I am a computational geometer specializing in (a) quality mesh refinement, and (b) dynamic geometric algorithms. While computational geometry is beautiful in its own right, what drives my work is its application to scientific computing – in particular, the finite element method. My focus straddles theory and practice: I intend to implement all my algorithms, and I intend to mathematically analyze all my programs. Beyond my specialty, I have research or development experience spanning much of computer science: auction theory and game theory; combinatorial optimization; AI, in particular planning and scheduling; and programming languages, from type theory to optimizing compilers.

My current long-term research goal is to design a set of techniques that will allow running large simulations where the geometric domain undergoes changes through time. I have contributed to the following results toward that goal: (1) The first optimal-time mesh refinement algorithm that works on PLC\(^1\) inputs in any dimension, under usual restrictions [HMP06]. The algorithm is easy to implement and is fast in practice. (2) The first parallel algorithm for the same [HMP07]. (3) The first optimal-time dynamic algorithm for meshing point clouds, along with evidence that (1) and (2) should dynamize well [AH06, HA07]. (4) A technique for performing rotations in higher-dimensional analogues to binary trees [HM04, HMS05].

All of my work has been collaborative, hence the ever-present ‘we.’ My typical mode of research is to meet a coauthor over coffee, writing utensils at the ready, debate until exhaustion, then write up the results of the conversation. I am actively working with researchers at CMU, Berkeley, and TTI-C to extend my recent results.

1 Recent results and the near future

As a member of the NSF-funded Sangria project [SAN], I worked on technologies for simulating blood flow. The driving application was to design a ventricular assist device (VAD): a pump that helps – but does not replace – an ailing heart. Simulations of VADs are complex affairs because the setting is highly kinetic: the pump blades rotate; the blood vessel walls flex; the heart muscle beats; blood cells deform.

A Finite Element Method (FEM) simulation runs as follows: the input is a domain, given as a Piecewise Linear Complex (PLC); and a set of Partial Differential Equations (PDEs), such as the Navier-Stokes equations that describe fluid flow. First, we mesh the domain. A mesh is a discretization of the input geometry into elements: triangles or tetrahedra. For the FEM to produce numerically valid results, the elements should have good quality: all the interior angles should be bounded away from \(180^\circ\). Generally, this requires adding Steiner points. Second, given the mesh, FEM theory describes how to turn the PDEs into a sparse system of linear equations. We assemble an appropriate matrix and vector. Finally, we numerically solve the system, giving us an approximate solution to the PDEs.

Finite element simulations with moving elements normally first generate a mesh for the initial geometry, solve once for the velocity, move the points according to the velocity field, then throw away the entire mesh and create a new mesh from scratch. This approach worked well when the dominant cost of the simulation was numerically solving the differential equations. However, solvers have improved dramatically in the past decade, whereas meshers have not kept pace. My recent work focuses on fixing this imbalance.

---

\(^1\)Piecewise Linear Complex: an arrangement of points, segments, polygons – that is, convex polytopes in any dimension. Any two polytopes can only intersect in a lower-dimensional polytope [MTTW95].
Sparse Voronoi Refinement (SVR): Along with Gary Miller and Todd Phillips [HMP06], I developed a mesh refinement algorithm for PLC inputs that achieves the near-optimal time bound of \( O(n \lg L/s + m) \) in any dimension\(^2\). This is the first result that runs in sub-quadratic time in dimension three or higher. The algorithm reverses the usual assumption that the mesh should always conform to the input, and instead imposes that the mesh should always have good quality. Through this invariant, we show that the mesh always has bounded degree, dramatically simplifying both algorithm and proof.

Implementation: Given code for performing geometric tests [She96, NBH03] and code for maintaining a Delaunay triangulation, it took me just 1085 lines of commented SML code to implement the SVR algorithm in two dimensions. Extending to higher dimension required another 30 lines per dimension. On a 20,000-point pathological example in three dimensions, I found that SVR was much faster than Shewchuk’s Pyramid code [She98]: Running on my aging laptop with 1.25 GB of RAM, my implementation returns its result in under one minute, whereas Pyramid hits swap and does not terminate overnight. A more careful C++ implementation of SVR, still in development, is twice as fast as Pyramid even on non-pathological examples such as the Stanford bunny.

Parallelization: Also with Miller and Phillips, I parallelized the SVR algorithm [HMP07]. The algorithm is work-efficient and has shallow parallel depth \( O((\lg(L/s) \lg m)) \), which is only a factor \( O(\lg m) \) off optimal. In practical terms, the algorithm is easy to implement and will speed up linearly in the number of processors. Ours is the first parallel algorithm with both work and depth guarantees in dimension three or higher.

Dynamization: With Umut A. Acar, I have worked on dynamizing SVR. That is, instead of being given up-front a geometry to mesh, we are asked to maintain a quality mesh with no more than the optimal number of vertices in it. Updates come in any order, asking to add or remove input features (points, segments, etc). An SML library for self-adjusting computation [Aca05, ABB+06] allowed us to develop an implementation of dynamic SVR before a full analysis, as we reported at the Fall Workshop in Computational Geometry [AH06]. We found that the dynamized SVR algorithm performs well: insertions and removals of points appear to take polylogarithmic time. Using some techniques from the parallel analysis of SVR, we have also proven that we can maintain a quadtree [BEG90, BET99, MV92] mesh over a point cloud in time \( O(\lg L/s) \) per update. We have submitted this result to ICALP [HA07].

Data structures: In work I presented at CCCG’04 [HM04], Miller and I showed how to perform rotations in a structure that maintains the history of incremental insertions into a Delaunay triangulation. With additional help from Shewchuk, we extended the results to arbitrary dimension and cleaned up the terminology and proofs [HMS05]. In general, it is unclear how to guarantee that the structure will be shallow and small, so we did not publish. However, SVR uses an incremental Delaunay triangulation algorithm as a subroutine, and the parallelization proves that indeed the history of SVR is of bounded depth \( O(\lg L/s) \) and size \( O(m) \). A dynamic SVR algorithm would adapt this work for its underlying data structure without needing to use a self-adjusting computation library.

\(^2\)Throughout this discussion, \( m \) is the optimal output size for any quality mesh. Also, \( L/s \) is the spread of the input: \( L \) is the diameter of the input space (the farthest distance between any two input points), while \( s \) is the smallest distance between any two features that do not intersect. A common assumption in computational geometry is that \( L/s \) is polynomial in \( n \), in which case \( \lg L/s \) can be read to mean \( \lg n \). Under that assumption, \( O(n \lg L/s + m) = O(n \lg n + m) \), which is optimal due to a sorting lower-bound.
2 Longer-term future directions

Given the parallel and dynamic meshing algorithms described above, we now know how to remesh quickly, reducing the meshing time to be only commensurate with that of the numerical solver. However, we can do better: during the TUMBLE project [TUM], we noted that the topology of the mesh requires little change between timesteps to maintain good quality and sizing guarantees. Thus, it should be possible to remesh in sublinear time per timestep. The techniques and analyses in the prior dynamic problem will be applicable to this kinetic problem.

With sub-linear time remeshing, the numerical solver time will once again predominate. To speed it up, we could use *space-time meshing* [US02]. Assuming that interesting dynamics cluster around the geometric features, large unencumbered areas in space will need less careful resolution in time. SVR over three dimensions of space and one of time will automatically do this. If the dynamics affect the geometry, the mesher and solver must interact in ways yet to be resolved.

SVR has harsh input restrictions, as is common in meshing work: All inputs must intersect at obtuse or right angles. The PLC features must individually be convex. There can be no curves in the input. The topology must be watertight. For an algorithm that scientists and engineers can unthinkingly apply (which is my goal), all of these restrictions must be lifted. Other significant improvements include better sliver removal; using the constrained Delaunay rather than conforming Delaunay triangulation; distributed, out-of-core, and streaming meshing; and myriad other projects.

It is also interesting to consider using mesh refinement algorithms in novel ways. SVR puts Steiner points on bands around the medial axis; it seems likely that this fact could be exploited for point location problems, surface reconstruction, and other such geometric questions.

3 Work outside Geometry

Earlier in my graduate student career, I worked with Tuomas Sandholm on methods to reduce communication costs in combinatorial auctions, where bidders bid on *sets* of items, rather than one item at a time. For instance, a bidder may place high value on the set \{ flight, hotel, rental car \} but zero value on any single item, since without the entire set, the trip cannot happen. In experiments, we found we could sizeably reduce communication costs by having the auctioneer incrementally ask the bidding agents for their value on only a few packages without needing to compute their full valuation on all subsets [HS02, HS04].

At NASA Ames Research Center, I helped ready the Livingstone version 2 (L2) model-based diagnosis engine for space flight. The software is given a model of the spacecraft, along with access to sensor readings. If the sensor readings are anomalous, L2 will diagnose the problem: most likely, a valve has failed, but perhaps instead a pipe is leaking. L2 can then recommend a workaround, such as opening a backup valve. Initially on a team of three developers, I came to be the main programmer on the project. I found some algorithmic problems, sped the code up more than one thousand-fold, and removed constructs such as `malloc` deemed “unsafe” by the flight team. L2 is now in orbit around Earth on board the EO-1 spacecraft [HSC04].

Also at NASA, I worked on implementing the Hybrid Concurrent Constraint language, which allowed expressing a system of Ordinary Differential Equations which occasionally changes discontinuously [CG98]. In close collaboration with a mechanical engineer (Adam Sweet), I improved the system so that he could run simulations of a Mars-based rocket fuel factory. Sadly, for intellectual property reasons, my work could never be publicly released.

While an undergraduate at Brown, I helped Preparata, Upfal, and Frieze develop their sequencing by hybridization algorithm by experimentally testing the algorithms they proposed [PFU99, Hud99]. I also worked on the Data Structures Library in Java (JDSL) project led by Tamassia and Goodrich [GHHT99, TGV+01], which led to the development of their well-known textbook.
References


[TUM] TUMBLE project. rioja.sangria.cs.cmu.edu/tumble/index.html The Sangria project’s software package for running moving mesh simulations.