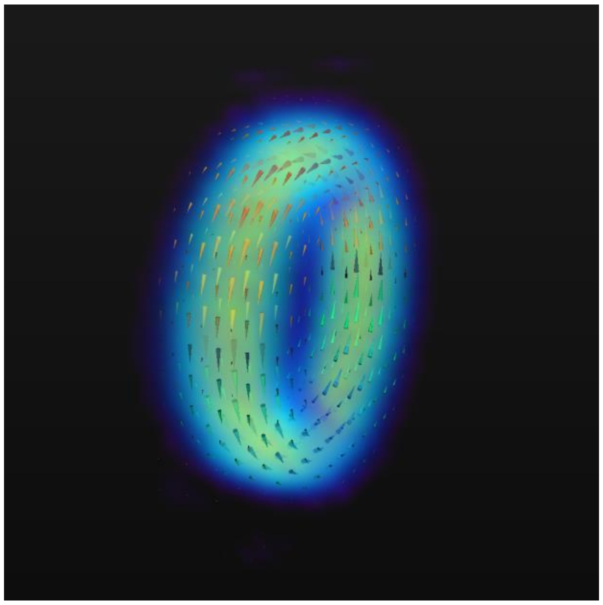
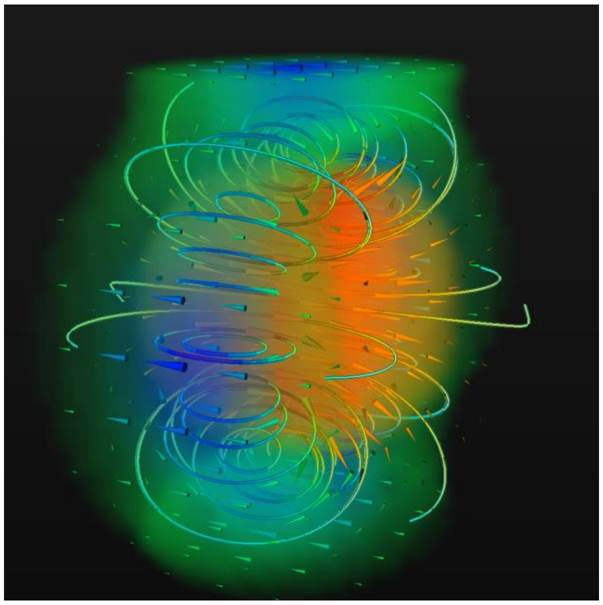
Fluid Simulation for Video Games (part 6)

By Dr. Michael J. Gourlay





Differential Velocity Solvers

This article, the sixth in a series, describes a radically different technique for computing velocity from vorticity, one of the cornerstones of the fluid simulation presented in these articles. The first article summarized fluid dynamics; the second surveyed fluid simulation techniques; and the third and fourth presented a vortex–particle fluid simulation with two-way fluid–body interactions that runs in real time. The fifth article demonstrated how to obtain and use CPU usage profiling data to optimize and further parallelize the code so that it ran faster.

This article describes a differential technique for solving velocity from vorticity and contrasts its results and performance with the integral treecode technique presented in Part 3. The Poisson solver presented in this article runs faster than the treecode, but its results look different—and are potentially less satisfying.

Velocity from Vorticity, Revisited

Recall from the second article that you can compute velocity $\boldsymbol{v}$ from vorticity $\boldsymbol{\omega}$, where $\boldsymbol{\omega} = \nabla \times \boldsymbol{v}$, using the Biot-Savart law:

$$\boldsymbol{v}(\boldsymbol{r}) = \frac{\mathbf{1}}{\mathbf{4}\boldsymbol{\pi}} \int \frac{\boldsymbol{\omega}(\boldsymbol{r}') \times (\boldsymbol{r} - \boldsymbol{r}')}{|\boldsymbol{r} - \boldsymbol{r}'|^3} \boldsymbol{d}^3\boldsymbol{r}' \qquad (1)$$

Part 3 described a fluid simulation algorithm that used this law to calculate velocity from vorticity, using a “treecode” to approximate this integral. This article describes an alternative way to compute velocity, using a differential approach.

Helmholtz Decomposition

The Helmholtz theorem states that a vector function $\boldsymbol{v}$ can be decomposed into an irrotational part ($\boldsymbol{\nabla}\varphi$) that has only divergence and a solenoidal part ($\boldsymbol{\nabla} \times \boldsymbol{A}$), which in turn has only curl:

$$\boldsymbol{v} = \boldsymbol{\nabla} \times \boldsymbol{A} - \boldsymbol{\nabla}\varphi \qquad (2)$$

where:

$$\boldsymbol{A}(\boldsymbol{r}) = \frac{1}{4\pi} \int \frac{\boldsymbol{\nabla}' \times \boldsymbol{v}(\boldsymbol{r}')}{|\boldsymbol{r} - \boldsymbol{r}'|} d^3\boldsymbol{r}' \qquad (3)$$

If $\boldsymbol{v}$ is velocity, then $\boldsymbol{\nabla}' \times \boldsymbol{v}(\boldsymbol{r}',)$,—the curl of velocity—is vorticity, $\boldsymbol{\omega}$. Because $\boldsymbol{\nabla} \times (\boldsymbol{\nabla}\varphi) \equiv \mathbf{0}$ for any φ, plugging the decomposed $\boldsymbol{v}$ (eqn. 2) into the formula for vorticity gives $\boldsymbol{\omega} = \boldsymbol{\nabla} \times (\boldsymbol{\nabla} \times \boldsymbol{A})$. Apply to this the following vector identity:

$$\boldsymbol{\nabla} \times (\boldsymbol{\nabla} \times \mathbf{A}) = \boldsymbol{\nabla}(\boldsymbol{\nabla} \cdot \mathbf{A}) - \nabla^2\mathbf{A} \qquad (4)$$

Notice that if $\mathbf{A}$ had an irrotational part, it would not contribute to $\boldsymbol{v}$. So for the sake of simplicity, choose $\boldsymbol{\nabla} \cdot \mathbf{A} \equiv \mathbf{0}$ to obtain:

$$\nabla^2\mathbf{A} = -\boldsymbol{\omega} \qquad (5)$$

This is called a *Poisson equation,* and solving it facilitates obtaining velocity from vorticity—our immediate goal. (Technically, (5) is a *vector Poisson equation,* because all the quantities are vectors, but this amounts to nothing more than 3 *scalar Poisson equations.*) This equation is one of the most studied partial differential equations, so you have many choices for solving it. This article and the code that accompanies it explore several techniques for solving this equation.

Discrete Poisson Solvers

As described in Part 2, to solve this equation numerically, you must discretize the Poisson equation, and then formulate an approximate solution. Use a Taylor series to expand a function $f(x)$ about $x = x_0$ (defining $X \equiv (x - x_0)$ for brevity):

$$f(x) = f(x_0) + \left.\frac{\partial f}{\partial x}\right|_{x_0} X + \left.\frac{\partial^2 f}{\partial x^2}\right|_{x_0} \frac{X^2}{2!} + \left.\frac{\partial^3 f}{\partial x^3}\right|_{x_0} \frac{X^3}{3!} + O(X^4) \qquad (6)$$

Evaluate (6) at $x_+ \equiv x_0 + h$ and $x_- \equiv x_0 - h$, using the abbreviations $f_{x+} \equiv f(x_+)$, $f_{x-} \equiv f(x_-)$ and $f_0 \equiv f(x_0)$:, as shown in Figure 1.

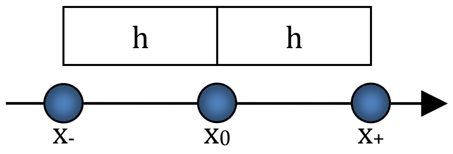


Figure 1. Evaluation points of a Taylor series expansion of a function

$$f_+ = f_0 + \left.\frac{\partial f}{\partial x}\right|_{x_0} h + \left.\frac{\partial^2 f}{\partial x^2}\right|_{x_0} \frac{h^2}{2!} + \left.\frac{\partial^3 f}{\partial x^3}\right|_{x_0} \frac{h^3}{3!} + O(h^4) \qquad (7)$$

$$f_- = f_0 - \left.\frac{\partial f}{\partial x}\right|_{x_0} h + \left.\frac{\partial^2 f}{\partial x^2}\right|_{x_0} \frac{h^2}{2!} - \left.\frac{\partial^3 f}{\partial x^3}\right|_{x_0} \frac{h^3}{3!} + O(h^4) \qquad (8)$$

Adding these equations causes the odd terms to cancel. Rearrange terms to obtain a finite difference approximation for the second derivative:

$$\left.\frac{\partial^2 f}{\partial x^2}\right|_{x_0} = \frac{f_+ - 2f_0 + f_-}{h^2} + O(h^2) \qquad (9)$$

Extending (9) to three dimensions yields a finite difference *Laplacian* operator, ∇^2:

$$\nabla^2 \equiv \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2} + \frac{\partial^2 f}{\partial z^2} \approx \frac{f_{x+} + f_{x-} + f_{y+} + f_{y-} + f_{z+} + f_{z-} - 6f_0}{h^2} \qquad (10)$$

Figure 2 shows the so-called *stencils* for this finite difference Laplacian operator—that is, the geometric arrangement of points involved in calculating this discrete Laplacian.

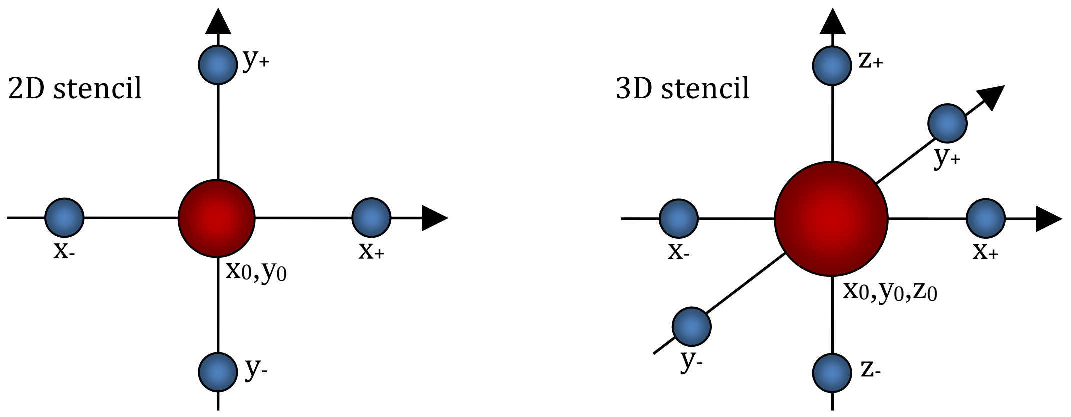


Figure 2. Stencils of a finite difference Laplacian operator. Node size indicates the relative magnitude of its contribution, while color indicates sign: Blue is positive; red is negative.

Plug (10) into each component, A_i, of $\mathbf{A} = \langle A_x, A_y, A_z \rangle$ in the vector Poisson equation (5) to obtain the discrete vector Poisson equation:

$$\frac{A_{i_{x+}} + A_{i_{x-}} + A_{i_{y+}} + A_{i_{y-}} + A_{i_{z+}} + A_{i_{z-}} - 6A_{i_0}}{h^2} = -\omega_\text{i} \qquad (11)$$

Solving (11) for A_{i_0} turns this algebraic equation into the basis for an iterative solver:

$$A_{i_0} := \left(A_{i_{x+}} + A_{i_{x-}} + A_{i_{y+}} + A_{i_{y-}} + A_{i_{z+}} + A_{i_{z-}} + h^2\omega_\text{i}\right)/6 \qquad (12)$$

Applying (12) repeatedly to each cell in a grid leads to a solution of the Poisson equation.

Computing (12) is either called the *Jacobi method* or the *Gauss-Seidel method,* depending on whether the result is stored in a new location or in the original location. Because the result does not require a separate output buffer, the Gauss-Seidel method takes less memory. It also converges more rapidly, because the updated results stored in a grid cell immediately contribute to the solution of its neighbors.

Figure 3 shows that for each iteration of (12), information propagates only between adjacent grid cells. The number of times you have to apply (12) depends on the *rate of convergence* of the method. Convergence, in turn, depends on the number of grid points. If the grid has more points, it takes more iterations to find a solution, because more update steps are necessary for information to propagate between grid cells far from each other.

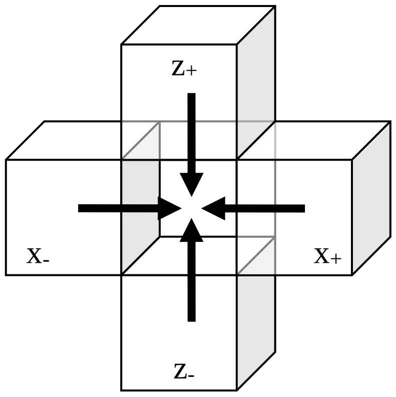


Figure 3. When using the Jacobi or Gauss-Seidel method to solve the Poisson equation for a cell, each of its neighbors contributes part of the solution.

Figure 4 shows the gradual convergence of the Gauss-Seidel method. Unfortunately, it converges rather slowly. On a grid with N points, it takes approximately N^2 iterations to solve the equation.

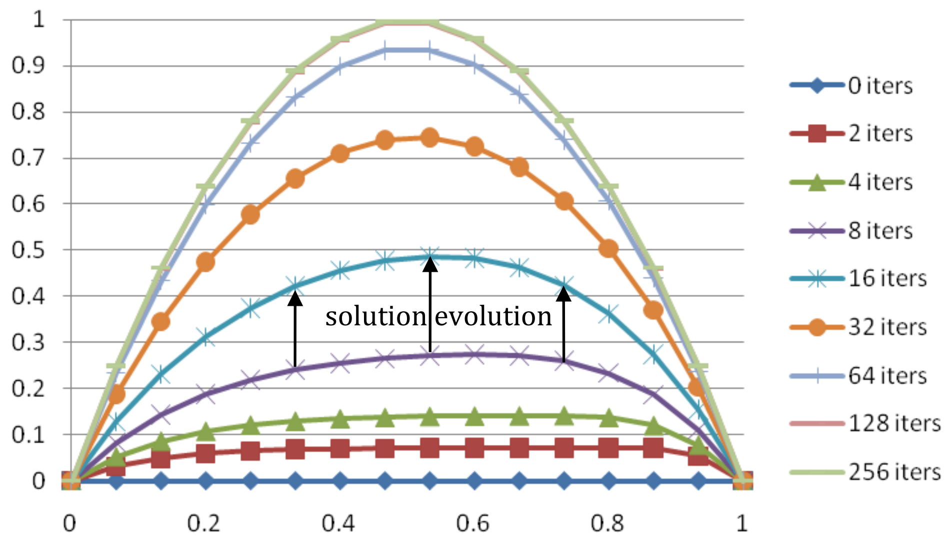


Figure 4. Evolution of the solution to a Poisson equation as solved by the Gauss-Seidel method for a 1D grid with 16 points.

Speeding Up Gauss-Seidel

Fortunately, one technique—almost as simple as Gauss-Seidel—converges much faster. Consider that each iteration of Gauss-Seidel produces a different approximate solution, each closer to the actual solution than the previous (just as Figure 4 shows). The *difference* between curves in Figure 4 indicates the *direction* of the actual solution relative to the *current* approximation of that solution. In other words, given the curves $f^{(j-1)}$ for iteration j-1 and $f^{(j)}$ for iteration j, the difference $f^{(j)}$-$f^{(j-1)}$ gives a good sense of where the solution is headed (as the arrows in Figure 4 depicted). A technique called *Richardson extrapolation* combines multiple approximations so that their errors mostly cancel. This idea leads to the successive over-relaxation (SOR) formula—an almost trivial extension of (12). Given approximations $\mathbf{A}^{(j)}$ and $\mathbf{A}^{(j-1)}$ for the value of the vector potential at a given grid point (from 12), use the following formula to assign a new value to $\mathbf{A}$:

$$\mathbf{A} := \alpha\mathbf{A}^{(j)} + (1-\alpha)\mathbf{A}^{(j-1)} \qquad (13)$$

where α is the *relaxation parameter.* Useful values of α lie between 1 and 2, typically near 1.7, but the optimal value depends on the number of grid points along each direction. This seemingly minor change drastically improves convergence; instead of needing N^2 iterations, $N^{3/2}$ iterations suffice.

Note: Aficionados of physics simulation will likely encounter a linear algebraic solver called the *conjugate gradient* (CG) method. Although this method has additional merits when solving systems other than the Poisson equation, CG has the same order of convergence as SOR, and SOR is much simpler to understand and implement.

Parallelizing Gauss-Seidel

Because the output of each iteration of the Jacobi method is written into a separate storage area from the input, that method is an embarrassingly parallelizable algorithm. In contrast, the Gauss-Seidel method overwrites the same grid from which it reads, so in its simplest form it does not safely and efficiently parallelize. Fortunately, a simple trick helps avoid complicated synchronization issues: Partition the grid into a checkerboard pattern, as Figure 5 shows.

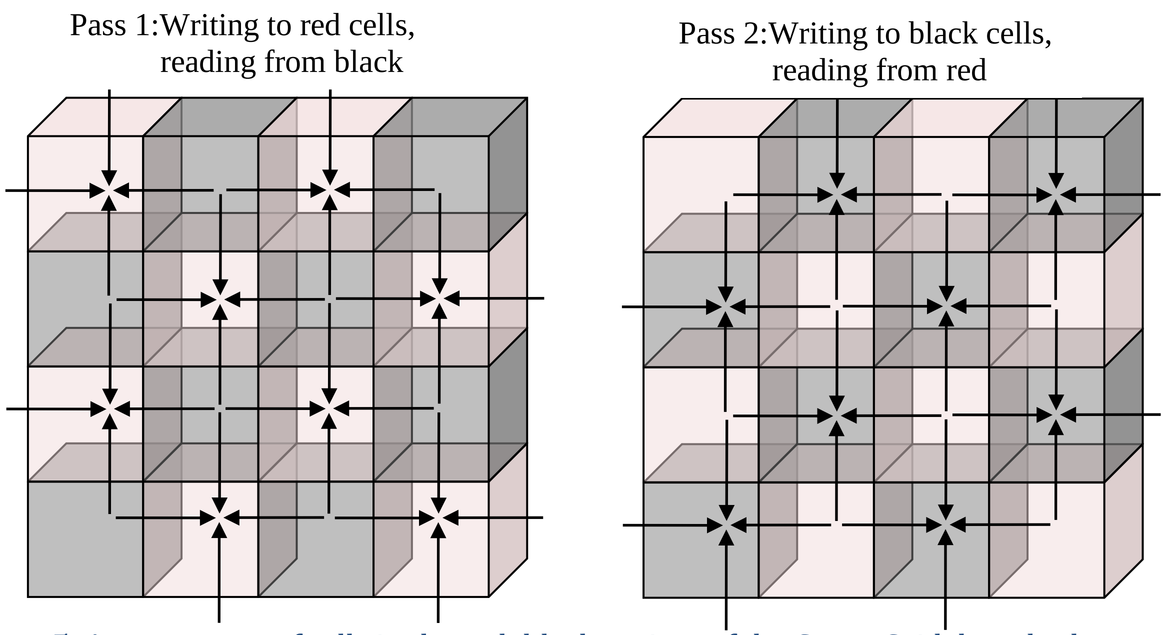

Pass 1:Writing to red cells, reading from black
Pass 2:Writing to black cells, reading from red

Figure 5. Arrangement of cells in the red–black variant of the Gauss-Seidel method

Consider grid cells as being either red or black. When writing to red cells, all inputs come from black cells and vice versa. Each iteration of the original Gauss-Seidel method therefore becomes two "half" passes: a red pass and a black pass, where each of those updates only half of the grid cells.

The `SolveVectorPoisson` routine shows the front-end wrapper routine that implements the red–black Gauss-Seidel method using Intel® Threading Building Blocks (TBB). Note that the worker routine, `StepTowardVectorPoissonSolution`, is called twice: once for red and once for black. That worker routine implements equations (12) and (13) for each grid point. For further details, see the code that accompanies this article.

```
/*! \brief Function object to solve vector Poisson equation using Threading Building Blocks
*/
class UniformGrid_StepTowardVectorPoissonSolution_TBB
{
    UniformGrid< Vec3 > &       mSolution   ;  ///< Address of object containing solution
    const UniformGrid< Vec3 > & mLaplacian  ;  ///< Address of object containing Laplacian
    const GaussSeidelPortion    mRedOrBlack ;  ///< Whether to operate on red or black cells
  public:
    void operator() ( const tbb::blocked_range<size_t> & r ) const
    {  // Compute subset of velocity grid.
      StepTowardVectorPoissonSolution( mSolution, mLaplacian, r.begin(), r.end(), mRedOrBlack );
    }
    UniformGrid_StepTowardVectorPoissonSolution_TBB( UniformGrid< Vec3 > & pSolution
      , const UniformGrid< Vec3 > & pLaplacian , GaussSeidelPortion redOrBlack )
      : mSolution( pSolution ) , mLaplacian( pLaplacian ) , mRedOrBlack( redOrBlack ) {}
} ;
```

```
void SolveVectorPoisson( UniformGrid<Vec3> & soln , const UniformGrid<Vec3> & lap , size_t numSteps )
{
  soln.Init( Vec3( 0.0f , 0.0f , 0.0f ) ) ;
  const size_t gridDimMax  = MAX3(   soln.GetNumPoints( 0 )
                    ,  soln.GetNumPoints( 1 )
                    ,  soln.GetNumPoints( 2 ) ) ;
  const size_t maxIters   = ( numSteps > 0 ) ? numSteps : gridDimMax ;
  for( size_t iter = 0 ; iter < maxIters ; ++ iter )
  {
    const size_t numZ   = soln.GetNumPoints( 2 ) ;
    const size_t grainSize = MAX2( 1 , numZ / gNumberOfProcessors ) ;
    parallel_for( tbb::blocked_range<size_t>( 0 , numZ , grainSize )
          , UniformGrid_StepTowardVectorPoissonSolution_TBB(soln, lap, GS_RED  ) ) ;
    parallel_for( tbb::blocked_range<size_t>( 0 , numZ , grainSize )
          , UniformGrid_StepTowardVectorPoissonSolution_TBB(soln, lap, GS_BLACK ) ) ;
  }
}
```

Even Faster Poisson Solvers

Figures 3, 4, and 5 paint a grim picture for the Gauss-Seidel method. Figures 3 and 5 indicate that, at each step, the Gauss-Seidel method communicates information locally only between adjacent cells, whereas Figure 4 clearly indicates that the solution propagates globally across the entire grid. Ultimately, what keeps Gauss-Seidel from being fast is the fact that information needs to get across the entire grid but can only do so one tiny grid cell–sized step at a time.

The *multigrid method,* shown in Figure 6,addresses this problem in a straightforward way: The solution for a coarse grid resembles the solution for a fine grid, yet the solution on a coarse grid takes fewer steps to obtain. Furthermore, the solution obtained for the coarse grid can “prime” the solution for a fine grid. Combining these grid types, the large-scale, low-resolution features are solved quickly on the coarse grids, and the small-scale, high-resolution features are “filled in” on the fine grids. This method turns out to have the fastest convergence, O(N), and is easy to implement.

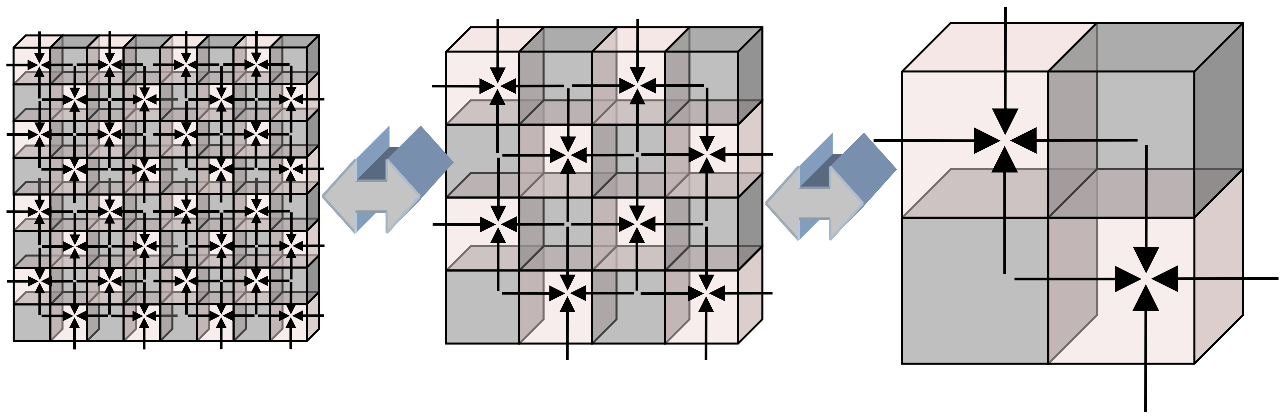


Figure 6. A fine grid, a medium grid and a coarse grid in a multigrid solver

The following code snippet shows how to implement a multigrid algorithm to solve the Poisson equation, reusing the `SolveVectorPoisson` from earlier:

```
unsigned maxValidDepth = 0 ;
for( unsigned iLayer = 1 ; iLayer < mVorticityMultiGrid.GetDepth() ; ++ iLayer )
{
  const unsigned minDim = MIN3( mVorticityMultiGrid[ iLayer ].GetNumPoints( 0 )
              , mVorticityMultiGrid[ iLayer ].GetNumPoints( 1 )
              , mVorticityMultiGrid[ iLayer ].GetNumPoints( 2 ) ) ;
  if( minDim > 2 )
  {
    mVorticityMultiGrid.DownSampleInto( iLayer ) ;
    vectorPotentialMultiGrid.DownSampleInto( iLayer ) ;
    SolveVectorPoisson( vectorPotentialMultiGrid[ iLayer ], mVorticityMultiGrid[ iLayer ] , 3 ) ;
  }
  else
  {
    maxValidDepth = iLayer - 1 ;
  }
}
for( unsigned iLayer = maxValidDepth ; iLayer >= 1 ; -- iLayer )
{
  vectorPotentialMultiGrid.UpSampleFrom( iLayer ) ;
  SolveVectorPoisson( vectorPotentialMultiGrid[ iLayer-1 ], mVorticityMultiGrid[ iLayer-1 ], 3 ) ;
}
```

See the code that accompanies this article for further details.

Note: The Poisson equation can also be solved using spectral methods. Derivatives and integrals in a spatial domain are the same as multiplication and division in a spectral domain. Converting between domains involves Fourier transforms, fast versions of which (FFTs) run in O(N log N) time.

Boundary Conditions

A complete treatment of any partial differential equation must include boundary conditions. As Part 1 mentioned, typical choices include *essential* (where you specify the value of the solution at the boundaries) and *natural* boundary conditions (where you specify the value of the derivative at the boundaries). The code that accompanies this article readily handles either, and the demos use natural boundary conditions. Note that the *interior boundaries*—that is, the boundaries between the fluid and bodies immersed in the flow—receive the same treatment presented in Part 5, which is quite different and entirely unrelated to the *exterior boundaries,* which are simply the edges of the grid.

Velocity from Vector Potential

Solving the Poisson equation (5) yields the vector potential, **A**. Take the curl of that vector potential to obtain velocity, as in equation (2). Part 3 already presented code to perform that computation, which is reused here.

The rest of the fluid simulation algorithm is identical to what Parts 3 and 4 presented. Here is an outline of the algorithm:

1. Initialize vorticity stored in vortons (vortex particles).
2. For each simulation time step, perform the following tasks:
    a. Interpolate vorticity from vortons onto a grid.
    b. Compute velocity from vorticity, again onto a grid.
    c. Advect (move) vortons and tracer particles according to the velocity.

d. Stretch and tilt vortons.
e. Diffuse (redistribute) vorticity between adjacent vortons.

Notice that only the details of step 2b vary between this article and Part 3.

Results

Compare the results of this velocity solver to the direct summation algorithm presented in Part 3. Figure 7 shows a vortex ring whose vorticity comes from the same vortons in both cases. Then, the velocity solver obtains a velocity field, either through direct summation or through the Poisson solver. Then, the program computes the curl of that velocity field to produce the images in Figure 7. To the extent that (a) and (b) resemble each other, the Poisson velocity solver has succeeded.

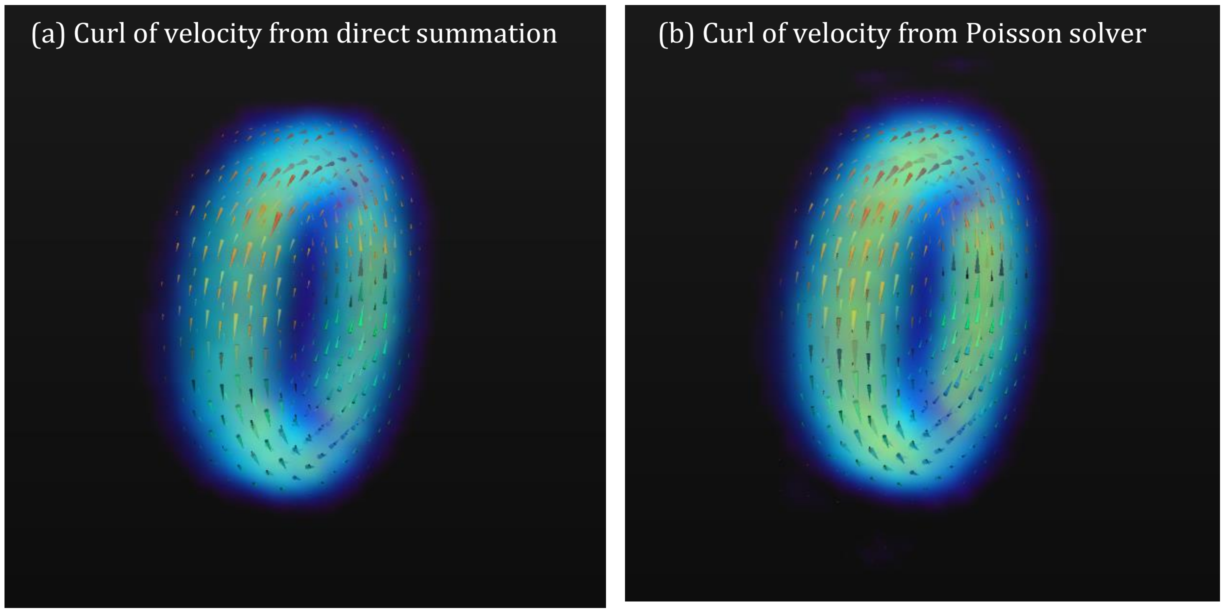

(a) Curl of velocity from direct summation
(b) Curl of velocity from Poisson solver

Figure 7. The curl of velocity of a vortex ring obtained by two different methods. Arrows indicate vorticity. The colored cloud indicates vortex intensity.

Next, look at the velocity field induced by vortons of the vortex ring computed in two different ways: direct summation and the Poisson solver. Once again, the extent to which the images in Figure 8 resemble each other indicates a degree of success for the fast velocity solver.

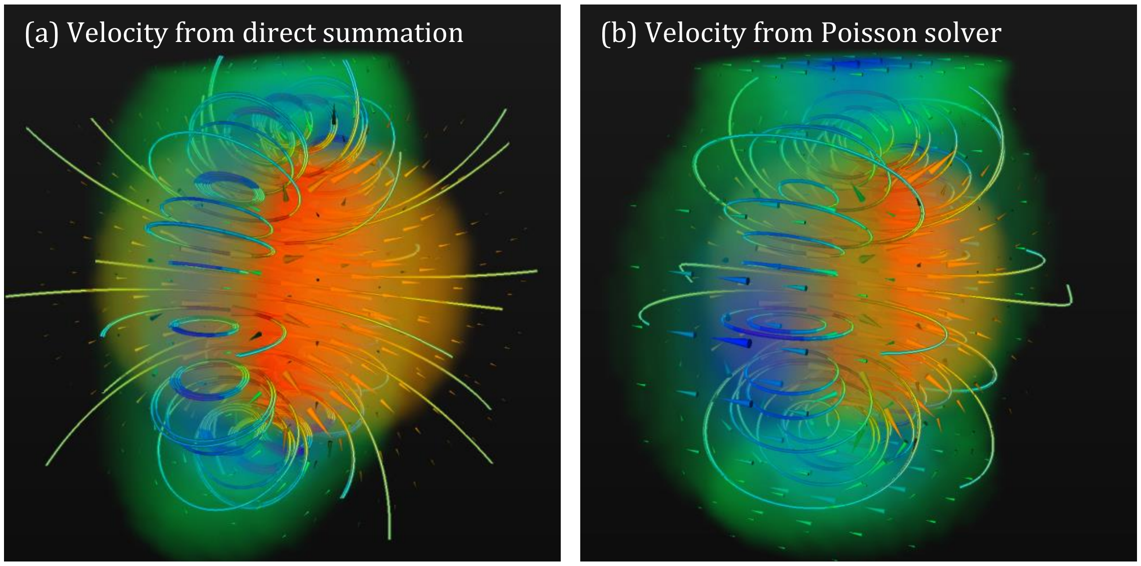

(a) Velocity from direct summation
(b) Velocity from Poisson solver

Figure 8. Velocity field recovered from the vorticity of a vortex ring using (a) direct summation and (b) the Poisson solver. Arrows indicate the velocity flow field. The orange/red cloud indicates velocity moving along +X, and the blue/green cloud indicates velocity moving along –X. Lines indicate streamlines—that is, paths particles would take as they follow the flow.

Figure 8 clearly shows the impact of different boundary conditions; the direct summation (and the treecode algorithm) has boundary conditions something akin to "radiation" boundaries, which do not impose any influence on the flow, as a wall would. In sharp contrast, the Poisson solver cannot as readily facilitate radiation boundary conditions, so the next best thing is to try to keep the boundaries far enough away from the "interior" of the flow that the flow acts nearly as though it's a free flow (meaning far from boundaries). These simulations apparently do not use enough "padding," because the influence of the boundaries is evident in the simulations.

When you run the simulation code that accompanies this article, you can switch between treecode and Poisson solvers to observe the difference in how the flow evolves. The Poisson solver tends to confine the flow to a narrower corridor.

Performance

For the simulations presented, the Poisson solver runs 1.85 times faster than the treecode algorithm, although that speedup is deceptive. The treecode algorithm runs in O(N log N) time, whereas the SOR Poisson solver runs in $O(N^{3/2})$ time. So, for large enough grids, the treecode algorithm would run faster. Meanwhile, the multigrid solver runs in O(N) time, so eventually, the multigrid solver would run fastest of all, but the overhead associated with restriction and interpolation dominates the runtime for grids as small as those used in these simulations.

The world of fluid simulation includes more algorithms than this series of articles surveys, and another method remains that interested readers should consider. Fast multipole methods (FMM) reside in the integral family akin to treecodes, and they can yield O(N) runtimes, just like multigrid methods. The math behind FMM is, however, much more involved and so is the overhead. You can reduce the FMM to its simplest form, which could be called a *monopole method,* and indeed I have written such an algorithm. The results are fast but considerably less accurate; they give only a vague impression of the flow field and do not stand up to scrutiny, although they sometimes suffice for visual effects in video games.

Summary

This article presented an alternative algorithm for computing velocity from vorticity that departed drastically from the treecode method presented in Part 3, yet both algorithms produce similar results. For the simulations presented in this article, the SOR Poisson solver runs faster than the treecode, but for larger problems, you should expect the treecode to outrun SOR. And for even larger problems, the multigrid method will ultimately run fastest. Although the Poisson solver runs faster and its results can even be more accurate, it has difficulty simulating free flows, whereas the treecode algorithm readily handles free flows.

The next article in this series will present additional components that demonstrate how to incorporate these techniques into a visual effects subsystem typically present in a video game.

Further Reading

- Course notes for Jim Demmel's "Applications of Parallel Computers." Available from http://www.cs.berkeley.edu/~demmel/cs267_Spr06/Lectures/lectures.html.
- M.L. Smedinghoff (2005). *Solving the Poisson Equation with Multigrid.*
- A.K. Mitra. *Finite Difference Method for the Solution of the Laplace Equation.*