Tensoral present and future

By Eliot Dresselhaus

1. Motivation and objectives

The coding of high-performance fluids simulations requires significant knowledge of both numerical and computational details. The magnitude and complexity of low-level details is often enough to discourage many *users* of turbulence data wishing to study more important, higher-level fluid dynamical questions. These same complexities are often a practical barrier to simulation *experts* who develop, verify and maintain the codes which generate this data. Future fluids codes, with high resolution and complex geometries, are likely to involve far more coding complexity.

My research — the design and implementation of the `Tensoral` computer language — aims to greatly ease the coding of today's simulation and post-processing codes and at the same time provide a general computational tool for future simulations.

2. The current `Tensoral`

`Tensoral` is a very high-level language. To the user seeking to analyze turbulence data, `Tensoral` speaks the language of the Navier–Stokes equation: three dimensional tensor calculus and statistics. With `Tensoral` a user can perform efficient high-level analysis of simulation data without any knowledge of the underlying numerical and computational complexities necessary to manipulate this data.

The simulation expert is responsible for teaching `Tensoral` how to realize this tensor language with executable computer code. Specifically, such an expert must code the basic building blocks of a numerical method in a `Tensoral` "back-end." A back-end defines how a fluid field is to be represented on a particular computer system and how operations (e.g. derivatives, integrals, statistics, etc.) are to be performed on this representation. Once these building blocks are in place, post-processing and simulation codes can be constructed from them using the `Tensoral` compiler.

The current implementation of `Tensoral` — described in detail in previous CTR research briefs — can generate efficient code for general computations involving the arithmetic, statistics, and calculus of numerically represented tensor fields. Currently, the only complete back-end is for isotropic turbulence. In principal, the current language and compiler is sufficient to meet the needs of today's users and experts alike. In practice, however, the current language and compiler has significant limitations.

The current `Tensoral` is not coherent. Inside `Tensoral` there is a language the user sees (tensors) and a language that experts sees (the back-end language). Coding with the current back-end language is practically difficult and conceptually obscure. This inhibits the generation of new back-ends with the current system.

The current `Tensoral` is not modular. Back-ends cannot share code in a flexible way. As many of the codes at CTR use similar numerical and data management strategies, this lack of modularity is a significant problem.

3. The new `Tensoral`

The new `Tensoral`, a generalization and renovation of the current system, is presently being developed. The new system is both coherent and modular. For the remainder of this document we outline the new system and give specific examples of its abstractions. In particular, we introduce the `E` programming language, in which the new `Tensoral` is implemented. Next we introduce abstractions for tensor representation and for the data management of large arrays. Along the way we show how several different numerical strategies employed at CTR are realized with these abstractions.

3.1 Coherence: `C` within `E`

The new `Tensoral` software has a coherent structure. In the new system users and experts both see the same language. Users see high-level abstractions such as tensors embedded in `C` syntax. Experts define new abstractions in terms of lower-level abstractions, also embedded in `C` syntax. Thus, the new system has no separate back-end language. This new general programming system — called `E`— distills and generalizes the basic programming ideas of the current `Tensoral`. The new `Tensoral` will be implemented in this new system.

The `E` compiler scans blocks of `C` code delimited by `{}` brackets for special syntax, for example tensor or back-end syntax. This syntax is then associated with further `E` code which itself may contain other special syntax. This process continues recursively until all special syntax is resolved into `C` code. The resulting `C` code is then organized for efficient execution. This basic plan allows for very general high-level computations to be hierarchically reduced into low-level computations.

3.2 Modularity: Syntax within `E`

New abstractions in `E` are introduced by adding new syntax to the system. Syntax rules matched by the compiler are transformed into `E` code which may contain yet other syntax rules.

`E`'s programmable syntax allows for notation to be suited to the problem at hand. Languages with fixed syntax (e.g. `C` or Fortran) require problems to be translated this fixed syntax. `E` encourages problems to be expressed in their most natural syntactic form. For example, mathematical formulae could be notated with `TeX` syntax, two-dimensional computer graphics operations could be notated with `Postscript` syntax, etc. What ever the syntax, the compiler reduces this syntax eventually into executable code.

3.3 Tensors and their representation

Variables in `Tensoral` — such as a fluid velocity field — are instances of the `tensor` abstraction which we describe here. Tensors are indexed: they have rank and dimension. Tensors depend on `coordinate` directions. Tensor arithmetic is

performed point-wise. Derivatives, integrals and averaging may be taken with respect to these coordinates. Coordinates are defined by the values they may take as well as by how functions of them are to be represented.

Suppose a programmer wants a real valued function f of a variable $0 < t < 1$, represented on a fixed grid of size I by values $f(t_i)$, $t_i = i/I$, $i = 0 \ldots I-1$. A coordinate t is introduced

```
coordinate t = 0 ..  1, size I;
```

and f is declared to depend on t

```
real f(t);
```

The function `f` may be added to other functions (compatibly represented), may be differentiated with a finite difference stencil, averaged, etc.

Of course functions may depend on more than one coordinate and may be represented by orthogonal function expansions or by splines (rather than on a fixed grid). For example, the isotropic turbulence simulation represents velocity fields by N^3 Fourier coefficients in a cube:

```
fourier coordinate x y z = 0 ..  2pi, size N;
```

The coordinate system for the channel flow simulation would need a Chebyshev direction:

```
chebyshev coordinate w = -1 ..  1, size N;
```

The `fourier` and `chebyshev` (and other) packages contains all of the relevant details of how tensor representations are realized, how derivatives are taken, how the Laplace operator is inverted given boundary conditions, etc.

Once these coordinate systems have been defined tensors may be declared to depend on them. With coordinates as above, an isotropic velocity field `u` would be declared as `real u_i(xyz)`, a channel flow field as `real v_i(xwz)`. Such tensors, once declared, may now be used for computation.

3.4 Split arrays

Fluids simulations represent velocity fields with a small number of large three-dimensional arrays. To attain the highest possible resolution these arrays must be as large as possible. Modern computer systems have finite resources: typically, $\approx 10^2$ processors connected by a fast network, each processor having $\approx 10^6$ fast storage elements (e.g. RAM) and $\approx 10^9$ slow storage elements (e.g. disk or tape). One of the painful details of high-resolution coding is fitting the largest possible problem onto a given set of computational resources.

Such large problems are fit on specific computer systems by splitting arrays so that only one or two dimensional fragments of the entire array are in fast memory at a time. The remainder of the array can only be accessed through communication with other processors or by accessing mass storage (disk or tape).

How an array is to be split may depend on what operations are being performed on it. Transform (e.g. spectral and spline) methods typically require that the

array direction being transformed be in local memory. Such methods may require different array geometries for each direction to be transformed. Data management operations must be introduced to put data in the right geometry at the right time.

The new `Tensoral` has a general high-level abstraction for such split multi-dimensional arrays. Split arrays are general. They may be used in `Tensoral` back-ends for representing tensors or may be used elsewhere. Just as a `tensor` is referenced via a `coordinate` system, `Tensoral` arrays are referenced via array indices. Indices are declared

```
index x = 1 ..  nx;
```

and arrays declared as functions of these indices `float f[x]`. Indices may be `split` in a hierarchical and programmable manner. Declarations

```
index y = 1 ..  ny | cpu;
index z = 1 ..  nz | cpu;
```

would introduce indices `x` and `z` that are split across processor in a multi-computer. `cpu` refers to a package which knows how interface with inter-processor communication software in the operating system. Splitting packages would be provided for various inter-processor communication schemes or for communication with mass storage devices (such as disks or tapes). Once a system of indices are introduced, arrays may be declared (`float f[xyz], g[xyz]`) and operated upon (`f = g + 1`).

When arrays are referenced, splittings may be explicitly given. Thus, `f[xy|z]` would generate code so that each `cpu` had `xy` planes of data (for a "planes" code); `f[x|yz]` would generate code so that each `cpu` had `x` pencils of data (for "pencil" codes). This allows for explicit control over data management.

5. Status

It is clear that the system outlined here is a powerful and general extension of the current `Tensoral` system. Current work focuses on realizing the design presented here.