Tensoral for post-processing users and simulation authors

By Eliot Dresselhaus

The CTR post-processing effort aims to make turbulence simulations and data more readily and usefully available to the research and industrial communities. The *Tensoral* language — which provides the foundation for this effort — is introduced here in the form of a user's guide. The *Tensoral* user's guide is presented in two main sections. Section 1 acts as a general introduction and guides database *users* who wish to post-process simulation databases. Section 2 gives a brief description of how database *authors* and other advanced users can make simulation codes and/or the databases they generate available to the user community via *Tensoral* database *backends*.

The two-part structure of this document conforms to the two-level design structure of the *Tensoral* language. *Tensoral* has been designed to be a general computer language for performing tensor calculus and statistics on numerical data. *Tensoral*'s generality allows it to be used for stand-alone *native* coding of high-level post-processing tasks (as described in section 1 of this guide). At the same time, *Tensoral*'s specialization to a minute task (namely, to numerical tensor calculus and statistics) allows it to be easily *embedded* into applications written partly in *Tensoral* and partly in other computer languages (here, C and *Vectoral*). Embedded *Tensoral* — aimed at advanced users for more general coding (e.g. of efficient simulations, for interfacing with pre-existing software, for visualization, etc.) — is described in section 2 of this guide.

1. Tensoral user's guide

Overview

The post-processing problem entails computing quantities derived from given base quantities such as a velocity vector field $\vec{u}(\vec{x},t)$, a scalar field $\phi(\vec{x},t)$, or a vorticity field $\vec{\omega}(\vec{x},t)$. (Which base quantities are present will vary from database to database.) Derived quantities are typically those commonly arising in theories of fluid mechanics, turbulence, and in practical problems; all of these quantities involve performing calculus and statistics on numerically represented tensor quantities.

A *Tensoral* post-processor canonically starts with one or more given fields (e.g. $\vec{u}(\vec{x},t)$) and computes one or more derived quantities and outputs the results of these computations in some form. For example, given a velocity field one may wish to calculate pressure, strain, vorticity, strain times vorticity, mean and mean square velocity, skin friction, etc.

Tensoral presents users with two main abstractions: *tensors* and *operators*, which we presently introduce. All quantities in *Tensoral* (whether base or derived) are

represented as *tensors*. *Tensoral* tensors correspond loosely to mathematical tensor fields. This correspondence is loose in that *Tensoral* tensors are not defined by how they transform under coordinate change. Instead, *Tensoral* tensors are "computational" tensors: that is, they are indexed numerical arrays (for example, `a_ij(x,y,z)`) — with one set of tensor indices (`ij`) and one set of coordinate indices (`xyz`). *Tensoral* tensors have *rank* and *dimension* which respectively define the number and range of tensor indices; `a_ij` is a rank 2 tensor and indices `i` and `j` take integral values from 1 to d, the dimension of `a`. Coordinate indices describe which coordinates (if any) a tensor depends on. Normally, coordinate indices are transparent to post-processing users — operations on tensors always apply to the entire array. Adding two tensors, for example, adds array values at corresponding spatial points and for corresponding tensor indices. Explicit coordinate values are also available via projection (introduced below).

Tensoral tensors are modified and combined with *operators*. Standard *Tensoral* operators include tensor assignment (`=`), algebraic operations (addition, subtraction, multiplication, division, exponentiation), differentiation, integration, averaging, and projection. Such operators are built into *Tensoral* syntax and are hence "standard." Tensor rank and coordinate dependencies are appropriately updated when variables are assigned, algebraically combined, differentiated, averaged, or indices contracted (dot product). Thus, performing a derivative increases rank by one, averaging removes coordinate dependencies, etc.

User defined operators can be provided at will by database authors. Useful examples of such operators include reading and writing databases, output of tensors for visualization or graphing, etc. *Tensoral* provides an extremely flexible mechanism for such operators to be added by database authors. However, available user defined operators must be documented by a database author for users to be able to effectively use them.

It should be mentioned here that *Tensoral* tensors and operators are abstract notions. How an abstract tensor is represented numerically (e.g. as an array in memory, across processors in a multi-computer, split between memory and disk, etc.) and how operators operate (e.g. derivatives as finite differences or as multiplication in wave space, etc.) is completely determined by a database *backend*. Such backends (described in section 2 of this guide) are provided by simulation authors and give all of the information necessary to convert *Tensoral* post-processors into an executable computer program to perform the intended computation and output the result.

Mathematical syntax

Mathematical notation in *Tensoral* is a super-set of *Vectoral* notation and as with *Vectoral* aims to present a syntax as close as possible to standard mathematical notation. Thus, given tensors `a` and `b`, `a+b`, `a-b`, `a*b`, `a.b`, `a/b`, `a^b` and `|a|` represent point-wise addition, subtraction, cross product, dot product, division, exponentiation and absolute value (for scalars), respectively. Juxtaposition `a b` can also be used for outer (tensor) product (but only if `a` and `b` are within parenthesis or are not within parentheses but are on the same input line). Floating point constants

(which are tensors of rank 0 with no coordinate dependencies) are entered as in *Vectoral*: as sequences of base ten digits and optional decimal point, followed with an optional `e` or `E` for exponent and optional `i` or `I` for the imaginary unit ($\sqrt{-1}$). Any balanced parenthesis (`()`, `{}` or `[]`) may be used for grouping mathematical expressions.

Tensor notation

In addition *Tensoral* supports tensor notation as follows. Tensor indices are introduced by the underscore character `_` and followed by arbitrary mixtures of single digits or single letter coordinate directions (for indices with explicit values – e.g. `a_12` ≡ `a_xy`), or single letters which are not coordinates (for dummy indices – e.g. `a_ij`). No spaces are allowed before or after indices or the leading `_`. Indices whether explicit or dummy must be single letters or digits. One should think of *Tensoral* indexed expressions as atomic variable references which — as in most other computer languages — contain no white space.

Any tensor index which is not a number or a coordinate direction (as defined in a particular simulation backend) is assumed to be a dummy index. Dummy indices label how tensor indices are to be combined in an expression — for example, distinguishing the statements `a_ij = b_j c_i` and `a_ij = b_i c_j`. Dummy indices are also often used in conjunction with the summation convention: namely, that repeated dummy indices in a product are summed over. Use of the summation convention in indexed products is controlled by whether `*` or `.` or juxtaposition is used for multiplication. Products involving pure juxtaposition imply the summation convention; otherwise, summation is not implied. *Tensoral* provides the standard symbols δ_{ij} (totally symmetric) and ϵ_{ijk} (totally anti-symmetric) as `delta` and `epsilon`. Here are some illustrative examples of tensor index notation:

`c_k = epsilon_ijk a_i b_j`	cross product of rank 1 `a`, `b`.
`c = a_i b_i`	dot product of rank 1 `a`, `b`.
`c_ik = a_ij b_jk`	matrix multiplication of rank 2 `a`, `b`.

Tensor expressions need not have explicit indices. If indices are missing *Tensoral* deduces the tensor rank (either from the tensor assigned to the given expression or zero if there is no such tensor) and inserts dummy indices in missing slots from left to right. Summation over repeated dummy indices is implied for index-free expressions, independent of which form of multiplication is used. Also, for index-free expressions the multiplication operators `*` and `.` generate dummy indices for cross and dot products, respectively. Using index free notation, the above examples are coded as follows:

`c = a * b`	cross product.
`c = a . b`	dot product.
`c = a b`	matrix multiplication.

Coordinate indexing: projection

Indexing (also referred to as *projection*) is also supported for spatial coordinates, but with a different notation than for tensor indexing. A tensor `u` depending on

coordinates xyz can be evaluated at particular x, y or z values for $x = 17$ planes as `u(17,y,z)`, for $x = 17$, $z = 19$ pencils as `u(17,y,19)`, or at a single point $x = y = z = 0$ as `u(0,0,0)`. All coordinate names in *Tensoral* are single characters, whose definition, semantics, and values are determined by database backends.

Tensoral coordinates are not required to be the native coordinates of a simulation. A database author can provide various coordinate systems for a single simulation as appropriate and physically meaningful. Typical coordinates for Rogallo's wave space isotropic turbulence simulation would include, for example:

`x y z`	standard Cartesian coordinates,
`X Y Z`	Cartesian wave space,
`r`	the radial coordinate $r^2 = x^2 + y^2 + z^2$,
`k`	the wave vector magnitude $k^2 = X^2 + Y^2 + Z^2$.

Tensoral operators

Mathematical functions (e.g. sine, cosine, log, exponential), differentiation, laplace and curl operator inversion, among others) all appear in *Tensoral* as operators. Operators in *Tensoral* act from the left and apply to a given number of tensors or tensor expressions on the right. Operator arguments are flanked by parentheses (one of `()`, `{}` or `[]`) and separated by commas as in standard mathematical notation. (Additional operator notation is provided at statement level and will be discussed below.) If an operator takes a single operand (for example, square root), these parenthesis may be omitted so long as the argument is a tensor. Thus, `sqrt z` is permissible in place of `sqrt(z)`.

Operators are either built into *Tensoral* or are defined by database authors' backends. Since they must be specially defined by backends or are built into *Tensoral*, operators can be recognized as syntactically differentiated from tensors, making it possible to differentiate tensor projection (e.g. `a(0,0,0)` for tensor `a`) from operator notation (e.g. `f(0,0,0)` for operator `f`).

The standard mathematical functions in *Tensoral* are as in *Vectoral* and are listed in the following table:

`conj z`	Complex conjugate
`exp x, log x, log10 x`	Exponential, log, log base 10.
`sqrt x`	Square root
`sin x, cos x, tan x`	Trigonometric functions.
`arcsin x, arccos x`	
`arctan x`	Trigonometric inverse functions.

Like tensors, operators can also have rank and be indexed (like any other tensor). In particular, differentiation (`diff` or `grad`), integration (`int`), averaging (`ave`), minimum (`min`), and maximum (`max`) are all indexed operators in *Tensoral*.

Differentiation has rank one and can be explicitly indexed (e.g. `diff_y u_x` for $\partial_y u_x$) or can be index free:

`w = grad * u`	encodes the curl of u;
`divu = grad . u`	encodes its divergence.

In addition, a special indexed shorthand is available for derivatives: any dummy indices following a comma are taken as derivatives. Thus, `u_i,j` is shorthand for `diff_j u_i`, and `v,ii` generates a Laplacian $\nabla^2 v$.

The remainder of the indexed operators listed above are special in that they do not have fixed rank. Consider `ave` as a typical example. `ave_x` performs an average over the x coordinate direction (as defined by the database backend); `ave_xy` performs averaging over both x and y coordinate directions. The remainder of the average-like operators (`int`, `min`, `max`) behave in a similar fashion: operator indices determine which coordinates are to be integrated and minimized or maximized over.

Tensoral statements

There are only two forms of statement in *Tensoral*: assignments and statement-level operator expressions. Assignments can use multiple left-hand sides as long as they are tensors and multiple assignments may be performed in parallel as in *Vectoral* with the `&` character joining the multiple assignments. In parallel assignment, right hand sides of all `&`-linked assignments are evaluated *before* any assignments are performed, so that the statement `a = b & b = a`, for example, swaps tensors `a` and `b`.

Statement-level operator expressions may optionally use a special operator syntax, different from standard functional notation (e.g. `f(a,b,c)`). At statement-level operator expressions may be written without parenthesis. If all arguments are tensors (either indexed or index-free) an operator expression maybe written without commas. In either case, the final argument is terminated by a newline, which replaces the closing parenthesis of functional notation. The following are valid examples of statement-level operators `f` and `g`: `g a+b, a-b` and `f a b`. (Of course, both of these examples must be terminated with a new line.)

Example and usage

Here we illustrate what has just been presented and give a complete example of how *Tensoral* can be used to perform a simple post-processing task. Suppose — for the sake of example — one wants to study vorticity generation in an evolving incompressible boundary layer (evolving along the x direction). A simple question to ask would be "what does the plane-averaged vortex stretching term look like as a function of x?" Suppose further that one desires to measure this stretching in an exponential sense, i.e. to calculate $d/dt \log(\vec{\omega}^2) = \vec{\omega}.\mathbf{S}\vec{\omega}/\omega^2$, and average it along yz directions. One would code the following *Tensoral* program into a file `test` on the computer disk:

```
S_ij = 1/2 (u_i,j + u_j,i)
w_k = 1/2  epsilon_ijk (u_i,j - u_j,i)
print ave_yz (w .  S w / w .  w)
```

To execute the *Tensoral* program `test` on simulation restart file `run1`, one would execute a command (for example, to a Unix shell program) `tl test run1` and the entire x direction of mean exponential vorticity production should be output on the computer console. (Details, of course have been omitted here: in particular, `run1` must be associated with some author-contributed *Tensoral* database backend.)

2. Tensoral author's guide

This author's guide seeks to introduce simulation and database authors to how the *Tensoral* compiler operates and to how database backends interact with and control this operation. Thus, we give here a general introduction to the inner workings of the *Tensoral* compiler and follow it with a brief description of the tools with which backends are coded.

Overview

The *Tensoral* system compiles high-level tensor expressions and statements — either in the form of a native post-processor (as described in the above user's guide) or as embedded within a lower-level *host* computer language — into *host* language code which numerically realizes these tensor operations. The lower-level language program output by a *Tensoral* compilation is itself compiled by another (e.g. *C* or *Vectoral*) compiler into an executable computer program. *Tensoral* has been designed to be easily adapted to generate any sufficiently powerful host language. Thus, the prototype system currently under development has separate versions for *C* and *Vectoral* as host languages.

By compiling tensor operations into a host language, *Tensoral* can be simultaneously general and efficient. Also, this design allows for *Tensoral* to be flexibly embedded within non-tensor specific host language code. In this way *Tensoral* specializes in numerical tensor computations and leaves other language features (input/output, file handling, graphics calls, etc.) to the host language.

The process of converting *Tensoral* into host code is mediated by the *Tensoral* compiler and is controlled by database backends. The *Tensoral* compiler presents backends with several constructs for describing exactly how abstract *Tensoral* tensors and the operators which combine them are realized in host code. In particular, the compiler presents database authors with mechanisms for host-coding both tensor and coordinate indices, for host-coding operators and how they combine tensors in mathematically meaningful ways, and for host-coding loops to iterate over tensors' coordinates.

The backend constructs for looping, operators, and tensor indexing are given using a parenthesized Lisp-like notation: *Tensoral* employs the *Scheme* dialect of Lisp for both its internal coding and as an extension language. Host code is specified within *Scheme* in the form of a simple *template* language. Templates are fragments of host code which can refer to other templates or arbitrary *Scheme* code, can have other templates substituted in them, and can be split and subsequently inserted onto the loops which iterate over tensor's dependent coordinates. The details and syntax for templates, as well as for the backend looping, indexing, and operator constructs just mentioned, will be touched upon in the following and detailed elsewhere.

How Tensoral works

How then is a *Tensoral* program compiled into host language code? Tensors in *Tensoral* must somehow correspond to numeric arrays. Hence, operations involving tensors must correspond to host code which iterates numerical operations over the

elements of these arrays. The first step in generating host code must then involve specifying how *Tensoral* tensors and expressions are to be iterated over.

The *Tensoral* compiler represents this iteration with a scaffolding of nested host language *loops* — `for` loops in *C* or *Vectoral*; `do` loops in *Fortran*. Looping constructs are defined by simulation authors' backends with the `loop` function and are meant to be flexible and general so as to support various data management strategies such as splitting data into one dimensional pencils, two dimensional planes, groups of planes, or splitting data across processors of a multi-computer (such as the Intel Hypercube or Paragon systems).

The ordering and nesting of these loops is dynamic and under the control of either the database backend or the *Tensoral* program (or both). Loop nests are determined and changed either implicitly through the *Tensoral* operators present in an expression (the typical case for native *Tensoral* post-processors) or explicitly in *Tensoral* code (the typical case for embedded *Tensoral*).

Once the loop scaffolding has been erected, host code templates for tensor operations and expressions can be built around and inside it. *Tensoral* statements are first parsed by the compiler into *Scheme* code. Parsing involves mapping *Tensoral* operator notation (for example, `f(a,b,c)` and `a = b*c`) to corresponding *Scheme* function calls (`(f a b c)` and `(= a (* b c))`) for appropriately defined or re-defined *Scheme* functions (`f`, `=`, and `*`). This *Scheme* code, which only involves function calls and tensor references, is then recursively evaluated by the *Scheme* interpreter. The results of evaluating *Tensoral* operators at each recursion level are a template representing the operator applied to its operands in host code and a representation of which coordinates this expression depends on. Both of these evaluation results come in the form of *Scheme* strings.

When tensors are encountered while evaluating *Tensoral* expressions, special templates are used to generate host code for them. These special templates are given by backends and completely implement how tensor and coordinate indexing behaves. Coordinate dependency information, built into to how tensors are internally represented by the *Tensoral* compiler, determines how this indexing is to be performed. Tensor indexing templates are generally the most complex in a database backend since they almost completely implement how tensors are represented numerically. Template and coordinate dependency information are returned as results of this evaluation.

The evaluation of *Tensoral* operator expressions also involves generating both a template and corresponding coordinate dependency information. Operator evaluation begins by recursively evaluating the operator's operands giving the operands templates and coordinate dependencies. All *Tensoral* operators have templates associated with them. These templates are either built into *Tensoral* or are given by database backends via the `operator` command. In either case, these operator templates specify where and how operand templates are to be placed within them. In this way, expression templates are formed. All *Tensoral* operators also have coordinate dependency information which specify how they combine the coordinate dependencies of their operands. Thus, evaluation results in a template and

coordinate dependencies.

After expression and tensor operands have been substituted, host code is generated from templates by "cut and paste." Substituted templates — whether evaluated from statement level or nested within an expression — are split (the "cut" operation) at points specified in the template and the pieces of the split template ("sub-templates") are inserted (the "paste" operation) onto the loop nest according to the nesting levels specified along with these split points. The sub-template after the final split point of a template (or the entire template if no split point is present) is taken as the *value* of the template. Template values represent host code for the value of a template. For nested expressions, template values are inserted into the expression which recursively contains them (e.g. the * template value in `(= a (* b c))`); template values at the statement-level (e.g. = value in `(= a (* b c))`) are inserted into the loop nest according to the statement-level coordinate dependency information generated in the evaluation process.

Once all tensor expressions have been evaluated and all templates generated and split, the loop nest contains only host language strings and *Scheme* code. Any *Scheme* code on the loop nest needs to be further evaluated and will, presumably, generate more templates and/or *Scheme* code. The evaluation process just outlined is repeated until no more *Scheme* code remains on the loop nest and the entire loop nest may be output as host code. Initially host and *Scheme* code exist on a single loop nest; however, as further *Scheme* code is evaluated, further structure may be added under the direction of this the *Scheme* code. One of the most important uses of *Scheme* code within templates is to structure loop nests as appropriate to a given computation.

3. Current status and future direction

At present a prototype of a pre-*Tensoral* language — a lower-level language than *Tensoral* as described here — has been completed and is operational. This language generates *Vectoral* post-processors given pre-*Tensoral* code and includes many of the backend concepts described here. However, this pre-*Tensoral* language is missing many of the features and even some of the general concepts described in this document. The full *Tensoral* language described here is currently under development and will hopefully be completed in a matter of months.

As for the future, I hope to have a prototype *Tensoral* system functional for the next CTR summer program. Use of *Tensoral* in a summer program should provide significant experience towards how to use *Tensoral* effectively and how to refine its design to increase its utility. For the near future, my goal is to have Rogallo's isotropic turbulence simulation and database post-processing coded purely in *Tensoral*.