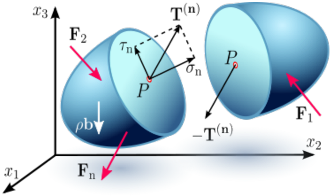
- Internal forces are caused by deformation, ϕ:



- Linear elasticity: simple model where force is a linear function of displacement field, $u(X)$
- Linear FEM Discretization
  - Per-vertex displacement vector $u_i = u(X_i)$
  - Linear interpolation within tets (Piecewise linear deformation)

What Breaks?

- Given a deformation, will the model break?
- We predict a model (element) will break if internal forces exceed a threshold.
- Internal forces: $\mathbf{T}^{(\mathbf{n})} = \sigma \mathbf{n}$ (σ is the 3×3 stress tensor)



  - Intuitively: force after element sliced by plane with normal $\mathbf{n}$
- So: maximum eigenvalue of σ gives internal force in maximum direction.

Image Source: Wikipedia

What Breaks?

- For each tetrahedral element, e, compute σ_e
- Thousands and thousands of 3×3 eigenvalue problems
- James was using scipy, and this was SLOW.
- SIMD operation–great candidate for OpenCL.

Image Source: Wikipedia

3×3 Eigenvalues

- Could solve cubic polynomial–unstable!
- Alternate approach: Jacobi's eigenvalue algorithm.
  - Iteratively conjugate by orthogonal matrices: $A_{k+1} = U_k^T A_k U_k$
  - Choose a nonzero off-diagonal and solve for a "Givens Rotation" matrix that zeroes it:

$$U_k = G(i, j, \theta) \begin{bmatrix} 1 & \cdots & 0 & \cdots & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & & \vdots & & \vdots \\ 0 & \cdots & \cos(\theta) & \cdots & -\sin(\theta) & \cdots & 0 \\ \vdots & & \vdots & \ddots & \vdots & & \vdots \\ 0 & \cdots & \sin(\theta) & \cdots & \cos(\theta) & \cdots & 0 \\ \vdots & & \vdots & & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & \cdots & 0 & \cdots & 1 \end{bmatrix}$$

  - Each iteration moves nonzero "mass" to the diagonal
  - Converges quickly for small matrices.
  - With approximations, can be modified to avoid branching.

Performance

- $\approx$ 120ns per 3×3 using OpenMP
- $\approx$ 40ns per 3×3 matrix using 4-wide SSE! (OpenCL)
- Negligible time for even the largest problem size I could fit on the GPU–might as well keep it on the CPU.
- Interesting tidbit: ffast-math replaces:

  `float w = 1.0f / sqrtf(a12_sqr + a11_m_a22_sqr);`

  With RSQRT + Newton Raphson Correction step:

```
rsqrtss %xmm1,%xmm3
mulss   %xmm3,%xmm1
mulss   %xmm3,%xmm1
mulss   0x00003680(%rip),%xmm3
addss   0x00003674(%rip),%xmm1
mulss   %xmm3,%xmm1
```

Modal Analysis

- We must effectively search over all possible deformations.
- **Modal Analysis**, a common technique in accelerating simulations, is key in making this search tractable.
- Consider how a displacement evolves: (Newton's Law):

$$\mathbf{K}\mathbf{u} = \mathbf{M}\ddot{\mathbf{u}}$$

  K: Stiffness matrix (maps displacement to force)
  M: Mass matrix
- Solution can be decomposed into eigenvectors whose components vary sinusoidally in time.
- Eigenvalue gives corresponding frequency/energy

Modes

- Modes are given by the generalized eigenvalue problem:

$$\mathbf{Kx} = \lambda \mathbf{Mx}$$

  K: Symmetric Positive Semidefinite, Sparse
  M: Symmetric Positive Definite, Sparse

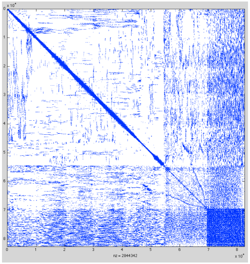
- Can be solved as-is, or transformed into traditional eigenvalue problem using Cholesky factorization $\mathbf{M} = \mathbf{LL^T}, \quad \mathbf{y} = \mathbf{L^T x}$

$$\mathbf{L}^{-1}\mathbf{K}{\mathbf{L}^{-1}}^{\mathbf{T}}\mathbf{y} = \tilde{\mathbf{K}}\mathbf{y} = \lambda \mathbf{y}$$

  Cheap if we use a lumped mass approximation (**M** diagonal)

- What do these look like? Demo.

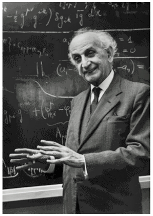
Sparse Eigenvalue Problem



- We want the smallest eigen pair of sparse SPD matrix $\mathbf{A}$
- Only tractable techniques: iterative
- Power Method: find largest eigenvalue/eigenvector
  - Repeatedly apply $\mathbf{A}$ to random $\mathbf{x_0}$, blowing up largest eigenvector's component.
  - $\mathbf{x_k} = \mathbf{A^k x_0}$ approximates eigenvector
- But, we can do much better by considering the entire Krylov space:

$$span(\mathbf{x_0}, \mathbf{A x_0}, \ldots, \mathbf{A}^k \mathbf{x_0})$$

Sparse Eigenvalue Problem: Lanczos



- Lanczos Method: find SET of largest/smallest eigen pairs
  - Build Krylov basis by repeatedly applying $\mathbf{A}$ to random $\mathbf{x_0}$
  - The Krylov subspace is optimal for approximating largest and smallest eigenvectors of $\mathbf{A}$
  - Orthonormal Krylov basis tri-diagonalizes $\mathbf{A}$
    $\Rightarrow$ Finding approximate eigen pairs is cheap tri-diagonal eigenvalue problem
  - Can derive 3-term recurrence relation for orthonomal basis.

Image Source: physics.ncsu.edu/lanczos/

Lanczos Problems

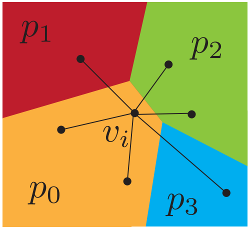
- Main point: dominant cost is the sparse matvec $\mathbf{Ax}$
- Wrinkle: plain Lanczos doesn't converge to my smallest eigenvalues (though it efficiently finds the largest).
- Work-around: "Shift-Invert Lanczos" on $(\mathbf{A} - \sigma\mathbf{I})^{-1}$
    - Converges wonderfully, but...
    - Requires factorization (slow, 4x fill-in)
    - Back substitions instead of plain matvecs–hard to parallelize!
- Other "matvec"-only techniques I tried:
    - Using CG to apply $(\mathbf{A} - \sigma\mathbf{I})^{-1}$ (Doesn't converge)
    - RCG/LOBPCG: Directly minimize Rayleigh quotient $\frac{x^T AX}{x^T x}$ with nonlinear CG (Doesn't converge)

Sparse Matvec

- To have something to show today, I decided to go ahead with optimizing/parallelizing the sparse matvec.
- Challenge: sparse matvecs do only two FLOPs per every four memory accesses!
- GPU Option: not particularly exciting–just ways make matrix element access more contiguous
- Also not particularly fast (CUSP, NVIDIA's sparse matrix library)
    - only gets $\approx$ 1.38 GFLOPS on my GeForce 320m
    - still only $\approx$ 6.5 GFLOPS on a GeForce GTX 590

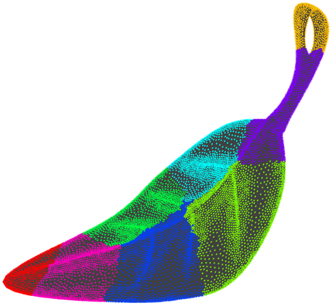
MPI Sparse Matvec

- I decided to implement a parallel sparse matvec in MPI
- Inspired by James Demmel's talk, "Communication-Avoiding Algorithms..."
- Insight:
  - Rows/columns i of K correspond to mesh vertices, v_i
  - Nonzero entries (i, j) in K correspond to mesh edges (v_i, v_j)
  - i.e. result component $(\mathbf{Kx})_i = \mathbf{K}(i,i)\mathbf{x}_i + \sum_{j \textit{ adj to } i} \mathbf{K}(i,j)\mathbf{x}_j$
- This means if we cluster adjacent nodes together, only communicate boundary node contributions.



Partitioning

- Optimal partitioning (minimizing communication) is NP-hard
- I need a FAST partitioning algorithm
- I came up with a greedy BFS-based algorithm, with pretty good results:



MPI Algorithm

- Root breaks mesh into P partitions
- Root constructs and distributes to each process
  - re-indexed sub-matrices and sub-vectors
  - list of elements that must be communicated with neighbors
- All P processors do a sub-matrix, sub-vector matvec
- All P processors exchange/reduce boundary node contributions

Performance Notes

- Cache performance is improved because re-indexed subproblems are more likely to fit.
- Individual sub-matrix matvec can be optimized independently/run on GPU
- Current performance: ≈ 1 GFLOPS on my Core i5 520m (demo)

Future (Current) Work

- OpenCL Sparse Matvec
- GPU FEM matrix construction
  - Most FEM codes build per-element matrices and compile them into a full matrix.
  - Embarrassingly parallel operation followed by careful reduction
- Find/devise a working matvec-only small eigensolver! (Nontrivial!)