Tensoral: a system for post-processing turbulence simulation data

By Eliot Dresselhaus

1. Motivations and objectives

1.1 General motivations

Many computer simulations in engineering and science — and especially in computational fluid dynamics (CFD) — produce huge quantities of numerical data. These data are often so large (consider the roughly 1 Gbyte needed for a single scalar variable in 512^3 isotropic turbulence simulations) as to make even relatively simple post-processing of this data unwieldy. The data, once computed and quality-assured, is most likely analyzed by only a few people (usually only the simulation's authors) and from at most a few perspectives (usually only those at which the authors are most concerned and knowledgeable). As a result, much useful numerical data is under-utilized. Since future state-of-the-art simulations will produce even larger datasets, will use more complex flow geometries, and will be performed on more complex super-computers (for example, super-computers with many loosely coupled processors), data management issues will become increasingly cumbersome.

My goal is to provide software which will automate the present and future task of managing and post-processing large turbulence datasets. My research has focused on the development of these software tools — specifically, through the development of a very high-level language called "*Tensoral*". The ultimate goal of *Tensoral* is to convert high-level mathematical expressions (tensor algebra, calculus, and statistics) into efficient low-level programs which numerically calculate these expressions given simulation datasets. For example, a user's program to calculate vorticity would be coded in *Tensoral* as something akin to $\vec{\omega} = \nabla \times \vec{u}$. *Tensoral* would process this "program" — at least for the case of homogeneous turbulence on the Cray Y-MP — into a roughly 200-line *Vectoral* program to calculate vorticity.

This approach to the database and post-processing problem has several advantages. Using *Tensoral* the numerical and data management details of a simulation are shielded from the concerns of the end user. This shielding is carried out without sacrificing post-processor efficiency and robustness. Another advantage of *Tensoral* is that its very high-level nature lends itself to portability across a wide variety of computing (and super-computing) platforms. This is especially important considering the rapidity of changes in supercomputing hardware.

1.2 Specific motivations and objectives

The fundamental scientific goal of fluids research is to reach an understanding of the correlation between the Navier–Stokes equations

$$\partial_t \vec{u} + (\vec{u} \cdot \nabla)\vec{u} = -\nabla p/\rho + \nu \nabla^2 \vec{u}$$

(whether incompressible or compressible) and the observed and theoretically predicted features of the velocity field $\vec{u}(\vec{x},t)$, the pressure field $p(\vec{x},t)$, and other related quantities. $\vec{u}(\vec{x},t)$ is the fundamental hydrodynamic quantity: all other quantities, both dynamic and statistical, are derived from it (at least for incompressible flows). Turbulence theory, modeling, and experiments are all phrased in terms of quantities derived from the velocity field $\vec{u}(\vec{x},t)$. The quantities arising in this theory, modeling, and experiments are precisely the ones we desire to compute; they include:

- the vorticity vector field, $\vec{\omega}(\vec{x},t) = \nabla \times \vec{u}(\vec{x},t)$,
- the strain rate tensor, $S_{ij} = (\partial_i u_j + \partial_j u_i)/2$,
- the pressure scalar, $p(\vec{x},t) = -\nabla^{-2}\ \nabla \cdot (\vec{u} \cdot \nabla)\vec{u}$ (for incompressible flows),
- the kinetic energy dissipation density $\epsilon(\vec{x},t) = \nu \sum_{ij} S_{ij}S_{ij}$ (incompressible),
- the wave-space velocity field $\vec{u}(\vec{k},t)$ and associated energy spectrum $E(k) = |\vec{u}(\vec{k},t)|^2$,
- the density scalar $\rho(\vec{x},t)$ (for compressible flows),
- the stream function $\vec{\psi}$, $\nabla \times \vec{\psi} = \vec{u}$,
- the helicity density, $\vec{u} \cdot \vec{\omega}$.

Statistical quantities of interest include:

- mean velocity profiles and other averaged velocity components,
- the Reynolds stress tensor $R_{ij} = \langle u_i u_j \rangle$,
- the correlation tensor $\langle u_i(\vec{x}) u_j(\vec{x}+\vec{x}') \rangle$,
- the total energy dissipation $\epsilon \propto \sum_{ij} \langle S_{ij}S_{ij} \rangle$,
- the enstrophy (mean square vorticity) $\langle \vec{\omega}^2 \rangle$,
- the pressure strain correlation $\langle pS_{ij} \rangle$.

We desire to compute quantities such as the above using data from several families of turbulence simulations. These datasets solve either the incompressible or compressible Navier–Stokes equations for a variety of geometries and boundary conditions. Different geometries and boundary conditions imply that the velocity field is represented by different grids or with various different orthogonal function expansions (for spectral methods). Even though these datasets simulate roughly the same underlying equations, such dissimilarities in geometry and boundary conditions require dissimilar numerical methods and data management schemes. Some simulations use orthogonal functions (e.g. Fourier, Chebyshev, Jacobi eigenfunctions) to satisfy the boundary conditions; derivatives are calculated spectrally. Other simulations use finite-difference methods to calculate derivatives and are likely set in complex geometries (relative to the spectral simulations). Certain simulations use curvilinear grids. Some simulations evolve the evolution equation for $\vec{u}$; others evolve its curl, $\vec{\omega}$. Thus, some databases contain the velocity field $\vec{u}$ itself while others contain its curl. On a more mundane level, the simulations are performed on several different super-computers (Cray Y-MP, Intel Hypercube, Thinking Machines CM-2) and retain some degree of machine specificity even at the database level (e.g. machine byte-order, floating point format, machine-specific optimized Fourier transform routines, etc.).

2.2 Current post-processing

Currently all post-processing of turbulence data is done "by hand." That is, for each simulation and for each desired quantity, someone must either add the required code to an existing post-processor or develop a specific new post-processor, perhaps with an existing one as a model. If the databases in question were small and simple, either of these options would be straightforward. Since the databases are very large and have numerical quirks to them, both options involve significant effort.

A simple example will illustrate this. Suppose we desire to calculate the physical-space pressure $p(\vec{x})$ given a wave-space space velocity field snapshot $\vec{u}(\vec{k})$ from an isotropic turbulence database (the simplest to post-process in the above table). Here is an outline of what must be done to calculate $p(\vec{x}) = -\nabla^{-2} \sum_{ij} \frac{\partial u_i}{\partial x_j} \frac{\partial u_j}{\partial x_i}$

- Read in $\vec{u}(\vec{k})$ in k_x-k_y planes and calculate necessary y derivatives in wave space (multiplying by ik_y).
- Fourier transform these derivatives from wave y space to physical y space. This is the first of three sub-transform steps that make up a full three-dimensional Fourier transform.
- Read in data in x-z planes, still in wave space, and calculate necessary x and z derivatives (multiplying by ik_x and ik_z).
- Fourier transform both x and z axes into physical space. At this point, all of the required velocity derivatives are in physical space.
- Form the source term $\sum_{ij} \frac{\partial u_i}{\partial x_j} \frac{\partial u_j}{\partial x_i}$ in physical space.
- Transform source term, now fully calculated, back into full wave space and invert ∇^2 (divide by $-\vec{k}^2$).
- Transform result back to full physical space.

Much of the complexity of this example stems from the fact that the complete velocity field is too large to fit into even a super-computer's central memory. Thus, the data must be split into "pencils" or "planes" of one or two dimensional data. For more complex databases, even more steps must be taken to perform a similar computation: for example, for certain simulations the physical space product must be dealiased on a 3/2 size grid; for others the derivatives involve Chebyshev and Fourier transforms rather than just Fourier transforms as above.

Considering the above example, one can see that a post-processor which computes many quantities can become significantly complex and inscrutable to the uninitiated. In fact, for certain simulations the post-processing software is a more complex code than the simulation code itself. This is particularly true when time and space optimization issues are important. It is important to realize, however, that these complexities can all be understood and are fairly algorithmic. In particular, it is plausible that an expert system (such as the *Tensoral* language) can be taught to *generate* code to perform the above and similar post-processing tasks.

To summarize, the post-processing of turbulence data involves performing tensor calculus and statistics on a number of dissimilar numerical dataset types. Numerical operations must be performed in a manner consistent with the simulation which generated the database. Currently post-processors are written entirely by hand and are specific not only to the simulation in question but also to one or more

particular quantities of interest. Moreover, these post-processors are quite complex codes in their own rights and provide significant barriers to the uninitiated who desire to distill scientific understanding from the myriad computational details. It is entirely plausible that this task — the creation of database post-processors — can be automated.

2. Accomplishments: tensoral design and μtensoral implementation.

2.1 Tensoral by example

The best way to introduce a new computer language is by example: suppose we desire to study the role of the pressure strain term $\langle pS_{ij} \rangle$ in the mean Reynolds stress $\langle u_i u_j \rangle$ evolution

$$\frac{\partial}{\partial t} \langle R_{ij} \rangle = 2 \langle pS_{ij} \rangle + \nu \langle u_j \nabla^2 u_i + u_i \nabla^2 u_j \rangle .$$

This hypothetical study would calculate $\langle pS_{ij} \rangle$ for various turbulence databases. To calculate $\langle pS_{ij} \rangle$ given a database file `db`, one would code the following *Tensoral* program in a file `ps.tl`:

Line 1 `A_ij = db:u_i,j`
Line 2 `S_ij = (A_ij + A_ji)/2`
Line 3 `w_k = 1/2 epsilon_ijk A_ij`
Line 4 `p = -unlaplacian(S_ij*S_ji - w_k*w_k)`
Line 5 `print "Mean pressure strain", <p S_ij>`

Line 1 defines the velocity gradient tensor `A_ij`. Lines 2 and 3 form the strain `S_ij` tensor and vorticity `w_k`. Line 4 inverts the Poisson equation for the pressure `p`. Line 5 averages the pressure strain and writes the result to the console. To complete our study, we would run this program for several database files from several different simulations.

2.2 Tensoral design: from top to bottom

Exactly *how* is the program `ps.tl` turned into an efficient post-processor to perform the indented task? The overall answer to this question is illustrated in figure 1 and is described in what follows. The *Tensoral* compiler takes the program supplied by the end database user (e.g. `ps.tl`), determines the appropriate numerical methods and data management techniques for the database file `db` (found in a "database description"), and uses this information to output a post-processor in a lower-level language (relative to *Tensoral*). This low-level post-processor is automatically generated and may in principal be any sufficiently powerful language such as *Vectoral*, Fortran, or *C*; we use the *Vectoral* language in our prototype because of several of its unique features. Finally, this low-level post-processor is compiled and combined with the requisite library routines (e.g. Fourier, Chebyshev, Jacobi transforms, Poisson solvers) to make an executable post-processor which can then be used to visualize data, to make graphs, or to transport post-processed data to other sites.

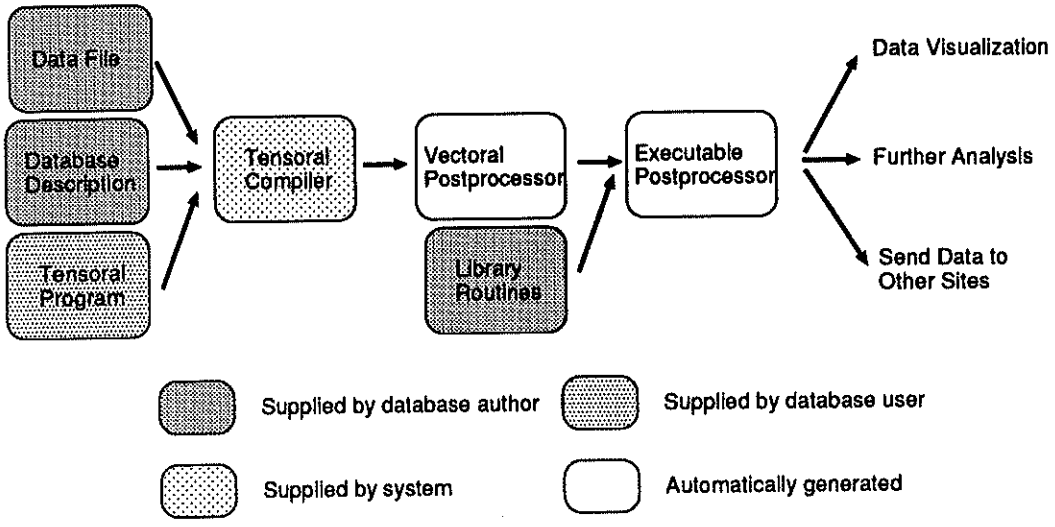

Figure 1

Clearly, the most involved step of this automated system is the generation of a low-level *Vectoral* post-processor given a high-level *Tensoral* program. This step is performed by the *Tensoral* compiler, which compiles very high-level tensor operations into a low-level "machine code" (called μ*Tensoral*). μ*Tensoral* machine code consists of the primitive database operations such as reading a pencil or plane of data into main memory, Fourier transforming that pencil or plane, etc. These primitive operations are defined in the database description. Seen this way, the *Tensoral* compiler is an expert system which figures out how to generate and order primitive μ*Tensoral* database operations to accomplish a given task. If the database description is properly coded, the compiler should be able to generate code which is nearly as efficient as that generated by hand.

2.3 Tensoral implementation: from the bottom up

Implementation of the *Tensoral* proceeds from the bottom up. That is, my goal is to develop a fully functional and robust low-level μ*Tensoral* machine code before launching into the *Tensoral* compiler development which will eventually generate this low-level code. Using this approach, the design outlined above can be proved and tuned from its foundations up. In particular, database descriptions and μ*Tensoral* codes can be written and refined as experience is gained.

Currently a functional μ*Tensoral* system has been implemented. This system is built atop a small Lisp interpreter. The description of numerical methods and data management schemes provided to μ*Tensoral* by database descriptions drive Lisp code which sets up an environment in which μ*Tensoral* code is then executed. Thus, slight differences between simulations can be conditionalized in Lisp (using `if` statements, for example) so that one database description file can actually handle

multiple related simulations. Also, using such conditionals one can write *μTensoral* programs which are common between closely related simulations. The Lisp system is only seen by those programming in *μTensoral* and writing database descriptions, and only then in the remote background. In particular, Lisp will never be seen by *Tensoral* programmers and almost never by *μTensoral* programmers.

A *μTensoral* program to calculate vorticity $\vec{\omega}(\vec{x},t) = \nabla \times \vec{u}(\vec{x},t)$, given spectral $\vec{u}(\vec{k},t)$, is coded as follows:

```
(define-tensor w 1)

; Read u in xy planes; calculate y derivatives;
; transform to physical y.
(loop-memxy
(bs->memxy u_1 u_3)
(->memxy w_3 w_1)
(= w_1 (diff_2 u_3))
(= w_3 (diff_2 u_1))
(wavey->physy- w_1 w_3)
(memxy->bs w_1 w_3))

; Read u in xz planes; calculate other derivatives;
; form vorticity.
(loop-memxz
(bs->memxz u)
(bs->memxz w)
(= w_1 (- w_1 (diff_3 u_2)))
(= w_2 (- (diff_3 u_1) (diff_1 u_3)))
(= w_3 (- (diff_1 u_2) w_3))
(wavez->physz- w)
(wavex->physx- w)
(memxz->bs w))
```

This program is essentially a high-level "shorthand" for the two passes through the database needed to calculate vorticity. It is particularly noteworthy that the roughly 20 line *μTensoral* shorthand shown here generates an approximately 180-line *Vectoral* program. This large code expansion indicates that even the embryonic *μTensoral* system can be used as a useful post-processing tool, even though it assumes a programmer be familiar with the precise numerics of a simulation (as represented by its database description).

Current work focuses on extending *μTensoral*. Currently *μTensoral* only "knows" about tensor algebra and calculus but not statistics. General statistical functions are now being added to average over combinations of x, y, and z coordinates to form correlations and probability density functions (PDFs). Thus far only one database description has been completed; others will follow.