Symbolic computation of turbulence and energy dissipation in the Taylor vortex model

Richard J. Fateman
Computer Science Division, EECS Department
University of California at Berkeley

April 8, 1998

Abstract

Using a classic example proposed by G. I. Taylor, we reconsider through the use of computer algebra, the mathematical analysis of a fundamental process in turbulent flow, namely: How do large scale eddies evolve into smaller scale ones to the point where they are effectively absorbed by viscosity? The explicit symbolic series solution of this problem, even for cleverly chosen special cases, requires daunting algebra, and so numerical methods have become quite popular. Yet an algebraic approach can provide substantial insight, especially if it can be pursued with modest human effort.

The specific example we use dates to 1937 when Taylor and Green[8] first published a method for explicitly computing successive approximations to formulas describing the 3-dimensional evolution over time of what is now called a Taylor-Green vortex.

With the aid of a computer algebra system, we have duplicated Taylor and Green's efforts and obtained more detailed time-series results. We have extended their approximation of the energy dissipation from order 5 in time to order 14, including the variation with viscosity.

Rather than attempting additional interpretation of results for fluid flow (for which, see papers by Brachet *et al*[2,3], we examine the promise of computer algebra in pursuing such problems in fluid mechanics.

1. Introduction

First we briefly review the nature of the mathematical and physical model of the Taylor-Green vortex. This is not intended as a replacement for Taylor's original study[8] or for the extensive updating of the structure of this vortex given by Brachet *et al*[2] where numerical methods are used. Such numerical methods, even with high-precision software floating-point, still has certain limitations not present in this work. We provide only a brief introduction to the problem and its notation, and move on to a discussion of the computing involved. In particular we demonstrate how the use of a computer algebra system can extend the symbolic results available.

1.1. *Background*

Since we are not experts on turbulent flow, we begin by quoting a recent account of the rationale behind continuing to study this problem (from Brachet[2]):

> The fundamental dynamical mechanism involved in homogeneous three-dimensional turbulent flows is the enhancement of vorticity by vortex-line stretching and the consequent production of small-scale eddies. This process controls the turbulent-energy dynamics and hence the global structure and evolution of the flow. A prototype of this process is given by the Taylor-Green vortex [8] denoted as TG below) which is perhaps the simplest system in which to study the generation of small scales and the resulting turbulence.

1.2. *The Vortex Model*

This section is a reiteration of the initial statement and development of Taylor's model. The initial flow of an incompressible fluid is chosen as represented by the following equations:

$$\left.\begin{aligned} u &= A\cos ax\sin by\sin cz \\ v &= B\sin ax\cos by\sin cz \\ w &= C\sin ax\sin ay\cos cz \end{aligned}\right\} \tag{1.1}$$

Using the fact that ρ, the density, is constant for an incompressible fluid, and the equation of continuity:

$$\rho_t + (\rho u)_x + (\rho v)_y + (\rho w)_z = 0$$

we can derive the following consistency equations for our model:

$$Aa + Bb + Cc = 0$$

$$u_x + v_y + w_z = 0$$

The equations of motion are:

$$\left.\begin{aligned} -u_t &= uu_x + vu_y + wu_z + P_x/\rho - \nu\nabla^2 u \\ -v_t &= uv_x + vv_y + wv_z + P_y/\rho - \nu\nabla^2 v \\ -w_t &= uw_x + vw_y + ww_z + P_z/\rho - \nu\nabla^2 w \end{aligned}\right\} \tag{1.2}$$

We solve for P, the pressure by taking its Laplacian. That is, applying $\partial/\partial x$ to the first equation of (1.2), $\partial/\partial y$ to the second, $\partial/\partial z$ to the third and summing up the three new equations. We obtain

$$-\frac{1}{\rho}\nabla^2 P = u_x^2 + v_y^2 + w_z^2 + 2(v_x u_y + w_y v_z + u_z w_x) \tag{1.3}$$

The periodic solution of this equation can be found easily. Substituting this solution and (1.1) into (1.2), we obtain an expression for u_t that can be integrated to give the first approximation:

$$u = A(1-\theta\nu t)\cos ax\sin by\sin cz + \frac{A_3}{a}t\sin 2ax\cos 2by - \frac{A_2}{a}t\sin 2ax\cos 2cz$$

where $\theta = a^2 + b^2 + c^2$, and the constants A_2 and A_3 are obtained by cyclic permutation of the letters a, b, and c from a constant A_1 where

$$A_1 = \frac{b}{4}\big(A^2 b \frac{a^2}{b^2 + c^2} + ABa\big)$$

Similar expressions are obtained for v and w. To obtain the next approximation, we repeat this process by replacing (1.1) with the newly calculated expressions for u, v and w.

1.3. *Pursuing the General Case*

Deriving the successive approximations indicated in the previous section quickly become intractable. While the second approximation has 11 terms of the form $P(t) \cdot \cos lax \sin mby \sin ncz$ the third has over 500 terms, and the fourth over 3000.

"Putting this on the computer" by means of suitable programming is the topic of this paper. However, we wish to emphasize the computer algebra approach of series expansions with arbitrary parameters. Instead of particular numerical solutions for one point in parameter space, one can argue that the computer algebra approach is more physical and that more information can be extracted from such results.

Having decided to use computer algebra, there is still an interesting point of mathematical style that distinguishes the computer algebra approach from that of classical manipulation. Taylor and Green preferred to express their solutions in terms of several parameters and a substantial number of auxiliary variables invented for convenience in printing and manipulation. These are defined through side relations and symmetries (such as the A_i of the last section). By contrast, when a brute force computation scheme is set up, the presence of any extra parameters in the formulation typically requires additional computation and storage. As for additional "intermediate" variables, the computer programs are not ordinarily written to "make up" such names. Indeed, if any such variables are designated by the computer, they tend to be out of a kind of desperation in printing large expressions, rather than by arguments of symmetry. In our particular example, if one knows that $A + B + C = 0$, then the computer program likely proceeds more efficiently by replacing C by $-A - B$ at the outset. In a similar vein, while Taylor and Green use $\sin(ax)$ etc. for scaling the frequency explicitly, if we can get away with setting $a = 1$ with an implicit re-scaling by stretching the axis, we save more time and space.

The full calculation with all variables can proceed with a program available from the author (a small variation on the one displayed in the appendix). This permits arbitrary A, B, and C, as well as a, b, and c. The cost of this generality over the special case given in the next section is considerable.

To make a judgment, we will use as a benchmark the time for computing a particular summary statistic, the mean rate of energy dissipation in a fluid:

$$\overline{W} = \mu\overline{(\xi^2 + \eta^2 + \zeta^2)} = \mu\overline{\omega^2}$$

where ξ, η and ζ are the three components of vorticity $\omega = \text{curl } \mathbf{v}$. For this problem,

we can calculate $\overline{W}/\mu$ using the following formula:

$$\frac{\overline{W}}{\mu} = \overline{u_x^2} + \overline{u_y^2} + \overline{u_z^2} + \overline{v_x^2} + \overline{v_y^2} + \overline{v_z^2} + \overline{w_x^2} + \overline{w_y^2} + \overline{w_z^2} \tag{1.4}$$

where

$$\overline{u_x^2} = \frac{1}{\pi^3} \int_0^\pi \int_0^\pi \int_0^\pi u_x^2 dx\ dy\ dz$$

etc. Using arbitrary a, b, c as well as arbitrary A, B, and C, the computation seems impractical beyond t^3, a computation of about 4.4 minutes. This is how far Taylor and Green computed the general result by hand. On t^4, we exhausted memory after two hours*. Using $a = b = c = 1$, but arbitrary A, B, and C, the computation to order t^4 took about 3.6 minutes. Using $A = 1$, $B = -1$ and $C = 0$, as in the next section, reduces the time to 50 seconds.

1.4. *A Special Case*

Again following the lead of Taylor and Green, we simplify the problem so we can more easily compute the series further, and simultaneously standardize our results so we can compare them to published literature. Using the final parameter settings of the previous section, but taking advantage of symmetries observed by Taylor and Green, further reduces the computation time from 50 to 8.5 seconds. We also take some care to eliminate from the computation certain terms that we know will not contribute to the result to a given target order in t. Using these ideas, Taylor and Green completed this special computation by hand up to t^5, something that takes us 22 seconds. However, we were able to extend it considerably further: to t^{14}.

2. The Program

2.1. *Overview*

In writing a program to re-do a calculation, it is tempting to try to duplicate in detail all the manipulations that (we hypothesize) that Taylor and Green pursued in the form they carried out the algebra. This paper is somewhat more explicit than most in describing their algebra, but unfortunately, they used slightly different methods for each approximation. This minimized the human labor of keeping track of the terms. We imagine them using large sheets of paper, tabular listings, blackboards, and other record-keeping aids to control the dozens and eventually hundreds of terms. Duplicating this approach (but doing the raw algebra by computer algorithms) would be possible but would be error prone. Since programs rarely work the first time, it is hard to debug a complicated program if it is used once, and that is what we would be doing for each order. If we use the same method for each extension, we might be more confident in the program's correctness.

Another possible approach to massive algebraic computation is to analyze the structure of the problem to the point at which all dependence on symbols is implicit

*on a 200Mhz Pentium II computer with 64 megabytes of RAM, running an optimized version of Macsyma 2.2 using Allegro Common Lisp.

in the storage and sequence of operations, and the results require only numerical manipulation. In such a case one can (usually with substantial analysis and programming effort) convert the calculation to Fortran or some numerically-oriented language. Brachet *et al*[2] as well as Pelz and Gulak[4] pursued this path. In so doing, they made the computation extremely specific to the problem as analyzed, with settings of parameters and symmetries "cast in stone" before the computation. An additional substantial concern in such an approach can be assuring the precision of the arithmetic available suffices provide accuracy of the numerical calculations. In fact, Brachet offered an *ad hoc* estimate of the delicacy of floating-point accuracy, which our evidence supports as correct.

The third approach, and the one we followed, is to consider fitting the calculation into an appropriate form for efficient symbolic computation. That is, we re-use standard efficient methods, not specifically developed for this problem. In general one must find an efficient data model for the problem: mapping it into canonical representations that can be easily manipulated by already-existing routines. Even if these representations are not an exact fit for the computation and may consequently "waste" some time or space, the advantage to the mathematician/programmer may be quite substantial: human time and effort is saved; and the results may be extended to higher order by simple extensions of the canonical method rather than reprogramming to go to the next step. In this approach the division of labor between human and computer is appropriate: Computers are not averse to grinding away using some brute-force method as long as the computation time required is available, whereas the relative uniformity of the program in a reasonably high-level language makes it easier for a human to write and check.

We first tried reproducing this particular calculation by computer in 1971. Using a naive approach on a machine that was quite puny by today's standards, it became apparent that a more sophisticated reformulation was needed. This eventually led to representing all non-trivial expressions as so-called *Poisson series*. That is, expressions are written in the form of

$$\sum_i p_i \left\{ \begin{matrix} \sin \\ \cos \end{matrix} \right\} (a_i x + b_i y + c_i z) \tag{2.1}$$

where a_i, b_i, c_i are small positive or negative integers, and p_i is a coefficient that may be (for example) a polynomial or, as in our case, a power series in other variables. Thus a typical term from Taylor's paper initially written as $A \cos ax \sin by \sin cz$ is transformed, for computational purposes into the Poisson series

$$-\frac{A}{4}\cos(ax+by+cz)-\frac{A}{4}\cos(-ax+by+cz)+\frac{A}{4}\cos(ax-by+cz)+\frac{A}{4}\cos(-ax-by+cz)$$

This form admits of a particularly compact representation where the trigonometric arguments are bit-packed into a vector (perhaps a machine word or several words).

It is interesting to contrast the approach in Brachet [2] where forms for the flow were encoded as 4-D arrays of coefficients implicitly representing triply-nested sums of products of two cosines and one sine.

First, they reformulated the initial conditions slightly for symmetry,

$$\left.\begin{aligned} v_x(\mathbf{r}, t=0) &= \tfrac{2}{\sqrt{3}} \sin\left(\theta + \tfrac{2\pi}{3}\right) \cos y \cos z \\ v_y(\mathbf{r}, t=0) &= \tfrac{2}{\sqrt{3}} \sin\left(\theta - \tfrac{2\pi}{3}\right) \cos y \cos z \\ v_z(\mathbf{r}, t=0) &= \tfrac{2}{\sqrt{3}} \sin\theta \cos y \cos z \end{aligned}\right\} \tag{2.2}$$

where for the TG flow, $\theta = 0$. This preserves the remarkable quality of TG in that the initial conditions specify two-dimensional streamlines but the flow is three-dimensional for all $t > 0$.

Brachet specified a solution for the flow that corresponds to

$$\begin{aligned} v_x(\mathbf{r}, t) &= \textstyle\sum_{m=0}^{\infty} \sum_{n=0}^{\infty} \sum_{p=0}^{\infty} u_x(m, n, p, t) \sin mx \cos ny \cos pz \\ v_y(\mathbf{r}, t) &= \textstyle\sum_{m=0}^{\infty} \sum_{n=0}^{\infty} \sum_{p=0}^{\infty} u_y(m, n, p, t) \cos mx \sin ny \cos pz \\ v_z(\mathbf{r}, t) &= \textstyle\sum_{m=0}^{\infty} \sum_{n=0}^{\infty} \sum_{p=0}^{\infty} u_z(m, n, p, t) \cos mx \sin ny \sin pz \end{aligned} \tag{2.3}$$

where $\mathbf{u}(m, n, p, t)$ vanishes unless m, n, and p are either all even or all odd integers. In their calculation, each component was ultimately a high-precision floating-point number in a 4-dimensionally indexed structure. To be faithful to our intentions of carrying along the viscosity information, if we were to duplicate this, we would need to store a polynomial in ν, of degree $\max(m, n, p) - 1$ in each cell, as well as support a version of the FFT over such a coefficient domain.

By contrast, the advantage of using a computer algebra system is that the program is considerably shorter and more perspicuous: it looks much like the mathematical formulation. Furthermore, we can do exact coefficient computations without fear of roundoff error. We also gain by using a Poisson series form that provides substantially better compactness than the general tree-style representation of an algebraic expression in a computer. This form uses far less memory internally, and also supports efficient routines for taking derivatives, integrals, and the multiplication of series. Because these operations are kept within a closed domain of Poisson canonical expressions, their results are kept simplified at all times.

In this particular example (1.3) can be solved easily since

$$\nabla^2 A \cos(ax + by + cz) = A(a^2 + b^2 + c^2) \cos(ax + by + cz)$$

The energy dissipation (1.4) can be computed easily since

$$\overline{\left(\sum A(t) \cos(ax + by + cz)\right)^2} = \sum A^2(t) \overline{cos^2(ax + by + cz)} = \frac{1}{2} \sum A^2(t) \tag{2.4}$$

2.2. *Details*

As in many other computational matters, the devil is in the details. A totally brute-force method may not work because an explosive growth of terms can force the computational effort beyond available resources.

In addition to using the Poisson representation, we further optimize the program by trimming off the terms that are not needed for computing the energy. Suppose that we want to compute the energy dissipation to the t^s term.

This is computed by summing the squares of coefficients in a Poisson series (2.4), a typical coefficient Q being a Taylor series in t and ν computed to order h. The coefficient Q certainly need not be computed to any higher order in t than s at any point, and this suggests a truncation that leads to some savings. If Q were allowed to grow as a polynomial in t and ν it would grow as n^2 at the n step. As described by Taylor and Green, one can truncate to an even tighter series if one can be sure that Q has no constant term, e.g. $Q = q_1(\nu)t + q_2(\nu)t^2 + \cdots$. In such a case one need only retain terms to t^{s-1} because in the penultimate step, squaring to compute the energy dissipation, the highest term in Q would be multiplied by t^1 or a higher power. If terms of t^s or higher were retained, upon squaring these would become terms in t^{s+1} or higher: these have no effect on the t^s approximation. In one approach to optimization one can compute an index k for each coefficient such that $Q = t^k(p_k + p_{k+1}t + \cdots ... + p_{s-k}t^{s-k})$ (with $p_0 \neq 0$), retaining far fewer than $s+1$ terms in general. The complexity of characterizing such terms in the Poisson representation was tricky, and we ended up ignoring this optimization.

Independently, it is easy to show by induction that in computing approximations to $\mathbf{u}$, the new terms have a minimum power of t associated with them. The terms look like $t^{k_j}B_j(t)\sin(a_jx+b_jy+c_jz)$ where $k_j \geq m_j - 1$ and $m_j = \max(|a_j|, |b_j|, |c_j|)$. We can use this fact to drop terms whose coefficients are necessarily of too high a degree.

Without such optimizations the program bogs down rather spectacularly; We tried removing all such speedups and wrote a totally naive program directly mimicking the mathematics. We found it could not complete in one hour what the optimized program could do in two seconds.

In the process of trying to push this calculation as far as possible, we examined in some detail yet other optimizations based on floating-point arithmetic. Although we ultimately did not use this technique in the work in this paper, graduate student Phil Liao pursued this approach and revealed a subtlety not present in exact computation following the same algorithm. If a coefficient term that is in fact exactly zero is computed as 10^{-19} (where other coefficients are typically 10^1 to 10^{-5}) then the symbolic terms that are attached to such small coefficients will be retained instead of being dropped, and enter into the computation further. Thus the "cost" of a rounding error is not limited to deflecting a computed number, one which would be computed anyway, slightly away from its correct value. Instead, the cost is much higher: we suffer the retention of an additional term to be manipulated. This can represent a substantial computational cost, especially as the intermediate algebra grows (exponentially, in most cases).

Thus it is very valuable to be able to deduce, either *a priori* by symmetry or other arguments that certain terms must be zero and need not be computed, or that terms which are below some threshold are necessarily zero and should be chopped off. An alternative that we implemented is to *probabilistically estimate the "zero-ness" of a small term by the use of finite field arithmetic.* That is, we associate with each floating-point numerical coefficient the exact value the coefficient would assume if all computations were done modulo some prime P. The integers modulo P form an algebraic field that supports rational operations, which is all that we are

doing here. If the real answer were 0, the finite-field computation would necessarily be 0 as well. There is a chance that a number computed in this finite field as 0 is in fact some non-zero multiple of P, and it is only its modular image that is 0, not its actual value. The probability is small, though how small depends on the sequence of operations and the size of P. We can improve our protection against error by using another prime P' and repeating the calculation modulo P' as well. A crude estimate, assuming that operations continue to distribute values throughout the field, would be that the probability of a "false zero" appearing, given that P is a 31-bit prime, is about $4.6 \cdot 10^{-10}$. Using two primes, we would estimate about $2.1 \cdot 10^{-19}$. If we make this check only for numbers that, as floating-point values, are already much small numerically than some norm of the expressions in which they appear, the chance of error is further reduced.

Given the additional programming and testing complexity, and the need, ultimately, for software high-precision "bigfloats" rather than hardware floats, we returned to the simpler and, for our purposes here, still adequately-fast exact arithmetic.

3. Results

The computation of the energy dissipation "by hand" was accomplished by Taylor and Green to 5th order. This was, we believe, a monumental task, although following the tradition of mathematical publishing, the effort is not mentioned explicitly in the paper. Our 1971 attempt to duplicate this used about 5 CPU hours on a PDP-10 computer with about 1.2 megabytes of memory. We recently duplicated this computation on a 200Mhz Pentium P6 computer with 64 megabytes of RAM, running an optimized version of Macsyma 2.2 using Allegro Common Lisp 4.3 as a base. Taylor's computation to t^5 took about 22 seconds.

We then extended the results to orders 6, 7, and 8 in times of about 2, 3.7 and 20 minutes, respectively. Jumping to order 12 took 16:14 (hours:minutes) and to order 14, 90:27. This last computation used a maximum of 39 megabytes of RAM.

The change in computer time from the general case to the more special case of previous sections is related to several apparently minor programming changes: taking advantage of symmetry, reducing the number of variables in the coefficients: instead of polynomials in ν, we had polynomials in ν, A, B, and C.

In the interests of finding ever more terms, or reproducing those we had computed but at a decreased cost, we sought ways to simplify the default arithmetic on exact rational coefficients with very long numerators and denominators. One approach required switching to double-precision floating-point arithmetic. While this sped up the computation by a factor of two, comparison with exact computation showed that some coefficients were contaminated by roundoff, and terms that should have been zero in the intermediate forms were not. Of all the advantages of using floating-point data, the one that seemed to dominate was the compactness of the print form of the floating-point numbers, compared to looking at the ratio of two hundred-digit integers.

The coefficients given in Equation (3.1) below confirm equations (46) and (47) in Taylor[8] up to order 5, and extend it substantially. For the special case of inviscid

flow displayed in table 5 of Brachet[2] we also confirm their first few terms up to t^{14}; By comparison, Brachet's table extends to much higher powers (t^80) but that table omits all dependence on ν, setting it to 0.

Although we have computed all the terms of the expansion as exact rational numbers, some of the coefficients are over 80 characters long, and the floating-point version is more readable. The exact version is available from the author; here we show the first few and last few computed:

$$\begin{aligned}
&\frac{\overline{W}}{\mu} = \\
&= \frac{3}{4} - \frac{9}{2}\nu t + \left(\frac{5}{64} + \frac{27}{2}\nu^2\right) t^2 + \left(-\frac{5}{4}\nu - 27\nu^3\right) t^3 + \cdots \\
&\quad \left(\cdots + \frac{4260702167694743128 37\,\nu^8}{1717910498304000} + \frac{13017506311230613\,\nu^{10}}{19978358400}\right. \\
&\quad \left. + \frac{481982151269\,\nu^{12}}{27243216} + \frac{118098\,\nu^{14}}{175175}\right) t^{14}
\end{aligned} \tag{3.1}$$

Next we provide this result in double-precision floating-point. Note that the only computation done in floating-point was the final division of two exact integers for printing. Indeed, as noted above, executing this whole computation using double-precision for intermediate calculations shows some significant departures from the correct values below. Here then is a value for $\frac{\overline{W}}{\mu}$:

$$\begin{aligned}
&0.75 - 4.5\,\nu\,t + 0.078125\,t^2 + 13.5\,\nu^2\,t^2 - 1.25\,\nu\,t^3 - 27.0\,\nu^3\,t^3 \\
&+0.0059185606060606606\,t^4 + 9.557291666666666\,\nu^2\,t^4 + 40.5\,\nu^4\,t^4 \\
&-0.1922940340909091\,\nu\,t^5 - 47.5625\,\nu^3\,t^5 - 48.6\,\nu^5\,t^5 \\
&-2.784360642534798\cdot 10^{-4}\,t^6 + 2.8573626893939394\,\nu^2\,t^6 + 175.17013888888889\,\nu^4\,t^6 + 48.6\,\nu^6\,t^6 \\
&-0.00378606084312667\,\nu\,t^7 - 27.138546176046177\,\nu^3\,t^7 - 512.25\,\nu^5\,t^7 - 41.65714285714286\,\nu^7\,t^7 \\
&+6.104820966212543\cdot 10^{-5}\,t^8 + 0.30972541546704224\,\nu^2\,t^8 \\
&+189.76911356872296\,\nu^4\,t^8 + 1243.218998015873\,\nu^6\,t^8 + 31.242857142857144\,\nu^8\,t^8 \\
&-0.0029308018157472 84\,\nu\,t^9 - 6.603690029327987\,\nu^3\,t^9 - 1056.1445404566498\,\nu^5\,t^9 \\
&-2581.133267195767\,\nu^7\,t^9 - 20.82857142857143\,\nu^9\,t^9 \\
&-9.636134093101083\cdot 10^{-6}\,t^{10} + 0.09173564793613308\,\nu^2\,t^{10} + 86.540815544480 11\,\nu^4\,t^{10} \\
&+4912.313357483566\,\nu^6\,t^{10} + 4686.1221009700175\,\nu^8\,t^{10} + 12.497142857142856\,\nu^{10}\,t^{10} \\
&+2.809251289033533\cdot 10^{-4}\,\nu\,t^{11} - 2.077271456546748\,\nu^3\,t^{11} - 838.4713473274443\,\nu^5\,t^{11} \\
&-19734.583287884252\,\nu^7\,t^{11} - 7564.562433862434\,\nu^9\,t^{11} - 6.816623376623377\,\nu^{11}\,t^{11} \\
&+1.2376451325674272\cdot 10^{-6}\,t^{12} + 0.002114184073730675\,\nu^2\,t^{12} + 35.21107616269811\,\nu^4\,t^{12} \\
&+6537.562421814222\,\nu^6\,t^{12} + 70094.23212559852\,\nu^8\,t^{12} + 10999.573964178826\,\nu^{10}\,t^{12} \\
&+3.4083116883116884\,\nu^{12}\,t^{12} \\
&-5.75664300410366\cdot 10^{-5}\,\nu\,t^{13} - 0.30948436405964\,\nu^3 t^{13} - 468.753395585854\,\nu^5\,t^{13} \\
&-43094.1764469689\,\nu^7\,t^{13} - 223907.350033079\,\nu^9\,t^{13} - 14558.9748316498\,\nu^{11}\,t^{13}
\end{aligned}$$

$$
\begin{aligned}
&-1.57306693306692\,\nu^{13}t^{13}\\
&-1.3001673399488\cdot 10^{-7}\,t^{14}+0.00240077640406\,\nu^{2}t^{14}+9.80925915972072\,\nu^{4}t^{14}\\
&+5127.6476628604\,\nu^{6}t^{14}+248016.539388598\,\nu^{8}t^{14}+651580.377656585\,\nu^{10}t^{14}\\
&+17691.8228474934\,\nu^{12}t^{14}+0.67417154274296\,\nu^{14}t^{14}
\end{aligned}
\tag{3.2}
$$

It would seem that if we took Brachet's path and set ν to 0, we would save considerable time and space. In fact, much of the busywork in maintaining data structures is the same, although the entries in the structure are no longer Taylor series in t and ν, just series in t. The savings for setting $\nu = 0$ is that the 12th order computation took 5 hours and 53 minutes, constituting about 36 percent of the time needed for carrying ν along as well.

4. Further notes on the computation

Computer algebra systems (CAS) such as Macsyma[9] have become more popular in recent years as powerful desktop workstations capable of running such programs have become widespread. The sales literature for these programs seems to suggest that one would use a CAS primarily for access to advanced algorithms such as finding closed-form integrals or perhaps factorizations of polynomials. A second use, as shown here, is *massive repetitive algebraic manipulation*, where speed and the ability to manipulate expressions with many thousands of algebraic terms and trigonometric terms, is the primary requirement.

In many computer applications, standard numeric computations can be extended to the point of swamping even the largest computers simply by demanding overly-large or overly-detailed results. In algebraic calculations it is even easier to swamp a computer because the units of representation are not limited to objects of pre-determined numeric size (e.g. 64-bit floating-point numbers), but can grow as the calculation proceeds. Furthermore, the time for multiplying two "expressions" is not correlated with megaflops, but instead depends on the size and complexity of the expressions, and may not involve any floating-point operations at all. Even if the end result is small, as is almost inevitable if it is to be comprehensible to humans, the intermediate expressions and the times to manipulate them can be huge. The cost of going from one order to the next in this computation using exact rational arithmetic can be seen to be quite substantial and growing.

In this example, as in others we have encountered, the choice of the right representation can mean the difference between failure and success. In this case, the efficiency of the Poisson series representation in Macsyma was critical. In similar computations,the introduction of floating-point representations for the coefficients in our Taylor series may result in another savings in time, partly by eliminating the need to reduce unwieldy fractions to lowest terms, and partly by reducing the storage requirement. If we were pressing the limits of resources, we believe this combination of representations could better take advantage of parallel computers. The bulk of the computation time was spent in the multiplication of Poisson series. This operation can be performed in an "embarrassingly parallel" fashion, with an

expected nearly-linear speedup as additional processors are made available. That is, we expect that for this critical operation, a nearly n-fold speedup is attainable by using n processors rather than one[†]

We should also mention that Poisson series in Macsyma are more general than is evident from our example. In particular, the coefficients of the Poisson series here were Taylor series in t (time); and the coefficients in these series had only one other parameter ν, and were themselves either exact rational numbers or floating-point numbers. Instead of Taylor series, it is plausible that one can use rational functions, partial fraction expansions, Padé approximates, or other forms for coefficients, so long as the representations are closed under the (rational arithmetic or differential) operations used.

5. Further work

The production of series leads to several lines of questions. Here we have series, in two variables, and presumed to be asymptotic in nature. The truncated series is certainly not convergent for all time, and so one might reasonably ask for estimates of the radius of convergence, perhaps some other expressions formed by analytic continuation, and other justification of the validity of this approach, perhaps by direct numerical simulation. Computer algebra systems have routines for assisting in the investigation of radii of convergence through generation of Padé approximates. These rational approximations can be examined to find the zeros of the denominators, giving an estimate of a plausible radius of convergence. Starting with the expressions here, we can generate (somewhat speculative) conclusions regarding the convergence of the series (3.1) in time, and the likely presence of singularities on the real time line, an issue whose importance is discussed at great length by Brachet[2] (and in a similar situation, Pelz[4]). For example, if we set $\nu = 0$ and look at all of the zeros of the denominators of the twelve-term expression whose argument is in $[-\pi/4, \pi/4]$ we find a pair of zeros at distance from the origin 2.7719 in the complex plane. If one insists on real zeros, the closest ones in this sector are at $t = 2.07069$. Another way of gauging the location of poles is to consider zeros of the inverse of the Taylor series for (3.1). This has zeros in a circle around the origin at a distance between 2.602 to 2.821, and a real positive zero at 2.753, suggesting a pole of the original expression may be near there.

We believe we can extend this series somewhat further simply by letting our workstation run longer; we used only about 39 megabytes to t^{14} on a system configuration that today (1998) costs about $1000 for hardware. An approach using parallel computers to manipulate the "embarrassingly parallel" Poisson series could cut this time substantially. Alternatively one could consider the a specific reformulation the methods of Brachet[2], or for that matter, a numerical simulation which might be valid for larger t in any case.

While the search for real finite time singularities in this particular problem may be a dead issue (the belief being there are none), in general the approach from series

[†] This does not translate directly into an n-fold speedup of the whole program: If the multiplications take 90% of the time, even an infinite speedup of this operation would yield only a factor of 10 total speedup.

expansion and Padé approximates can give additional perspective on complex time singularities.

6. Conclusion

What have we learned from this exercise?

1. We confirmed Taylor and Green's results, as far as they computed them.

2. We can reproduce this result in less than 30 seconds of computer time.

3. We can extend their results to far more terms, and plot results easily for different values of the perturbation parameters, for example, as surface plots of functions of t and ν.

4. Although we have not done so in this report, we, and others using this program, can re-examine the premises of the model and its simplifications, recompute the energy dissipation, or other quantities of interest.

5. We believe that other similar perturbation models of explicit symbolic computation in fluid mechanics, previous ignored because of the monstrous algebraic manipulation required, can be fruitfully re-examined using similar symbolic methods.

6. While some careful feeding and attention has paid off in this effort to get the most out of a computer algebra system, we hope that examples of computations such as this can suggest to investigators choices for mapping their mathematical models into plausible representations. We hope that the published ideas here will encourage others to adopt this general approach.

We realize that extending calculations of this model, originally chosen for its simplicity, may not represent an advance in the current knowledge of vortices. Yet it can continue to serve as a check on numerical methods or other simple computer algebra programs.

If computer algebra is to contribute to our understanding of fluid mechanics, we must show that from these simple (yet technically challenging) examples, we can generalize techniques for the algebraic computation of approximate solutions to other more pressing related mathematical problems in this or other domains. We would find it exciting to see such methods considered more seriously as an alternative to direct numerical methods, to provide more delicate insights into the behavior of fluids.

In the current scientific environment in which numerical simulation has all but taken the place of physical experimentation in some fields, and in which questions arise as to whether a phenomenon has physical relevance or is perhaps an artifact of numerical roundoff or truncation error, some mathematical algebraic confirmation may provide a welcome relief from otherwise knotty problems.

For many investigators taking advantage of the current advanced state of numerical methods for high performance computers, we believe it is more likely that

computer algebra will not be used as an end in itself to produce symbolic expressions, but as a complementary technique to developing more efficient programs more rapidly and with higher confidence of correctness.

7. Acknowledgments

Graduate student Hsin-Chao Phil Liao assisted in programming, finding bugs, timing some test runs, developing the finite-field coefficient routines to assist in detecting floating-point zeros. Liao also plotted the results of some earlier computations. Dr. Theodore Einwohner provided useful advice and located some valuable references. Prof. Richard Pelz and Y. Gulak of Rutgers provided additional insight into vortex calculations and the search for singularities.

References

1. G. K. Batchelor ed., *The Scientific Papers of Sir Geoffrey Ingram Taylor*, vol. II, Cambridge University Press, 1960.
2. Marc E. Brachet, Daniel I. Meiron, Steven A. Orszag, B.G. Nickel, Rudolf H. Morf, Uriel Frisch, "Small-scale structure of the Taylor-Green vortex," J. Fluid Mech. (1983) vol 130 pp. 411—452.
3. Marc E. Brachet, Daniel I. Meiron, Steven A. Orszag, B.G. Nickel, Rudolf H. Morf, Uriel Frisch, "The Taylor-Green Vortex and Fully Developed Turbulence," J. Statistical Physics (1984) vol 34 pp. 1049—1063.
4. R.B. Pelz, Y. Gulak, " Evidence for a real-time singularity in hydrodynamics from time series analysis," Phys. Rev. Letters, vol.79, no.25 (Dec. 1997) pp. 4998—5001.
5. G. I. Taylor, "On the Dissipation of Eddies," *Reports and Memoranda of the Advisory Committee for Aeronautics*, no. 598, 1918, reprint[1].
6. G. I. Taylor, "Statistical Theory of Turbulence, Part I," *Proceedings of the Royal Society*, A, vol. CLI, pp. 421—44, 1935, reprint[1].
7. G. I. Taylor, "Statistical Theory of Turbulence, Part II," *Proceedings of the Royal Society*, A, vol. CLI, pp. 444—54, 1935, reprint[1].
8. G. I. Taylor and A. E. Green, "Mechanism of the Production of Small Eddies from Large Ones," *Proceedings of the Royal Society*, A, vol. CLVIII, pp. 499—521, 1937, reprint[1].
9. *Macsyma: Mathematics and System Reference Manual*, 16th ed., Macsyma, Inc., 1996.

Appendix A

An electronic version of the program below, as well as several variants, can be obtained from the author, email `fateman@cs.berkeley.edu`.

```
/* Taylor-Green vortex calculation in Macsyma, February 1998.
Many of the detailed specifications below are motivated not
by mathematical or even computational necessity, but by a
quest for efficiency in computation time or space. */

/*Declare that we will be using "rational variables" named
   nu,  t.  The ordering of these variables in displayed
```

```
  output is specified by this list.  In particular  t is the main
  variable, while u is more constant.  Thus we would display
  u*t^2 ++ u^2*t +u^2+u+1  rather that (t+1)*u^2+ (t^2+1)*u +1
*/
 /* advice for compiling */
declare(t,special)$ declare(lim,special)$declare(n,special)$
ratvars(nu,t)$ ratweight(t,1)$ ratweight(nu,0)$
[ratexpand:true, powerdisp:true]$

/* Declare the angle variables in the Poisson series package: x,y,z.
For computing symmetries we allow for an additional variable w. */

 poisvars:[w,x,y,z]$ poislim:60$

/* In order to make our programs more compact, we define certain
 infix operations as shorthand for built-in functions that
 manipulate Poisson series.  We also specify the precedences
of these infix operations. */

(".+."(x,y):=poisplus(x,y),  infix(".+.",100,100,expr,expr),
 ".*."(x,y):=poistimes(x,y), infix(".*.",120,120,expr,expr))$

/* Some utility functions:
Omega fun converts u,v,w into W/mu: it computes
        u_x^2 bar + u_y^2 bar + ... .
trimit, poistrim, shorten: remove terms that are not going to cause
a change in the dissipation (see TG paper) */

omega(m):=block([res:0],poismap(m,omegafun,omegafun),res)$
omegafun(co,arg):=
  block([],
        res:res+co^2/2*(coeff(arg,x)^2+coeff(arg,y)^2+coeff(arg,z)^2),
        return(0))$

/*   tfun(co,arg) is used in computing the inverse of the Laplacian,
      eq (45) in Taylor/Green */

tfun(co,arg):= co/(coeff(arg,x)^2+coeff(arg,y)^2+coeff(arg,z)^2)$

shorten(co,arg):= /*simply convert to rational form and truncate */
 rat(co)$

/* trimit is a function that used with poismap.
  It removes terms of too high a frequency from being
  carried along in the computation. */
```

```
trimit(co,arg):=
 block([a:abs(coeff(arg,x)),b:abs(coeff(arg,y)),c:abs(coeff(arg,z))],
  if (max(a,b,c) > lim+1-n) then return(co) else return(0))$

[v[1]:x, v[2]:y, v[3]:z]$

/* By symmetry we do not have to really compute the y component, just
manipulate the x component. Makesym, given a series with terms like
sin(ax+by+cz) makes a new series with ax+by+cz changed to ay-bx+cz*/

makesym(m) := poissubst(-x,w,poissubst(y,x,poissubst(w,y,m)))$

/* main pgm: compute u[1,n], u[3,n] by integration and trimming */

compute(n):=
  block([p:0,u2,mx,my,mz,ratwtlvl:lim],
   for i in [1,3] do /* for u[1,..] and u[3, ..] */
     u[i,n]:u[i,0].+.(-1).*.  /* add in const of integration */
     poisint(            /*integrate du/dt wrt t */
      u[1,n-1]              .*.(mx:poisdiff(u[i,n-1],x))
        .+.makesym(u[1,n-1]).*.(my:poisdiff(u[i,n-1],y))
        .+.u[3,n-1]         .*.(mz:poisdiff(u[i,n-1],z))
        .+.(-nu).*.(poisdiff(mx,x).+.poisdiff(my,y).+.poisdiff(mz,z)),
        t),
  /* the 1/rho dP/dx term is added below*/
        p:poisdiff(u[1,n],x),
        p:makesym(p).+.p.+.poisdiff(u[3,n],z),
        p:poismap(p,tfun,tfun),
        for i in [1,3] do
          u[i,n]:poismap(u[i,n-1],trimit,trimit)
                .+.poismap(u[i,n].+.
                       poisdiff(p,v[i]),shorten,shorten),
        p: 2*omega(u[1,n])+ omega(u[3,n]),
        /*optional: wipe out old array entries to save memory */
        if n>1 then for i in [1,3] do u[i,n-1]:0,
        return (ratexpand(p)))$

/* initial vortex to order t^0 */
u[1,0]:intopois(cos(x)*sin(y)*sin(z))$
u[2,0]:intopois(sin(x)*cos(y)*sin(z))$ /*computed from symmetry, though */
u[3,0]:intopois(0)$  /* since C=0, but otherwise would be sin sin cos */

poistrim(u,a,b,c) :=
  block([m],
```

```
        mododeclare([u,a,b,c,m],fixnum),
        m:max(abs(a),abs(b),abs(c)),
        if 2*(m-1)>lim or m>(lim+1-n) then
          return(true), /*trim it*/
        return(false))$

om(low,lim) := /* low is 1 unless restarting from a checkpoint.
                  lim is the maximum order for t */
        for i:low thru lim do omeg[i]:compute(i)$

compile(shorten,poistrim)$ /*speed up these functions by compiling */
/* to run TG's computation, type om(1,5);
   for our longest run,  type om(1,14);
   to set a specific value for nu, say nu=0, just type nu:0; */
```