Database post-processing in Tensoral

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, introduced in this document and currently existing in prototype form, is the foundation of this effort. *Tensoral* provides a convenient and powerful protocol to connect *users* who wish to analyze fluids databases with the *authors* who generate them.

In this document we introduce *Tensoral* and its prototype implementation in the form of a user's guide. This guide focuses on use of *Tensoral* for post-processing turbulence databases. The corresponding document — the *Tensoral* "author's guide" — which focuses on how authors can make databases available to users via the *Tensoral* system — is currently unwritten.

Section 1 of this user's guide defines *Tensoral*'s basic notions: we explain the class of problems at hand and how *Tensoral* abstracts them. Section 2 defines *Tensoral* syntax for mathematical expressions. Section 3 shows how these expressions make up *Tensoral statements*. Section 4 shows how *Tensoral* statements and expressions are embedded into other computer languages (such as C or *Vectoral*) to make *Tensoral* programs. We conclude with a complete example program.

1. Basic notions

Post-processing in the abstract

The post-processing of fluids data entails computing quantities *derived* from *base* quantities such as the velocity vector field $\vec{u}(\vec{x},t)$, scalar field $\phi(\vec{x},t)$, or vorticity field $\vec{\omega}(\vec{x},t) = \nabla \times \vec{u}$. Base quantities are those found in the databases output by simulation codes and may vary from simulation to simulation. Derived quantities are typically those commonly arising in theories of fluid mechanics, turbulence, and in practical problems such as velocity profiles, Reynolds' stresses, probability densities, etc.

The canonical post-processor starts with one or more base fields, computes one or more derived quantities, and outputs the results of these computations. An example post-processor might, given a velocity field, calculate pressure, strain, vorticity, strain times vorticity, mean and mean square velocity, Reynold's stresses, or skin friction, and generate tables or graphs of these quantities. In general terms, post-processing involves calculus and statistics applied to numerical tensor data.

1.1 Tensoral in the abstract

Tensoral has been designed to apply to a very general class of numerical problems. *Tensoral* applies whenever it is useful to separate the high-level coding and low-level implementation of calculus and statistics operations on numerical data. Applied to

turbulence databases *Tensoral* programs can flexibly and efficiently serve as post-processors. This user's guide will focus on such post-processing applications of the *Tensoral* language.

Tensoral, like other computer languages, provides abstractions to aid the user in understanding and using the system. There are three basic abstractions in *Tensoral*: a type of variable called a *tensor*, *operators* which *operate* on tensors and produce new tensors, and a general and programmable notion of *state* in which tensors exist and in which operators act. These basic abstractions when combined make up the description of a data *type* which is defined by an author's *back-end*.

1.2 Tensors

All mathematical quantities in *Tensoral* are represented by *tensors*. *Tensoral tensors* correspond (loosely) to mathematical tensor fields. They are more usefully thought of as "computational" tensors — that is, as indexed numerical arrays (e.g. `a_ij(xyz)`) with two sets of indices: a set of tensor or array indices (`ij`) and a set of *coordinate* indices (`xyz`). Each coordinate value (e.g. `(xyz)`) corresponds to an array of numbers (e.g. `a_ij`) which may or may not be indexed like a mathematical tensor.

1.3 Operators

Mathematical and other operations on tensors are represented by *Tensoral operators*. Certain *standard* operators are invoked by elements of *Tensoral* syntax (e.g. `a+b` invokes the addition operator). Non-standard operators may be introduced by back-end authors to extend the functionality coded into *Tensoral* syntax. Standard operators in *Tensoral* include tensor assignment, algebraic operations (addition, subtraction, multiplication, division, exponentiation), differentiation, integration, and averaging. *Tensoral* also provides standard operators for the reading and writing of files in various formats. Standardization is intended to allow mathematically similar operations to be coded with the same syntax even if such operations have different back-end numerical implementations. Ideally, such standardization would allow post-processing codes for different back-ends to be identical. In addition to standard operators *Tensoral* provides a highly flexible mechanism for integrating new operators into the language. Such operators, as well as re-implementations of standard operators, may be provided at will by database authors.

Typically *Tensoral* operators apply to tensors as arrays: that is, they apply to all tensor (or array) indices and coordinate indices unless explicitly instructed otherwise. Thus, *Tensoral* is an array language. Adding two tensors, for example, adds array values for all tensor indices and at all spatial points (i.e. grid points) defined for a given simulation. Explicit tensor or coordinate indices can be specified with tensor indexing and coordinate indexing.

The implementation specifics of tensors and (potentially all) operators that act upon them are also provided by database authors. The numerical representation of an abstract tensor (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 author's *back-end*. Such back-ends provide all of the information necessary for the *Tensoral* compiler to convert high-level post-processors into an executable computer programs. Back-ends and especially back-end programming will not be covered in this document.

1.4 State

Every tensor at every stage in a computation is in a certain *state*. This state controls how back-ends represent tensor references and operations on them in terms of executable computer code. The state of a tensor is represented by a set of *state variables* taking a specified set of values (possibly Boolean true or false, integer, floating point, string, or otherwise). For example, each tensor has pre-defined integer valued state variables `rank` and `dim` which respectively define the number and range of tensor indices that may be validly used to index a tensor. Thus, `a_ij` is valid reference to a `rank` 2 tensor with indices `i` and `j` taking integral values from `1` to `dim`, the dimension of `a` (which must be an integer). Similarly, a tensor's coordinate dependencies are given by a built-in state called `dep`. Such dependencies change when variables are averaged over or new coordinates are introduced in an expression. Thus, the tensor `a_ij(xyz)` has state `{rank 2, dim 3, dep xyz}` — we shall use this notation throughout to denote state variables (here `rank`, `dim` and `dep`) with values (here 2, 3 and `xyz`) of a tensor's state. Other state variables can be introduced at will by database authors to encode specification of a simulation's grid, how a tensor is currently being represented (e.g. in physical or wave space), the state and type of a tensor's data management (e.g. three-dimensional tensor in xy planes), etc.

2. Expression syntax

Tensoral syntax is introduced here from bottom to top. First we show how to reference tensor variables. Tensor references are the building blocks of mathematical expressions. Such expressions are themselves the building blocks of *Tensoral* statements. *Tensoral* statements may introduce new tensors, assign tensors to expressions, read and write files, perform back-end specific operations, etc.

2.1 Tensor variables

Tensor references — for example, `a_12(xyz)` — have up to three components: a required tensor name `a`, optional array (tensor) indices `_12` and optional coordinates indices `(xyz)`. The notation `a_12(xyz)` corresponds to the mathematical notation $a_{ij}(\vec{x})$ for a tensor field of rank 2. Both array and coordinate indexing specialize a tensor reference. That is, `a` refers to $a_{ij}(\vec{x})$ for *all* indices ij and spatial points $\vec{x}$; `a_12` refers to $a_{12}(\vec{x})$ for all spatial points $\vec{x}$; `a_12(x=10)` corresponds to all values of $a_{12}(10, y, z)$ for all y and z. Array and coordinate indexing will be detailed below.

Tensor variable names are given by sequences of lower and upper case letters, underscores `_`, and primes `'`. Digits may also be used in tensor names but not as the first character. The following are all valid distinct tensor names: `u`, `U`, `v'`, `velocity`, `u1`, `u1'`, `long_name`, `str'ange_na'me`. Tensor indices are introduced by the final `_` character in a tensor name, but only if the preceding characters actually

refer to a tensor variable. Thus, `long_name_ij` either refers to an unindexed tensor named `long_name_ij` or refers to the `ij` component of a tensor named `long_name` depending on which alternative has been declared to be a tensor variable.

2.2 Coordinates

Coordinates are notated by single lower or upper case letters, optionally followed by prime `'` characters. Thus, the following are valid and distinct *Tensoral* coordinates: `x`, `y`, `z`, `x'`, `X`, `X'`, `r`, `k` — as long as they have been provided for by back-end authors.

Tensors may depend on any unique combination of coordinates — as long as they have been defined by a back-end. A back-end author can provide various coordinate systems for a single simulation as appropriate and physically meaningful. For example, coordinates on a Cartesian grid are typically

`x y z`	Cartesian coordinates x, y and z,
`r`	the radial coordinate $r^2 = x^2 + y^2 + z^2$.

Coordinates in *Tensoral* are special tensors of `{rank 0}` which depend on themselves and when referenced in expressions generate corresponding coordinate values. The values taken on by coordinates are defined by back-end authors. Thus, a `z` coordinate might go from 0 to 2π for isotropic turbulence and from 0 to ∞ for a boundary layer.

2.3 Tensor indexing

Tensor indices, introduced by the underscore character `_`, consist of a sequence of coordinates or single digits or dummy indices (*explicit* indexing) or may not be present at all (*implicit* indexing).

Explicit index values (e.g. `a_12` or `a_xy`) may be either digits or coordinates. No spaces are allowed before or after indices or the leading `_`: indexed expressions are atomic variable references which contain no white space. If a coordinate `c` is used to index a tensor, the tensor must depend on `c` (i.e. `c` must be present in the tensor's `dep` state) — otherwise a compiler error is issued; the value of a coordinate index is the position that `c` takes in the dependency state of the tensor. The value of a digit index is the specified number between `1` and `dim` (the tensor's dimension). Thus, if `a` has `{rank 2, dep xyz}` state, then `a_12`, `a_xy`, `a_1y`, and `a_x2` are all valid and equivalent indexed expressions.

Any tensor index which is not a digit or a coordinate direction (as defined in a particular simulation back-end) is assumed to be a dummy index. Dummy indices in *Tensoral* aim for the same semantics as in standard mathematical tensor notation. Thus, a dummy indexed expression `a_ij` (assuming neither `i` or `j` are coordinates) refers to all 9 components of `a` (assumed to be `{rank 2}`). Dummy indices label how tensor indices are to be repeated and combined in an expression — for example, distinguishing the assignment statements `a_ij = b_j c_i` and `a_ij = b_i c_j`. Dummy indices are also used in conjunction with the summation convention: namely, that repeated dummy indices in a product are summed over. We defer further details of dummy indexing until we introduce the operators which control their interpretation.

Tensor references need not have explicit indices; they may be indexed *implicitly*. If indices are not specified, dummy indices are introduced in a standard way. Currently, this standard has not been fully decided upon. The user is encouraged (for now) to use explicit dummy indices.

2.4 Coordinate indexing

Indexing is also supported for the coordinates a tensor depends on. A tensor `u` depending on coordinates `xyz` can be evaluated at particular `x`, `y` or `z` values — for example, at $x = 17$ planes with `u(17,y,z)` (or equivalently, `u(x=17)`), at $x = 17$, $z = 69$ pencils as `u(17,y,69)` (or `u(x=17,z=69)`), or at a single point $x = y = z = 0$ as `u(0,0,0)` (or `u(x=y=z=0)`). If a coordinate is either not specified or is specified only by name (with no explicit value given), this coordinate is assumed to take on all possible values. Thus, `u(17,y,z)` refers to an array indexed by coordinates `yz`.

2.5 Algebraic notation

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. Numerical 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 ($\sqrt{-1}$). Numerical constants can be thought of as tensors with zero rank and no coordinate dependencies (i.e. state `{rank 0, dep 0}`). Given expressions a and b, we have the following operators:

a+*b*, *a*-*b*	addition, subtraction,
a/*b*	division,
a *b*, *a*.*b*, *a**b*,	multiplication: juxtaposition, dot, star
a^*b*	exponentiation,
a$	complex conjugate,
-*a*	negation,
|*a*|	absolute value.

Any balanced parenthesis (`()`, `{}` or `[]`) may be used for grouping mathematical expressions. These operators — like most operators in *Tensoral*— apply to tensors as arrays: that is, they apply to all array and coordinate indices which have not been explicitly specified.

Precedence of operations in the above table increases from left to right, top to bottom. To avoid ambiguity between *a*-*b* being subtraction (*a*-*b*) and juxtaposition (*a*) (-*b*) negation is not allowed with juxtaposition: juxtaposition may be used, however, with all other operators of lower precedence. Also, all non-commutative operations (i.e. subtraction, division, multiplication of tensors, and exponentiation are left-associative. Thus, the exponentiation *a*^*b*^*c* is grouped as (*a*^*b*)^*c*, etc.

2.6 Three types of multiplication?

There are three types of binary multiplication in *Tensoral*: star `*`, dot `.` and juxtaposition (which has no symbol). These three forms of multiplication differ in how they treat array indices.

Explicitly indexed, . and juxtaposition both imply the summation convention — that is, repeated indices are summed, for example making `a_ij b_j` or `a_ij . b_j` equivalent to matrix multiplication. Explicitly indexed multiplication with * does not sum repeated indices: Thus, `u_i * u_j` is the outer (tensor) product of u with itself. This convention roughly corresponds to the usual tensor notation: . is an inner (dot) product (contracts indices) and * is an outer (tensor) product.

In connection with use of the summation convention, *Tensoral* provides the standard tensors `delta` and `epsilon`. `delta_ij` is the totally symmetric unit tensor (Kronecker delta) and `epsilon_`$i1 \ldots iN$ is the totally anti-symmetric unit tensor: equal to 1 for even permutations of $1 \ldots N$, -1 for odd permutations, 0 otherwise).

Here are some illustrative examples of explicitly indexed multiplications:

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

(Implicit indexing will in the future be different for the three multiplications. The details of this difference is currently unresolved and are not given here.)

2.7 Operator notation

Algebraic operations and mathematical functions (e.g. sine, cosine, log, exponential), differentiation, Laplace inversion and curl operator inversion, among others, all invoke *Tensoral operators*. Such operators which are a part of standard *Tensoral* syntax, as well as those that are non-standard, can all be invoked explicitly with operator notation.

Operators in *Tensoral* act from the left and apply to operand expressions on the right. Operands are flanked by parentheses (one of `()`, `{}` or `[]`), and operands beyond the first are separated by commas as in standard mathematical notation. Additionally, if an operator takes one or zero operands these parenthesis and commas may be omitted so that the operator and operand are juxtaposed. Operators are specially introduced to *Tensoral*, so that it is possible to syntactically differentiate operator notation (e.g. `o` or `o(0,0,0)` for operator `o`) from tensor references (e.g. `a` or `a(0,0,0)` for tensor `a`). The following are examples of both types of *Tensoral* operator expressions:

```
sqrt(x)     random     sqrt x     f(x,y)     sqrt x + y
```

The last expression above is equivalent to `sqrt(x) + y` since juxtaposition of operands has higher precedence than addition (and other operators in the above table).

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

`random`	A random number between 0 and 1,
`conj z`	Complex conjugate,
`exp z, log z, log10 z`	Exponential, log, log base 10,

`sqrt z`	Square root,
`sin z, cos z, tan z`	Trigonometric functions,
`arcsin z, arccos z`	
`arctan z`	Trigonometric inverse functions.

All functions taking arguments `z` in the above table may operate on either real or complex quantities.

2.8 Calculus, statistics, indexed operators

Like tensors, operators also have state and, therefore, `rank` and can be indexed. Standard indexed operators are:

`grad`	numerical differentiation,
`int`	numerical integration,
`ave, sum`	averaging, summation,
`min, max`	minimum, maximum.

Differentiation `grad` mimics the gradient operator ∇: it has `rank 1` and can be explicitly indexed as if it were a tensor (e.g. `grad_y u_x` for $\partial_y u_x$). As with products, the summation convention applies to repeated indices. 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 short hand for `grad_j u_i`, and `v_j,ii` generates the Laplacian $\nabla^2 v_j$.

The remaining indexed operators mentioned above do not have fixed rank. That is, their operation is determined by how they are indexed. Consider `ave` as a typical example. `ave_x` performs an average over the x coordinate direction (as defined by the database back-end); `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.

2.9 Operators and back-ends

Currently, new operators may only be defined or re-defined (for standard operators) by back-ends. At some point the introduction of new operators will become standard *Tensoral*. The isotropic turbulence back-end `iso`, for example, re-defines the three product operators `*`, `.`, and juxtaposition. For `iso` tensors, multiplication can either be in physical or wave (Fourier) space. To select between these two possibilities, the back-end author (in this case E. D.) has defined `*` to imply wave space and juxtaposition ... to imply physical space operands: that is, if any operand of `*` is not in wave space it is transformed into wave space (and similarly for physical space and juxtaposition). This mimics the notation $f \star g$ for convolution which, of course, is multiplication in wave space. The third multiplication `.` implies nothing about the wave and physical space representations of its operands.

3. Statement syntax

There are three kinds of statement in *Tensoral*: declarations, assignments, and statement level operator expressions. Declarations introduce new tensors to the

compiler. Assignments alter the value and state of existing tensors. Statement level operator expressions — whose syntax is identical to the operator expressions described above — allow for non-standard (i.e. back-end specific) operations to be performed.

3.1 Declarations

As in *Vectoral* or C, all variables in *Tensoral* must be declared. A declaration introduces a certain variable name (whose syntax is defined above) to be a tensor of a certain *type*. The type of a tensor corresponds to a back-end which describes this type. For example, the isotropic turbulence back-end defines a tensor type called `iso`. Tensors may have more than one type. For example, an `iso` tensor `a_ij` may be declared to be `symmetric`, indicating to the compiler that it may equate `a_ij` with `a_ji`.

The type or types in a tensor declaration define all of the state variables which may control how an abstract tensor is represented by a numerical tensor field. For example, `iso` tensors represent fields either in physical or in Fourier space and have a corresponding state variable called `wave` which is true or false depending on whether a given tensor is in wave or physical space at a given point in a program. (This state variable could have more complex semantics to represent tensors in mixed wave and physical space.) Other types may introduce other state variables or may override those already defined. For example, if a tensor is declared `symmetric iso`, `symmetric` states and operators have higher precedence than `iso` states and operators. (Such type precedence allows for back-ends to be constructed in a modular fashion — minimizing the duplication of effort.)

When a tensor is declared its initial state must also be declared. In particular, declarations may give values for the standard *Tensoral* state variables `rank`, dimension `dim`, and coordinate dependency `dep`. Other back-end specific state information may also be initialized, for example, declaring a tensor to have a certain back-end specific mesh parameters, to be in wave or physical space, etc. All state variables have *default* values. The standard states, for example, might default to values `{rank 0}`, `{dim 3}`, and `{dep 0}`. Back-end specific defaults are provided and documented by back-end authors.

The general syntax of *Tensoral* declarations is as follows:

$$type_1 \ \ldots \ type_L \ \{init_1 \ , \ \ldots \ , \ init_M\} \ tensor_1 \ , \ \ldots, \ tensor_N$$

This declares $N > 0$ tensors all having the specified $L > 0$ types with decreasing state precedence from left to right and initial state specified by $M >= 0$ initializations. Each initialization field may have two forms:

variable value or *tensor*

This first form explicitly initializes a given state variable to an optionally given value (which defaults to `true`); the second uses the current state of a tensor to initialize the state of another. In addition a tensor name (e.g. `T(xyz)`) may be followed by a list

of coordinates in parentheses (), {}, or [] to initialize its coordinate dependencies (e.g. to xyz). Uninitialized state variables take on their default values.

An author may declare certain tensors which can themselves be referenced to initialize other tensors. Such tensors are *templates* for other tensors and do not correspond to variables in *Tensoral* programs. All turbulence simulations are likely to have definitions of the velocity and/or scalar fields (for example, called `velocity` and `scalar`), which can be inherited by other tensors. This allows for simple declarations of derived quantities, for example, the declaration

```
iso {velocity} u, {rank 2} du, {rank 0} p
```

which might declare an `iso` velocity field `u`, its derivative `du`, and pressure `p`, a scalar.

3.2 Assignments

Tensoral assignments assign tensors on the left-hand side of an equals sign `=` to an expression on its right-hand side. Assignments can use multiple left-hand sides as long as they are tensors whose `rank` is mutually compatible with the right-hand side `rank`. 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 assignments linked with `&` characters are evaluated *before* any assignments are performed, so that the statement `a = b & b = a`, for example, swaps tensors `a` and `b`. Assignments transfer both the state and value of tensors.

In addition tensors may be read from and written to files with assignment notation. File names in *Tensoral* are delimited by double quotes `"restart10"` if constant or by angle brackets `<file>` if variable. Files are read if they occur on the right-hand side of assignments and are written if they occur on the left-hand side of assignments. Currently, file operations may not be mixed with other expressions. This may change in the future.

4. Program syntax

Tensoral statements exist *inside* of a *host* language. *Tensoral* itself was designed to be minimal: *Tensoral* supports tensors and operations on them — no more, no less. The host language is relied upon for everything else. The semantics of how programs are organized, say into different files and different subroutines, how variables are scoped (global versus local), etc. are all lifted from the host language. A user is not forced to learn another computer language; instead the user must only learn how to combine *Tensoral* and host code.

Back-ends also use host language code to implement operators, tensor references, declarations, file operations, etc. Thus, eventually a *Tensoral* program is transformed into a host language program which can then be compiled by the host-language compiler.

4.1 Tensoral in C

In the current *Tensoral* prototype, C is used as the host language. This is referred as *Tensoral* in C. For the remainder of this section, we will take C as the host language to explain how *Tensoral* codes fits into host code.

Tensor variables and *Tensoral* containing functions may be introduced anywhere that C variables and functions may be introduced. *Tensoral* containing functions have the same syntax as C functions except their declared names are prefixed by tensor types just as they would appear in a regular tensor declaration. These types — with the usual precedence rule — make up the environment of back-end template tensors, state variables, coordinates, and operators available to the *Tensoral* code in the function. Tensors as arguments to functions and as local (called "automatic" in C) variables as in C are local to the {} block of their definition. (The user must be careful to keep distinct names for C and *Tensoral* variables — use of the same C and *Tensoral* variable names can cause confusing results!)

Tensoral statements may appear wherever C statements may. Since all tensor variables are specifically introduced as tensors, *Tensoral* code may be differentiated from C code. Because of this differentiation *Tensoral* expressions and statements may appear anywhere C expressions and statements may. The presence of *Tensoral* expressions may change the semantics of C code. *Tensoral* code may involve array operations and hence imply iteration over coordinates. This iteration also includes iteration over the C statement which contains the *Tensoral* array operation. For example, the function call `printf ("%g", f(xyz));` will print `f` for all coordinate values `xyz`. On the other hand, some C statements such as the looping constructs `for` and `while` are not iterated over *Tensoral* coordinates; others (`switch`, `if`, function calls, etc.) are iterated.

4.2 Example: the isotropic turbulence back-end

Rogallo's isotropic turbulence simulation and the corresponding `iso` *Tensoral* back-end represent tensors on a uniform computational mesh of size N^3. Since N here is desired to be as large as possible, tensors must have their data *managed* — that is, since arrays of size N^3 are potentially too large to fit into a computer's main memory, all tensors are split up into data plane groups of size MN^2 or into groups of lines of size $M'N$. At one time only M planes or M' lines of a tensor's data is operated on; the remainder of the tensor is kept on backing store (i.e. disk or other processors of a multi-computer). Data management code is automatically generated by the `iso` back-end.

Derivatives are computed by transforming into wave space; certain products are formed in physical space. Fourier transforms (whether 1, 2, or 3 dimensional) are automatically coded when derivatives are taken or when physical space products are requested. For `iso` tensors recall that star `*` implies wave space, juxtaposition implies physical space, and dot `.` implies neither. Thus, `a * b` will transform `a` and `b` to wave space if necessary before multiplying them; `a b` will transform both to physical space (as necessary); `a.b` will simply multiply `a` and `b` as is.

`iso` tensors have special state variables associated with them. The standard states `rank`, `dep`, and `dim` default to 0, 0 and 3 — by default, tensors are scalars, are not dependent on any of the coordinates, and have indices that range from 1 to 3. Tensors can be either in wave space (for state `{wave true}`) or in physical space (`{wave false}`). Currently, tensors are represented either in groups of xy planes (for state `{memxy true}`) or xz planes (`{memxy false}`).

The mesh size and planing factor are given by the state variables `mesh` (which corresponds to N above) and `plane` (which corresponds to M). These state variables may take on integer values, specifying a constant size mesh and constant planing factors; for example,

```
iso {rank 1, mesh 32, plane 2, wave} u
```

declares a 32^3 wave-space velocity field which is split into groups of data planes of size 2×32^2. Similarly, `mesh` and `plane` may take on values that are the names of external C variables which contain the mesh size and planing factors at run time; for example,

```
iso {rank 1, mesh N, plane M} u
```

declares `u` to get its run time size and planing from C variables `N` and `M` (which had better exist!). Also, mesh and planing factors can be initialized when a field is read from a restart file.

The `iso` back-end also provides template tensors which can be used to initialize tensors in user's programs. The tensor `velocity` sets up a velocity field which can be read from a simulation restart file.

4.3 A complete example

Here we illustrate a simple post-processor written in *Tensoral*. Let's suppose — for the sake of example — we want to compute pressure statistics for a series of restart files `run1`, `run2`, ... One codes the following *Tensoral* (in C) program in a file `p.tlc`:

```
1   iso main (int argc, char * argv[]) {
2     int f;
3     iso {velocity} u, {u, rank 2} A, {u, rank 0} p;
4     iso {} mean, rms;
5     for (f = 1; f < argc; f++) {
6       u = <argv[f]>;
7       A_ij = u_i,j;
8       p = -unlaplacian (A_ij A_ji);
9       mean = ave_xyz (p);
10      rms = sqrt (ave_xyz (p^2) - mean^2);
11      printf ("pressure:  min %g, max %g, mean %g, rms %g\n",
12              min_xyz (p), max_xyz (p), mean, rms);
13    }
14  }
```

According to C standards `main` is the function which is called by the operating system with string arguments `argv` of size `argc`. `main` has been declared with type `iso` to indicate that it contains *Tensoral* code; `iso` has been previously introduced to the compiler as part of its library of back-end types. Line 2 declares a C integer variable which is used to loop through the restart files assumed to be given on the command line array `argv`. Line 3 declares the tensor variables we need. The velocity field `u` is initialized using the `iso` template `velocity`. `A` is the velocity

derivative tensor, declared to be like u but with `rank 2`. p is the pressure and `mean` and `rms` are both scalars (`{rank 0, dep 0}`) used for statistics. Line 5 loops over restart files; array operations inside this loop are iterated inside the loop (as for all `for` and `while` statements). Line 6 reads in a velocity field from the command line argument `argv[f]`. Line 7 forms the velocity gradients and line 8 forms the pressure. Lines 9 and 10 compute simple statistics and lines 11 and 12 print them out.

To execute this *Tensoral* program one first compiles it with the command

```
tl -o p p.tlc
```

This produces an executable file p (the `-o` *f* flag names compiler output). Now p can be applied to the restart files `run1`, `run2`, ...

```
p run1 run2 ...
```

and will output the desired statistics.