Fast Triangulated Vortex Methods for the 2-D Euler Equations

Giovanni Russo
Universita dell'Aquila
Dipartimento di Matematica
Via Vetoio, Loc Coppito
67010 L'Aquila, Italy

and

John A. Strain[1]
Department of Mathematics
and
Lawrence Berkeley Laboratory
University of California
Berkeley, CA 94720 USA

January 13, 2000

[1]Research supported by NSF Grant DMS-9114308 at the Institute for Advanced Study, and by a NSF Mathematical Sciences Postdoctoral Research Fellowship, by AFOSR Grant AFOSR-92-0165, and by the Applied Mathematical Sciences Subprogram of the Office of Energy Research, U.S. Department of Energy under Contract DE-AC03-76SF00098, at the University of California, Berkeley.

Abstract

Vortex methods for inviscid incompressible two-dimensional fluid flow are usually based on blob approximations. This paper presents a vortex method in which the vorticity is approximated by a piecewise polynomial interpolant on a Delaunay triangulation of the vortices. An efficient reconstruction of the Delaunay triangulation at each step makes the method accurate for long times. The vertices of the triangulation move with the fluid velocity, which is reconstructed from the vorticity via a simplified fast multipole method for the Biot-Savart law with a continuous source distribution. The initial distribution of vortices is constructed from the initial vorticity field by an adaptive approximation method which produces good accuracy even for discontinuous initial data.

Numerical results show that the method is highly accurate over long time intervals. Experiments with single and multiple circular and elliptical rotating patches of both piecewise constant and smooth vorticity indicate that the method produces much smaller errors than blob methods with the same number of degrees of freedom, at little additional cost.

Generalizations to domains with boundaries, viscous flow and three space dimensions are discussed.

Contents

1 Introduction

Vortex methods simulate fluid flow by moving a collection of markers carrying vorticity. They are grid-free, with little or no numerical diffusion, and naturally adaptive, and they preserve moments of the vorticity. They have been generalized in many directions and applied to complex high-Reynolds-number flow [15, 16, 29, 14, 4, 1, 36, 18, 40, 42].

The classical vortex-blob method due to Chorin [15] is based on smoothing point vortices [39] into smooth blobs of vorticity, to obtain higher accuracy and a more robust method. Various high-order methods have been constructed [5, 6], but numerical tests show that the order of accuracy decreases sharply when the flow becomes disorganized. This paper presents an efficient and accurate new vortex method which maintains second-order accuracy during long time integrations.

Different approximations of the vorticity within the same Lagrangian framework lead to other vortex methods. Piecewise constant approximation of the vorticity has been used to study the evolution of vortex patches [11, 49]. Piecewise linear approximation has been used for smooth flows in [2, 13]. In these methods, the velocity is computed from a piecewise polynomial approximate vorticity field, either from the Biot-Savart law or by solving a Poisson problem. At each time step, the vertices of the triangulation are moved according to the computed velocity and the vorticity at the vertices is passively advected. At the next time step the vorticity is again approximated by a piecewise linear function on the triangulation and the process is repeated. These methods converge as the size of the triangles goes to zero. We briefly recall this background material in Section 2.

In this paper we present a vortex method based on the piecewise linear approximation of vorticity on a triangulation. We introduce three important new features which make the method far more accurate, efficient and robust: Delaunay triangulation, fast velocity evaluation, and adaptive initial triangulation. Our algorithm is summarized in Section 3.

We update a *Delaunay triangulation* of the vortices at each time step. A Delaunay triangulation is *locally equiangular*, so it maintains a uniform accuracy over long times. This triangulation can be constructed in $O(N \log N)$ operations, using a fast method described in Section 4.

The second new feature is the fast evaluation of the velocity field. The velocity field due to a piecewise linear vorticity on a triangulation can be evaluated exactly [13]. A straightforward evaluation method, however, results in an $O(N^2)$ computational cost. The complexity can be reduced by using a fast multipole method [12]; we implemented a simplified $O(N^{4/3})$ version. For $N = 51200$ our velocity evaluation is 200 times faster than direct evaluation, and the breakeven point is about $N = 100$. Our fast velocity evaluation method is described in Section 5.

A triangulation allows more flexibility than equal-size blobs in approximat-

ing the initial vorticity. We take advantage of this flexibility to construct the initial triangulation adaptively, to resolve the initial vorticity with few degrees of freedom. As a result, our method can be used to model discontinuous vortex patches as well as smooth vorticity fields. Our adaptive triangulation method is discussed in Section 6.

In Section 7 we present numerical results for smooth and non-smooth initial vorticity fields. We compute the evolution of single and multiple circular and elliptical patches of smooth and constant vorticity, and compare with the exact solution when available. Convergence studies for multiple patches are performed by differencing. We compare our method with vortex-blob and Lagrangian finite element methods and show the long-time accuracy, efficiency and robustness of our method.

In Section 8 we discuss generalizations of the method. We consider viscosity, boundary conditions, three-dimensional problems, higher-order methods. In Section 9 we discuss our conclusions.

2 Vortex methods

In this section we review the vorticity formulation of the 2-D Euler equations, the vortex blob method and the Lagrangian finite element method on which the present method is based.

The Euler equations of two-dimensional incompressible inviscid flow are

$$\frac{\partial u}{\partial t} + (u \cdot \nabla)u = -\frac{1}{\rho}\nabla p, \tag{2.1}$$

$$\nabla \cdot u = 0, \tag{2.2}$$

where ρ is the (constant) density of the fluid, $u = (u_x, u_y)$ is the velocity and p the pressure. Both u and p are functions of $z \equiv (x, y)$ and t. (We will find it convenient on several occasions to use complex notation, in which $z = (x, y) = x + iy$ identifies a point in $\mathbb{R}^2$, thought of as the complex plane.)

The curl of (2.1) gives the vorticity equation

$$\frac{\partial \omega}{\partial t} + (u \cdot \nabla)\omega = 0, \tag{2.3}$$

where

$$\omega := \partial_x u_y - \partial_y u_x \tag{2.4}$$

is the vorticity. Thus the vorticity is transported passively along streamlines. By (2.2), u is the curl of a vector field; in two dimensions the vector field has only one non-zero component, the stream function ψ. Then

$$u_x = \frac{\partial \psi}{\partial y}, \quad u_y = -\frac{\partial \psi}{\partial x}. \tag{2.5}$$

and (2.4) becomes a Poisson equation for the stream function:

$$-\Delta\psi = \omega.$$

In unbounded flow with zero velocity far from the origin, this equation can be solved with the boundary condition $\nabla\psi \to 0$ at ∞ to get the "Biot-Savart law"

$$u(z,t) = \int_{I\!R^2} K(z-z')\omega(z',t)\,dz', \tag{2.6}$$

where

$$K = \frac{1}{2\pi|z|^2}\begin{pmatrix} -y \\ x \end{pmatrix}. \tag{2.7}$$

Flow in a domain with boundary will be considered in Section 8.

The flow can also be described by the flow map $z : I\!R^2 \times [0,T] \to I\!R^2$ defined so that $z(\zeta,t)$ is the position of the fluid particle which at time $t=0$ is at the position ζ.

By (2.6), $z(\zeta,t)$ satisfies

$$\frac{dz}{dt}(\zeta,t) = \int_{I\!R^2} K(z(\zeta,t)-z')\omega(z',t)\,dz'. \tag{2.8}$$

Putting $z' = z(\zeta',t)$ inside the integral gives

$$\begin{aligned} \frac{dz}{dt}(\zeta,t) &= \int_{I\!R^2} K(z(\zeta,t)-z(\zeta',t))\omega(z(\zeta',t),t)\,d\zeta' \\ &= \int_{I\!R^2} K(z(\zeta,t)-z(\zeta',t))\omega_0(\zeta')\,d\zeta' \end{aligned} \tag{2.9}$$

since the Jacobian of $z(\zeta,t)$ is unity.

Vortex methods are based on various recipes for evaluating the Biot-Savart integral with a quadrature formula. Discretizations based on the formulation (2.9) give Lagrangian methods, where the space variable is the initial location of the fluid marker ζ. The convergence study of vortex-blob methods is often based on this formulation, which has the weakness common to most Lagrangian methods: they become inaccurate as the grid is greatly distorted. A "Free-Lagrangian" method based on approximation of the vorticity at time t in (2.8) overcomes this difficulty and helps provide a more accurate approximation of the velocity.

The "point-vortex method" [39] approximates (2.9) by

$$\frac{dz_i}{dt} = \sum_{j\neq i} K(z_i - z_j)\omega_0(\zeta_j)h^2; \tag{2.10}$$

it is very physical since it moves N point vortices with circulations $\Gamma_i = \omega_0(\zeta_i)h^2$. Although the method converges [22], it presents some difficulties. If two vortices come too close together, the velocity approximation becomes unbounded. Also,

a distribution of point vortices is usually a poor approximation to a smooth vorticity distribution.

Chorin [15] observed that the singularity can be mollified by convolving the kernel with a blob function $g_\delta(z)$ to get a smoothed kernel

$$K_\delta = K * g_\delta, \quad g_\delta(z) = \frac{1}{\delta^2} g\left(\frac{z}{\delta}\right).$$

The resulting "vortex-blob" method is

$$\frac{dz_i}{dt} = \sum_{j=1}^{N} K_\delta(z_i - z_j)\omega_0(\zeta_j)h^2. \tag{2.11}$$

Convergence results for this method are given in [24, 5, 1, 25]. The numerical behavior of this method has been studied in [34, 42]; it has been very widely used in practice and generalized to model three-dimensional turbulent flows with boundaries and combustion [29, 14, 16].

Lagrangian finite element methods, on the other hand, approximate ω in (2.8) by a piecewise linear function on a triangulation. For each t let $\mathcal{T}_h(t) = \{\tau_i(t)\}_{i=1}^{N_T}$ be a triangulation covering the support of ω with N vertices $\{z_j(t)\}_{j=1}^{N}$, and let

$$V_h = \{v(z) \in C^0(I\!R^2) : v|_{\tau_i} \text{ is linear for each } i\}$$

be the space of continuous piecewise linear functions over $\mathcal{T}_\langle(\sqcup)$. At each time t the vorticity $\omega(z,t)$ is approximated by the piecewise linear interpolant $\omega_h(z,t) \in V_h$. The velocity is approximated by

$$u_h(z,t) = \int_{I\!R^2} K(z-z')\omega_h(z',t)\,dz' = \sum_{i=1}^{N_T} \int_{\tau_i} K(z')\omega_h(z',t)\,dz' \tag{2.12}$$

in [13] and by solving a Poisson problem in [2]. A natural algorithm is then obtained by transporting the vertices of the triangulation along the streamlines defined by

$$\frac{dz_i}{dt} = u_h(z_i,t)$$

and leaving the topology of the triangulation unchanged.

In this paper, we use (2.12) to approximate the velocity. Each term

$$\int_\tau K(z-z')\omega_h(z',t)\,dz' \tag{2.13}$$

in the sum (2.12) can be evaluated exactly, so the evaluation of the velocity at one vertex costs $O(N)$ operations and the cost of the velocity evaluation is $O(N^2)$. To evaluate (2.13), we follow [13]; fix a triangle τ and a vertex z, and take a coordinate system with origin at z. Then we can write

$$\omega_h(x,y,t) = a + bx + cy$$

on τ. For each i and j, let

$$F^{ij} = \int_\tau K(z - z')x'^i y'^j \, dz'. \tag{2.14}$$

Then

$$\int_\tau K(z - z')\omega_h(z') \, dz' = aF^{00} + bF^{10} + cF^{01}.$$

Let z_1, z_2, z_3 be the vertices of τ, as in Figure 1, and set $z_4 = z_1, z_5 = z_2$ for convenience. We compute the three integrals F^{ij} by splitting τ into three triangles with vertex z, as in Figure 1, and writing

$$\int_\tau = \sum_{j=1}^{3} \sigma_j \int_{\tau_j}$$

where $\sigma_j = 1$ if point z is to the left of $\overline{z_{j+1}z_{j+2}}$ and $\sigma_j = -1$ otherwise.

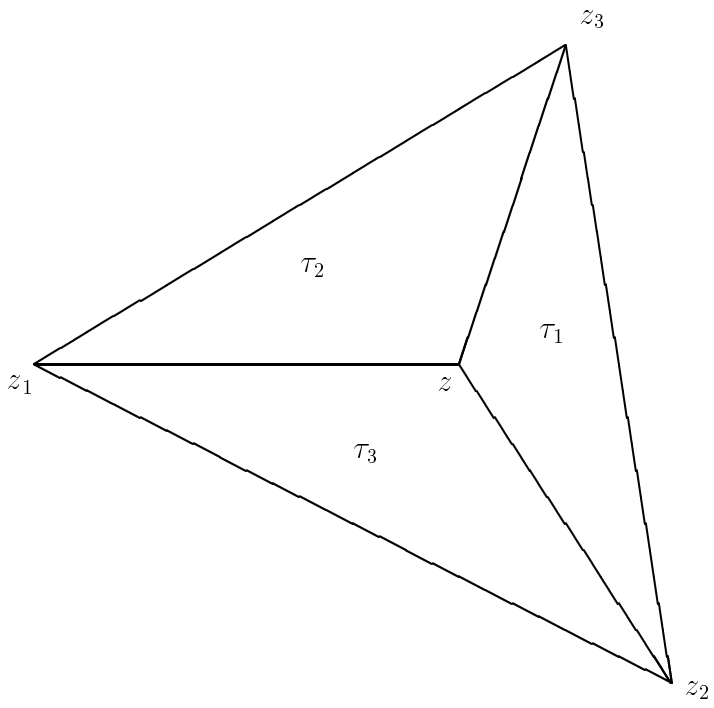


Figure 1: Splitting up the computation of the Biot-Savart integral.

On each subtriangle τ_k, each term can be expressed in polar coordinates and evaluated:

$$F^{00} = \begin{pmatrix} -d_x \log r + d_y \theta \\ -d_x \theta - d_y \log r \end{pmatrix}$$

$$F^{10} = \begin{pmatrix} A\sin\beta\cos\beta + \frac{1}{2}(d_x^2 - d_y^2)\log r - d_x d_y \theta \\ A\cos^2\beta + \frac{1}{2}(d_x^2 - d_y^2)\theta + d_x d_y \log r \end{pmatrix}$$

$$F^{01} = \begin{pmatrix} F_y^{10} - A \\ -F_x^{10} \end{pmatrix}$$

where (see Figure 2), $d_x = x_2 - x_1$, $d_y = y_2 - y_1$, $r = |z - z_1|/|z - z_2|$, θ is the angle $z_2\widehat{z}\tilde{z}z_1$, β is the angle that $\tilde{z} - z$ forms with the x-axis and A is the area of triangle τ_3.

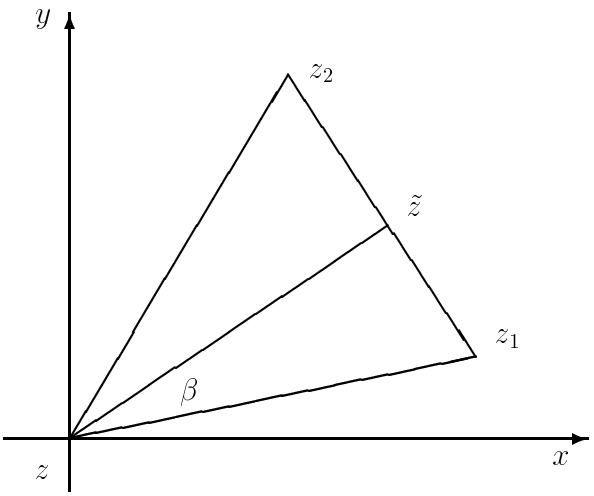


Figure 2: Triangle τ_3.

3 Formal description of the algorithm

In this section we summarize our algorithm in a procedural form. The next three sections will be devoted to a detailed description of the new features we have added.

Algorithm

Step 1 - Input.

Read the initial data from a file. The initial file contains:

- time integration parameters
  final time, time step, order of Runge-Kutta method
- output control parameters
- triangulation parameters
  frequency of retriangulation
  type of triangulation method (McLain, uniform, adaptive)
- fast velocity evaluation parameters

number of terms in expansions
number of neighbor cells
cutoff for refinement
- type of initial data
  vorticity profile (smooth or discontinuous)
  number of vortex patches
- parameters for the adaptive initial grid
  error tolerance
  maximum level of refinement

Step 2 - Initial conditions.

Generate the initial distribution of vertices $\{z_i^0, i = 1 \dots N\}$ according to the initial vorticity. The following options are available:

- read the initial triangulation from a file
- uniform or randomly generated vertices
- adaptive triangulation to resolve ω [described in Section 6]

Assign the initial values of the vorticity $\omega_i = \omega_i(z_i^0)$.

Step 3 - Main loop.

do $n = 1 \dots M$
Compute the velocity associated to ω *and* z [see *Velocity evaluation* below]:
$u^{n-1} = F(\omega, z^{n-1})$
Store the output [every N_s time steps]
if the exact solution is known **then**
evaluate L_1 and L_∞ relative error by comparison with the exact solution
end if
Write output [errors, timing, triangulation, and so on] to files.
Move the points
if [Euler's method]**then**
$z^n = z^{n-1} + u^{n-1}\Delta t$
else if [second-order Runge-Kutta] **then**
$\tilde{z} = z^{n-1} + u^{n-1}\Delta t$
$\tilde{u} = F(\omega, \tilde{z})$
$z^n = z^{n-1} + (u^{n-1} + \tilde{u})\Delta t/2$
else [fourth order Runge-Kutta]
$\tilde{z} = z^{n-1} + u^{n-1}\Delta t/2$
$\tilde{u}^1 = F(\omega, \tilde{z})$
$\tilde{z} = z^{n-1} + \tilde{u}^1\Delta t/2$
$\tilde{u}^2 = F(\omega, \tilde{z})$
$\tilde{z} = z^{n-1} + \tilde{u}^2\Delta t/2$
$\tilde{u}^3 = F(\omega, \tilde{z})$
$z^n = z^{n-1} + (u^{n-1} + 2\tilde{u}^1 + 2\tilde{u}^2 + \tilde{u}^3)\Delta t/6$
end if
end do

Velocity evaluation.

$$u = F(\omega, z)$$

- construct a Delaunay triangulation of $\{z_i\}$ [described in Section 4]
- evaluate the velocity u at each point z_i, $i = 1 \ldots N$
  with the fast summation method [described in Section 5]

4 Delaunay triangulation

4.1 Triangulation and interpolation

Given a set Z of N points z_j in $\mathbb{R}^2$, there are many ways to connect the points into a mesh $\mathcal{T}$ of triangles covering the convex hull C_Z of Z. If function values $f_j = f(z_j)$ are given at the vertices, each triangulation $\mathcal{T}$ produces a piecewise linear interpolant $T(z)$, the unique function which is linear on each triangle of $\mathcal{T}$, continuous and has $T(z_j) = f_j$ for each j. The *error*

$$e_{\mathcal{T}}(f) = \max_{z \in C_Z} |f(z) - T(z)|. \tag{4.1}$$

in such an interpolant can be bounded in terms of the second derivatives of f, the longest edge length of $\mathcal{T}$, and the "condition number" of $\mathcal{T}$, a measure of the angles occurring in $\mathcal{T}$ [50].

We cannot control the second derivatives of f, but we can minimize the error in linear interpolation given Z by choosing the best triangulation for a class of f. Bad triangulations, for most classes have long thin triangles and long edges. Good triangulations have short edges and very few long thin triangles. A simple example is shown in Figure 3. The best triangulation for a given f can be very expensive to find.

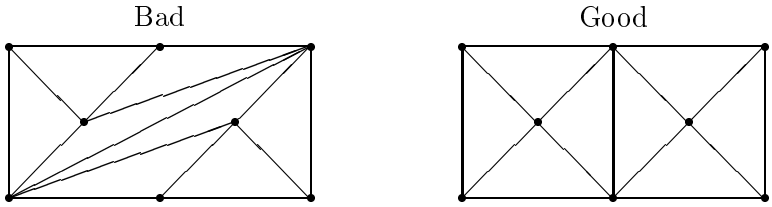


Figure 3: Good and bad triangulations of a simple point set Z.

An affordable alternative is provided by the "Delaunay triangulation". It is almost optimal for error bounds, yet can be constructed in $O(N \log N)$ time. Indeed, [50] shows that no other triangulation can reduce the error bounds by more than a factor of two, while many fast methods for constructing the Delaunay triangulation have been proposed [9, 19, 21, 23, 27, 28, 30, 33, 43, 44]. In this section, we describe the Delaunay triangulation and a fast method for its construction, following [44].

4.2 Definitions and data structures

The Delaunay triangulation can be (and historically has been) defined in many ways. Currently one popular definition is in terms of the Voronoi diagram.

Suppose $Z = \{z_j : j = 1, 2, \ldots, N\}$ is a set of N points in a set $\Omega \subset \mathbb{R}^2$; for convenience we assume Ω has a polygonal boundary. The *Voronoi diagram* of Z is the set of polygons P_j defined by

$$P_j = \{z \in \Omega : |z - z_j| \leq |z - z_i| \text{ for all } i \neq j\}. \tag{4.2}$$

Thus P_j is the set of points in Ω which are closer to z_j than to any other point z_i in Z. See Figure 4 for an example. The Voronoi diagram of Z is a useful tool for identifying nearest neighbors, because the nearest neighbors of z_j are precisely those points z_i whose Voronoi polygons P_i share an edge with P_j. The Voronoi diagram is used to solve closest point problems in computational geometry, for precisely this reason, in [9] and [35].

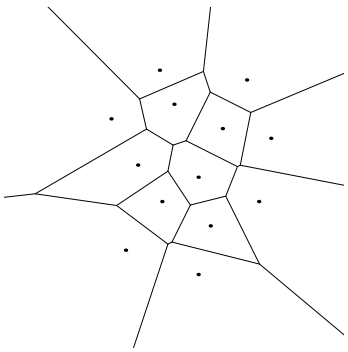
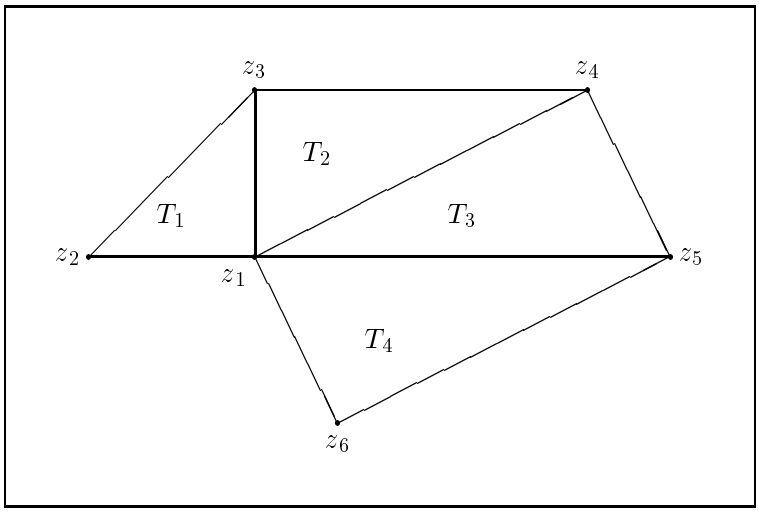


Figure 4: Voronoi diagram associated with a set of points.

The dual of the Voronoi diagram is the Delaunay triangulation, obtained by connecting two points with a triangle edge iff their Voronoi polygons share an edge. In the exceptional case when four points of Z lie on a circle, some edges of their Voronoi polygons have zero length, and one can triangulate the cocircular points in any nondegenerate way, so the resulting Delaunay triangulation is not unique. This possibility requires careful treatment, because the Delaunay triangulation changes by passing through such a case [47].

Another definition, which leads to our method of construction, is through the *circumcircle criterion*; the circumcircle of any triangle contains no other point of Z in its interior. This determines the Delaunay triangulation up to the nonuniqueness caused by cocircular points.

Before discussing the construction of the Delaunay triangulation, we must specify how it is to be stored. We store a triangulation by giving two integer arrays, `itt` and `itv`, in addition to the two real arrays needed to store the coordinates x_i and y_i of the points in Z. Let N_T be the number of triangles in the Delaunay triangulation. (By Euler's formula, $N_T \leq 2N$, which simplifies the assignment of storage considerably.) Then $k = \texttt{itv}(i,j)$ is the index of the ith vertex z_k of triangle T_j, for $i = 1$ to 3 and $j = 1$ to N_T. Also, $k = \texttt{itt}(i,j)$ is the index of the triangle T_k which lies across edge i of triangle T_j. If edge i of triangle T_j lies on the convex hull of Z, we set $\texttt{itt}(i,j) = 0$. See Figure 5 for an example of `itt` and `itv`.



j	1	2	3	4
$\texttt{itv}(1,j)$	2	1	1	1
$\texttt{itv}(2,j)$	1	4	5	6
$\texttt{itv}(3,j)$	3	3	4	5

j	1	2	3	4
$\texttt{itt}(1,j)$	0	3	4	0
$\texttt{itt}(2,j)$	2	0	0	0
$\texttt{itt}(3,j)$	0	1	2	3

Figure 5: A small triangulation and the corresponding triangle to triangle and vertex pointers `itt` and `itv`.

4.3 McLain's method

Next we describe an algorithm due to McLain [31], which starts with a triangle belonging to the Delaunay triangulation and adds triangles one at a time until done, using the circumcircle criterion.

To construct the first triangle T_1, we choose a vertex, z_i say, at random from Z. Then the second vertex, say z_j, is chosen as a closest point to z_i. The third vertex z_k of T_1 is chosen by the circumcircle criterion, applied to each side of $\overline{z_i z_j}$. This criterion says that we select the next vertex z_k so that a) z_k lies outside $\overline{z_i z_j}$ and b) no other point of Z lies in the interior of the circumcircle of the resulting

triangle; see Figure 6. This means that z_k minimizes the signed distance $t(z)$ of the circumcircle center from the line through z_i and z_j

$$t(z) = \frac{(z - z_i) \cdot (z - z_j)}{2(z - z_m) \cdot n}$$

where z_m is the midpoint of $\overline{z_i z_j}$, n is the unit normal to $\overline{z_i z_j}$, and $\cdot$ is the dot product. Any minimizer of $t(z)$ may be chosen as the third vertex of T_1.

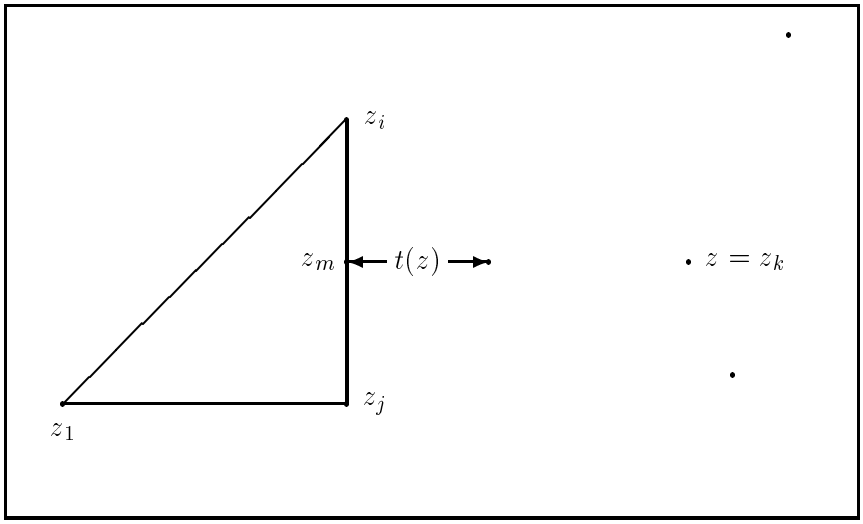


Figure 6: Geometry of McLain's method.

We now have the first triangle T_1. We store the indices of z_i, z_j and z_k in the array $\texttt{itv}(m, 1)$, and set $\texttt{itt}(m, n) = -1$ initially for $1 \le m \le 3$ and $1 \le n \le 2N$. We also swap two vertices if necessary to orient $z_i z_j z_k$ counterclockwise.

The triangulation is now built one triangle at a time—each triangle belongs to the final Delaunay triangulation. We loop through the indices n of existing triangles, adding a triangle (if possible) to each edge m of triangle n which is not already occupied. It may be that it is impossible to add a triangle to edge m, because there are no points of Z outside the line extending that edge. In that case, we mark m as an edge of the convex hull of Z by setting $\texttt{itt}(m, n) = 0$, and proceed to the next edge. If possible, however, we find the third vertex of the new triangle by the circumcircle criterion, as a minimizer of $t(z)$ over Z. If the minimizer is unique, it is taken as the third vertex of the new triangle. Otherwise, there are four or more cocircular points in Z; the two vertices of m and the minimizers of $t(z)$. We then triangulate all cocircular vertices in any nondegenerate way.

We now add the new triangle to `itt` and add its vertices to the next empty location in `itv`. The new triangle may also be a neighbor of some previously constructed triangle which we have not yet accounted for, and if so the appropriate entries must be made in `itt`.

We now proceed to the next edge and repeat. When we run out of unoccupied edges, the Delaunay triangulation will be complete.

4.4 A uniform cell method

McLain's method is robust and easy to program, but can be quite slow when N is large. To speed it up, we introduce a cell structure and vertex-to-triangle pointers. Cells were also used in [9, 30, 33] to speed up Voronoi diagram calculations. The basic idea is that only nearby vertices can affect the addition of a new triangle, if the vertices are reasonably uniform. Thus we can organize the vertices into a spatial data structure [41] and search only nearby vertices. The circumcircle criterion allows us to check that we have included all the vertices which matter. Let C be the circle produced by minimizing $t(z)$ over a subset of Z. Then no point outside C can be a global minimizer of $t(z)$. Thus any candidate for a new vertex excludes all vertices of Z outside C.

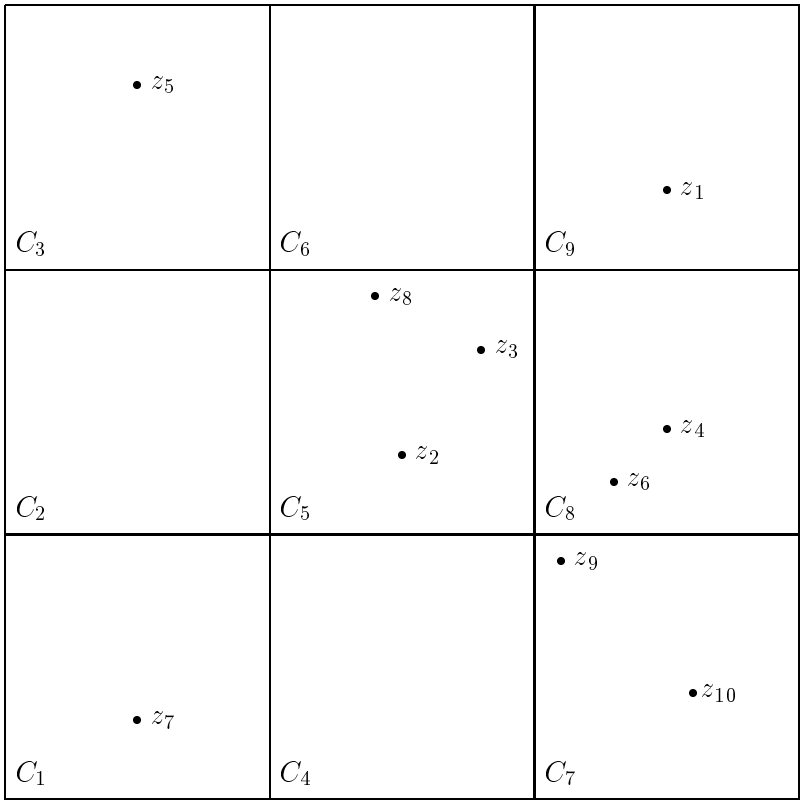
There are two stages of the triangle addition process which require $O(N)$ work. First, we have to find the minimizer of $t(z)$ over Z. Second, we have to check all previously found triangles to find those sharing an edge with the new triangle.

We reduce the cost of the minimization step by organizing the vertices Z into a data structure according to their spatial location. We first put Z in a rectangle C with sides parallel to the coordinate axes. Then we subdivide C into $N_C = O(\sqrt{N}) \times O(\sqrt{N})$ rectangular cells and store each z_i in the cell where it lies. To do this, we use an array `icv` of length N which contains the index of each vertex and an array `icv1` of length N_C which contains, in its jth location, the location in `icv` where storage for the vertices in cell j begins. Thus the points z_j in cell i have their indices j stored in `icv` between addresses `icv1`(i) and `icv1`$(i+1)-1$ inclusive; we set `icv1`$(N_C+1) = N+1$ for convenience. This data structure can be constructed in $O(N)$ work. An example is shown in Figure 7.

Now we reduce the cost of minimizing $t(z)$ as follows. Say we are finding minimizers of $t(z)$ outside $\overline{z_i z_j}$. Find the cells i_1 and i_2 which contain z_i and z_j (usually $i_1 = i_2$) and construct the smallest rectangular union R of cells in the cell structure which contains both i_1 and i_2. Rather than minimizing $t(z)$ over all points, we now find only those minimizers of $t(z)$ which lie in R.

If R contains no points outside $\overline{z_i z_j}$, we revert to McLain's procedure for this edge. If there is a point in R on the correct side of $\overline{z_i z_j}$, then we will find a minimizer z_k of $t(z)$ over $Z \cap R$. This point may not be the global minimizer, because R may not contain the latter. But *any* minimizer of $t(z)$ over all N points of Z will lie inside the circumcircle C of $z_i z_j z_k$. In practice, the minimizer of $t(z)$ over R will be the global minimizer almost all the time, if the point distribution is reasonably uniform.

Hence if $C \subset R$, we have already found the minimizer of $t(z)$ over Z. Otherwise, we expand R until it contains C, and search the new R. This produces all



i	1	2	3	4	5	6	7	8	9	10
`icv1`(i)	1	2	2	3	3	6	6	8	10	11

j	1	2	3	4	5	6	7	8	9	10
`icv`(j)	7	5	2	3	8	9	10	4	6	1

Figure 7: Ten points in a uniform cell data structure.

the global minimizers of $t(z)$.

If more than one minimizer is found, we must check previous triangles to avoid degeneracy. The new triangle can cross only triangles which have all three vertices on C. To check these triangles efficiently, we need pointers from the points of Z to triangles having them as vertices. This requires $3N_T \leq 6N$ integer locations, but each point belongs to six triangles only on the average. Hence the storage method must allow for variations in the length of triangle storage from point to point, and this structure must be constructed simultaneously with the triangulation rather than all at once.

This situation is ideal for the use of a a *linked list*. This is a single array $\texttt{ivt}(i,j)$, where $i = 1$ to 2 and $j = 1$ to $3N_T$, with the triangle indices for a given point stored in a chain of non-contiguous locations, with each triangle index stored in $\texttt{ivt}(1,j)$ and $\texttt{ivt}(2,j)$ occupied by a pointer to the next triangle index. To get started, a triangle T_k to which z_j belongs is stored in $\texttt{ivt}(1,j)$ for $1 \leq j \leq N$; then $\texttt{ivt}(2,j)$ points to the location in `ivt` where the index of the *last* triangle (in order of creation) to which z_j belongs is stored. If this location is k and $l = \texttt{ivt}(1,k)$ then T_l is the last triangle to which z_j belongs and $\texttt{ivt}(2,k)$ is the location in `ivt` where the *next to last* triangle index for z_j is stored. The storage proceeds backwards in this way until the end of the triangle list for the jth point is signaled by a -1 in $\texttt{ivt}(2,n)$ for some n. We add a triangle to the list of z_j simply by resetting the end link $\texttt{ivt}(2,j)$ and adding the triangle to the next empty location at the end of `ivt`. See Figure 8 for an example of the linked list.

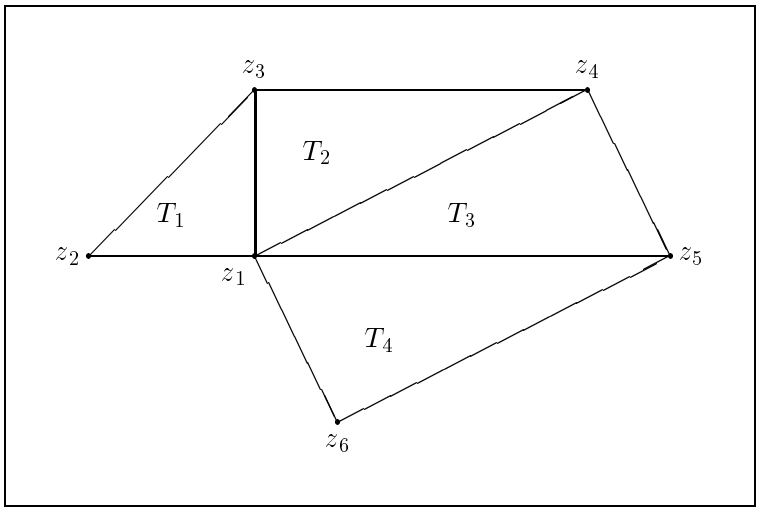
Given this storage arrangement, we can easily look up all triangles having z_k as a vertex, check if all three vertices lie on the circumcircle, and check for degeneracy if necessary.

The linked list also speeds up the second $O(N)$ stage of the triangle addition process; check all previously constructed triangles and find those sharing a common edge with the new triangle, to add to `itt`. This is easy to speed up, because `ivt` points from vertices to triangles containing them; hence we can find all the desired triangles immediately in time proportional to their number and independent of N.

Finally, we update the pointers and proceed to the next edge of the growing triangulation. When there are no more edges to be augmented, the triangulation is concluded.

4.5 An adaptive cell method

The uniform cell method is highly efficient when the points are reasonably uniform. Unfortunately, in applications, we do not have uniform points. Even for interpolation of a function, we want more data points where the function varies more rapidly [37]. Practical situations often lead to highly nonuniform point distributions, for which both numerical experiments and theory indicate that the uniform cell method requires close to its worst-case $O(N^2)$ time. Even worse, the



j	1	2	3	4	5	6	7	8	9	10	11	12
$\texttt{ivt}(1,j)$	1	1	1	2	3	4	2	2	3	3	4	4
$\texttt{ivt}(2,j)$	11	-1	8	10	12	-1	-1	-1	7	-1	9	-1

Figure 8: A small triangulation and the corresponding linked list of vertex-to-triangle pointers.

uniform method can be fooled simply by adding a few outlying points at a large distance from the rest of the points; it will then construct a cell structure which is much too coarse, and the only remedy for this is adaptivity.

In this section, we present an *adaptive* cell method which runs much faster than the uniform method on nonuniform point distributions. The idea is to sort points into cells of varying size, with no more than s points per cell. This is done by recursively subdividing the rectangle C until no cell contains more than s points.

At the end of the construction, we have partitioned C into N_C subcells of varying sizes, as shown in Figure 9 for a small example with $s = 3$ and $N_C = 22$. For each cell i, we store a) data on its spatial location and b) the indices j of the points z_j lying in cell i. Part a) is achieved by storing three pointers per cell, arranged in a $3 \times N_C$ array $\texttt{icxy}(n, i)$; $L = \texttt{icxy}(3, i)$ is the *level* of i in the sense that cell i is 2^{-L} times smaller in each dimension than the original cell C. Two more pointers $n_x = \texttt{icxy}(1, i)$ and $n_y = \texttt{icxy}(2, i)$ give the spatial location of the cell, as if it were part of a regular grid on C composed entirely of cells of level L; its lower left corner is at the point $(x = a_x + n_x \cdot h_x, y = a_y + n_y \cdot h_y)$. Here $C = [a_x, b_x] \times [a_y, b_y]$ while the sides of i have lengths $h_x = 2^{-L}(b_x - a_x)$ and $h_y = 2^{-L}(b_y - a_y)$ respectively. Part b) is achieved by storing a list `icv` of points lying in each cell. Additional pointers `icv1` and `icv2` give the addresses in `icv` of the beginning and end of the list of points in cell i. Thus cell i contains

(x_j, y_j), where $j = \texttt{icv}(k)$ for $k = \texttt{icv1}(i), \ldots, \texttt{icv2}(i)$.

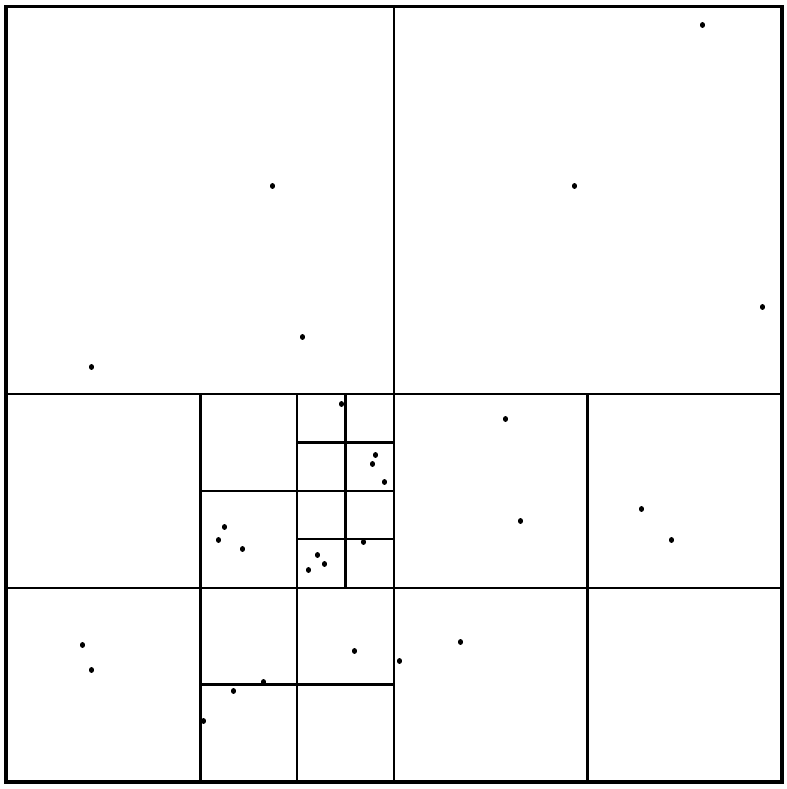


Figure 9: Adaptive cell structure with no more than three points per cell.

The cells are sorted lexicographically within each level, and arranged by level. Thus we use also pointers `ilc1` such that all the cells on level L are given by $i = \texttt{ilc1}(L), \texttt{ilc1}(L) + 1, \ldots, \texttt{ilc1}(L + 1) - 1$. The purpose of lexicographic ordering on each level is to speed up the operation of searching for a cell with given values of n_x, n_y and L; we simply carry out a binary search of $\texttt{icxy}(1, i)$ and $\texttt{icxy}(2, i)$ for i between $\texttt{ilc1}(L)$ and $\texttt{ilc1}(L + 1) - 1$. This operation is important when we construct the list of neighbors of a given cell or when we find all cells which intersect a given geometric object. This data structure is similar to that used in [12, 48].

Next we describe the construction of the adaptive cell structure. We begin with the rectangle C and subdivide it into four cells by bisecting each coordinate. We assign each point z_j to the cell in which it lies. These cells constitute level 1 of the structure. To construct level 2, we run through cells created at level 1 and bisect any which contain more than s points, reassigning points to the subcells in which they lie. The resulting cells are added to the end of `icxy`, `icv`, `icv1` and `icv2` in the order in which they were formed. Cells which are subdivided are marked for deletion, and when the level 2 cells have all been created, the

subdivided cells from level 1 are deleted and storage is reassigned. Thus empty cells are kept but subdivided cells are eliminated; the result is a partition of C into cells with disjoint interiors. After deletion, pointers `ilc1` are made. The algorithm now proceeds recursively one level at a time. At each level, the cells created in the previous level are subdivided where necessary, and the new cells assigned numbers `icxy` and storage in `icv1` and `icv2`. Subdivided cells are deleted and storage moved up.

When this process terminates, either because the maximum number of levels is reached or because no cell has more than s points in it, the cells on each level are sorted and rearranged in lexicographic order. Finally we make pointers `ivc` from points to cells, showing which cell a point lies in, and we are done.

We need to carry out two primitive operations on this data structure. First, we have to find the nearest neighbor cells of a given cell i, all cells having a point in common with i. If all the cells were the same size, the spatial location numbers of the desired cells would be obtained from $\texttt{icxy}(n, i)$ by adding 0, -1 or $+1$ to $\texttt{icxy}(1, i)$ and $\texttt{icxy}(2, i)$. A search through the cells on level $L = \texttt{icxy}(3, i)$ would produce them. The cells are not all the same size, so we must look on all levels for neighbors.

For example, suppose we are looking for the lower left corner neighbor of i. We begin on the same level as i by setting $n_x = \texttt{icxy}(1, i) - 1$ and $n_y = \texttt{icxy}(2, i) - 1$. These are the values $\texttt{icxy}(1, j)$ and $\texttt{icxy}(2, j)$ would have if a cell j of the same size as i occupied the lower left corner position. Thus we search through cells on level $L = \texttt{icxy}(3, i)$ for a cell with numbers n_x and n_y. If the search succeeds, we are done. If it fails, we must look for a larger or smaller cell. A larger cell is easier to find in general, so we set $n_x \leftarrow \lfloor n_x/2 \rfloor$, $n_y \leftarrow \lfloor n_y/2 \rfloor$ and $L \leftarrow L - 1$. and search level L for the cell (n_x, n_y). This procedure is repeated until either we find the cell or we reach the top level. If the latter occurs, we need a smaller cell. The corners and sides differ here because on the corners we are looking for a single cell, while on the sides we are looking for several smaller cells. On the lower left corner, for example, we seek a smaller cell by putting $n_x \leftarrow 2 \cdot n_x + 1$, $n_y \leftarrow 2 \cdot n_y + 1$, $L \leftarrow L + 1$, and searching on level L, then repeating this procedure as needed until the cell is found.

On the sides, the search for smaller neighbors is slightly more complicated. We begin, say on the left side, with $n_x \leftarrow n_x - 1$ and $n_y \leftarrow n_y$. If no cell on level L with numbers (n_x, n_y) exists, then we look for smaller neighbors, possibly several of them. First, we subdivide (n_x, n_y) into four cells and stack the right-hand two cells. The left two cells are discarded. We now run through the stack, searching for each cell on the level where it should exist. If found, it is added to the neighbor list and we continue with the next stack entry. If no such cell exists, it is subdivided, the right-hand two cells are stacked and the left-hand ones discarded, and we continue with the next stack entry. When this process terminates, we have the list of neighbors.

Another operation we need to carry out with this data structure is to find all cells which intersect a given geometrical object Ω such as a square or the

intersection of a circle with a half-space. A fast method uses recursion: Stack the four top-level cells. Examine each for existence and intersection; if it exists and intersects Ω it is added to our list, if it does not intersect it is discarded, and if it does not exist but intersects, then it is subdivided, its subcells are stacked, and we proceed.

An adaptive cell method for Delaunay triangulation is now a straightforward extension of the uniform method. Only the search strategy changes, as follows.

The first step is to search the cell or the two cells containing the vertices z_i and z_j of the current edge. If z_k minimizes $t(z)$ over this search area, we compute the circumcircle of $z_i z_j z_k$ and test whether it is contained in the search area. If it is, we have found the global minimizer and can proceed. Otherwise, we must enlarge the search area.

Our next step is then to find the nearest neighbor cells of the one or two cells of the first search area and take their union as the second search area. We expect a single layer of nearest neighbors to be sufficient in most cases because they will "screen" the current edge from further points. The second search can again have three outcomes. First suppose no point has yet been found when the second search terminates. Then it is quite likely but not certain that $\overline{z_i z_j}$ is on the boundary of the convex hull of Z; thus we find all cells intersecting the half-space outside $\overline{z_i z_j}$ and take their union as the third search area. If, on the other hand, we have a local minimizer z_k, let C be the circumcircle of z_i, z_j and z_k. If the interior of C is contained in the second search area, we have found the global minimizer and can proceed.

Otherwise, we must enlarge our scope to the third and final search area, comprising all cells which intersect C. After searching the third search area, we have either found all global minimizers of $t(z)$ which lie outside $\overline{z_i z_j}$, or determined that $\overline{z_i z_j}$ lies on the boundary of the convex hull of Z, and can proceed.

A considerable speedup is obtained by precomputing all neighbors of nonempty cells and storing them. This eliminates the necessity of repeatedly finding the neighbors of cells, a considerable savings when s is large.

4.6 Numerical results

We have implemented the three algorithms described in this section in Fortran and tested their performance on many sets of data points. Results from only one set of test data will be reported here. The data consists of four sets of $N/4$ normally distributed points, centered at four points in $[0,1]^2$ and with variances given by $\sigma = 0.15, 0.15/7, 0.15/7^2, 0.15/7^3$. An example with $N = 800$ is shown in Figure 10, where the fourth set of points is inside the third set and therefore invisible.

Table 1 reports the results of triangulating this set of data points, with N ranging from 100 to 51,200. The column headings have the following meanings;

N is the number of data points.

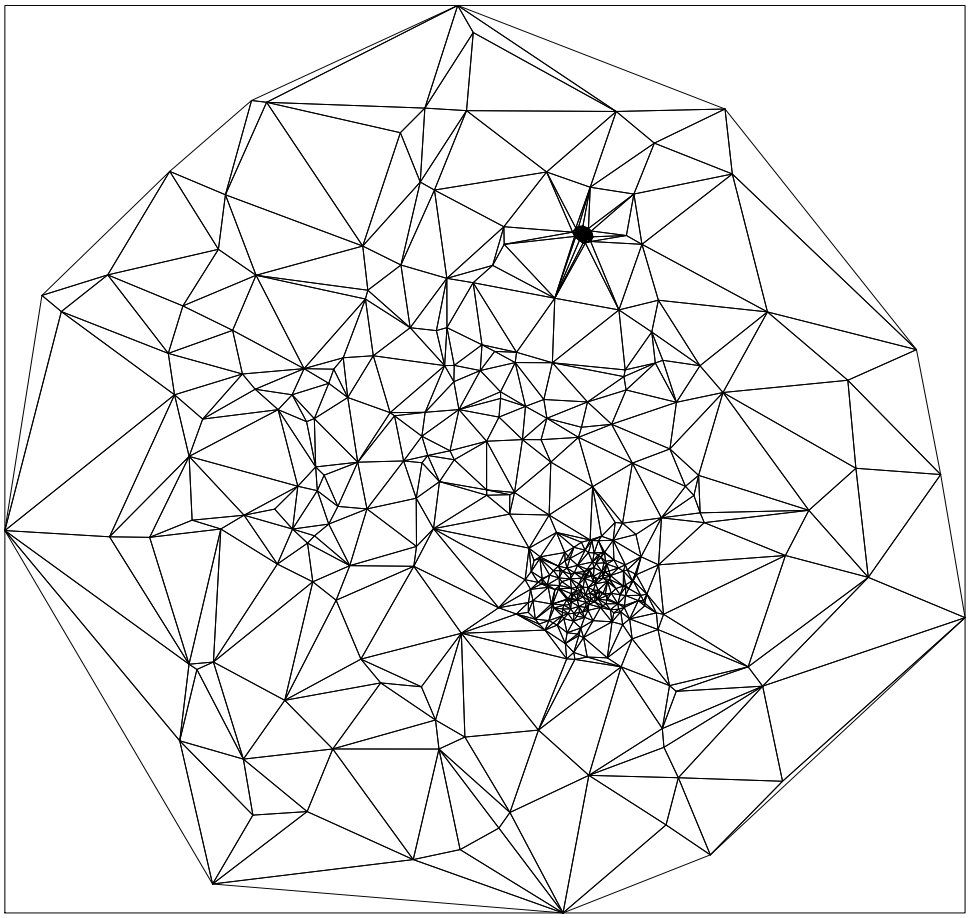


Figure 10: Sample Delaunay triangulation of $N = 800$ nonuniform points.

N_T is the number of triangles produced.

T_q is the CPU time in seconds required by our implementation of McLain's method, estimated by extrapolation for $N > 10,000$.

T_u is the CPU time required by the uniform cell method, with $N_C = (\lfloor\sqrt{N}\rfloor)^2$ cells, estimated by extrapolation for $N > 20,000$.

T_a is the CPU time required by the adaptive cell method, using $s = 25$.

N_C is the number of cells created by the adaptive method.

C_T $= 10^4 \cdot T_a / N \log N$ is the scaled CPU time constant for the adaptive method.

L is the highest level used in construction of the adaptive cell structure.

We can draw the following conclusions from this table; first, both the uniform and adaptive methods are faster than the quadratic method as soon as $N \geq 200$. Thus they are to be preferred for large problems if sufficient memory is available. The uniform method requires about $26N$ integer memory in addition to $2N$ real storage for x and y; about $12N$ of the integer storage is used just to store the triangulation. Thus the uniform method uses only about twice the minimum amount of memory. The adaptive method typically has similar storage requirements, despite the larger amount of information it stores, because we take bigger cells and hence have fewer of them. It is difficult to give a tight upper bound for its memory usage, especially when the number of points per cell is chosen very small. However, on this example, with neighbor lists stored, it required about $4N$ additional integer locations for large N.

Second, on these nonuniformly distributed points, the uniform method runs quickly when N is small, but degenerates to $O(N^2)$ performance when N gets large. This is to be expected. The adaptive method, on the other hand, displays a gratifyingly regular $O(N \log N)$ performance throughout the whole range of N. It beats the uniform method consistently when $N \geq 400$, and outperforms the quadratic method as soon as $N \geq 200$. The constancy (and even slight decrease) of C_T indicates that the adaptive cell method is $O(N \log N)$ or better, even on these extremely nonuniform point distributions.

5 Fast velocity evaluation

We now consider the most expensive part of our method, the velocity evaluation. Given ω piecewise linear on the triangulation $\mathcal{T}$, we need to evaluate the velocity

$$u(z) = \int K(z - z')\omega(z')dz' \tag{5.1}$$

at each vertex z_j of $\mathcal{T}$. Here we omit a constant factor and ignore conjugation for simplicity, so $K(z) = 1/z$, and we integrate over $\mathcal{T}$, the support of ω, with respect to $dz' = dx'dy'$.

N	T_q	T_u	T_a	C_T	N_C	L
100	0.14	0.11	0.20	4.3	28	8
200	0.54	0.32	0.44	4.2	40	10
400	2.1	1.1	0.97	4.0	58	10
800	8.8	3.9	2.1	3.9	109	11
1600	38	13.3	4.4	3.7	178	12
3200	136	51	9.2	3.6	319	13
6400	566	198	19.6	3.5	583	14
12800	2267*	779	41	3.4	1144	14
25600	9068*	3118*	82	3.2	2275	15
51200	36274*	12475*	169	3.1	4426	16

Table 1: Timings for constructing the Delaunay triangulation of N nonuniformly distributed points, using the quadratic, uniform and adaptive cell methods. Asterisks denote timings obtained by extrapolation for the quadratic and uniform methods.

Directly evaluating u at the N vertices of $\mathcal{T}$ via (2.12) costs $O(N^2)$ work with a large constant. We now present a fast algorithm which requires $O(N^{4/3}\log\epsilon)$ time to evaluate N values of u within an error tolerance ϵ, when $\mathcal{T}$ is quasi-uniform (when there are upper and lower bounds proportional to N on the number of triangles in any fixed area). Our algorithm is based on the fast multipole method [12], but differs in forming moments of a continuous source rather than point charges.

5.1 Splitting

Our first step is to split the velocity u at each vertex into *local* and *far-field* parts u_L and u_F. To do this, let C_0 be a rectangle containing $\mathcal{T}$ and divide C_0 into N_C square cells C of equal side length, say $2h$. Fix a vertex $z = z_j$. Then

$$\begin{aligned} u(z) &= \sum_C \int_C K(z-z')\omega(z')dz' \\ &= u_L(z) + u_F(z) \end{aligned} \tag{5.2}$$

where u_L is the sum of those terms due to cells C within r cells of z, and u_F is the remainder (see Figure 11). Thus

$$u_F(z) = \sum_{\mathrm{d}(z,C)>(2r+1)h} \int_C K(z-z')\omega(z')dz' \tag{5.3}$$

where the distance from a cell C (with center c) to the point z is defined by

$$\mathrm{d}(z,C) = \max\left(|\Re(z-c)|, |\Im(z-c)|\right). \tag{5.4}$$

Note that $\omega(z')$ is piecewise linear on the triangulation $\mathcal{T}$; thus the integral over each cell C is a sum of integrals over subtriangles of triangles intersecting C. The most efficient way to evaluate these integrals is to carry out a preliminary step in which the triangulation is refined wherever necessary to make each triangle lie completely within a single cell C. This refinement is implemented recursively; we stack all triangles, then cut each one which crosses a cell boundary and put the resulting pair of triangles back on the stack. When the stack is exhausted, no triangle crosses a cell boundary.

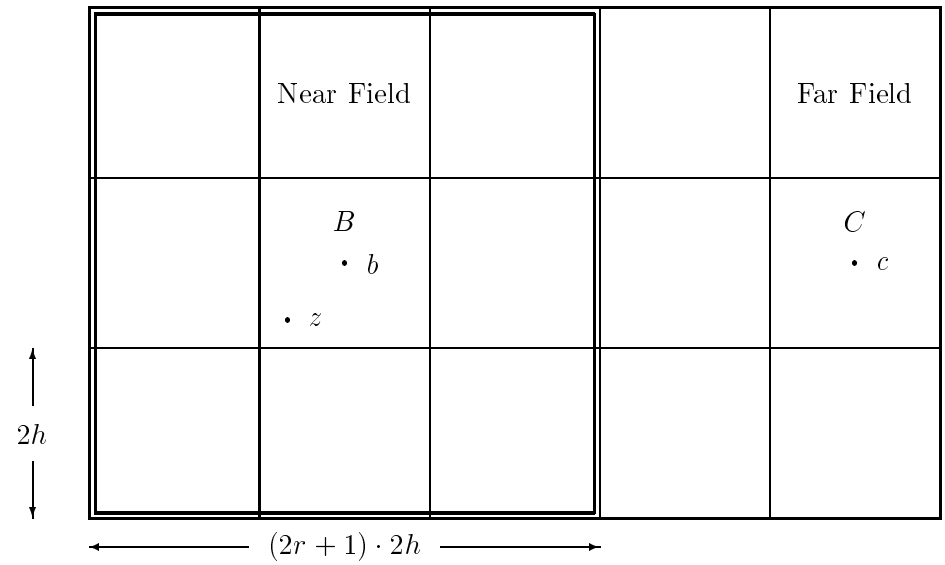


Figure 11: Cells and centers for fast velocity evaluation, with $r = 1$.

5.2 Laurent expansion of u_F

Now consider the far-field. Let C be a given cell (with center c) contributing to the far-field velocity evaluated at point z. Then for each $z' \in C$, we can expand K about c in a Laurent series:

$$\begin{aligned} K(z - z') &= \frac{1}{z - z'} \\ &= \frac{1}{z - c} \sum_{n=0}^{\infty} \left(\frac{z' - c}{z - c} \right)^n . \end{aligned} \tag{5.5}$$

How well does this series converge? By elementary geometry, we have

$$\left| \frac{z' - c}{z - c} \right| \leq \frac{\sqrt{2}}{2r + 1} = \rho. \tag{5.6}$$

(Typically $r = 1$ and $\rho = 0.4714$.) Hence the error (relative to $1/(z-c)$) in truncating the Laurent series of K after the pth term is bounded by

$$E_p = \left| \sum_{n=p+1}^{\infty} \left(\frac{z'-c}{z-c} \right)^n \right| \leq \frac{\rho^{p+1}}{1-\rho} \leq \rho^p \tag{5.7}$$

since $\rho \leq 1/2$. If $r = 1$, for example, we can guarantee $E_p \leq 10^{-k}$ with $p \sim 3k$ since $\rho^3 \leq 0.105$. In any case, we now assume an error tolerance ϵ has been specified, and r, h and p are chosen to make $E_p \leq \epsilon$.

Then we have, within error $\epsilon|\omega|_1 = \epsilon \int |\omega|$,

$$\begin{aligned} u_F(z) &= \sum_{\mathrm{d}(z,C)>(2r+1)h} \int_C \sum_{n=0}^{p} \frac{1}{z-c} \left(\frac{z'-c}{z-c} \right)^n \omega(z')dz' \\ &= \sum_{\mathrm{d}(z,C)>(2r+1)h} \sum_{n=0}^{p} C_n (z-c)^{-n-1} \end{aligned} \tag{5.8}$$

where the coefficients C_n for cell C are defined by

$$C_n = \int_C (z'-c)^n \omega(z')dz'. \tag{5.9}$$

Since ω is piecewise linear and we subdivided the triangulation where necessary to make it compatible with the cell structure, we have

$$C_n = \sum_{T \subset C} T_n \tag{5.10}$$

where

$$T_n = \int_T (z'-c)^n \omega(z')dz' \tag{5.11}$$

is the moment of a linear function over a single triangle T. Clearly we need only evaluate the modified moments

$$T_n^{\alpha\beta} = \int_T \frac{(z-c)^n}{n!} x^\alpha y^\beta dxdy \tag{5.12}$$

where $\alpha + \beta \leq 1$, and we have added a factor of $n!$ to simplify later formulas.

Let's write $z - c = ax + by - c$ for the time being, where a and b are arbitrary. Then after evaluating T_n^{00}, we can get the rest by differentiation:

$$T_n^{\alpha\beta} = \left(\frac{\partial}{\partial a} \right)^\alpha \left(\frac{\partial}{\partial b} \right)^\beta T_{n+\alpha+\beta}^{00}. \tag{5.13}$$

To evaluate T_n^{00}, we apply the Divergence Theorem to get

$$T_n^{00} = \int_T \nabla \cdot F dxdy = \int_{\partial T} F \cdot \nu ds. \tag{5.14}$$

Here

$$\frac{z^n}{n!} = \nabla \cdot F = \nabla \cdot \left(\frac{z^{n+1}}{a(n+1)!}, 0 \right) = \nabla \cdot \left(0, \frac{z^{n+1}}{b(n+1)!} \right) \tag{5.15}$$

and

$$\nu ds = (\Delta y_k, -\Delta x_k) d\theta, \qquad 0 \le \theta \le 1 \tag{5.16}$$

if $z_k = x_k + iy_k$ are the three vertices of T. Here Δ is the forward difference operator $\Delta f_k = f_{k+1} - f_k$ with respect to the index $k = 1, 2, 3$ and we put $z_4 = z_1$. It follows that

$$T_n^{00} = \sum_{k=1}^{3} \frac{\Delta y_k}{a \Delta z_k} \Delta \left(\frac{z_k^{n+2}}{(n+2)!} \right) \tag{5.17}$$

$$= \sum_{k=1}^{3} \frac{-\Delta x_k}{b \Delta z_k} \Delta \left(\frac{z_k^{n+2}}{(n+2)!} \right) \tag{5.18}$$

To simplify the calculation of T_n^{10} and T_n^{01}, we differentiate (5.17) with respect to b and (5.18) with respect to a. Finally, we set $a = 1$ and $b = i$ to get

$$T_n^{00} = \frac{i}{2} \sum_{k=1}^{3} \frac{\Delta \overline{z_k}}{\Delta z_k} \Delta \left(\frac{z_k^{n+2}}{(n+2)!} \right) \tag{5.19}$$

$$T_n^{10} = i \sum_{k=1}^{3} -\frac{(\Delta x_k)^2}{(\Delta z_k)^2} \Delta \left(\frac{z_k^{n+3}}{(n+3)!} \right) + \frac{\Delta x_k}{\Delta z_k} \Delta \left(\frac{x_k z_k^{n+2}}{(n+2)!} \right) \tag{5.20}$$

$$T_n^{01} = \sum_{k=1}^{3} -\frac{(\Delta y_k)^2}{(\Delta z_k)^2} \Delta \left(\frac{z_k^{n+3}}{(n+3)!} \right) + \frac{\Delta y_k}{\Delta z_k} \Delta \left(\frac{y_k z_k^{n+2}}{(n+2)!} \right). \tag{5.21}$$

Note as a check that $T_n^{10} + iT_n^{01} = (n+1)T_{n+1}^{00}$. Note also that this calculation works for *any* polygon, not just a triangle.

5.3 An $O(N^{3/2} \log \epsilon)$ algorithm

Separation of local interactions from the far-field and Laurent expansion of the latter leads already to algorithms which cost $O(N^{3/2} \log \epsilon)$ time with N quasi-uniformly distributed triangles and an error tolerance ϵ. To construct such an algorithm, divide $\mathcal{T}$ into N_C cells C of side length $2h$, each containing $O(N/N_C)$ triangles (we can ignore preliminary subdivision as it only affects the constant) and choose parameters r and $p = O(\log \epsilon)$ to make $E_p \le \epsilon$. The number N_C will be chosen later to achieve maximal efficiency. Then evaluate $u_L(z_j)$, for each vertex z_j, directly in $O(N/N_C)$ time per vertex or $O(N^2/N_C)$ total time. For the far-field, form Laurent coefficients for each cell in $O(pN)$ time (since each triangle contributes to p coefficients) and evaluate $O(N_C)$ p-term Laurent series at N points in $O(pNN_C)$ time. Choosing $N_C = O(N^{1/2})$ minimizes the total time which is then $O(N^{3/2}p)$ where $p = O(\log \epsilon)$. Thus this gives an $O(N^{3/2} \log \epsilon)$ algorithm with N quasi-uniformly distributed triangles.

5.4 An $O(N^{4/3}\log\epsilon)$ algorithm

We next add a further observation which reduces the time to $O(N^{4/3}\log\epsilon)$ with N quasi-uniformly distributed triangles. The observation is that the far-field is smooth, hence well approximated by a Taylor series in each cell. The Taylor series can be computed by summing over the far-field contributions from each far-field cell, then evaluated once and for all at each vertex. This further decoupling of sources from points of evaluation leads to an $O(N^{4/3}\log\epsilon)$ algorithm.

Thus consider a cell B, with center b, containing triangle vertices z_j where we wish to evaluate $u_F(z)$. Each term in each Laurent series has a Taylor expansion

$$\frac{1}{(z-c)^{n+1}} = \sum_{m=0}^{\infty} \left(\begin{array}{c} n+m \\ m \end{array} \right) (b-c)^{-n-m-1}(b-z)^m \tag{5.22}$$

about the cell center b. Thus,

$$u_F(z) = \sum_{m=0}^{\infty} B_m (b-z)^m \tag{5.23}$$

where the Taylor coefficient B_m in cell B is given by

$$B_m = \frac{1}{m!} \sum_C (b-c)^{-m} \sum_{n=0}^{p} (n+m)!(b-c)^{-n-1}\frac{C_n}{n!}. \tag{5.24}$$

The error in truncating the Taylor series after p terms is bounded by

$$\begin{aligned} E_p &= \left| \sum_{m=p+1}^{\infty} B_m (b-z)^m \right| && (5.25) \\ &\leq \sum_{m=p+1}^{\infty} 2|\omega|_1 (2(r+1)h)^{-m-1}(\sqrt{2}h)^m && (5.26) \\ &\leq \frac{|\omega|_1}{(r+1)h}\rho^p && (5.27) \end{aligned}$$

with $\rho = 1/(\sqrt{2}(r+1))$. Clearly this can be made $\leq \epsilon$ by choice of p once r and h are fixed, and $p = O(\log\epsilon)$.

This transformation leads to an $O(N^{4/3}\log\epsilon)$ algorithm as follows. As before, we divide T into N_C cells each with $O(N/N_C)$ triangles. The local part costs $O(N^2/N_C)$ as before. The far-field part costs $O(pN)$ to form Laurent coefficients, $O(N_C^2 p^2)$ to convert Laurent to Taylor series and $O(Np)$ to evaluate Taylor series. Hence $N_C = O((N/p)^{2/3})$ gives the minimum time and it is $O(N^{4/3}|\log\epsilon|^{2/3}) = O(N^{4/3}\log\epsilon)$.

5.5 Refinements

There are three or four refinements to the final algorithm which collectively produce a factor of three or four speedup for large N, and one which makes the algorithm $O(N\log N\log\epsilon)$ for large N.

First and most trivially, empty cells containing zero vorticity should be ignored in forming moments, and the powers of vertices required to form and transform the moments should be precomputed and stored.

Somewhat less trivially, we observe that the far-field becomes smoother at longer distances. Thus more distant cells need contribute to fewer terms in Laurent or Taylor series. If we need p_0 terms for the nearest far-field cells to get error ϵ, then a cell at distance $(2ih, 2jh)$ from the evaluation cell need only contribute to p coefficients where

$$p = p_0 \frac{\log((2r+1)^2 h^2/2)}{\log(((2i-1)^2 + (2j-1)^2)h^2/2)}. \tag{5.28}$$

This refinement usually speeds up large computations by a factor of 2.

Another refinement concerns the preliminary subdivision of triangles to make them lie precisely in cells. Clearly we want to cut as few triangles as possible, since the cost of the local part increases with the number of triangles. Also, it is not necessary to have triangles *completely* contained in cells if they are *nearly* contained. Thus we specify a distance q by which a triangle may extend outside a cell boundary, so a triangle must go $2qh$ outside to be cut. Typical cut triangulations for various values of q are shown in Figure 12. The error bounds will be affected by q since the far-field can come nearer, but in practice even such large values as $q = 0.32$ produce little or no change in the error. This is because most triangles are far away, where q is irrelevant. The CPU time, however, can be drastically reduced by taking q large, because many fewer local interactions need be computed. For example, the number of triangles is cut in half by taking $q = 0.32$ instead of $q = 0.02$, with no increase in the error. This leads to a factor of two speedup in the local interactions.

The algorithm requires a choice of cell size, and its speed depends on the choice. Such a parameter is difficult to estimate a priori; cells too small require too many subdivisions, and too many Laurent-Taylor conversions, while cells too large require too much local work. The real remedy for this is adaptivity, as used in our Delaunay triangulation method or [12, 48], but this complicates the handling of Taylor expansions. We implemented instead a simple method for choosing cell sizes, based on minimizing the CPU time at each step. We keep an increment $i = \pm 1$, and do $n_x \leftarrow n_x + i$ at each step, where n_x is the number of cells in the x-direction. The number of cells in the y-direction is chosen to keep the cells approximately square. The increment i changes sign whenever the CPU time required for the current fast velocity evaluation exceeds the CPU time required for the last one. This choice of parameter keeps us within one cell of a local minimum of CPU time, even if we start the computation with the wrong number of cells. It also adapts automatically to odd-shaped distributions of vorticity.

We also observe that the algorithm can easily be made to run in $O(N \log N \log \epsilon)$ time on quasi-uniform triangulations. To do this, we simply observe that the formula (5.24) which converts Laurent to Taylor coefficients at a cost of $O(N_C^2 p^2)$

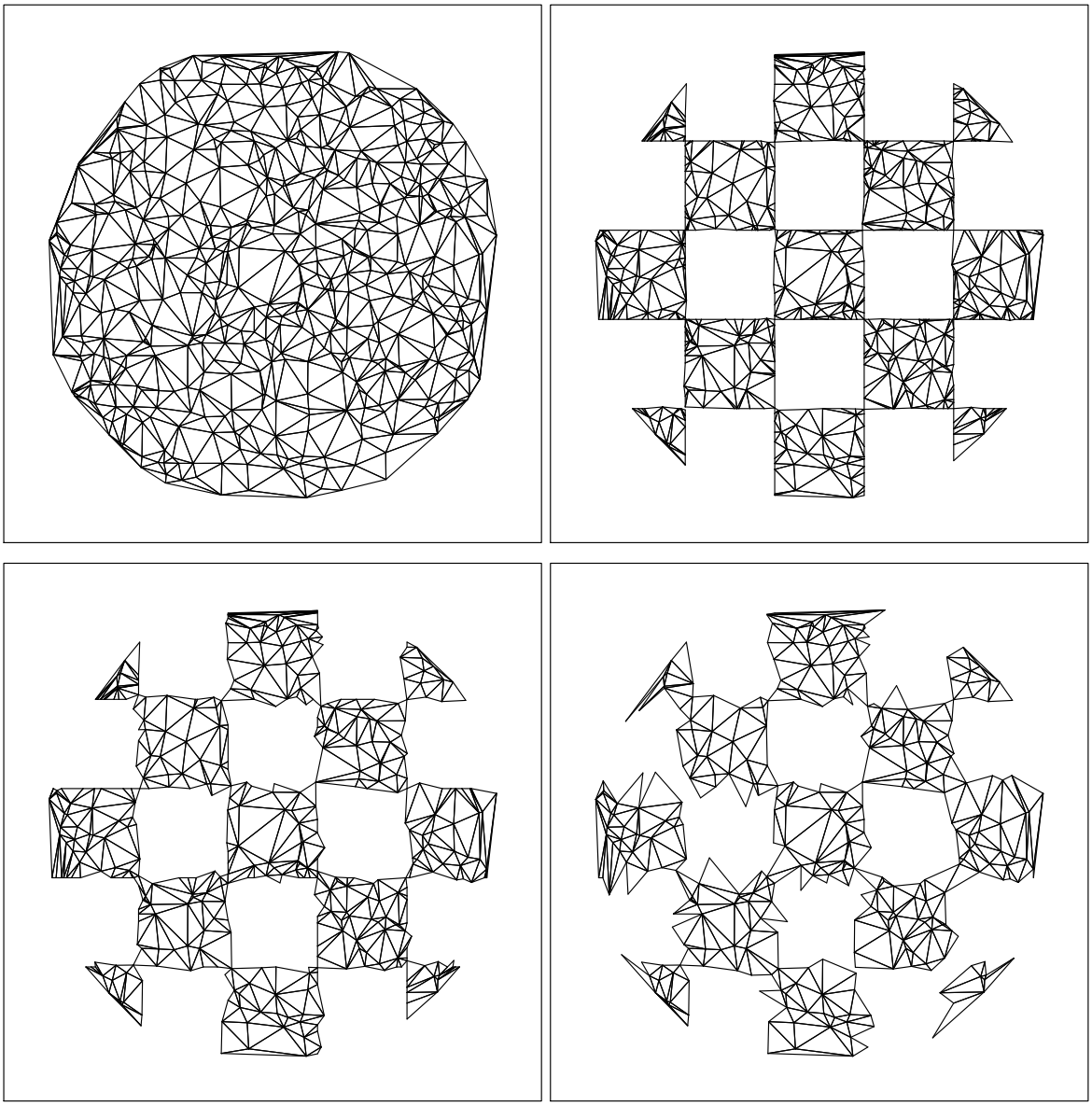


Figure 12: The original and (alternate cells of the) subdivided triangulation with $N = 500$ and $q = 0.02$, 0.08 and 0.32. The subdivided triangulations have 1146, 750 and 507 vertices respectively.

is a *correlation* which can be computed in $O(N_C \log N_C p \log p)$ with the FFT. Then choosing $N_C = O(N)$ gives an $O(N \log N \log \epsilon)$ algorithm. However, we believe the overhead of this approach would be large enough that little speedup would result in practical problems; hence we have not implemented it. It would be important in three-dimensional problems.

Finally, the restriction to quasi-uniform triangulations can be removed by making the algorithm adaptive, exactly as in [12]. In our computations, however, we did not implement an adaptive method because of its complexity.

5.6 Numerical results

We now present numerical results which show that our algorithm achieves considerable speedups over direct evaluation. Table 2 gives the result of fast and direct velocity evaluations for uniformly distributed random vortices in $[-1,1]^2$ with random ω values uniformly distributed on $[-1,1]$. We take $q = 0.2$ and $\epsilon = 10^{-3}$, which requires $p = 10$ with $r = 1$. The other numerical parameters are given in the table along with the times T_d and T_f for direct and fast evaluation and the maximum relative error E_f in fast evaluation. Here N is the number of vertices, N_T the number of triangles, and N_T' is the number of triangles after the subdivision of the triangulation required to put each triangles within q of lying in a single cell.

N	N_T	T_d	T_f	$100T_f/N^{4/3}$	E_f	$\sqrt{N_C}$	N_T'
100	183	2.31	2.36	0.51	0.48×10^{-6}	4	243
200	378	9.63	4.88	0.42	0.72×10^{-6}	6	493
400	773	39.5	10.9	0.37	0.11×10^{-5}	9	1053
800	1566	161	24.3	0.33	0.90×10^{-6}	12	2020
1600	3161	658	55.5	0.30	0.48×10^{-6}	15	3840
3200	6352	2655	122	0.26	0.54×10^{-6}	20	7377
6400	12744	10683	271	0.23	0.77×10^{-6}	27	14405
12800	25529	42732*	627	0.21	0.14×10^{-5}	34	27576
25600	51115	170928*	1453	0.19	0.92×10^{-6}	45	54197
51200	102295	683712*	3431	0.18	0.93×10^{-6}	59	106686

Table 2: Timings for fast and direct velocity evaluation methods with N_T triangles. Asterisks denote timings obtained by extrapolation for the direct method.

We observe that the fast method breaks even for about $N = 100$ and achieves a speedup of about 200 when $N = 51200$. For $N \geq 1000$, we get a tenfold speedup. The fast velocity evaluation is slightly faster than $O(N^{4/3})$ in practice, and the error is much smaller than the error bound.

6 Initial triangulation

We must address one more computational issue, in order to have a robust method: where do we put the vertices initially? Say we are given an initial vorticity field; smooth, discontinuous, or worse. Then we should place the vertices z_j to minimize the error in representing ω by a piecewise linear function on the Delaunay triangulation of the vertices. To do this, we use adaptive refinement of a coarse initial triangulation.

Thus we begin with a uniform square mesh covering the support of ω, and cut each square into a pair of isosceles right triangles. This is our coarse initial triangulation, which we now refine as follows. We put all the triangles on a stack, and sweep through the stack, testing whether each triangle needs to be subdivided. To test a triangle, we first evaluate ω at the node which would be produced by subdividing the triangle. We also evaluate the linear interpolant at the same node, and compute the difference between the two values. If ω is within a tolerance ϵ (relative to the maximum value of ω so far encountered) of the interpolant at the new node, the triangle is accepted. Otherwise, the triangle is subdivided by Mitchell's newest-node bisection method [32], maintaining compatibility by subdividing neighbors as necessary, and the new triangles are stacked. We then repeat the procedure with the next triangle in the stack, until the stack is finished. Using the maximum value of $|\omega|$ so far encountered produces a triangulation on which the error is likely to be smaller than $\epsilon|\omega|_\infty$ rather than larger, though of course any method can be fooled into accepting a substandard triangle with errors which are actually too large.

Mitchell's subdivision procedure begins by assigning one vertex of each triangle in the initial triangulation as a "peak," and the side opposite the peak as the base. (In our case, the initial triangulation consists of isosceles right triangles and the peak is the vertex at the right angle, opposite the hypotenuse.) Then it subdivides triangles by dividing the base and the neighboring triangle opposite the peak, with the new vertex being assigned as the peak of each of the four new triangles created by the subdivision. Compatibility is maintained by always subdividing compatible pairs of triangles; if the neighbor opposite the peak is not compatibly subdivisible, it is itself divided recursively until compatibility is maintained: See Figure 13. Because we begin with isosceles right triangles, the recursion is always finite. The fact that we subdivide triangles compatibly gives a slight safety factor, because even if a triangle is wrongly accepted, it will still be subdivided if one of its neighbors with peak opposite it is subdivided.

7 Numerical results

In this section we present numerical results that show the accuracy, efficiency and robustness of the method. Our results show that the method maintains its accuracy for very long periods of time, on simple and complex test cases. It is flexible and robust, and can compute even discontinuous solutions, with no

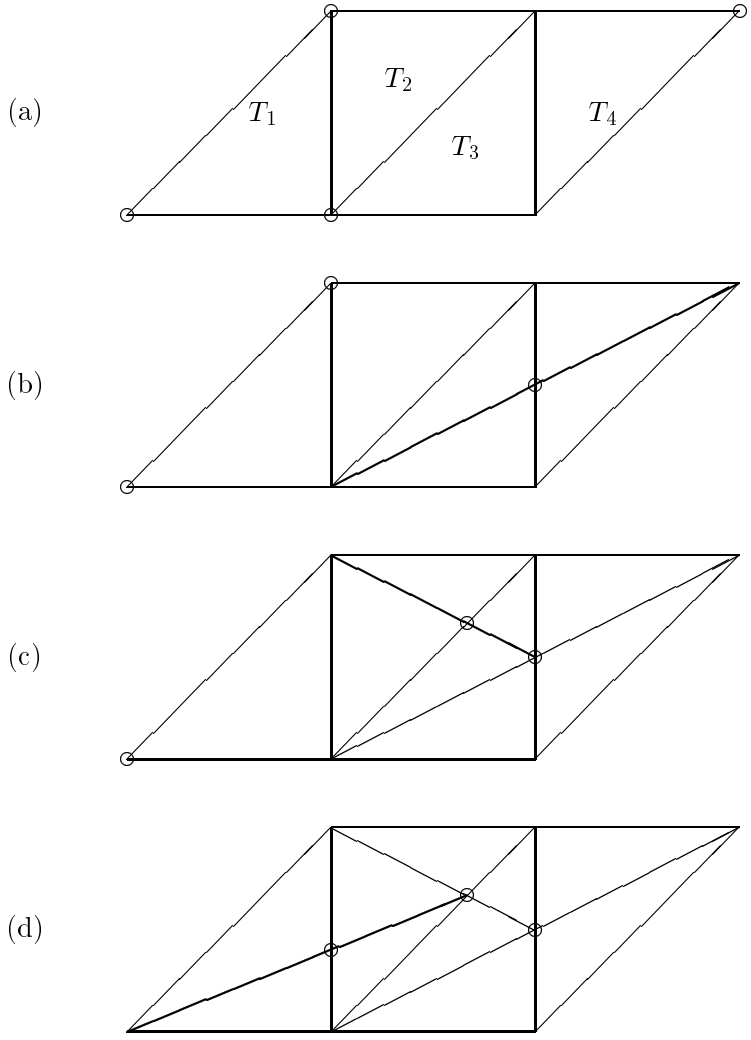


Figure 13: An example of Mitchell's recursive newest-node bisection. Triangle T_1 is flagged for subdivision, but the peak (circle) of its neighbor T_2 does not lie opposite T_1. Hence we must refine T_2 and its neighbor T_3. Similarly, the peak of T_3 does not lie opposite T_2, so we must refine T_4 and T_3. The peak of T_4 lies opposite T_3, so the recursion stops here. We then divide triangles backwards in pairs as shown in (b) through (d), until we have divided T_1. The final triangulation is shown in (d).

numerical parameters except the resolution and time step, and little numerical diffusion.

First, we discuss the norms and conserved quantities we plan to measure. There are six reasonable quantities to measure, the L^1, L^2 and L^∞ errors in the velocity and the vorticity, and for smooth solutions the results are essentially independent of the choice of norm. Since we present numerical results with nonsmooth as well as smooth vorticity fields, we prefer the errors in velocity, a smoother quantity and a primitive variable with direct physical meaning. The error in velocity is appropriate for comparing our method to other methods, since the representation of the vorticity in other methods is quite different. The L^∞ norm seems more appropriate than L^1 or L^2, since the velocity field of a single vortex blob is not in L^1 or L^2, so these norms depend on the support of the grid, and not only on h, even if the vorticity has compact support. Thus our main measure of error is the relative L^∞ norm of the velocity error

$$E_u \equiv \frac{\max_z |u(z,t) - u_h(z,t)|}{\max_z |u(z,t)|}$$

where u is the exact and u_h the computed velocity field. The maximum over z is approximated by the larger of the maximum over the vertices and the maximum error in the linear interpolant of u_h at one random point per triangle.

There are also two useful conserved quantities which we check, the circulation

$$\Gamma := \int_{I\!R^2} \omega(z,t)\, dz \tag{7.1}$$

and the second moment of the vorticity

$$M_2 := \int_{I\!R^2} |z|^2 \omega(z,t)\, dz. \tag{7.2}$$

Conservation of circulation follows from conservation of vorticity along the streamlines;

$$\int_{I\!R^2} \omega(z,t)\, dz = \int_{I\!R^2} \omega(z(\zeta,t),t)\, d\zeta = \int_{I\!R^2} \omega_0(\zeta)\, d\zeta.$$

The Jacobian $|\partial z/\partial \zeta| = 1$ because the flow is incompressible. Conservation of the second moment is proved for example on page 528 of [3]. The second moment is a good measure of numerical diffusion, because physical diffusion makes the second moment nonconstant in time.

7.1 Comparison with vortex-blob methods

In a vortex-blob method the velocity field is computed using (2.11). The convergence properties of such a method depend on the blob function. Several convergence results for vortex-blob methods are given in [1]. Here we briefly recall some of the main results. Let $u(z,t)$ be the exact velocity field at position z and

time t, and $u_{h,\delta}(z,t)$ the velocity field produced by a vortex-blob method of grid size h and blob size δ. The discrete L^2 error, or "consistency error" is defined by

$$e_2(t) = \left\{ \sum_i |u(z_i,t) - u_{h,\delta}(z_i,t)|^2 h^2 \right\}^{1/2},$$

where $z_i(t)$ is the exact position of the i-th particle of fluid at time t. It can be shown that, for a finite time interval T and for smooth initial conditions, the following estimate holds:

$$\max_{0\le t\le T} e_2(t) \le C\left(\delta^p + \left(\frac{h}{\delta}\right)^L \delta\right).$$

Here the constant C depends on the initial condition and on T, and the constants p and L are determined by the blob function. For Gaussian blob functions, $L=\infty$, so with $\delta = h^q$, the error estimate becomes

$$\max_{0\le t\le T} e_2(t) \le Ch^{pq},$$

where p is the order of the blob function. In theory it is possible to obtain an arbitrarily high order of convergence by choosing p large and q close to one. Experience shows, however, that for p large and q close to one there is a considerable loss of accuracy after a short time. We used the Gaussian blob function of order $p=4$ [34]

$$g(r) = \frac{1}{\pi}(2e^{-r^2} - \frac{1}{2}e^{-r^2/2}),$$

where $r=|z|$.

As a test problem we consider Perlman's test case [34] with vorticity

$$\omega(z) = \begin{cases} (1-|z|^2)^7 & |z|\le 1 \\ 0 & |z|>1 \end{cases}. \tag{7.3}$$

The corresponding velocity field is given by

$$u(z) = f(|z|)\begin{pmatrix} y \\ -x \end{pmatrix}, \tag{7.4}$$

where

$$f(r) = \begin{cases} -\dfrac{1}{16r^2}(1-(1-r^2)^8) & r\le 1 \\ -\dfrac{1}{16r^2} & r>1 \end{cases}.$$

The flow is radially symmetric and rotates about the origin. The particles near the origin complete one rotation at time $t=4\pi$, while the particles on $|z|=1$ complete one rotation at $t=32\pi$.

At time $t = 0$ we place the particles on a regular square grid of size $h = 0.4$, 0.2, 0.1 or 0.05, inside a circle of radius $R = 1.2$, and set $\omega_i = \omega(z_i)$ where ω is defined by (7.3). The system of ODEs

$$\dot{z}_i = u^\delta(z_i, t)$$

is integrated up to $t = 32\pi$ by a Runge-Kutta method of order 4, with time step $\Delta t = 0.05$. The results reported in Figure 14 show that large q gives a higher order of convergence for short times, but for smaller q the accuracy is maintained for longer times.

We repeated the calculations with our algorithm, using the same initial conditions, and solving the system of ODEs by a Runge-Kutta method of order 4, with $n = 64$, 96, 128 and 192 time steps up to $T = 32\pi$. Figure 15 plots the relative L^∞ error in velocity and the moment errors versus time. It is evident that our method is more accurate than the vortex-blob method for a given number of degrees of freedom. The errors are not smooth functions of time, because of the reconnection, but remain uniformly small.

The computation of the velocity field with a fast multipole-based vortex-blob method is slightly faster than with our method; Figure 15 also plots the time-averaged errors against the total CPU time. However, our method achieves better accuracy, so the average error decreases faster as CPU time increases. Note, however, that we are comparing very simple versions of these algorithms; little can be inferred about the relative performance of possible production versions of these codes.

Vortex-blob methods preserve some conservation properties of the Euler equations. In a vortex-blob method, the discrete analogues of Γ and M_2 are

$$\Gamma^{(N)} := \sum_{i=1}^{N} \omega_i h^2 \tag{7.5}$$

$$M_2^{(N)} := \sum_{i=1}^{N} \omega_i |z_i|^2 h^2 \tag{7.6}$$

and $\Gamma^{(N)}$ is obviously conserved. It is easy to show that $M_2^{(N)}$ is also conserved [40]. Conservation of the second moment is important, since the rate of change of the second moment is related to the numerical diffusion of the method. This is one reason why vortex-blob methods are attractive for inviscid flow. They have been used for slightly viscous flow in conjunction with random and deterministic methods for the treatment of vorticity diffusion [15, 18, 40].

Our method does not preserve the circulation and the second moment. Figure 15 plots the relative errors in the circulation and second moment; with $h = 0.05$ the second moment errors are less than 0.4% up to $t = 32\pi$. In view of the extension of the method to the Navier-Stokes equations, we use this to estimate the minimum viscosity that is possible to treat with such a method for a given grid parameter h. For the Navier-Stokes equations

$$\omega_t + u \cdot \nabla \omega = \nu \Delta \omega$$

the second moment of Perlman's test case evolves according to

$$M_2(t) = M_2(0) + 4\nu t.$$

Since $M_2(0) = \pi/72$,

$$m_2 \equiv \frac{M_2(t) - M_2(0)}{M_2(0)} = \frac{288\nu t}{\pi}.$$

At $t = 32\pi$, $m_2 = 9216\nu$. An error of 0.4% corresponds to $\nu = 8 \times 10^{-7}$, a fairly small viscosity. This suggests that our method can be combined with the method of [40] to solve the Navier-Stokes equations for small Reynolds number flows. Note, however, that we do not observe a linear growth of error with time. Rather, the second moment is roughly constant, suggesting that our method may be even less diffusive than this simple estimate would imply.

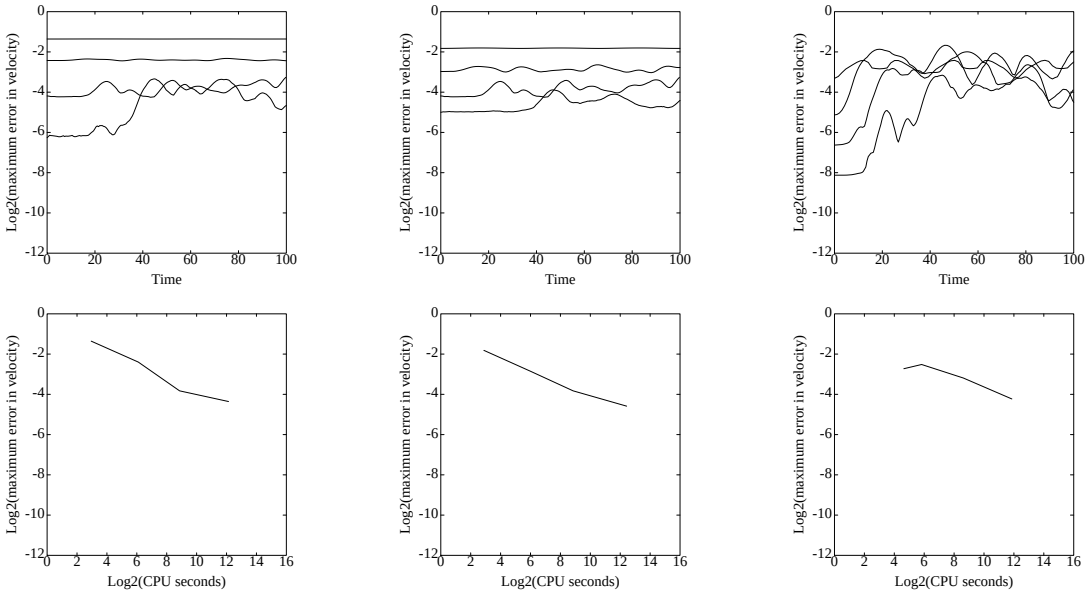


Figure 14: Errors versus time for the vortex blob method with $h = 0.4$, 0.2, 0.1 and 0.05 and $n = 64$, 96, 128 and 192 time steps. Top row: Relative L^∞ errors in velocity versus time. Second row: Time-averaged relative L^∞ errors in velocity versus SPARC-2 CPU time in seconds. Left column: Supergaussian blob, $q = 0.7$. Center: Supergaussian blob, variable $q = 0.9$, 0.8, 0.7 and 0.6. Right: Finite-core blob, $q = 0.4$.

7.2 Reconnection versus fixed topology

We next reproduce the results presented in [13] and compare them with those produced with our method, for Perlman's test case (7.3). The initial triangulation

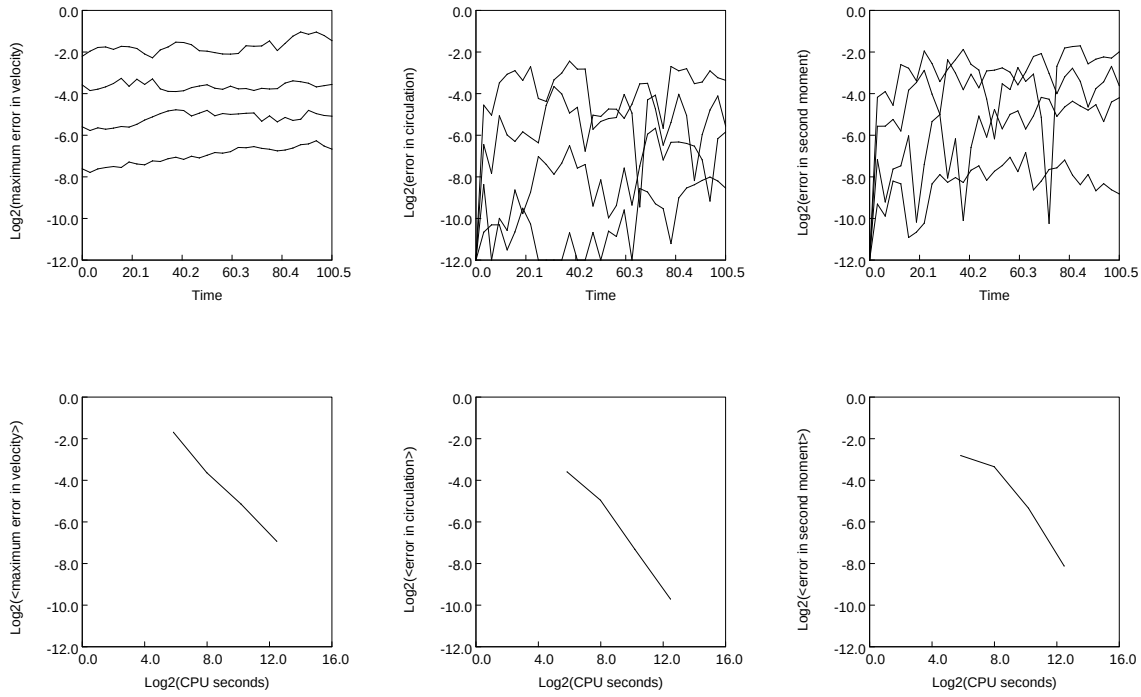


Figure 15: Comparison with vortex-blob methods; relative L^∞ velocity, circulation and moment errors (left to right) versus time (first row) and the corresponding time-averaged errors versus SPARC-2 CPU time (second row) for our method. The four runs plotted used the following parameters: Mesh sizes $h = 0.4$, 0.2, 0.1 and 0.05. $N = 29$, 113, 441 and 1793 vertices. $n = 64$, 96, 128 and 192 time steps.

was produced by the MODULEF library of finite element codes [10], with $h = 1/6, 1/8, 1/12, 1/16$ and $1/24$. The system of ODEs is integrated by a Runge-Kutta method of order 2, with $\Delta t = \pi/8$. For each value of h we make two runs. In the first we keep the topology of the triangulation unchanged and evaluate the velocity directly, as in [13]. In the second run we construct a Delaunay triangulation at each time step, using the adaptive cell technique of Section 4, and use the fast velocity evaluation of Section 5. Figure 16 compares the triangulations in the two cases, for $h = 1/12$ and $t = 0, 2\pi, 4\pi$, and also shows the Delaunay triangulation for later times $t = 8\pi, 16\pi, 32\pi$. Clearly without reconnection the grid becomes very distorted and degenerates after a certain time, when triangles with negative area form, while the Delaunay triangulation remains regular for long times. Figure 17 shows the velocity and moment errors as a function of time up to $T = 8\pi$. The distortion of the fixed grid causes a dramatic increase in the error, while with a Delaunay triangulation the error remains small. Note that thin triangles occur even with a Delaunay triangulation, but that the error remains small nonetheless. This is different from finite element methods, for example, where such thin triangles would produce disaster. Here, we use the triangulation only to evaluate the Biot-Savart integral, we need not solve a linear system. The time-averaged errors versus CPU times are reported in Figure 18. It is clear that the fast velocity evaluation method is essential for attaining small errors in reasonable computation times.

7.3 The adaptive method

We now test our method on Perlman's test case, without the handicap of a uniform initial grid. We first fix a grid refinement tolerance $\epsilon = 0.064$ and halve the time step until the first two digits of the errors do not change for $0 \leq t \leq 32\pi$, using fourth-order Runge-Kutta. This gives us a time step $\Delta t = \pi/2$ which makes time discretization errors negligible in comparison with spatial discretization errors for this ϵ. Then we run three more cases, with $\epsilon = 0.016$, 0.004 and 0.001, reducing Δt each time. Figure 19 displays the resulting triangulations at $t = 0$, $t = 4\pi$ and $t = 32\pi$. Figure 20 plots the errors against time and CPU time; they remain uniformly small over time and decrease very rapidly with increasing computational effort.

7.4 Kirchhoff's elliptical vortex

We now turn to a more challenging test case, a patch of piecewise constant vorticity. An exact circular patch of constant vorticity is easy to construct, but shares with Perlman's test case an unrealistic radial symmetry. We use a more interesting test case, the Kirchhoff elliptical vortex, a uniformly rotating *elliptical* patch of constant vorticity with exact velocity field given in Appendix A. The Kirchhoff vortex is of considerable physical interest [7] as well as numerically useful.

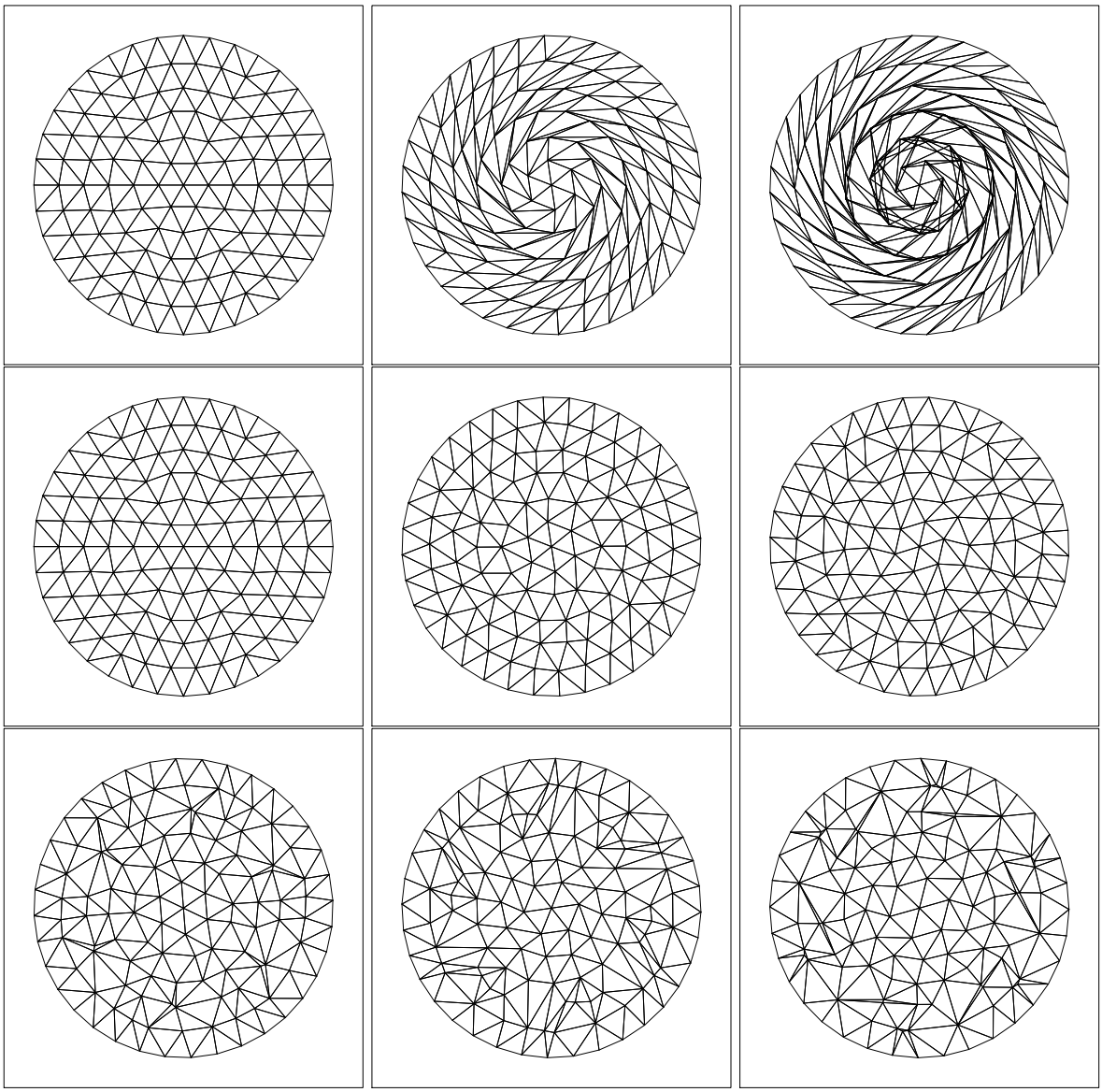


Figure 16: Fixed (top row) and Delaunay (second row) triangulations at times $t = 0, 2\pi$ and 4π (left to right), and Delaunay triangulations at later times $t = 8\pi$, 16π and 32π (last row).

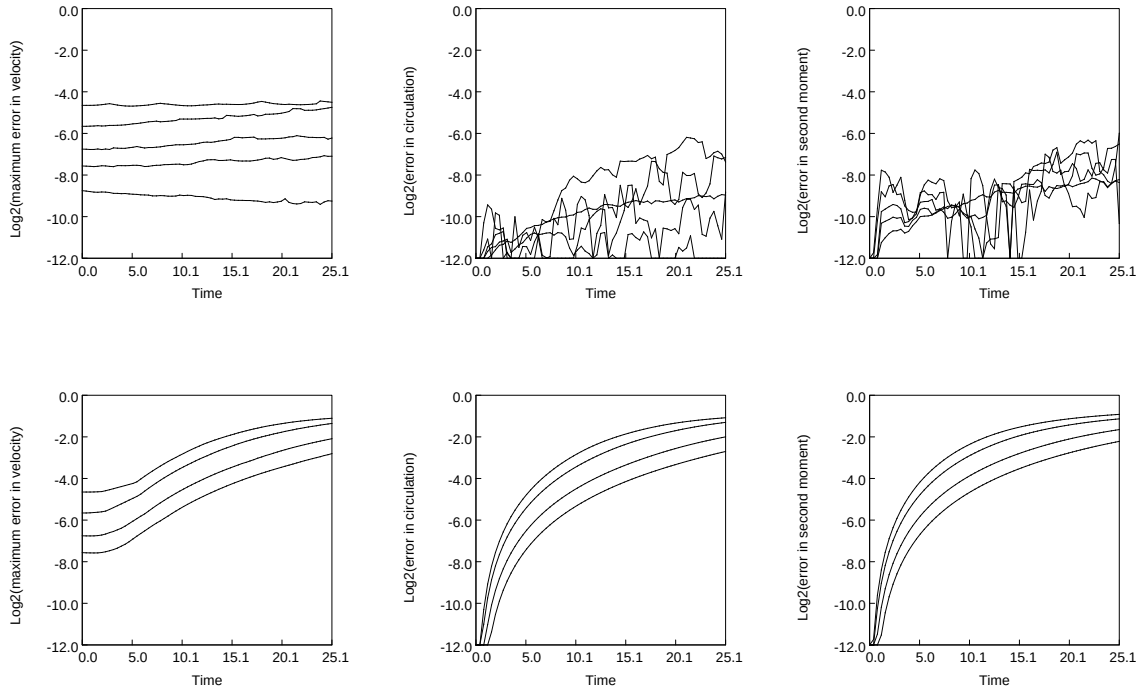


Figure 17: Comparison between fixed topology and Delaunay triangulation with $h = 1/6, 1/8, 1/12, 1/16, 1/24$. First row: Relative L^∞ error in velocity, circulation errors and second moment errors versus time, for Delaunay triangulation. Second row: the same quantities for fixed topology.

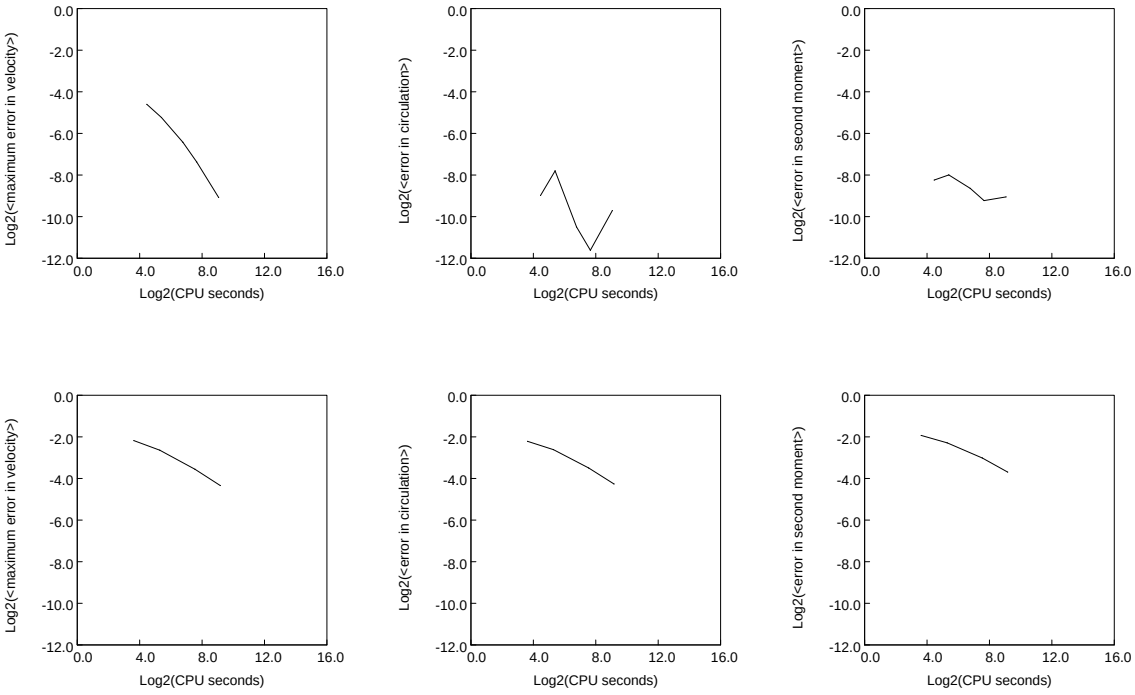


Figure 18: Comparison between fixed topology and Delaunay triangulation with $h = 1/6, 1/8, 1/12, 1/16, 1/24$. First row: Time-averaged relative L^∞ error in velocity, circulation errors and second moment errors versus Cray-2 CPU time, for Delaunay triangulation. Second row: the same quantities for fixed topology.

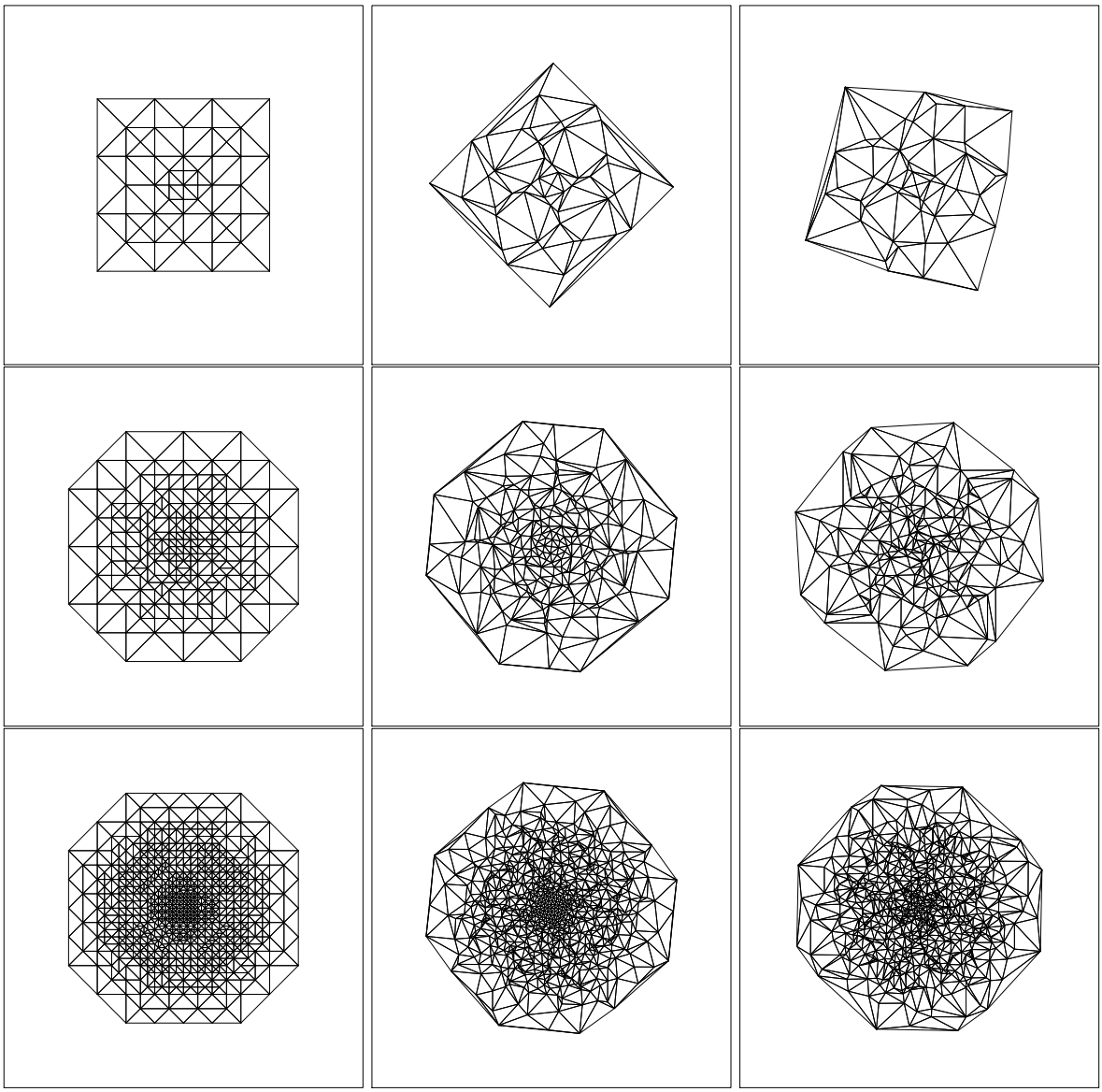


Figure 19: Adaptive triangulation of Perlman's test case at times $t = 0$, 4π and 32π (left to right), with $\epsilon = 0.064$ (top row), 0.016 (second row), and 0.004 (last row).

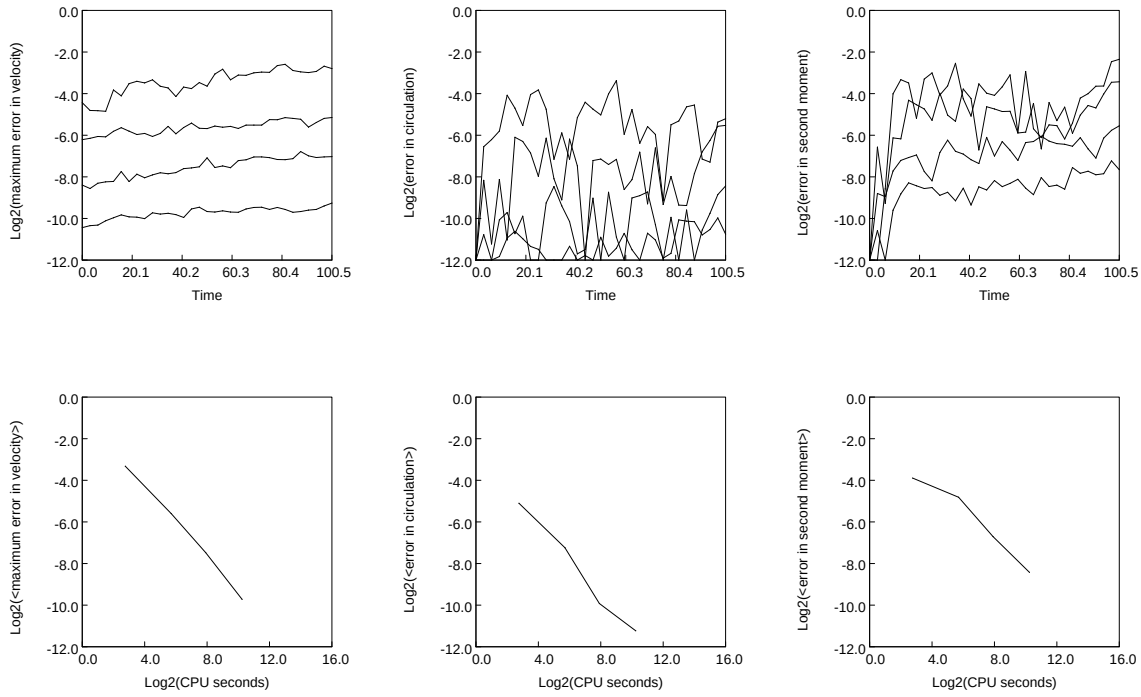


Figure 20: Errors for Perlman's test case, with an adaptive grid reconnected at every step. First row: Relative L^∞ error in velocity, circulation errors and second moment errors versus time. Second row: Time-averaged quantities versus Cray-2 CPU time. The four runs plotted used the following parameters: Mesh tolerances $\epsilon = 0.064$, 0.016, 0.004 and 0.001. $N = 50$, 205, 725 and 2709 vertices. $n = 64$, 96, 128 and 192 time steps.

Resolving the elliptical vortex with an adaptive grid requires that we vary the number L of refinement levels allowed together with the tolerance ϵ. We take $\epsilon = 0.064$, 0.016, 0.004 and 0.001, with $L = 8$, 10, 12 and 14 levels of refinement of an initial uniform grid with $h = 0.48$, and $n = 32$, 48, 64 and 96 time steps, using fourth-order Runge-Kutta.

Figure 21 shows the vorticity field at $t = 0$, $T/4$ and $T/2$, where $T = 9\pi = 28.274334$ is the rotational period of the Kirchhoff ellipse with aspect ratio 2, and strength 1. The vorticity field is plotted by giving each triangle a gray-scale value equal to $\bar{\omega}/|\omega|_\infty$, where $\bar{\omega}$ is the average over the triangle, 0 is lightest and 1 is darkest. In the more accurate calculations, the ellipse returns very closely to its original position after one period. Note that the fluid inside the ellipse rotates as a rigid body (since ω is constant there); the fluid outside undergoes a more complicated deformation.

Figure 22 plots the L^1 and L^∞ errors in velocity and vorticity and the moment errors against time. Clearly the L^∞ error in vorticity is $O(1)$, as one would expect, while the L^∞ error in velocity is uniformly small.

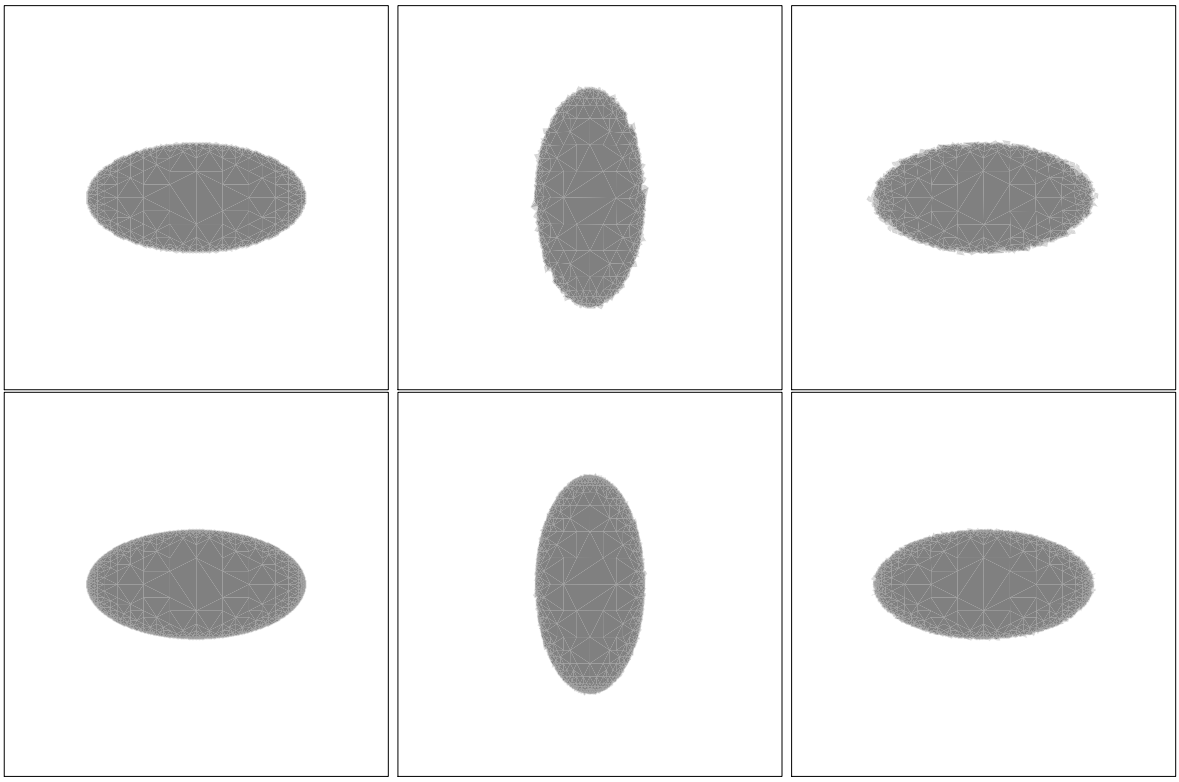


Figure 21: Grayscale plots of the Kirchhoff elliptical vortex at times $t = 0$, $T/4$ and $T/2$ (left to right), with $\epsilon = 0.016$ (top row) and 0.004 (last row).

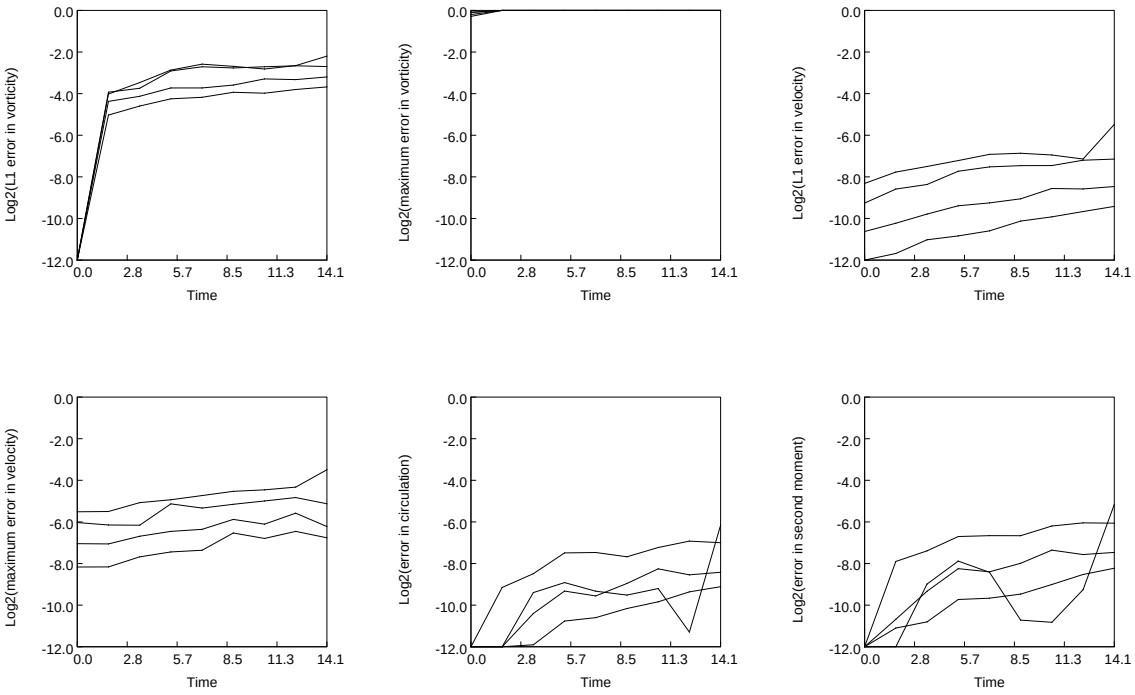


Figure 22: Errors for the Kirchhoff elliptical vortex. First row: Relative L^1 and L^∞ errors in vorticity and L^1 error in velocity versus time. Second row: Relative L^∞ error in velocity, circulation and second moment errors versus time. The four runs plotted used the following parameters: $L = 8$, 10, 12 and 14 levels of refinement. $N = 468$, 1064, 2320 and 4700 vertices. $n = 32$, 48, 64 and 96 time steps.

7.5 Interacting vortex patches

Our final numerical examples are flows composed of several interacting vortex patches. Since exact solutions are unavailable, we estimate the errors in vorticity and velocity by differencing. We evaluate the vorticity and velocity fields, stored on the triangulation, by linear interpolation to fixed uniform grid, then difference successive calculations. This gives error estimates which agree well with the exact errors when the latter are available.

Our first test case without an exact solution uses three randomly placed patches, each a scaled version of Perlman's test case. The triangulation is plotted in Figure 23, for $\epsilon = 0.016$ and 0.004, at times $t = 0$, 25 and 200. The errors estimated by differencing are plotted in Figure 24, using $\epsilon = 0.064$, 0.016 and 0.004 and fourth-order Runge-Kutta with $n = 96$, 128 and 192 time steps up to $t = 200$.

Our second test case is the interaction of circular patches of constant vorticity, as studied in [11, 51, 49] by specialized methods. We do not expect great accuracy from our general-purpose code; we are pushing the limits of adaptivity. Figure 25 shows the vorticity computed with $\epsilon = 0.004$ and 0.001, at times $t = 0$, 10 and 40. The errors estimated by differencing are plotted in Figure 26, using $\epsilon = 0.064$, 0.016, 0.004 and 0.001 with fourth-order Runge-Kutta with $n = 64$, 96, 128 and 192 time steps up to $t = 40$.

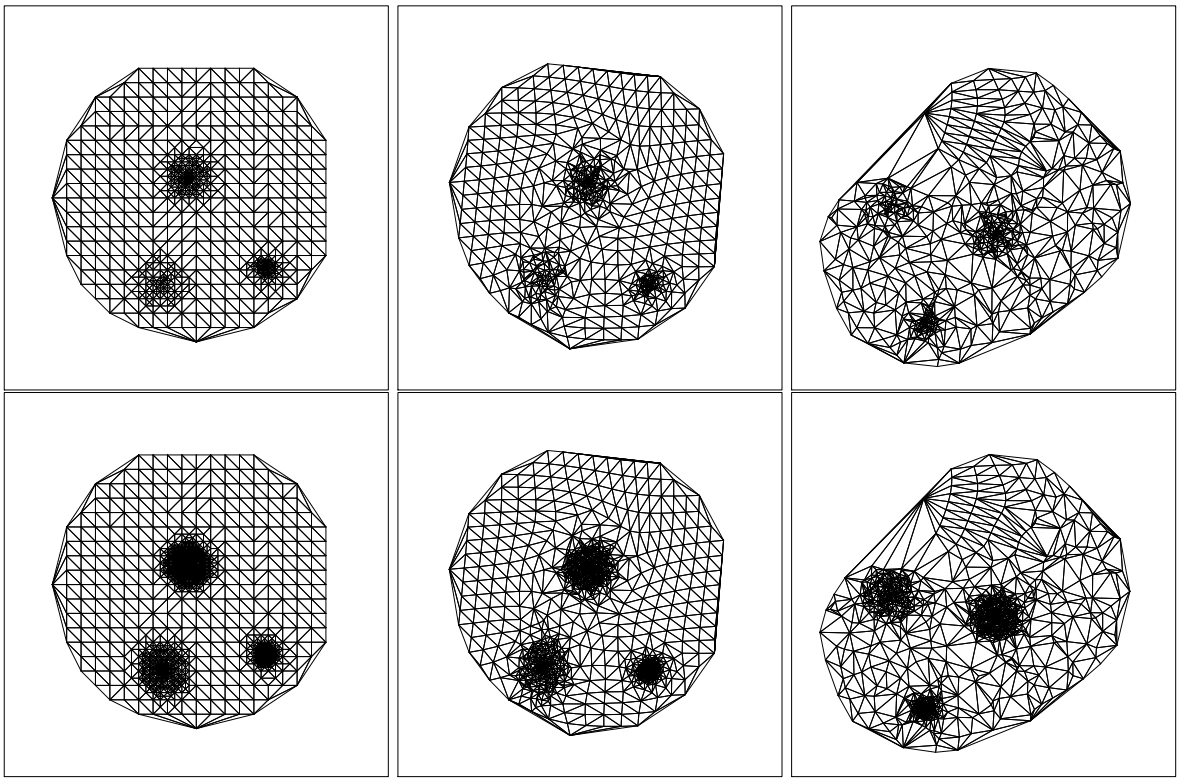


Figure 23: Three interacting patches of smooth vorticity, at times $t = 0$, 25 and 200 (left to right), with $\epsilon = 0.016$ (top row) and 0.004 (second row).

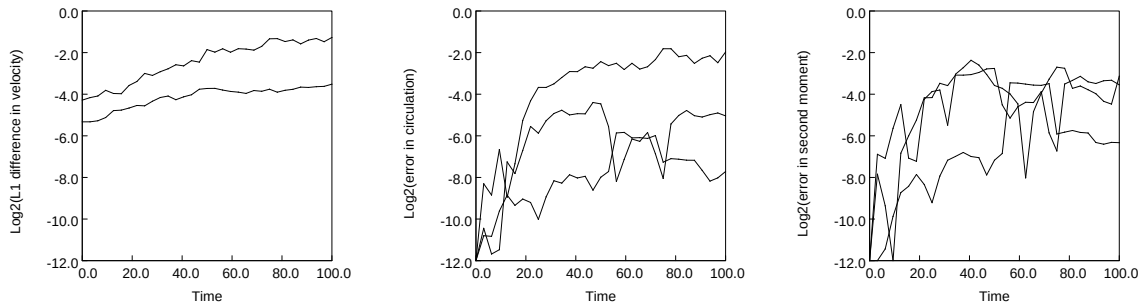


Figure 24: Errors for smooth patches of vorticity, estimated by differencing. Relative L^1 error in velocity, circulation errors and second moment errors vs time.

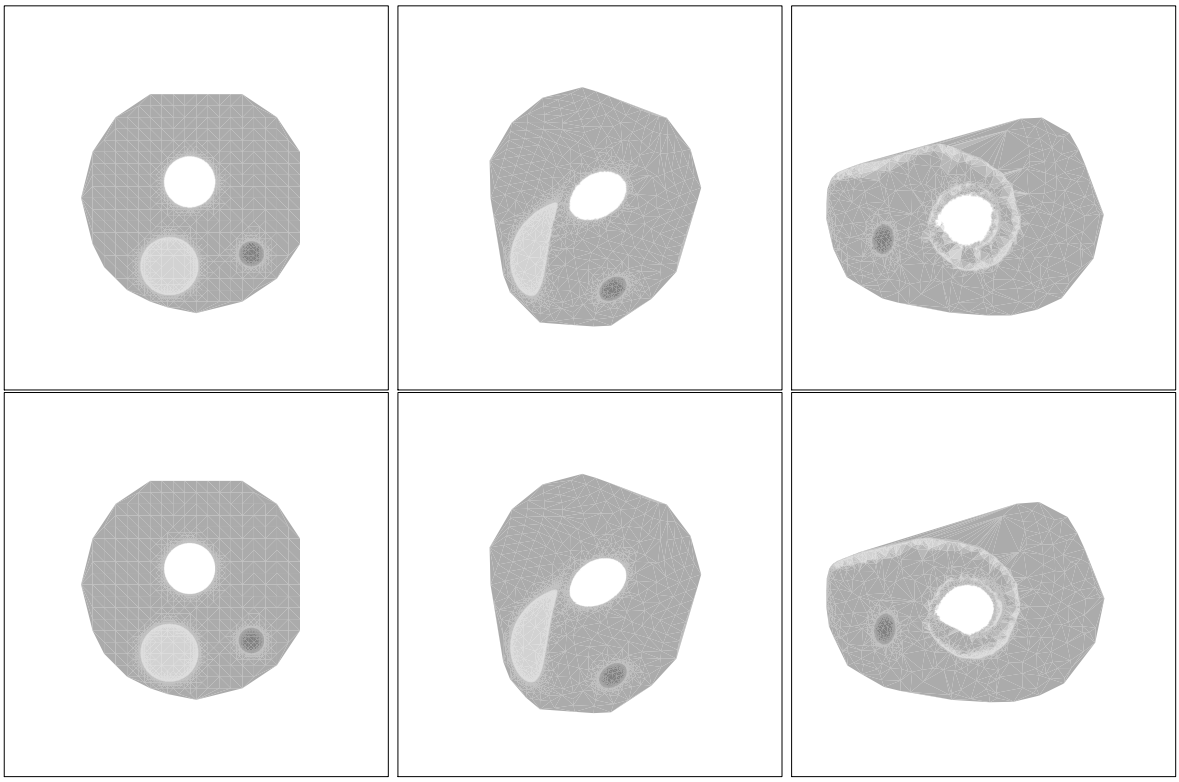


Figure 25: Three interacting patches of constant vorticity, at times $t = 0$, 10 and 40 (left to right), with $\epsilon = 0.004$ (top row) and 0.001 (bottom row).

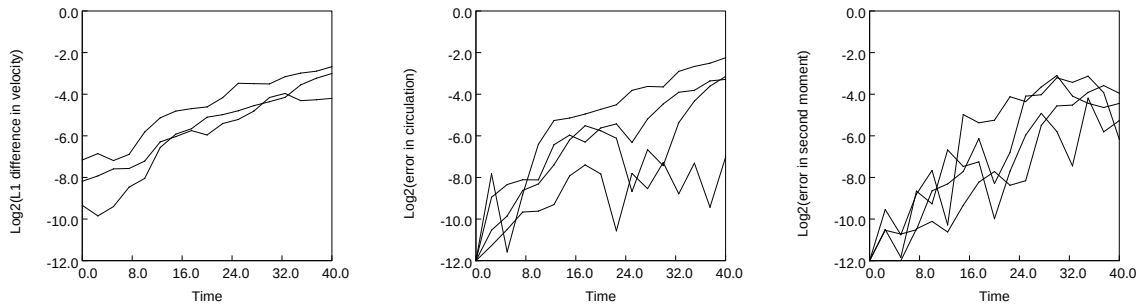


Figure 26: Errors for constant patches of vorticity, estimated by differencing. Relative L^1 error in velocity, circulation errors and second moment errors versus time.

8 Generalizations

Our method can be extended to model more general flows. In this section we consider the following generalizations:

- Boundary conditions for the 2-D Euler equations.
- The Navier-Stokes equations in $I\!R^2$.
- Boundary conditions for the 2-D Navier-Stokes equations and vorticity generation at the boundaries.
- Higher order methods.
- Euler and Navier-Stokes equations in $I\!R^3$.

We have not implemented these generalizations; this is work in progress.

8.1 Boundary conditions for the 2-D Euler equations

Let Ω be the domain containing the flow, $\partial\Omega$ its boundary, and ν the outward unit normal (see Figure 27).

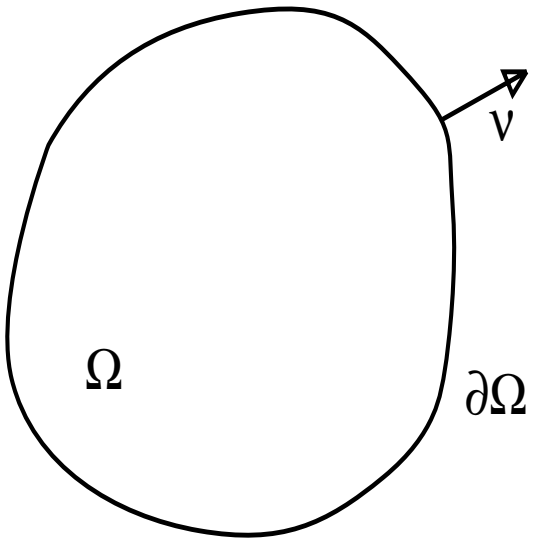


Figure 27: A domain Ω and its boundary $\partial\Omega$.

The no-flow boundary condition reads

$$u \cdot \nu = 0 \qquad \text{on } \partial\Omega. \tag{8.1}$$

In the vorticity formulation, this condition must be translated from the velocity to the vorticity. This can be obtained in the following way. From equation (2.5)

it follows that the tangential derivative of ψ along the boundary vanishes:

$$\frac{\partial \psi}{\partial \tau} = 0 \qquad \text{on } \partial\Omega,$$

where τ denotes the unit vector tangent to $\partial\Omega$. This means that ψ is constant along the boundary and, since the stream function is determined only up to a constant anyway, we can set it to be zero. The Poisson equation for ψ is therefore

$$\begin{cases} -\Delta\psi &= \omega & \text{in } \Omega \\ \psi &= 0 & \text{on } \partial\Omega \end{cases} . \tag{8.2}$$

This is a standard problem and there are many ways to solve it numerically. An attractive method in this setting is to represent ψ as the volume potential

$$V\omega(x) = \int_\Omega G(x-y)\omega(y)dy$$

of ω, with $G(x) = (2\pi)^{-1}\log|x|$ the free-space Green function of $-\Delta$, plus the solution ψ_∂ of

$$\begin{cases} -\Delta\psi_\partial &= 0 & \text{in } \Omega \\ \psi_\partial &= -V\omega & \text{on } \partial\Omega \end{cases} . \tag{8.3}$$

The velocity due to the volume potential is precisely what we evaluated in Section 5, using piecewise linear vorticity, while $\nabla\psi_\partial$ can be found by solving a second-kind Fredholm integral equation on $\partial\Omega$ [20]. The integral equation can be solved very efficiently by iteration and the fast algorithms of [38, 12, 45]. These approaches are particularly attractive if Ω is moving as time passes.

Alternatively, one could use a standard finite element Galerkin method with piecewise linear elements to solve (8.2), then calculate u by numerical differentiation.

If Ω is convex and simply connected, then the algorithm described in Section 4 can be used to construct the triangulation. For more general non-convex or non-simply-connected domains, that algorithm can easily be modified to remove triangles outside Ω.

8.2 The Navier-Stokes equations

We consider the 2-D Navier-Stokes equations in $\mathbf{R}^2$. The equation for the vorticity is

$$\frac{\partial\omega}{\partial t} + (u\cdot\nabla)\omega = \mu\Delta\omega,$$

where μ is the kinematic viscosity. The velocity can be reconstructed from the vorticity using the Biot-Savart law (2.6). We can solve the Navier-Stokes equations numerically by a fractional step method. In the convection step we move

the vertices as above, and in the diffusion step we solve the diffusion equation for ω. Let $\omega^n(z)$ denote the piecewise linear vorticity distribution at time t_n. After the convection step for a time step Δt, we have a new vorticity distribution $\tilde{\omega}^n(z)$, piecewise linear on a new triangulation. During the diffusion step we have to solve the equation

$$\begin{cases} \dfrac{\partial \omega}{\partial t} &= \mu \Delta \omega \\ \omega(z, t_n) &= \tilde{\omega}^n(z) \end{cases} \tag{8.4}$$

One way to compute $\omega^{n+1}(z)$ is to solve (8.4) exactly using the gaussian kernel $\Gamma(z,t) = (1/4\pi t)\exp(-|z|^2/4t)$. This would give

$$\bar{\omega}^n(z) = \int_{\mathbf{R}^2} \Gamma(z - \tilde{z}, \mu\Delta t)\tilde{\omega}^n(\tilde{z})\, d\tilde{z}. \tag{8.5}$$

Then the piecewise linear function $\omega^{n+1}(z)$ is obtained by projecting $\bar{\omega}^n$ into the space of piecewise linear functions with the same values at the nodes:

$$\omega^{n+1}(z_i) = \bar{\omega}^n(z_i), \qquad i = 1, \ldots, N.$$

This procedure, however, is not very accurate, because the projection onto piecewise linear functions introduces high frequency components in the vorticity distribution, producing a spurious diffusion. A simple 1-D calculation shows that the local truncation error is $O(h^2\sqrt{\mu\Delta t})$. This makes the method inaccurate for small time steps. The Green's function approach requires the computation of the integral (8.5). A naive implementation of this integral would give a computational complexity $O(N^2)$. Fast algorithms have been constructed for this case; the complexity is reduced to $O(N)$ [46]. One advantage of this approach is that it does not suffer from any stability restriction, and therefore it can be used with arbitrarily large values of $\mu\Delta t$. In view of these considerations, this approach seems interesting in the presence of large viscosities. For small values of the viscosity, alternative approaches can be considered. We propose here two possibilities, one based on the discretization of the Laplacian on a Voronoi mesh, the other obtained by collocation.

The discrete Laplacian is defined in the following way. For any simply connected bounded domain $\mathcal{P} \subset \mathbf{R}^2$ with regular boundary $\partial\mathcal{P}$, it is

$$\int_{\mathcal{P}} \Delta\phi \, dz = \int_{\partial\mathcal{P}} \frac{\partial \phi}{\partial \nu}\, ds$$

Discretizing this relation on a Voronoi polygon $\mathcal{P}_i$ (see Figure 4), one defines the discrete Laplacian B by

$$B\phi(z_i) = \frac{1}{A_i} \sum_{j \neq i} \frac{\phi(z_j) - \phi(z_i)}{|z_j - z_i|} l_{ij}$$

where A_i is the area of the Voronoi polygon corresponding to z_i, and l_{ij} is the length of the edge corresponding to points z_i and z_j.

The diffusion equation (8.4) for ω becomes

$$\left\{ \begin{array}{rcll} \dfrac{d\omega_i}{dt} & = & \displaystyle\sum_{j\neq i} \beta_{ij}\omega_j & \\ \omega_i(t_n) & = & \tilde{\omega}^n(z_i) & \end{array} \right. \qquad i = 1, \ldots, N \tag{8.6}$$

with

$$\beta_{ij} = \left\{ \begin{array}{ll} \dfrac{1}{A_i}\dfrac{l_{ij}}{|z_j - z_i|} & j \neq i \\ -\dfrac{1}{A_i}\displaystyle\sum_{k\neq i} \dfrac{l_{ik}}{|z_k - z_i|} & j = i \end{array} \right.$$

The Voronoi diagram and the Delaunay triangulation are dual structures. It is easy to obtain one, once the other is known.

The discrete Laplacian has been used for the solution of the Navier-Stokes equation in conjunction with the vortex-blob method for the computation of the velocity [40]. In that case the primary variables were the circulation associated to each vortex. It was possible to prove several conservation properties for the diffusion equation discretized on a Voronoi mesh. In our case, the vorticity distribution is a piecewise linear function on the Delaunay triangulation and such conservation properties no longer hold. It would be worthwhile to explore the properties of the discretization of the heat equation on a Voronoi mesh.

An alternative approach, which is more consistent with our framework, consists in a collocation-Galerkin method. Multiplying both sides of (8.4) by a test function $\phi(z)$ with compact support and integrating, we obtain

$$\frac{d}{dt}\int \phi(z)\omega(z,t)\,dz = -\mu \int \nabla\phi(z)\cdot\nabla\omega(z,t)\,dz. \tag{8.7}$$

We associate to a given triangulation a set of piecewise linear functions $\{\phi_k(z), k = 1, \ldots, N\}$ such that

$$\phi_k(z_i) = \delta_{ik},$$

and consider the projection of equation (8.7) on the space of piecewise linear functions on the triangulation. We obtain:

$$\sum_i M_{ki}\dot{\omega}_i = -\mu \sum_i K_{ki}\omega_i \tag{8.8}$$

where

$$M_{ki} = \int \phi_k(z)\phi_i(z)\,dz, \quad K_{ki} = \int \nabla\phi_k(z)\cdot\nabla\phi_i(z)\,dz.$$

The quantities M_{ij} and K_{ij} are easily computable from the triangulation. M_{ij} is the *mass matrix* and K_{ij} is the *stiffness matrix* associated to the triangulation [17]. These matrices can be easily computed from the triangulation. System

(8.8) could be discretized in time by a *Crank-Nicolson method* in order to avoid the stability restriction on the time step:

$$\sum_j M_{kj} \frac{\omega_j^{n+1} - \omega_j^n}{\mu \Delta t} + \sum_j K_{kj} \frac{\omega_j^{n+1} + \omega_j^n}{2} = 0 \tag{8.9}$$

It is not clear to the authors what is the best way to solve the large, sparse linear system (8.9) for ω^{n+1}. The LU factorization does not seem to be convenient, since the triangulation changes at every time step. Probably the best strategy consists of an iterative method such as a preconditioned conjugate gradient or GMRES.

8.3 Boundary conditions for the Navier-Stokes equations

We consider now the treatment of the boundary conditions for the Navier-Stokes equations in a bounded region Ω. The no-slip boundary condition for a boundary at rest reads

$$u = 0 \qquad \text{on } \partial\Omega.$$

In order to enforce this condition on $\partial\Omega$ we make use of Chorin's method, which consists in placing a vortex sheet on the boundary to compensate for the tangential component of the velocity induced by the vorticity distribution inside the domain [15].

We discretize the time and consider a fractional-step method for the semidiscrete Navier-Stokes equations. Let $\omega^n(z)$ be the vorticity distribution at time t_n. The system is updated in the following way:

a) Solve Eq. (8.2) for $\psi^n(z)$ and compute the velocity field $u^n = \nabla^\perp \psi^n$. This velocity field satisfies the no-flow condition, but not the no-slip condition.

b) Consider the intermediate vorticity

$$\omega^{n+\frac{1}{2}} = \omega^n + 2(u^n \cdot \tau)\delta_{\partial\Omega}$$

where τ is the unit vector tangent to the boundary. Solve the diffusion equation for ω:

$$\begin{cases} \dfrac{\partial \omega}{\partial t} &= \Delta \omega \\ \omega(z, t_n) &= \omega^{n+\frac{1}{2}}(z) \end{cases} \tag{8.10}$$

and determine $\tilde{\omega}^{n+1}(z)$.

c) Compute the velocity field corresponding to the vorticity distribution $\tilde{\omega}^{n+1}(z)$ and solve the Euler equations in the time interval (t_n, t_{n+1}).

The new vorticity distribution will be denoted $\omega^{n+1}(z)$. The convergence of this algorithm for the semidiscrete equations is proved in [8] in the case of the half plane. We propose here the following discretization of the algorithm. Let us suppose we know the vorticity distribution $\omega_\tau^n(z)$ which is associated to a given triangulation $\mathcal{T}^n$ at time t_n. The first step consists in solving the Poisson equation for ψ^n with Dirichlet boundary conditions. Then, once the velocity u^n is computed on the boundary, the diffusion step is discretized in the following way. First, the triangulation is extended beyond $\partial\Omega$, by reflecting the triangles with one side on $\partial\Omega$. If the size of the triangles is small compared to the radius of curvature of $\partial\Omega$, the triangulation on the exterior of Ω reproduces a symmetric copy of the first line of triangles, with a small distortion (see Figure 28). After the triangulation has been extended, the function $\psi^n(z)$ is extended symmetrically beyond $\partial\Omega$. This will provide a discretization of the zero Neumann condition for the diffusion equation.

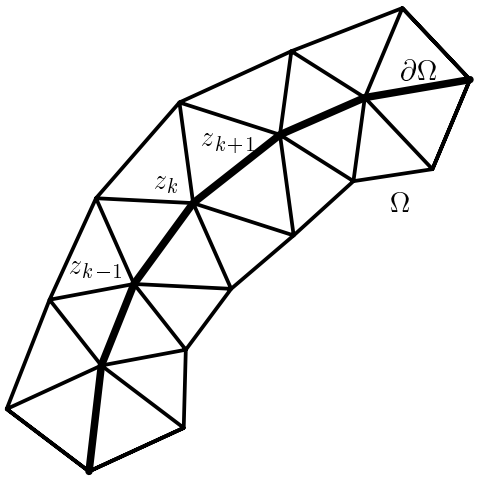


Figure 28: Symmetric extension of the triangulation beyond $\partial\omega$.

Next we multiply (8.10) by ϕ_k and integrate:

$$\frac{d}{dt}\int \phi_k(z)\omega^n(z)\,dz = \mu\int \phi_k\Delta\omega^n\,dz + 2\int \phi_k(u^n\cdot\tau)\delta_{\partial\Omega}\,dz, \quad k=1,\dots,N \tag{8.11}$$

If z_k is inside Ω then the second term on the right hand side is zero, and one obtains an equation of the form (8.8). If z_k is on $\partial\Omega$ then one obtains

$$\sum_i C_{ki}\dot{\omega}_i = \mu\sum_i M_{ki}\omega_i + 2W_k$$

where W_k is a line integral along the segments $z_{k-1}z_k$ and z_kz_{k+1}.

8.4 Higher order methods

Our method introduces several approximations; space and time discretization, and truncation of the series in the fast velocity evaluation. In Section 5 we saw how to control the error in the fast velocity evaluation, and time discretization error can be made small by using high order ODE solvers. Runge-Kutta or multistep methods can be used for this purpose. The main cause of inaccuracy lies in the spatial discretization. In this section we improve the spatial accuracy of the method.

The spatial discretization error in our method is due to the approximation of ω by a piecewise linear function. Such an approximation is second order in the size of the triangles. A better accuracy could be obtained by using basis functions that are polynomials of degree greater than one. There are several possibilities for obtaining higher order accuracy in the approximation of functions of two variables, which are commonly used in the finite element method. Most of these techniques, however, require values of the function at points that are not vertices of a triangle [17]. Such techniques have an intrinsic difficulty in this setting. Suppose we make use of the value of the function ω at the middle of the edge of the triangles. If we move these points with the flow, their vorticity is unchanged, but at the next time step their location will not be in the middle of the edge of a triangle. If we leave the point at the middle of the edge, then at the next time step the value of the vorticity at this point will change. We may think of correcting this effect by adding a term that takes into account the fact that the middle of the edge is not a Lagrangian point (up to $O(h^2)$), but then topological difficulties arise.

For these reasons, it is more convenient to use a higher-order approximation formula based on quantities defined at the vertices of the triangulation. We propose to use the space of piecewise cubic polynomials on the triangulation, with equal coefficients for the x^2y and xy^2 terms.

On each triangle such a function $\phi(x, y)$ is defined by 9 parameters:

$$\begin{aligned} \phi(x,y) &= a_1 + a_2 x + a_3 y + a_4 x^2 + a_5 xy + a_6 y^2 \\ &\quad + a_7 x^3 + a_8 (x^2 y + x y^2) + a_9 y^3. \end{aligned} \tag{8.12}$$

The nine parameters are uniquely defined by giving the value of the function and its partial derivatives at the three nodes of the triangle.

Let us denote by u and v the x and y components of the velocity u, and by ξ and η the components of $\nabla\omega$:

$$\xi \equiv \frac{\partial \omega}{\partial x}, \quad \eta \equiv \frac{\partial \omega}{\partial y}.$$

Then, by taking the x and y derivative of the equation for ω (Equation (2.3) in Section 2) one obtains the transport equations for ω, ξ and η:

$$\frac{d\omega}{dt} = 0,$$

$$\begin{aligned}\frac{d\xi}{dt} &= -\left(\frac{\partial u}{\partial x}\xi + \frac{\partial v}{\partial x}\eta\right),\\ \frac{d\eta}{dt} &= -\left(\frac{\partial u}{\partial y}\xi + \frac{\partial v}{\partial y}\eta\right),\end{aligned}$$

where

$$\frac{d}{dt} \equiv \partial_t + (u \cdot \nabla).$$

If we are able to compute u, v, u_x, v_x, u_y and v_y due to a piecewise cubic polynomial of the form (8.12) then we can solve the system of differential equations

$$\begin{aligned}\frac{dx_i}{dt} &= u(z_i),\\ \frac{dy_i}{dt} &= v(z_i),\\ \frac{d\xi_i}{dt} &= -(u_x(z_i)\xi_i + v_x(z_i)\eta_i),\\ \frac{d\eta_i}{dt} &= -(u_y(z_i)\xi_i + v_y(z_i)\eta_i.\end{aligned}$$

It is possible to extend our velocity evaluation method to compute such quantities. Indeed, u is split into a local term u_L and a far field term u_F. The first involves terms of the form

$$\int_\tau K(z - z')\omega_h(z', t)\, dz'$$

where ω_h is a polynomial of the form (8.12). Such integrals can be computed analytically as shown in [13]. The derivative of the field can be computed analytically as well;

$$\int_\tau \frac{\partial K}{\partial x}(z - z')\omega_h(z')\, dz'$$

can be integrated exactly as a line integral along the boundaries of the triangulation $\mathcal{T}$, and the far field contribution is automatically provided by the $O(N^{4/3}\log\epsilon)$ algorithm (see Section 5.4) which returns the first p terms of the Taylor expansion of the field.

A last observation concerns the expected order of accuracy of such an algorithm. Piecewise linear elements give $O(h^2)$, quadratic elements $O(h^3)$, and cubic elements $O(h^4)$. However we are not using the full subspace of piecewise cubic elements here, because each element has 9 free parameters instead of 10. This degrades the accuracy of the approximation to $O(h^3)$. For smooth flows the higher accuracy should compensate for the extra work needed to compute the derivative of the velocity field. We expect the computational time to be no more than twice the time required for the piecewise linear method for the same number of the points, because the far field terms are obtained for free. For very smooth flows, it might even be more economical to use quartic polynomials to get $O(h^5)$ accuracy, and evolve second derivatives as well.

8.5 Extension to three dimensions

The method here presented could be extended, in principle, to the incompressible Euler and Navier-Stokes equations in three dimensions. The extension, however, is not a trivial one.

The hardest problem is the computation of the Biot-Savart integral on a piecewise linear vorticity distribution. It is not clear whether a piecewise linear function times the Biot-Savart kernel can be integrated analytically on a tetrahedron in three dimensions. If it is not possible, then one should try to reduce the computation of such integrals to combinations of integrals that depend on fewer parameters. Then these new "special functions" could be tabulated and their values computed by interpolation. The feasibility of such a procedure, however, is questionable, since the next problem is the development of a fast algorithm for the computation of the far field in three dimensions. The fast multipole method in three dimensions is not as efficient as it is in two dimensions. This would make the velocity evaluation quite slow. Furthermore, the problem of the boundary conditions in ω in three dimensions is more complicated than in 2-D.

In view of these considerations, we think that a different approach could be more effective. A finite element method could be used to solve the Poisson equation for the vector velocity potential $\vec{\psi}(x, y, z, t)$:

$$-\Delta\vec{\psi} = \vec{\omega};$$

the velocity field is then

$$u = \nabla \times \vec{\psi}. \tag{8.13}$$

In order to discretize this equation, we need to construct a 3D grid which is the 3D analogue of the Delaunay triangulation. This can be done by dividing the space into Delaunay tetrahedra, that are defined in a way similar to the two dimensional case. Then we consider a basis $\mathcal{B}$ of piecewise linear functions on the triangulation, $\{\phi_i(x), i = 1, \ldots, N\}$. By going to a weak formulation and projecting on the subspace $\mathcal{B}$, the Poisson equation is discretized in the usual form:

$$\sum_j K_{ij}\vec{\psi}_j = \sum_j M_{ij}\vec{\omega}_j$$

where the matrices M and K are the mass and stiffness matrix corresponding to the given triangulation in space. In order to obtain the values of $u_i \approx u(x_i)$, we multiply equation (8.13) by ϕ_i and integrate. We obtain for $u_i^\alpha, i = 1, \ldots, N, \alpha = 1, 2, 3$, the following system:

$$\sum_{ij} M_{ij}u_j^\alpha = \sum_j \left(S_{ij}^\beta \psi_j^\gamma - S_{ij}^\gamma \psi_j^\beta\right) \tag{8.14}$$

where (α, β, γ) is a cyclic permutation of $(1, 2, 3)$, $(x^1, x^2, x^3) \equiv (x, y, z)$, and

$$S_{ij}^\alpha = \int \phi_i(x)\frac{\partial\phi_j(x)}{\partial x^\alpha}\,dx.$$

The Euler equations in three dimensions are

$$\frac{\partial \vec{\omega}}{\partial t} + (u \cdot \nabla)\vec{\omega} = (\vec{\omega} \cdot \nabla)u$$

This equation can be written as

$$\frac{d\vec{\omega}}{dt} = (\vec{\omega} \cdot \nabla)u \tag{8.15}$$

along the fluid lines $dx/dt = u$.

Let $\vec{\Omega}$ be the right hand side of Equation (8.15):

$$\Omega^\beta = \sum_{\alpha=1}^{3} \omega^\alpha \frac{\partial u^\beta}{\partial x^\alpha}.$$

Then we can compute a piecewise linear approximation of Ω^β at the nodes in the usual way. We obtain the system for Ω_i^β:

$$\sum_j M_{ij}\Omega_j^\beta = \sum_j \sum_{\alpha=1}^{3} \omega_j^\alpha S_{ij}^\alpha u_j^\beta.$$

Once u_i^α and Ω_j^α are known, the position and vorticity at the nodes can be updated by solving the equations

$$\begin{aligned} \dot{x}_i &= u_i \\ \dot{\vec{\omega}}_i &= \vec{\Omega}_i \end{aligned}$$

Of course there is no guarantee that the natural invariants of the three dimensional Euler equations are conserved in this discrete method. In particular, the total vorticity will not be conserved. One should check how well conservation of these physical invariants is maintained.

The extension to the Navier-Stokes equations could be done in a way similar to the two dimensional case, i.e. with a splitting method. The diffusion step for the vorticity vector could be treated by a Galerkin-Crank-Nicolson method.

9 Conclusions

We have presented an efficient and accurate new adaptive method for the 2-D Euler equations. Our method resembles the vortex method, but differs in approximating the vorticity by triangulation and interpolation rather than a sum of blobs. This alteration produces a method which is more accurate for long-time computations.

The efficiency of our method is due to an efficient Delaunay triangulation method, to a fast velocity evaluation technique related to the fast multipole

method, and to the construction of an adaptive initial grid. Our numerical results demonstrate that each of these techniques plays an essential role in making our computations accurate and efficient. We present a wide spectrum of numerical results, for simple classical test problems as well as complex problems without known solutions. In all cases, our method exhibits long-time accuracy. Even discontinuous initial data can be evolved accurately using our adaptive grid technique.

The method generalizes in various ways, to three-dimensional problems, viscous flow and domains with boundaries, and appears highly promising as a tool for engineering analysis of complex fluid flows.

A The Kirchhoff elliptical vortex

A rotating elliptical patch of constant vorticity is an exact solution of the Euler equations. A discussion of this topic can be found, for example, in [26]. We summarize it here for completeness.

Let $x'Oy'$ denote a fixed cartesian frame of reference in $\mathbb{R}^2$. Let us consider the 2D Euler equations with the following initial condition for the vorticity:

$$\omega(x', y', 0) = \begin{cases} \omega_1 & \text{if} \quad \dfrac{x'^2}{a^2} + \dfrac{y'^2}{b^2} < 1 \\ \\ 0 & \text{if} \quad \dfrac{x'^2}{a^2} + \dfrac{y'^2}{b^2} \geq 1 \end{cases}$$

where ω_1 is constant.

Let us make the *ansatz* that the solution for the vorticity distribution is an elliptical patch of constant vorticity which rotates without changing shape with an angular velocity ω_0. We shall prove that this *ansatz* is consistent with the Poisson equation for the stream function and we derive an expression for ω_0.

Let us consider a frame of reference which is at rest with the rotating ellipse and let us denote by x and y its coordinates. In this frame of reference the stream function does not depend on time. Let Ω denote the region with vorticity ω_1. In the region outside Ω the stream function satisfies the equation

$$\frac{\partial^2 \psi}{\partial x^2} + \frac{\partial^2 \psi}{\partial y^2} = 0 \tag{A.1}$$

The fluid lines can not cross the boundary $\partial\Omega$. The boundary condition on $\partial\Omega$ is therefore

$$-u \cdot \nu = \omega_0 r \cdot \tau$$

where ν and τ are unit vectors respectively normal and tangent to the boundary, and $r = (x, y)$. The boundary condition for ψ is therefore

$$-\frac{\partial \psi}{\partial \tau} = \omega_0 r \cdot \tau \tag{A.2}$$

It is convenient to make use of elliptical coordinates

$$x = c\cosh\xi\cos\eta, \qquad y = c\sinh\xi\sin\eta \tag{A.3}$$

where $c^2 = a^2 - b^2$. In these coordinates the Laplace equation for the stream function becomes:

$$\begin{cases} \dfrac{\partial^2\psi}{\partial\xi^2} + \dfrac{\partial^2\psi}{\partial\eta^2} = 0 & \text{in} \quad R^2\backslash\Omega \\ \dfrac{\partial\psi}{\partial\eta} = -\omega_0 c^2 \sin\eta\cos\eta & \text{on} \quad \partial\Omega \end{cases} \tag{A.4}$$

This equation can be solved by separation of variables.

Let $\psi = X(\xi)H(\eta) + \tilde{A}\xi + \tilde{B}\eta + \tilde{C}$. By inserting this expression into (A.4) and imposing that the velocity u vanishes at infinity, one finds

$$\psi = \frac{\omega_0}{4}(a+b)^2 e^{-2\xi}\cos 2\eta + \tilde{A}\xi \tag{A.5}$$

The constant $\tilde{A}$ is obtained by imposing that the circulation is the integral of ω:

$$\oint_{\partial\Omega} u\cdot d\tau = \pi ab\omega,$$

that is

$$-\int_0^{2\pi} \frac{\partial\psi}{\partial\xi}\, d\eta = \pi ab\omega.$$

This gives

$$\tilde{A} = \frac{1}{2}ab\omega.$$

The stream function inside Ω is obtained by the Poisson equation

$$\frac{\partial^2\psi}{\partial x^2} + \frac{\partial^2\psi}{\partial y^2} = -\omega_1, \tag{A.6}$$

with boundary conditions

$$\frac{ux}{a^2} + \frac{vy}{b^2} = \omega_0 y\frac{x}{a^2} - \omega_0 x\frac{y}{b^2}$$

equivalent to condition (A.2). Equation (A.6) is satisfied by a stream function of the form

$$\psi = -\frac{1}{2}\omega(Ax^2 + By^2) \tag{A.7}$$

provided $A + B = -1$ and $Aa^2 - Bb^2 = -\omega_0(a^2 - b^2)/\omega$. We have to check now that there is no slip across $\partial\Omega$. For this purpose we compute $u\cdot\tau$ from (A.5) and (A.7) and compare the two expressions. From (A.5),

$$u\cdot\tau = -\frac{\partial\psi_{\text{ext}}}{\partial\xi} = \frac{\omega_0}{2}(a+b)^2 e^{-2\xi}\cos 2\eta - \frac{1}{2}ab\omega$$

Suppose $\xi = \xi_0$ defines $\partial\Omega$. Then

$$c\cosh(\xi_0) = a, \quad c\sinh(\xi_0) = b$$

and therefore $(a+b)\exp(-2\xi_0) = c$. It follows that

$$\left.\frac{\partial\psi_{\text{ext}}}{\partial\xi}\right|_{\partial\Omega} = -\frac{1}{2}\omega_0 c^2\cos 2\eta + \frac{1}{2}ab\omega$$

In order to compute $u\cdot\tau$ from (A.7), let us express ψ in terms of ξ and η using (A.3):

$$\psi_{\text{int}} = -\frac{1}{2}\omega_1 c^2(A\cosh^2\xi\cos^2\eta + B\sinh^2\xi\sin^2\eta).$$

It is

$$\left.\frac{\partial\psi_{\text{int}}}{\partial\xi}\right|_{\partial\Omega} = -\frac{\omega_1 ab}{2}(A+B+(A-B)\cos 2\eta).$$

By equating the coefficients of $\cos 2\eta$ in the two expressions we obtain

$$\frac{\omega_1 ab}{2}(A-B) = \frac{1}{2}\omega_0 c^2,$$

which, together with equations for A and B gives

$$\omega_0 = \frac{ab\omega_1}{(a+b)^2}.$$

The relation between the coordinates (x,y) and (x',y') is therefore

$$\begin{aligned} x' &= x\cos\omega_0 t - y\sin\omega_0 t, \\ y' &= x\sin\omega_0 t + y\cos\omega_0 t. \end{aligned}$$

References

[1] C. Anderson and C. Greengard. On vortex methods. *SIAM J. Math. Anal.*, 22:413–440, 1985.

[2] C. Bardos, M. Bercovier, and O. Pironneau. The vortex method with finite elements. *Math. Comp.*, 36:119–136, 1981.

[3] G. K. Batchelor. *An introduction to fluid dynamics.* Cambridge University Press, 1973.

[4] J. T. Beale, A. Eydeland, and B. Turkington. Numerical tests of 3-D vortex methods using a vortex ring with swirl. In C. Anderson and C. Greengard, editors, *Vortex Methods and Vortex Dynamics.* Springer-Verlag, New York, 1992.

[5] J. T. Beale and A. Majda. Vortex methods II: high order accuracy in two and three dimensions. *Math. Comp.*, 39:29–52, 1982.

[6] J. T. Beale and A. Majda. High order accurate vortex methods with explicit velocity kernels. *J. Comput. Phys.*, 58:188–208, 1985.

[7] G. I. Bell. Vortex-induced radiation transported by a contour. *Physica*, 44D:203–228, 1990.

[8] G. Benfatto and M. Pulvirenti. Convergence of Chorin-Marsden product formula in a half space. *Comm. Math. Phys.*, 44D:203–228, 1986.

[9] J. L. Bentley, B. W. Weide, and A. C. Yao. Optimal expected-time algorithms for closest-point problems. *ACM Trans. Math. Softw.*, 6:563–580, 1980.

[10] M. Bernadou, P. L. George, A. Hassim, P. Joly, P. Laugh, A. Perronet, E. Saltel, D. Steer, G. Vanderbrock, and M. Vidrascu. MODULEF: A modular library of finite elements. Publication, INRIA, 1986.

[11] T. F. Buttke. A fast adaptive vortex method for patches of constant vorticity in two dimensions. *J. Comput. Phys.*, 89:161–186, 1990.

[12] J. Carrier, L. Greengard, and V. Rokhlin. A fast adaptive multipole method for particle simulations. *SIAM J. Sci. Stat. Comput.*, 9:669–686, 1988.

[13] T. Chacon Rebollo and T. Y. Hou. A Lagrangian finite element method for the 2-D Euler equations. *Comm. Pure Appl. Math.*, XLIII:735–767, 1990.

[14] Y. Choi, J. A. C. Humphrey, and F. S. Sherman. Random vortex simulation of transient wall-driven flow in a rectangular enclosure. *J. Comput. Phys.*, 75:359–383, 1988.

[15] A. J. Chorin. Numerical study of slightly viscous flow. *J. Fluid Mech.*, 57:785–796, 1973.

[16] A. J. Chorin. The evolution of a turbulent vortex. *Commun. Math. Phys.*, 35:517–535, 1982.

[17] P. G. Ciarlet and J. L. Lions, editors. *Finite Element Methods, Part 1*, volume II of *Handbook of Numerical Analysis*. Elsevier, 1992.

[18] G. H. Cottet, S. Mas-Gallic, and P. A. Raviart. Vortex methods for the incompressible Euler and Navier-Stokes equations. In B. Engquist, M. Luskin, and A. Majda, editors, *Computational fluid dynamics and reacting gas flow*, volume 12 of *IMA volumes in mathematics and applications*. Springer-Verlag, 1988.

[19] R. A. Dwyer. A faster divide-and-conquer algorithm for constructing Delaunay triangulations. *Algorithmica*, 2:151, 1987.

[20] G. B. Folland. *Introduction to Partial Differential Equations*, volume 17 of *Mathematics Notes*. Princeton University Press, 1976.

[21] S. Fortune. A sweepline algorithm for Voronoi diagrams. *Algorithmica*, 2:137, 1987.

[22] J. Goodman, T.Y. Hou, and J. Lowengrub. Convergence of the point vortex method for the 2-d Euler equations. *Comm. Pure Appl. Math.*, 43:415–430, 1990.

[23] P. J. Green and R. Sibson. Computing Dirichlet tesselations in the plane. *The Computer J.*, 21:168–173, 1978.

[24] O. H. Hald. The convergence of vortex methods, II. *SIAM J. Num. Anal.*, 16:726–755, 1979.

[25] O. H. Hald. Convergence of vortex methods for Euler's equations III. *SIAM J. Numer. Anal.*, 24:538–582, 1987.

[26] H. Lamb. *Hydrodynamics*. Dover (New York), sixth edition, 1945.

[27] C. Lawson. Software for C^1 surface interpolation. In J. Rice, editor, *Mathematical Software III*. Academic Press, New York, 1977.

[28] D. T. Lee and B. J. Schachter. Two algorithms for constructing a Delaunay triangulation. *Int. J. Comput. Inf. Sci.*, 9:219–242, 1980.

[29] A. Leonard. Computing three dimensional flows with vortex elements. *Ann Rev. Fluid Mech.*, 17:523–559, 1985.

[30] A. Maus. Delaunay triangulation and the convex hull of n points in expected linear time. *BIT*, 24:151–163, 1984.

[31] D. H. McLain. Two dimensional interpolation from random data. *The Computer J.*, 19:178–181, 1976.

[32] W. F. Mitchell. A comparison of adaptive refinement techniques for elliptic problems. *ACM Trans. Math. Softw.*, 15:326–347, 1989.

[33] T. Ohya, M. Iri, and K. Murota. A fast Voronoi diagram algorithm with quaternary tree bucketing. *Inf. Process. Lett.*, 18:178–181, 1984.

[34] M. Perlman. On the accuracy of vortex methods. *J. Comput. Phys.*, 59:200–223, 1985.

[35] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, New York, 1985.

[36] E. G. Puckett. Vortex methods: an introduction and survey of selected research topics. In R. A. Nicolaides and M. D. Gunzburger, editors, *Incompressible Fluid Dynamics — Trends and Advances*. Cambridge University Press, 1991.

[37] R. J. Renka and A. K. Cline. A triangle-based C^1 interpolation method. *Rocky Mountain J. Math.*, 14:119–139, 1984.

[38] V. Rokhlin. Rapid solution of integral equations of classical potential theory. *J. Comput. Phys.*, 60:187–207, 1985.

[39] L. Rosenhead. The formation of vortices from a surface of discontinuity. *Proc. R. Soc. Lon. A.*, 134:170–192, 1931.

[40] G. Russo. A deterministic vortex method for the Navier-Stokes equations. *J. Comput. Phys.*, 1993.

[41] H. Samet. *The design and analysis of spatial data structures*. Addison-Wesley, Reading, Massachusetts, 1990.

[42] J. A. Sethian and A. F. Ghoniem. Validation study of vortex methods. *J. Comput. Phys.*, 74:283–317, 1988.

[43] M. I. Shamos and D. Hoey. Closest-point problems. In *Proceedings 16th IEEE Symposium on Foundations of Computer Science*. IEEE, October 1975.

[44] J. Strain. An adaptive cell method for Delaunay triangulation. Technical report number LBL-32989, Lawrence Berkeley Laboratory, 1991.

[45] J. Strain. Fast potential theory II: Layer potentials and discrete sums. *J. Comput. Phys.*, 99:251–270, 1992.

[46] J. Strain. Fast adaptive methods for the free-space heat equation. *SIAM J. Sci. Comput.*, 15:185–206, 1994.

[47] M. Tanemura, T. Ogawa, and N. Ogita. A new algorithm for three-dimensional Voronoi tessellation. *J. Comput. Phys.*, 51:191–207, 1983.

[48] L. van Dommelen and E. A. Rundensteiner. Fast adaptive summation of point forces in the two-dimensional Poisson equation. *J. Comput. Phys.*, 83:126–147, 1989.

[49] H. Y. Wang. A high order vortex method for patches of constant vorticity in two dimensions. Research Report PAM-534, Center for Pure and Applied Mathematics, University of California, Berkeley, August 1991.

[50] F. W. Wilson, R. K. Goodrich, and W. Spratte. Lawson's algorithm is nearly optimal for controlling error bounds. *SIAM J. Numer. Anal.*, 27:190–197, 1990.

[51] N. J. Zabusky, M. H. Hughes, and K. V. Roberts. Contour dynamics for the euler equations in two dimensions. *J. Comput. Phys.*, 96:96–121, 1979.

AMS Subject Classifications: 76M10, 76M25, 65M50, 65M60, 65Y25

Key words: vortex methods, Delaunay triangulations, Voronoi diagrams, Euler equations, fluid flow, finite elements, free Lagrangian methods