Tensor Algebra with REDTEN

A User Manual
Edition 1.0

John F. Harper and Charles C. Dyer
Department of Astronomy
Scarborough College
University of Toronto

Email: harper@manitou.astro.utoronto.ca
dyer@manitou.astro.utoronto.ca
WWW: http://www.scar.utoronto.ca/%7Eharper/redten/

November 23, 1994

lifted this right out of the latex2html manual

Contents

Chapter 1

Introduction

1.1 History

The beginnings of REDTEN go back to 1983, when a summer project was begun with the goal of implementing a tensor algebra system in MAPLE[1]. After many difficulties this project was abandoned, but was later taken up again for a Masters thesis, this time using muMATH[2] as the base system.

muMATH was a small, relatively simple computer algebra system, running on IBM PC's and related INTEL-88 machines. As a LISP-based system, and with source code available for all but the kernel functions, it was particularly easy to introduce new data structures and functions, and, where necessary, re-define existing ones. Thus, in a relatively short time a basic working tensor system was produced. After a somewhat longer period a more advanced and capable system was available, with essentially all of the major features of REDTEN today.

From the outset, it was clear that a new system should implement tensor algebra, and not just a few operations with tensors. It was intended that the user be able to enter expressions and see results as close to the conventional form as was possible with a normal terminal screen and keyboard. Rather than calling a function to add two tensors, for example, the user would simply type the names and associated indices, and assign the result to a new tensor. If the sum involved three objects, the user would just add the third name; in other systems it might be necessary to break this operation into two parts, involving two sums and one intermediate object. It was this "serial construction" that was particularly to be avoided.

To accomplish this goal generally requires modifying the system parser to accept new operators. In muMATH this possibility is designed into the parser, and is therefore extremely simple. With a new assignment operator (in muTENSOR it was "::", in REDTEN it is "==") a new function is installed to handle the input: it receives the left and right hand sides of the assignment. Since the data structure representing algebraic expressions is known, it is a straightforward matter for the new system function to treat the expression

[1]maple ref
[2]mumath ref

as a tensor expression, and to produce the desired result.

As muTENSOR advanced, the amount of code involved soon exceeded that in the basic muMATH kernel. Since muMATH was capable of using only 256 Kilobytes of memory (even on machines with larger amounts of memory), the amount of working space (where algebraic calculations are carried out and results stored) gradually decreased. Eventually, there was almost no room left to introduce new functions, and large calculations could not be performed due to the space required to hold all the intermediate results in core. This prompted the development of a "virtual memory" sub-system, which allowed the storage of the components of a tensor in secondary storage, such as a hard-disk or ram-disk. A needed component was read in from the disk, used, discarded and the results of the computation possibly written to disk, all automatically; generally the time required to retrieve the component was much less than the time required to do algebra with it, so that the overhead was negligible. This allowed much larger problems to be successfully tackled, such as showing that the Kerr metric is vacuum, which required about 20 minutes on an IBM AT.

muTENSOR was demonstrated or described at several General Relativity conferences, and as a result, a moderately large group of users developed in several countries. However, the small amount of working space, even with the virtual memory sub-system, and the very simple algorithms used to implement algebra in muMATH[3] made it increasingly obvious that a larger more capable base system was needed if further advancement was to be made.

In late 1985 it was determined that REDUCE would provide the power needed for an improved version of muTENSOR, to be called REDTEN. Since REDUCE is also LISP-based, the transportation of the code to the REDUCE environment was relatively simple, the major difficulties being in the interaction with the I/O system and the internal data structure of expressions, which although similar in concept to those of muMATH, differed in detail. Otherwise, most of the code was simply translated directly from muSIMP (the base lisp of muMATH) to RLISP (the extended syntax of Standard Lisp) in which REDUCE is written. To this day there is code in REDTEN with an odd "flavour" due to the original influence of muSIMP.

For a time the two systems evolved in parallel, each receiving the same upgrades and bug fixes as the other. In REDTEN a virtual memory system was unnecessary[4], while at the same time more code could be developed to expand the system. Thus, REDTEN gradually became more advanced, with several new features not available in muTENSOR (such as frame and spinor packages). It was found the the REDTEN system was very capable and met all expectations.

An attempt was again made to implement a tensor-algebra system in MAPLE, using muTENSOR as a model. This system, called MapleTensor[5], was partially successful, but met difficulties with interacting with the MAPLE I/O system and with accessing the internal data structures used for expressions. It therefore lacked the same "feel" as

[3]Necessarily so, since sophisticated algorithms are usually much more complicated and occupy a larger amount of memory.

[4]It may be required for PC versions of REDTEN, but has not yet been developed.

[5]Implemented by summer students Jeff Rosenthal and Steve Allen.

muTENSOR and REDTEN and was not as convenient to use; support for MapleTensor has since been dropped.

In 1988(??) muMATH was discontinued, and a new product, DERIVE, was introduced by the Soft Warehouse. Unfortunately, DERIVE, as a much simpler system, does not include a programming environment. Hence, muTENSOR became an orphaned system, and its development has halted, while that of REDTEN has continued. Although a successor to muMATH may become available in the future, many of todays personal computers are capable of running REDUCE (and therefore REDTEN), so that the need for a small compact system is less of an issue.

Currently REDTEN is at version v4.0, and is continuing to evolve. Many improvements have been made in the past few years, and the kernel of the system has reached a relatively stable configuration. Recent improvements also include support for REDUCE 3.5(the current version of REDUCE) and a basic clean-up of the code. It is hoped that REDTEN will soon become a part of the REDUCE network-library of contributed programs, and therefore more widely available.

1.2 Concepts in REDUCE

It is assumed that the reader has a basic familiarity with REDUCE 3.5, and also of tensor analysis in general. The REDUCE Users Manual (Hearn 1987), or the books by Rayna (1987) and MacCallum and Wright (1991) are good references for REDUCE. Books by Stephani (19??) and Adler, Basin & Shiffler (19??) are good introductory texts in tensor analysis and cosmology.

1.2.1 Case Sensitivity

REDUCE is by default case insensitive, and depending on the installation the default behaviour may be to map upper case characters to lower case, or the reverse. This insensitivity is primarily to allow for greater portability of the REDUCE code, it is however, somewhat undesirable for a mathematical system, since the same symbol in different case is often used to represent different quantities.

By default, REDTEN is a case-sensitive system: it clears the control switches `raise` and `lower` at startup (these switches can be set by the user with the commands `on` or `off` [6]). In an upper-case system the `raise` switch is on, while the `lower` switch is off, the reverse may be true in lower-case systems, but the exact combination is implementation dependent. Improperly setting these switches can trap the user in the system, with no way to gracefully exit or reset the switches (the REDUCE quote character “`!`” can be used to inhibit the case-mapping of the system, see below).

If the REDUCE system is upper-case by default, a package to map the names of

[6] In some systems `lower` may not be declared as a switch, `on` and `off` will still set the switch, but will also give an error message.

many of the commonly used operators and functions to lower case is available[7] (see §7.4); this is not needed if the system is by default lower-case. In this document it is assumed that the REDUCE system is lower-case.

1.2.2 Notation

Throughout this document we shall use the following notation: square brackets (`[]`) shall denote the default value of a variable, braces ({}) denote optional items, and a name enclosed in angle brackets (`<>`) denotes the current value of that name. Square brackets are also used to delimit indices in REDTEN, however the usage should be clear from the context. REDUCE and REDTEN keywords, variables, and functions will be indicated in `typewriter font`. User input will be shown in a *`slanted typewriter font`*.

Certain REDUCE and REDTEN variables contain special characters that are not ordinarily considered to be alpha-numeric. These characters (most often '*', and occasionally '-') must be quoted with the escape-character, `!`, so for example to enter `coords*` one must type `coords!*`. Forgetting to escape the special character can have unexpected consequences, since these special characters usually have other meanings by themselves. In this manual we shall always indicate the escape character as if it were an actual part of the name, however, depending on the print mechanism used, identifiers with special characters may or may not be displayed with an embedded escape-character. The escape-character can also be used to suppress the case-folding of the system, if this has been re-enabled.

When showing the format of a call to a function, the type of each argument is shown following the argument name and separated by a colon. In some cases the type of the return value is indicated after the function call template. The argument types are described in C.1.

1.2.3 Lisp Variables

There are two operating modes in reduce: `algebraic` and `symbolic`. The first is the normal mode for REDUCE in which ordinary algebraic expressions may be entered. The symbolic mode, which accepts RLISP syntax, is an interface to the lower level lisp system, and allows the entry of more primitive operations. The setting of the control variable `lisp:!*mode` determines which mode is currently active. The commands `lisp` or `symbolic` [8] by themselves change the system to symbolic mode; if they precede a command on the same line, they cause only that command to evaluated in symbolic mode. The command `algebraic` acts similarly to change to algebraic mode.

Nearly all of the user interaction with REDTEN involves the usual REDUCE algebraic variables. However, some REDTEN control variables are defined in the underlying lisp system, and require care in setting or modifying. Most of these are default names for objects created by various functions, and a few are lists. Lisp lists are also used as arguments to some REDTEN functions and are used because they are the most convenient

[7] Only the lisp name `nil` cannot be mapped by this package, the user must ensure that this is entered in the correct case.

[8] Apparently, `symbolic` is the preferred command.

way to represent and input certain parameters and properties of tensors, such as the index structure or symmetries. A list has the form of a single quote mark `'`, followed by a parenthesized list of elements separated by spaces (*not* commas):

```
'(t h i s i s a l i s t).
```

A special case is the empty list, which is represented as `'()`. A special lisp symbol, `nil`, is used to represent the empty list and it is the usual way the lisp printer displays empty lists.

The REDUCE parser, when it encounters a quote mark on the first argument to a function, automatically parses the function call in `symbolic` mode, so that the following are equivalent[9]:

```
% this is a crummey example!
#: val := '(q w e);

(q w e)

#: lisp (val := '(q w e));

(q w e)
```

The brackets enclosing the command following the `lisp` command ensure that the whole command line is treated in symbolic mode.

The variable `val`, in the above example, is a "lisp variable", and its lisp value can be examined with

```
#: lisp val;

(q w e)

#: val;

val
```

and observe that the algebraic value of `val` is not affected. In this document, lisp variables used by REDTEN will be distinguished from algebraic variables by prepending `lisp:` to the variable name, so that, for example, `lisp:coords!*` is the lisp variable containing a list of the current coordinates.

In a similar manner, there are a number of functions that can only be accessed while in symbolic mode, these are not required for the ordinary operation of the system, but are occasionally of use. The property functions described below are of this type. These will also be indicated by prepending "`lisp:`" to the name, and it is understood that to use these functions, the command line should begin with the mode changing commands `lisp` or

[9]The REDUCE prompt consists of a sequence number followed by a colon. In all examples in this document the prompt is represented as `#:`.

`symbolic` . Throughout this document, the term "function" is applied to numerous names in REDTEN; it should be understood that these names are in fact the user handle through which the true lisp function is found and executed. If the user looks for a definition of, say, `riemann`, it will not be found, rather a property value under the key `simpfn` indicates that the real lisp function that is executed is named `riemann!*` (however, see §E.1 for an important consideration).

1.2.4 Property Lists

Lisp variables can have, in addition to a value, a list of associated pairs and flags called a property list (often abbreviated to *plist*). Most of the information kept both by REDUCE and REDTEN is in the form of property lists attached to various lisp identifiers. While the REDTEN user generally does not need to access a property list, there may be times when it is useful to make corrections or alterations directly. It is also useful to know how to examine the property list either for the sake of curiosity, to fix a problem, or to "cheat".

Although referred to as a list, the property list is more conveniently thought of as a series of keyword-value pairs, which can be accessed with the lisp functions `put()` and `get()` , and a set of flags (which are either present or not) that are accessed with the functions `flag()` and `flagp()` . Properties and flags can be deleted with the functions `remprop()` and `remflag()` . The entire property list can be accessed or replaced in REDTEN with the functions `prop()` and `setprop()` , these are defined in terms of standard lisp functions that vary from system to system (see appendix E). The formats of the calls to these functions is shown in C.1.

Chapter 2

Indexed Objects

This chapter is concerned with the basic data structure (the "indexed object") used in REDTEN and how it is created and accessed. The first part of the chapter describes the basic concept of the indexed object, its properties and how it is used with an index. The second part uses the Robertson-Walker metric as an example to illustrate the user interaction with the system.

2.1 The Indexed Object

The fundamental entity in REDTEN is the "indexed object", which consists of an identifier (a name) and an index enclosed in square brackets[1], for example:

```
R[a,b,c,d], g[a,1], G[1,2].
```

An index can either be composed of strictly non-negative integers (each of which must lie within the inclusive range for that index-element, see below) and which indicates a specific component of the indexed object, or it can contain identifiers that represent the full range of possible indices, or a combination of both. The contents of the index are described more fully in §2.3 below.

An indexed object is a REDUCE kernel: it is an irreducible algebraic entity. As such it can be part of any algebraic expression, although, depending on circumstances, it may or may not be evaluated to a "simpler" form[2]. Before using an index with a name that name must be made into an indexed object with the `mkobj()` command[3]. If an index is applied to a name that has not been declared as indexed, the system will prompt for the basic data required to define the object (the index-type and symmetries, defined below). It is advisable to fully declare indexed objects before they are used.

[1] Formally, this is an indexed object reference, but the term is used both in this case and for the data structure itself.

[2] i.e. something that is no longer an indexed object reference.

[3] In previous versions the `mktnsr()` function was used. This still exists for backward compatibility.

2.2 Using mkobj()

The user call to `mkobj()` has the form:

mkobj ('name:*id* or :*id-list*, 'indextype:*int-list*,{'symmetries:*list-list*},
{'implicit:*bool*}, {'itype:*id*}):*iobj*;

The first two arguments are required, the others are optional. If any intervening argument is undesired, use the value `'()`. Because some of these parameters are lists the input line must be parsed in symbolic mode. The usual way to ensure this is to apply a quote-mark to the object name. This will be made clearer in the examples below. `mkobj()` prints the indexed object with an attached alphabetic index (i.e. `a, b, c,` ...) to show the index-structure of the object created. Each of the arguments to `mkobj()` is described below.

2.2.1 name

In REDTEN the name of an indexed object is reserved: it stands for itself and cannot be assigned an algebraic value. The name carries a property list containing all the information needed to fully describe the object, including the index structure, symmetries, and component values. If the switch `mkobjsafe` is on and if `name` already has an algebraic value, dependencies, or is flagged `used!*` then `mkobj()` will warn that the name is in use and exit. This is intended to protect the user from accidentally supplying a name that might be part of an algebraic expression and which might lead to conflict between the uses of the name. The `used!*` flag (a REDUCE internal flag) is set whenever a name is typed to the algebraic parser (hence it also includes names with values or dependencies), thus it may be that this behaviour is somewhat restrictive. If the user immediately repeats the call to `mkobj()` with the same object name, the `mkobjsafe` switch is over-ridden and the object will be created.

In many cases an indexed object name can be used as a variable without serious difficulty, but problems, especially where dependencies are involved, can occur. If `mkobjsafe` is off, and if `<name>` originally had an algebraic value, that value is lost, and in fact, any values or properties associated with `<name>` are removed when it is made into an indexed object. The only other time `mkobj()` will fail to create a new object is when `name` is that of a kill-protected indexed object (see 5.2), or has been flagged `reserved`. If the user is creating an object whose name is that of a generic object (qv), the generic object is removed and a warning message is printed. The name of the new object is stored in the list `lisp:indexed!-names` .

As noted above, it is best to apply the lisp quote-character ' to the object name. Any special characters in the name must also be quoted with the escape character `!`. Because the name has been quoted, it cannot be an expression or operator that evaluates to a name. The `name` argument to `mkobj()` can also be a quoted list of names, each of which will be created with the same index structure and symmetries.

indextype element	type of index	display format	default range
—	scalar	—	—
0	array	a	0–3
1	tensor	a	0–3
2	frame	(a)	0–3
3	spinor (unprimed)	a	1–2
4	spinor (primed)	a'	1–2
5	dyad (unprimed)	a	1–2
6	dyad (primed)	a'	1–2
≥ 7	user-defined	user-defined	user-defined

Table 2.1: Index types and display formats

2.2.2 indextype

The structure of the objects' index is determined from the `indextype` list, a list of integers whose length is the number of indices that may be applied to the name (the rank of the object). Each integer element indicates the type of the corresponding index element, and the sign indicates whether it is covariant (negative) or contravariant (positive).

Table 2.1 shows the allowed types of indices and the format in which the example index element `a` will be displayed. For example, the `indextype` list `'(-1 -1)` indicates a rank-2 tensor with covariant indices. Whenever it is necessary to display an index with the object, the indices are printed according to both their type and location. Frame indices are enclosed in parentheses, while primed spinor and dyad indices are followed by a prime mark '. User defined index-types can be declared with the function `defindextype()` (qv). Covariant indices (with negative `indextype` elements) are displayed on the line below the object name, while contravariant indices are displayed on the line above. If the REDUCE switch `nat` is off, then the index is displayed following the object name and enclosed in square brackets, and no formatting is done.

When an object containing a spinor or dyad index is made, a conjugate object is automatically created, having each spinor `indextype` element exchanged with the other spinor type (i.e. primed indices become unprimed, and unprimed indices become primed) as defined by the control variable `lisp:conjugateindextypes!*` . This conjugate object is given the name `<name>_cnj` and forms a coupled pair with the parent object. The `printname` (qv) is the same as the parent object, but a bar is printed over the name to indicate conjugation. See §6.2 for more about conjugation operations applied to spinor and dyad objects.

The first line of table 2.1 shows no `indextype` for a scalar object (a rank-0 object), the `indextype` in this case is the empty list `'()` or `nil`. An indexed scalar must have an empty index attached (i.e. `[]`), it can only be non-empty if the first operator indicates differentiation. The index is printed as `[]` so as to indicate the indexed nature of the name. If `<name>` previously had an algebraic value this now becomes the indexed value of the scalar object. In general, an indexed scalar can be treated exactly as any other indexed object.

To create an object that is closely related to another is simplified by an extension to the above syntax. If an indexed object name is supplied instead of a list, the `indextype` parameter of that object is used to create the new object. If an indexed object name appears as an element in the index-type list, the `indextype` parameter of that object is inserted into the index-type list allowing the user to build objects of higher rank.

The `indextype` list is stored on the property list of `<name>` under the key `indextype`
.

2.2.3 Symmetries

After the structure of the index the most important property of an indexed object is the description of the intrinsic symmetries among its indices. The symmetries relate permutations of the index to a canonical form, and the values of the components so related are either identical or differ by a sign (and may be complex conjugates as well). A complicated symmetry can reduce the number of independent components of an indexed object from hundreds to just a few, and since all the related components are so easily obtainable, only the canonical components need be stored. This greatly reduces the amount of storage required for an object, and also ensures that the symmetrically related components are in fact consistent with one another. The symmetries are also used to find a canonical form for any index applied to an indexed object, which allows some simplification and cancellation of expressions even before the components are examined.

The canonical form of an index is that which has the lowest ordered indices moved to the left as controlled by the symmetry. The ordering of indices is "numeric-alpha", i.e. numerical indices are ordered lower than alphabetic indices. This rule is altered when an index-element is being shifted (qv), in which case indices shifting up are moved left, and indices shifting down are moved right. See the examples below.

The symmetry description in REDTEN is in the form of a list of lists, each element of which describes a specific relation between blocks of the object indices (an "independent" symmetry). A "block" is defined as a group of adjacent index elements that move together under the influence of the symmetry. The location of the n^{th} index element is often called a "slot", because the index elements are "dropped" into it when using the symmetries.

Four kinds of symmetry relations are supported:

- Positive symmetry — no sign change on odd permutations
- Negative symmetry (also called an anti-symmetry or a skew-symmetry) — sign change on odd permutations
- Trace symmetry (a diagonal symmetry) — a pseudo-symmetry similar to the positive symmetry, but the index elements must be identical, or the value is zero, and the block size must be unity.
- Hermitian symmetry — for spinor indices a Hermitian symmetry is implied.

Each of these may relate any number of adjacent blocks of index elements in an index.

The list describing a single independent symmetry relation is of the form:

$$(\{c\}\ b\ p_1\ p_2\ \ldots)\ \text{or}\ (\mathrm{h}\ p_1\ p_2\ \ldots)$$

The first form is used to describe the first three kinds of symmetry relations, while the second describes the Hermitian symmetry.

In the first form, b is the size of the block of indices that will be related by the symmetry. The sign of b indicates whether the symmetry is positive or negative. If $b = 0$, then a trace symmetry is implied, but the block size is unity. The p_i are pointers, starting from 1, to the beginning of each block of size b in the index. The optional flag 'c' (the literal character, in either upper- or lower-case) indicates that a conjugate is to be performed on odd permutations. Some examples follow.

`'(1 1 2)` — a simple symmetry, typical of metrics

`'(-2 1 3)` — an anti-symmetry in the first and second block of two indices, the outer Riemann symmetry

`'(0 1 2)` — a trace symmetry, used for diagonal metrics.

The Hermitian symmetry is indicated by the second form (where 'h' is a literal character, in either upper- or lower-case), and can be applied only to spinor indices, which must come in adjacent unprimed and primed pairs (i.e. the index-types are 3 and 4). If no pointers are given, the system will construct a Hermitian symmetry for the entire index.

Combining Symmetries

The independent symmetries can be combined to give more complicated relations among an objects' indices. The simplest combination is a set of symmetries that do not overlap, i.e. the indices involved in one symmetry are not involved in any other. In this case, the symmetries are simply listed one after another; it is conventional to order them by each leading pointer. Interleaved symmetries are not supported.

If the symmetries are nested, more care is required (see also §2.2.3 below). A symmetry that fits into a block of a larger symmetry should be matched by similar symmetries in each remaining block, if this is not done the system will generate incorrect results. The ordering of the symmetries is very important: the small non-overlapping symmetries must come before the larger ones that encompass them. The ordering should be first by block size, and then by leading pointer. The correct ordering is not checked by REDTEN, and unpredictable (and incorrect) results are likely. A Hermitian symmetry, if any, must be the last indicated.

The full description of an objects' symmetries is a list of independent symmetries, in order by block size and leading pointer. Some typical examples follow:

`'((-1 1 2)(-1 3 4) (2 1 3))` — The full Riemann symmetry: anti-symmetric in the first pair of indices, anti-symmetric in the second pair (the third and fourth indices), and symmetric in blocks of two indices.

`'((1 1 2))` — The symmetry typical of a metric (or many other rank-2 symmetric objects).

`'((c 1 2 3))` — An object symmetric in the second and third indices, where odd permutations also involve a conjugation (this is used for the spin-matrices, see §6.2).

As with the `indextype` parameter, an extension to the symmetry list syntax allows the user to create a new object with symmetries derived from another object. If an indexed object name appears in place of the symmetry list, that objects' symmetries are used for the new object. If an indexed object name appears in the symmetry list, that objects symmetries are inserted into the symmetry list for the new object. Currently, no adjustment of pointers is made, so this is useful only if new symmetries can be added after those of the other object.

Constraints

There are several constraints on the symmetry list that are checked by REDTEN, violating any of these will result in an error message.

- the block-size b must be an integer.
- there must be at least two pointers to each independent symmetry.
- the pointers must be non-negative integers, the first index element corresponds to $p = 1$.
- the pointers must be in ascending order.
- blocks cannot overlap.
- the corresponding `indextype` sublists must be identical (unless the symmetry is a trace symmetry).
- a block cannot extend beyond the end of the index.

There are several other constraints, as already mentioned, but a direct check for these is not made, and failing to observe them will lead to incorrect results.

The user should be aware that the internal form of the symmetry list is somewhat different, and is held on the `symmetry` property of the object.

2.2.4 implicit value

REDTEN stores the components of an indexed object in a list attached to the `tvalue` property of the object. The list is sparse: if a component is missing it is assumed to be zero. These components are referred to as "explicit" components. If the object is made with the implicit flag set, each non-existent explicit component is replaced by an "implicit" component (unless the symmetries prevent this). The implicit component is simply the object name and the index of the component. This is particularly useful for unknowns such as the Killing vectors, whose components are to be solved for from a set of equations. In this case, however, explicitly zero components must also be stored. See below for some examples of implicit objects.

2.2.5 itype

The final user-settable property is the `itype` of the object being created. The meaning of the `itype` is usually defined by the user, although the system has some types defined, such as `metric`, `connection`, and most functions set the `itype` of their output to something meaningful. Because some functions check the `itype` of their argument, `mkobj()` causes the user-entered value to be mapped to the native case of the underlying lisp system. Therefore, entering `itype` parameters in mixed case is pointless.

2.2.6 Other properties

There are a number of other properties attached to an indexed object either by `mkobj()` or by other functions in the system. A partial list of these property keys and the type of their values follows. Those flagged with an asterisk are shown by the `iprop()` function.

∗`altmetric:`*id-list* — A list of metrics to be used instead of those in `lisp:currentmetric`, settable by the user with the `altmetric()` function.

∗`conjugate:`*list* — The name (in a list) of the conjugate object associated with a spinor. The conjugate object has the primary object's name and a flag under this key.

∗`coords:`*id-list* — A copy of the `lisp:coords!*` environment variable at the time the object was created (see §2.4.1).

∗`cov:`*list* — The first element of the list is the order of the derivative, the second element is the name of the anti-derivative of this object (`nil` if this is the parent), the third element is the name of the derivative of this object (`nil` if it does not exist).

∗`description:`*string* — a description string that may be set or examined via the function `describe()`.

`det:`*aexp* — The value of the determinant of the object (which must be rank-2), as computed by the `det()` or `determ()` functions.

∗`div:`*id* — The name of the divergence object, created from this object by the function `div()`.

∗`indexed:`*id* — the property by which the system identifies indexed object names. The value under this property will be one of:

- `array` – if all indices in `indextype` are array indices
- `tensor` – if all indices are tensor indices
- `frame` – if all indices are frame indices
- `spinor` – if all indices are either primed or unprimed spinor indices
- `dyad` – if all indices are either primed or unprimed dyad indices
- `scalar` – if `indextype` is the empty list, indicating a scalar object

`mixed` – if `indextype` contains more than one kind of index.

other values may appear as defined by the user (see `defindextype()`).

$*$`indextype:`*int-list* — The user-input `indextype` list.

$*$`implicit:`*id* — The implicit parameter, usually the same as the object's name.

$*$`indices:`*list* — An internal representation derived from the `lisp:defindextype!*` lists.

$*$`multiplier:`*aexp* — A common factor, most often non-unity for metric (or array) inverses.

$*$`parent:`*id* — For a shifted object, this is the object from which it was derived.

$*$`printname:`*id* — The name used to display the object in indexed form. This may be different from the actual object name if the object is generic (see §4.3) or has shifted indices (see §3.1).

$*$`protection:`*int* — The internal format of the protection flags, the numbers 2, 3, or 6, which correspond to write-prtotection, kill-protection, or both; or `nil` for no protection.

$*$`odf:`*list* — The list is in the same format as for the `cov` property; it holds the names of ordinary derivatives.

$*$`restricted:`*list* — A restricted `indices` property, set by the user with the `restrict()` function.

$*$`shift:`*list* — A list of names of shifted objects derived from this one. Each of these objects will not have this property, but will have the `parent` property instead. See 3.1 for more details.

$*$`symmetry:`*list-list* — The internal format of the user input `symmetry` list.

$*$`itype:`*any* — The user input `itype` parameter.

`tvalue:`*alist* — the association list of indices and component values.

2.2.7 `mkobj()` Examples

Following are some simple examples of the creation of an indexed object, and the use of symmetries to canonicalize the index.

```
#: mkobj('q,'(1 1));     % q is a rank-2 contravariant tensor.
  a b  q
% w is like q, but is also symmetric.
#: mkobj('w,'(1 1),'((1 1 2)));
  a b
 w
% r is covariant, and is also flagged implicit.
#: mkobj ('r,'(-1 -1),'(),'t);
 r
  a b
#: q[b,a];   % this index will be unchanged.
  b a
 q
```

```
#: w[b,a];       % because w is symmetric, the index is mapped to a
             % canonical form.
  a b
 w
#: q[1,2];  % There are no components in q, so zero is returned.
 0
#: r[1,2];  % Since r is implicit, the implicit value is returned.
 r
  1 2
```

Finally, if `mkobj()` is given a single name as an argument, without even an empty index-type list, a "bare" declaration is made. The name is given the minimum number of properties that allow it to be correctly parsed with an index; it is assumed that the object will be correctly made later, usually during an indexed assignment. This use is somewhat sloppy, however, and is not encouraged.

2.3 The Index

Except for access granted by specialized functions, the index is the only way to manipulate the components of an indexed object. The index is also used to indicate operations to be performed on the object as a whole during the evaluation of an algebraic expression (see below, and Chapter 3).

As has been mentioned, the index itself consists of a set of integers and/or identifiers, separated by commas and enclosed in square brackets. There are two distinctly different kinds of index possible:

1. all indices are integers, the index then refers to a specific component of the object, which can be immediately read-out or assigned,
2. at least one index element is not an integer, and the index therefore is a pattern representing a set of components.

In the first case the index is called a "fixed" index, and in the second case it is called a "pattern" index.

The identifiers used in a pattern index are (aside from the indices used to indicate contractions) completely arbitrary. As such, they are not evaluated, so that any values they might have do not interfere with their use as indices. For example, the algebraic value of `a` does not affect its use in the index of `g`:

```
#: a := 324$
#: g[a,b];       % no problem with ``a'' having a value.
   g
    a b
```

To accommodate the possibility where the user may wish to use a REDUCE `for` loop to, for example, copy an indexed object to a REDUCE matrix, the switch `evalindex` is provided. If on, this switch causes all identifiers in an index to be evaluated; of course, if they do not evaluate either to an integer or an identifier, an error will result (in the above example, with `evalindex` on, an index-out-of-range error would occur).

There are three operators that can be used in an index which denote operations that are applied to individual indexed objects: differentiation, raising and lowering of indices via metric contractions ("shifting"), and symmetrization of indices. As of REDTEN v4.00 this last operation can also be applied across a product of indexed objects using the function `symz()` .

In general only the symmetrization operation is performed immediately (but this can be suppressed if the switch `symzexp` is turned off, see below), the other operations are held pending the use of the object in an indexed assignment (the subject of Chapter 3). In special cases, as noted elsewhere, the operations may be evaluated immediately.

2.3.1 Differentiation

In one conventional notation, a vertical bar placed after the index indicates differentiation, while a double bar indicates covariant differentiation (in another notation, the comma and semi-colon are used, but as these are special symbols to lisp and REDUCE they cannot be used here). In REDTEN the differentiation symbols are naturally `|` and `||`. The operation is indicated by placing these symbols after the normal object indices (i.e. after the number of normal index elements equal to the rank of the object) and adjacent to the following index element, for example: `g[a,b,|c]`. The comma preceeding the `|` must not be left out, or the parser will become confused (a comma *after* the `|` is harmless).

Any index with more elements than the rank of the object must indicate a differentiation with the first extra element. The remaining elements are taken to also be derivative indices, and do not require a differentiation symbol (unless the type of differentiation changes). Covariant differentiation is always done one index-element at a time (i.e. intermediate objects are created); it also requires the existence of the Christoffel symbols (which will be created from the tensor metric if need be), see §4.2. An ordinary derivative of an indexed object is computed by taking the derivative of each component with respect to each of the coordinate names stored on the `lisp:coords!*` variable, see §2.4.1. In either case a new object is created to store the derivatives, see §4.1 and §4.2.

2.3.2 Shift Operations

The process of raising and lowering indices via metric contractions (called shifting in REDTEN) is indicated by placing the `@` symbol before the index element that is to be moved. This generates a contraction with the appropriate metric and creates a new object. Shift operations are processed by the function `shift()` , described in more detail in §3.1.

2.3.3 Symmetrization

It is frequently the case in General Relativity that a new object of interest is constructed from some other object (or objects) by permuting its indices and adding the resultant objects together. These symmetrization operations can be indicated in the index of an object by enclosing the indices to be permuted in symmetrization operators: `:[` and `:]`; or anti-symmetrization operators: `:{` and `:}`[4]. The anti-symmetrization operators cause a sign change when swapping adjacent index elements, whereas the symmetrization operators do not. Using the objects created on page 14 we have, for example,

```
#: q[:[a,b:]];
    a b    b a
  (q    + q   )/2
#: w[:[a,b:]];
    a b
   w
#: w[:{a,b:}];
   0
#: 3*R[a,b,:{c,d,||e:}];
  R            - R             + R
   a b c d ||e    a b c e ||d    a b d e ||c
```

Each time the index is permuted according to the indicated symmetrization, the intrinsic symmetries of the object are applied to put the index into a canonical form. Thus it may be the case that significant simplification can occur before the actual evaluation of the expression is begun. The last expression above is an example of how the Bianchi identities can be calculated, assuming `R` had been given values.

Bach brackets can be used to isolate some indices from the symmetrization, and these are indicated by writing a `:&` symbol in the index as if it were a separate index element, i.e. it has commas on both sides, unlike the `@` and `|` operators.

```
#: R[:[a,:&,b,c,:&,d]:];
 (R        + R       )/2
   a b c d    a c b d
```

With REDTEN version v4.00 the previous restriction that a symmetrization involve no more than 4 indices was removed (although it will take much longer to evaluate larger operations, as the number of terms grows with the factorial of the number of indices). Also new in this version is the function `symz()` , which can be used to apply a symmetrization across the product of several indexed objects. The system still cannot apply intermixed symmetrizations separately to covariant and contravariant indices.

To use `symz()` simply give an indexed expression as the single argument to the function. At least one pair of matching symmetrization operators should appear (it's a no-op otherwise), but they are no longer restricted to appearing in the *same* index: the opening

[4] The use of these operators is new in v4.00.

op can appear in one index, and the closing op in another. Obviously, the opening op should appear before the closing op (the usual REDUCE reordering of expressions is suppressed until the symmetrization has been evaluated). Writing a distributed symmetrization that is not the argument to `symz()` will result in an error. For example,

```
#:  symz(q[a,:[b]*w[c:],d]);

   a b  c d    a c  b d
  q    w    + q    w
 -----------------------
            2
```

All the operators and rules that apply to a symmetrization in a single index apply here across the combined index of the terms in the expression.

If the switch `symzexp` is off, the symmetrization operations are not evaluated until the expression is involved in an indexed assignment. In this case, `symz()` returns unevaluated:

```
#: off symzexp;
#: symz(q[a,:[b]*w[c:],d]);
  a [b  c] d
 q     w
```

2.4 The Robertson-Walker Metric

The Robertson-Walker (RW) metric is well known in General Relativity as a representation of a universe filled with a pressureless dust. The line-element is usually written in the form:

$$ds^2 = dt^2 - R(t)^2 \cdot \left[d\omega^2 + s^2 \cdot \left(d\theta^2 + \sin^2(\theta)d\phi^2\right)\right] \tag{2.1}$$

where $R(t)$ is the time-dependent scale factor, and $s = \sin(\sqrt{k}\omega)/\sqrt{k}$ for $k = 0, \pm 1$.

In terms of a metric tensor, the RW metric is written as

$$g_{a\,b} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & -R(t)^2 & 0 & 0 \\ 0 & 0 & -s^2R(t)^2 & 0 \\ 0 & 0 & 0 & -\sin^2(\theta)s^2R(t)^2 \end{pmatrix}.$$

We shall use an indexed object to represent this metric, as described below.

2.4.1 `coords()`

Some of the properties of objects newly created by `mkobj()` are derived from lisp variables that define the "tensor environment" of the system, namely `lisp:coords!*`, which contains the coordinate names; and `lisp:defindextype!*`, which contains the definitions of index-types and the runs of indices (i.e. the integer range over which an index of a given type can vary). The length of the `lisp:coords!*` list is the dimension of the space time; the default value is `'(t r th ph)`. The `lisp:coords!*` variable is set with the function `coords()` and the `lisp:defindextype!*` variable is set with the function `defindextype()` .

The `coords()` function takes the list of coordinates as its first argument, and an optional second argument is used to determine the starting value for the tensor index-run, the default is 0. The default ranges of indices are shown in table 2.1. The `coords()` function also creates a coordinate-vector whose default name is the value of `lisp:mkcoords` .

For example, to change from the default spherical coordinates to those appropriate to the Robertson-Walker metric:

```
 #: coords '(t om th ph);
(t om th ph)
```

(Observe that as a function with one argument, the enclosing parenthesis are not required, but are required in the example below, where `coords()` has two arguments). If the user prefers to regard $x^4 = t$, then the following would be used:

```
 #: coords ('(om th ph t), 1);
(t om th ph)
```

and the tensor index range will be 1–4. `coords()` prints the new coordinate list, or the current list if no argument is given. The old coordinates are saved in `lisp:oldcoords!*` . The old coordinates can be reset with the command

```
 #: lisp coords oldcoords!*;
(t r th ph)
```

2.4.2 Entering Components

To make use of this metric we must first set the correct coordinates as above, and then create an appropriate indexed object and enter values for its components. Since this is a metric tensor, it will have two covariant tensor indices, and will be symmetric:

```
#: mkobj ('h, '(-1 -1), '((1 1 2)), '(), 'metric);
 h
  a b

#: depend rt, t;
#: let s = sin(sqrt(k)*om)/sqrt(k);
```

Notice that we have set up the object so that the `itype` is `metric`, and have used an empty list for the undesired `implicit` parameter. It is common to use the name `g` for a metric, but in REDTEN it has been made into a “**generic**” (qv) name, and should not be disturbed; we shall use h in this example instead. We have also set up the time-dependence of rt (using rt instead of R , since R is generic too) and defined a let rule to evaluate s. This last part could be done as easily with a REDUCE assignment.

There are two ways to enter the values of the components of h. The first method involves the expected assignment operation, where the left-hand side, referring to a specific element with a fixed index, is assigned the value on the right hand side. Unlike the normal REDUCE assignment operation which uses the symbol :=, the REDTEN indexed assignment operation uses the symbol ==. Attempting to assign a value to an indexed object element with the := operator will result in a REDUCE Missing Operator error message[5]. We can proceed as follows:

```
#: h[0,0] == 1;
   1
#: h[1,1] == -rt^2;
      2
   -rt
#: h[2,2] == -s^2*rt^2;
                     2   2
   - sin(sqrt(k) om)  rt
 -------------------------
             k
#: h[3,3] == -sin(th)^2*s^2*rt^2;
                     2        2   2
   - sin(sqrt(k) om)  sin(th)  rt
 ----------------------------------
                 k
```

As each value is entered, the system then reads the object to determine the actual value stored for that component, since it is possible for the assignment to fail due to symmetries or write-protection of the object. These are explicit components for h, and since h was not declared as an "implicit" object, any missing components are taken as 0. If any component already had a non-zero value, that value is overwritten (unless the object is write-protected).

When a large number of elements are to be assigned to the same object, a more convenient method is to use the function ias() . This function takes an indexed object as input and guides the user through the assignment process. At its simplest, ias() attempts to assign to the whole object, keeping in mind the intrinsic symmetries that are present. The user may also attach an index that indicates the assignments are to be to a particular row or column of the object. Using ias() the assignment to h would proceed as follows:

[5] If this happens, the function mclear() should be called to clear the parser.

```
#: ias(h);
  h[0,0] = 1;
  h[0,1] = ;
  h[0,2] = ;
  h[0,3] = ;
  h[1,1] = -rt^2;
  h[1,2] = ;
  h[1,3] = ;
  h[2,2] = -s^2*rt^2;
  h[2,3] = ;
  h[3,3] = -sin(th)^2*s^2*rt^2;
  h
```

At each prompt the user may enter the desired value; a semi-colon alone is equivalent to zero. If the user types `nil`[6] then the assignment process is terminated, and any remaining unassigned components will be unchanged.

The function `iclear()` can be used to remove all of the components of several indexed objects (and the multiplier), without disturbing any of the other properties.

So far we have created only a rank-2 object with some symmetries and values, the rest of the structure associated with a metric is not yet present. The function `metric()` is used to fully qualify an object as a metric: it attaches the required properties, adjusts some system variables and creates the metric inverse. We may apply `metric()` to `h` very simply:

```
#: metric(h);
 invert finished.
 h
```

At the end of this process `h` has been declared the "current-metric", and its name is installed in the system variable `lisp:currentmetric` . Among other things, this means that `h` will be used by `shift()` when raising or lowering tensor indices. The generic metric `g` will also refer to this object (see 4.3). The metric inverse will be given the name `h_inv`, it is created from the metric by the matrix function `invert()` (qv). The indexed object passed to `metric()` must be a rank-2 covariant object, and is assumed to be symmetric. If it is actually diagonal (as this example is), the symmetry will be adjusted to reflect this. A diagonal symmetry helps the system to run faster for many operations involving the metric, particularly in the raising and lowering of indices with `shift()` .

`metric()` can also be used to create a new indexed object suitable for filling with metric coefficients. To do this, give `metric()` no arguments, the object it creates will have a name in the usual sequence of metric objects described below. After the user has entered the metric coefficients, `metric()` should be called again as demonstrated above to finish the job of setting up the metric. In this situation, the newly created object must

[6] See 7.4 for information about entering `nil` in two-case systems.

be accessed by name, it is not the target of the generic metric, since it is not yet a proper metric.

A package for General Relativity is included in REDTEN (described in Chapter 4), and allows the user to enter a metric directly from the line-element:

```
#: ds2 := d(t)^2-rt^2*(d(om)^2+s^2*(d(th)^2+sin(th)^2*d(ph)^2));
                 2     2        2                 2        2   2
 ds2 := ( - d(om)  k rt  - d(ph)  sin(sqrt(k) om)  sin(th)  rt
                2          2                 2   2
         + d(t)  k - d(th)  sin(sqrt(k) om)  rt )/k
#: metric (ds2);
 computing g1
 invert finished.
 g1
```

In this case, the metric name is defined by the system. It will have the form `g<sequence number>`, where `<sequence number>` is initially 1, and increments for every new metric that is created in this fashion. An optional second argument allows the user to specify the name that will be used instead for the metric. This method is more convenient for simple metrics that are ordinarily presented in the form of a line-element, while the first method is often useful for very complicated metrics that are too troublesome to express in line-element form.

The line-element is written as a REDUCE expression containing the differential operator `d()` . This operator computes the total derivative of its argument with respect to the current coordinates; if the argument is one of the coordinate names, `d()` returns unevaluated. By examining the expression for `d()` operators the system can determine the structure of the metric, and can also tell if the metric is diagonal or simply symmetric.

Once the elements of the metric are in place, one needs a way to examine the object to ensure all is well.

2.4.3 Accessing and Displaying Components

There are a number of ways to examine the components of an indexed object. The simplest is to specify the fixed index that refers to the component of interest: the current value of the component will immediately be read-out, and is available for use in algebraic expressions. For example,

```
#: h[1,1];
      2
  - rt
#: df(h[2,2],t);
                                2
   - 2 rt  sin(sqrt(k) om)  rt
         t
 ------------------------------
              k
```

To examine the object as a whole, two display formats are provided. The first is obtained by attaching an empty index to the object, as in `h[]`. For an indexed scalar, this is the only kind of index allowed and results in the scalar value being returned. For any other kind of object the existing explicit components are displayed. Only those components that are actually stored in the data structure are shown, and no evaluation takes place, which also implies that a non-unit multiplier (qv) will not be combined with the component. Hence, an implicit object with no explicit elements will appear to be empty. Components shown by this display format may not match what would be shown by reading components with fixed indices. Because the index is empty some objects, notably those derived from the raising or lowering of indices of other objects, must be referenced with their actual name (or the generic equivalent). If the switch `iprop` is on, some of the properties listed in §2.2.6 are also shown.

The second display format is obtained when the literal character `?` is found (written as a normal index element) in a pattern or fixed index. Those indices preceding this special character are used as part (or all) of a pattern index, and any indices needed to fill out the index are automatically generated. Indices following the `?` are ignored. The simplest way to examine the entire object is with `h[?]` which (for a rank-2 object) is equivalent to `h[a,b,?]`, whereas to examine the first row one might use the command `h[1,?]`. This format will also display some of the object properties, depending on the setting of the `iprop` switch. The properties shown when the switch `iprop` is on can also be seen, regardless of the switch setting, with the function `iprop()` (qv).

The components are displayed by expanding the pattern index and reading each component from the object. Thus, the user can select segments of the object to display and will be shown the actual read-out value. If the switch `xeval` is on, an extra evaluation is made to reflect the current state of algebraic variables. This format shows exactly what is in the object, as would be shown by accessing each component directly with a specific fixed index, but only the canonical elements are shown. This format will also show those components that are zero (and usually are not stored and therefore not displayed in the first format) unless the REDUCE switch `nero` is on. The index can also be used to indicate a shifted object, so that only the parent name need be used, for example `h[@a,@b,?]`.

To see a directory of the objects created in the system, the function `dir()` is provided. In its simplest form, with no arguments, `dir()` displays all the objects in the system that have not been flagged with the `nodir()` (qv) function. The display shows the object name, its type, the number of non-zero explicit components, the state of the protection flags (qv), the coordinates and the index structure. With arguments `dir()` shows information about the named objects.

```
#:  dir();

   name          type          comp    prot  coordinates      indextype
   x             coordinates    4       w    (t om th ph)      (1)
 * h             metric         4       w    (t om th ph)      (-1 -1)
 * h_inv         metric         4       w    (t om th ph)      (1 1)
   3 objects,    Total components: 12
```

The current-metrics are indicated by the * at the beginning of the display line.

Chapter 3

Algebra with Indexed Objects

In this chapter we will show how to write expressions that both involve indexed objects, and which evaluate to indexed objects. It was intended from the outset in the design of REDTEN that the user be able to write tensor expressions in a convenient form, and have them displayed in as close to publication form as was practical.

Whenever the user enters an indexed object with a pattern index (an index in which at least one element is an identifier and not an integer) the indexed object is returned unevaluated (unless the intrinsic symmetries of the object allow an immediate simplification to zero). This has already been seen in the previous chapter, where the concept of an indexed object reference was demonstrated. Now we shall use this to build expressions that can then be used to create new indexed objects.

An "indexed expression" consists of the algebraic combination of indexed objects such that the indices involved obey the rules set out below. An indexed expression is a normal REDUCE expression and can be assigned to REDUCE variables, however, if this is done there is no checking of the index structure or other evaluation except for the usual canonicaliztion of indices until the expression is used in an "indexed assignment".

An indexed expression can be as simple as

```
#: r[a,b];
```

or as complicated as need be, given certain rules about indices and the combinations that are valid. In any indexed expression there will be some index elements that are non-integer (otherwise each indexed object reference can be immediately evaluated and a simple algebraic expression results). The labels used for each of these index-elements are arbitrary except for the following restrictions:

- In a product of indexed objects, a repeated index element represents the Einstein summation convention (a "contraction"), and the two indices must appear in covariant and contravariant positions. Having more than two repeated indices is an error, although this will not be indicated until the indexed assignment is evaluated. If the switch `extendedsum` is on, then the two repeated indices may also be both covariant or both contravariant.

- After the contractions are eliminated, the remaining indices must match the index structure of the output object in both location (i.e. either covariant or contravariant) and type.

These are of course just the usual rules for indices in tensor analysis.

The net index structure of the expression, obtained by eliminating the contractions and fixed indices, will be the same as that of the object to which the expression will eventually be assigned. Ideally, this object would be created by the user before the assignment takes place, so that proper checking of the index structure of the expression relative to that expected in the output object can be done. The user can also avoid declaring the object, in which case the system will prompt as soon as an index is placed on it, or the user can make a "bare" declaration (see page 15), and the system will determine the net index structure of the expression and place the `indextype` parameter on the object. This is not recommended.

The net symmetries of the expression are in general very difficult to determine by a simple algorithm, thus it is up to the user to correctly place these on the output object. Failing to do so can result in very long run times for some expressions, since the system uses the output objects' symmetries to generate the indices it uses to perform the evaluation.

Continuing with the Robertson-Walker metric example from the last chapter, we can now construct the two kinds of Christoffel symbols from the metric tensor. In tensor notation these are defined by:

$$[ij,k] = \frac{1}{2}\left(g_{ik|j} + g_{jk|i} - g_{ij|k}\right) \tag{3.1}$$

$$\left\{ {l \atop i\ j} \right\} = g^{lk}[ij,k] \tag{3.2}$$

where g is the metric tensor. Unfortunately, REDTEN does not have quite the flexibility of input for the user to use this exact formalism, so we must resort to making these rank-3 objects with, for example, the names `hc1` and `hc2` respectively. In actual fact, neither of the Christoffel symbols is a tensor, but in the GR package of REDTEN they are declared as such (with the generic names `c1` and `c2`), since certain combinations of these objects are tensors.

It can be seen that the first Christoffel symbol is symmetric in its first two indices i and j, while the second Christoffel symbol is symmetric in its second and third indices, again i and j, taking the upper index l to be the first. We can then declare these objects, giving them the proper index structure and symmetries:

```
#: mkobj ('hc1, '(-1 -1 -1), '((1 1 2)), '(), 'christoffel1);
 hc1
    a b c
#: mkobj ('hc2, '(1 -1 -1), '((1 2 3)), '(), 'christoffel2);
    a
 hc2
      b c
```

The Christoffel symbols can be evaluated by:

```
#: hc1[i,j,k] == (h[i,k,|j] + h[j,k,|i] - h[i,j,|k])/2;
                - h        + h        + h
                   i j |k     i k |j     j k |i
 hc1       = ----------------------------------
    i j k                   2
 hc1
    i j k
#: hc2[l,i,j] == h[@l,@k]*hc1[i,j,k];
    l        k l
 hc2       = h     hc1
      i j             i j k
    l
 hc2
      i j
#:  hc1[0,1,1];
  - rt  rt
      t
```

We first observe that indexed assignments are done using the same `==` operator that was used to write individual components to indexed objects, since that operation is a simple form of indexed assignment. Immediately after the input line has been parsed and depending on the setting of the switch `rewrite` the system will "pretty-print" the expression to show the user the index structure of the left and right hand sides. If these do not match, an error message will be forthcoming.

The system then begins the process of parsing the expression and evaluating each component for the left hand object. If the switch `showindices` is on, each components index is displayed on the screen as the assignment proceeds. This is useful for complicated objects, where the time to accomplish the assignment may be several minutes and it is desired to monitor the progress of the calculation. The proper display of these indices depends on the correct setting of the variable `lisp:upcursor` , see Appendix E for more information. A related switch is `peek` which also shows if the current component evaluates to zero or non-zero.

The expression for `hc1` involves the ordinary differentiation operator `|`; when this is encountered the system will create a new object to store the derivatives for future reference. This is described more fully in §4.1. In the calculation of `hc2` it will be observed that the shift operator `@` has been used on the metric indices. In general, when an indexed object reference is parsed, the system checks for these operators, and, if the corresponding shifted object exists, the reference is rewritten in terms of this object directly. Thus, typing `h[@a,@b]` is equivalent to typing `h_inv[a,b]`, since the inverse object already exists. If it did not, the operations would be held pending an indexed assignment, and at that time the `shift()` function would be called to construct the new object. That is, the system does as much simplifcation as it can on the expression as initially entered, but waits until it is involved in an indexed assignment to undertake any time-consuming calculations. The expression for `hc2` also contains a contraction representing the summation over the index `k`. It is here that the diagonal symmetry of the metric can be used effectively, since the summation in this case reduces to a single term.

It will also be observed that the indexed objects in the first of these calculations are involved in an algebraic expression (a quotient). This is a very simple case of a more general property of indexed expressions in REDTEN: an indexed object can be an argument to an algebraic function; the components of the indexed object are evaluated in the context of the function as the indexed expression is evaluated. For example,

```
#: k[a,b]==sub(s=1, h[a,b])$
#: l[a,b]==df(h[a,b],om)$
```

With the Christoffel symbols in hand, we can now proceed to evaluate the Riemann curvature tensor, which is defined in terms of the Christoffel symbols by:

$$R_{hijk} = \frac{d}{dx^j}[ik,h] - \frac{d}{dx^k}[ij,h] + \left\{ {l \atop i\ j} \right\}[hk,l] - \left\{ {l \atop i\ k} \right\}[hj,l]. \tag{3.3}$$

It can be shown from symmetry considerations that there are only 21 independent components of the Riemann tensor in a four-dimensional space-time, rather than the 256 components there appear to be[1]. The Riemann tensor is anti-symmetric in the first and second indices, anti-symmetric in the third and fourth indices, and symmetric in blocks of two indices starting at the first and third. In REDTEN notation, this symmetry list is written as `'((-1 1 2) (-1 3 4) (2 1 3))` and the Riemann tensor `hR` can be made with the command:

```
#: mkobj ('hR, '(-1 -1 -1 -1), '((-1 1 2) (-1 3 4) (2 1 3)), '(),
                                                          'riemann);

hR
  a b c d
```

Recall that the name `R` is declared generic in REDTEN and should not be used to create a Riemann tensor (the name `R` is however closely related to the Riemann tensors created with the GR package). The assignment to the Riemann tensor is made as follows:

```
#: hR[h,i,j,k] == hc1[i,k,h,|j]-hc1[i,j,h,|k]+hc2[l,i,j]*hc1[h,k,l]
  -hc2[l,i,k]*hc1[h,j,l];
                                  l                      l
hR          =  - hc1       hc2        + hc1       hc2
  h i j k          h j l       i k        h k l       i j
 - hc1         + hc1
     i j h |k      i k h |j
hR
  h i j k
```

[1]REDTEN does not make use of the cyclic symmetry that actually reduces the number of independent components of the Riemann tensor to 20.

If the user examines this object it will be seen that there are only 6 non-zero components. On a reasonably fast computer (eg. a 25Mhz 386 or a SUN) this calculation takes only a few seconds for this metric; for more complicated metrics it will of course take longer.

The next objects to create are the Ricci tensor and the associated Ricci scalar (also called the scalar curvature). These are defined by

$$R_{ij} = R^{k}{}_{ijk} = g^{lk} R_{lijk} \tag{3.4}$$

$$R = R^{i}{}_{j} = g^{ij} R_{ij} \tag{3.5}$$

The Ricci tensor is symmetric, hence:

```
#: mkobj('hRic, '(-1 -1), '((1 1 2)), '(), 'ricci);
 hRic
     a b
```

is the command to make the object. There are two ways to compute the components of both the Ricci tensor and the Ricci scalar, in REDTEN one method is somewhat more efficient than the other. The first method is to indicate a shift operation with the `@` operator, this will cause the system to create a new object (whose name is derived from the name of the object being shifted, see §3.1.2 for more detail). The internal contraction of the object is then done to yield the output object:

```
#: hRic[i,j] == hR[@k,i,j,k];
                  k
  hRic     = hR
      i j        i j k
computing hR_b
SHIFT finished.
  hRic
      i j
```

A side effect of this method is the creation of `hR_b`, a Riemann tensor with the first index raised, and, incidentally, fewer symmetries than the original Riemann tensor. Thus, it can take appreciably longer to compute this object since it has many more components (12 non-zero ones for this metric). Since we probably are not interested in this object this particular method is not the most efficient way to compute the Ricci tensor. Instead, we shall write the metric contraction explicitly, so that the system simply does a two-index contraction:

```
#: hRic[i,j] == h[@l,@k]*hR[l,i,j,k];
                               k l
 hRic     =  - hR          h
     i j          i l j k
 hRic
     i j
```

This calculation computes directly the object of interest and proceeds much more quickly than the first method. Since the contraction involves all the indices of the metric inverse, it will be read-out when the expression is processed (before the actual evaluation of output components begins) yeilding a substantial simplification. It will also be observed that the expression as echoed by the system is not identical to that typed in, since the indices have been canonicalized and, in the case of the Riemann tensor, a sign change was introduced.

We can similarly evaluate the Ricci scalar using the more efficient metric contraction:

```
#: hRsc == h[@i,@j]*hRic[i,j];
                    i j
 hRsc = hRic     h
            i j
                     2
  6 (rt    rt + rt    + k)
       t,2        t
 -------------------------
              2
            rt
```

Note that we did *not* declare `hRsc` in advance. The indexed expression on the right hand side evaluates to a scalar and REDTEN will accept a REDUCE scalar as the output object in this case. In fact, the left hand side can either be an indexed object or a scalar, and the right hand side can be any indexed expression, scalar expression, or simple scalar.

```
#: w[a,b] == 99; % write 99 into every component of w
  w
   a b
```

This example is not generally very useful unless perhaps to set every component of an object to zero. Of course, mixed indices may also be used, in order to limit the assignment to a row or column etc. Only if the right hand side is indexed is it necessary for the left-hand side to be similarly indexed. One can always assign an indexed expression to an object of higher rank; the extra indices may be fixed or pattern indices, if the latter, then the expression is assigned to every possible combination of indices. An error will occur if a pattern index-element on the right hand side is unmatched by a similar one on the left.

```
#: w[a,b] == p[a]; % this is ok
 w
  a b
#: p[a] == w[a,b]; % this is an error
Error: free index element b in w[a,b].
```

The first assignment will replicate `p` (assumed to have been created and given some values) into the columns of `w`. The second assignment fails because there is no index element corresponding to `b` on the left hand side (it is a "free" index element). If, however, `w` were zero (i.e. with no explicit components and also not implicit) then the assignment would proceed, because the expression

```
W
 a b
```

is replaced by zero before the index checking is done. If `b` were replaced by a valid integer then the assignment could again proceed.

As noted above, the index of the output object can be such that only a portion of the object is referenced during the indexed assignment operation. In this case, any components that are not included in the scope of the index remain unchanged; the new components are merged with the old. If the user decides that an indexed assignment applied to an object with previously defined components has been an error, the old value of the object can be recovered by calling `restoreobj()` immediately.

The function `seval()` is used to perform the evaluation of an indexed expression that reduces to a scalar, without the need for an indexed assignment using `==`. Thus, the dimension of the space-time can be found by

```
#: seval(h[@a,@b]*h[a,b]);
                 a b
 seval = h      h
          a b
  4
```

In fact, this function calls the indexed assignment operator and assigns the result to the name `seval` .

The left hand side of these scalar expression can also be an indexed scalar, declared by giving `mkobj()` an empty `indextype` property (see §2.2.2). In this case the empty index must be attached to the object name.

Another scalar of interest is formed by contracting the Riemann tensor with itself. The Kretschmann scalar is defined by

$$k = R^{abcd} R_{abcd}. \tag{3.6}$$

We can declare it to be an indexed scalar via

```
#: mkobj('hkrtch, '(), '(), '(), 'kretschmann);
 hkrtch
```

and evaluate it as follows:

```
#: hkrtch[] == hR[@a,@b,@c,@d]*hR[a,b,c,d];
computing hR_p
shift finished.
           2   2      4        2      2
  12 (rt      rt  + rt   + 2 rt   k + k )
        t,2          t        t
 ----------------------------------------
                      4
                    rt
```

In this instance we allowed the system to create the shifted Riemann tensor `hR_p` because it has the same symmetries as `hR` (and is not too expensive to compute), but the expression could have been written with the metric inverse explicitly. Note that forgetting the empty index on `hkrtch` is likely to lead to a syntax error. In order to examine the value of `hkrtch` it is also necessary to add the expty index, the name alone has itself as its value.

The remaining objects of interest for the RW metric are the Einstein tensor and the Weyl curvature tensor, defined by:

$$G_{ij} = R_{ij} - \frac{R}{2} g_{ij} \tag{3.7}$$

$$C_{hijk} = R_{hijk} + \frac{1}{n-2}\left(g_{hj}R_{ik} + g_{ik}R_{hj} - g_{ij}R_{hk} - g_{hk}R_{ij}\right) \tag{3.8}$$

$$+ \frac{R}{(n-1)(n-2)}\left(g_{hk}g_{ij} - g_{hj}g_{ik}\right) \tag{3.9}$$

These object are closely related to the Ricci tensor and the Riemann tensor, respectively, with the same index structure and symmetries and can be created in the following way using the extended `mkobj()` syntax described in §2.2.2 and §2.2.3:

```
#: mkobj('hG,'hRic,'hRic, '(), 'einstein);
 hG
   a b
#: mkobj ('hC, 'hR, 'hR, '(), 'weyl);
 hC
   a b c d
```

The evaluation of these objects is straightforward (some of the constants below depend on the dimension of the space; here $n = 4$):

```
#: hG[i,j] == hRic[i,j]-hRsc*h[i,j]/2;
 hG     =
   i j
                                    2                               2
   - 3 rt    h      rt - 3 rt    h      - 3 h      k + hRic      rt
         t,2  i j            t    i j       i j           i j
 --------------------------------------------------------------
                                        2
                                      rt
 hG
   i j
#: hC[h,i,j,k] == hR[h,i,j,k]+(h[h,j]*hRic[i,k]+h[i,k]*hRic[h,j]
   -h[i,j]*hRic[h,k]-h[h,k]*hRic[i,j])/2+hRsc/6*(h[h,k]*h[i,j]
   -h[h,j]*h[i,k]);
  <lots of output>
 hC
   h i j k
```

It will be observed that in the expression for the Weyl tensor the name `h` is used both for an index and as the metric. Ordinarily there is no conflict between a name used in an index and the same name used for any other indexed object or algebraic variable, unless the switch `evalindex` is on (qv). If the contents of the Weyl tensor are examined, it will be found that there are no explicit values, as is to be expected since the RW metric is conformally flat.

3.1 Metric Contractions

A common operation in tensor analysis and General Relativity is the raising and lowering of indices via metric contractions. The objects so created are not new entities in their own right, just re-expressions of the same fundamental entity. It is common to express these objects with the same name and let the index structure indicate which specific object is being referred to. Thus,

$$R_{abcd}, \quad R^{a}{}_{cbd}, \quad R^{abcd},$$

are all representations of the Riemann tensor. In REDTEN every indexed object must have a unique name in the system but the `printname` 's (the name used by the system to display the object) can be the same, indicating the existence of a fundamental relation.

The function `shift()` is used in concert with the `@` operator to "shift" indices, i.e. to raise or lower them by metric contractions. This function builds and maintains the data structure that relates the "parent" object to each of its "offspring". The entire set of offspring and the parent constitutes a "family", which is treated as a unit by some REDTEN functions.

The `@` operator causes an offspring object to be created in which each index-element operated on by `@` has been move to a position opposite its original location (i.e. covariant indices become contravariant and vice-versa). Two related operators are `@+` and `@-` which make "absolute shifts": `@+` shifts the index-element to a contravariant position (and is a no-op if it already was); `@-` shifts the index-element to a covariant position.

3.1.1 metrics

A prerequisite of using `shift()` is the existence of a metric for the appropriate index-type. We have seen in §2.4 how to construct a tensor metric from either an indexed object or a line-element. The function `metric()` , when it creates a metric, declares it to be the "current-metric" by updating the variable `lisp:currentmetric` . This variable is a list of names of metrics for each type of index defined by the `lisp:defindextype!*` variable, starting with the tensor metric (see table 2.1 for a list of the initial index-types defined in the system). After creating the metric `h` for the RW metric we can examine the value of this variable:

```
#: lisp currentmetric;
(h)
```

The current-metric for any index-type can be set with the function `setmetric()` and the name of the current-metric can be obtained with the function `getmetric()` . The argument to `setmetric()` is the name of a rank-2 covariant object that must be of `itype metric`, it should also have a metric inverse defined by applying the function `invert()` to it, if the object was not created by one of the system metric functions. If the tensor metric is being changed, the system coordinates (in `lisp:coords!*`) are changed to reflect those on the new metric. In addition, `setmetric()` also flags the old metric to be `nodir`, so that it and its family (see §4) are not displayed by `dir()` . With `setmetric()` it is possible for the user to maintain several metrics in the system simultaneously, switching between them as required. The GR package in REDTEN makes this even easier.

The argument to `getmetric()` is an integer index-type value whose sign is the *opposite* of the actual entries in the `indextype` property of the metric. Thus, the value 1 will return the tensor metric, while -1 will return the metric inverse:

```
#: getmetric(1);
  h
#: getmetric(-1);
  h_inv
```

It is an error to ask for a metric that has not yet been defined.

3.1.2 shift()

In order to perform a metric contraction that will raise or lower an index, the user types the index with the `@` operator preceeding the appropriate index element. The operation is performed when an expression containing the object appears as the right hand side of an indexed assignment. If the user wishes to form the shifted object immediately, the function `shift()` can be applied to the object.

The commands

```
#: shift(hG[@a,b]);
computing hG_b
   a
 hG
     b
#: hG_b[];
                       2
           - 3 (rt    + k)
   0                  t
 hG      = -----------------
     0               2
                   rt
```

```
                              2
            - 2 rt    rt - rt   - k
   1             t,2        t
 hG     = -------------------------
     1               2
                   rt
                              2
            - 2 rt    rt - rt   - k
   2             t,2        t
 hG     = -------------------------
     2               2
                   rt
                              2
            - 2 rt    rt - rt   - k
   3             t,2        t
 hG     = -------------------------
     3               2
                   rt
 hG_b
#: hG[@0,0];
           2
   - 3 (rt    + k)
          t
 -----------------
          2
        rt
```

create and display the "mixed" Einstein tensor for the RW metric, which has the form expected for a pressureless dust. The last command shows a short form for accessing specific elements of a shifted object, the full object name need not be typed, only the parent name with the proper shift operation. If such a reference is made before the offspring is computed, it will return unevaluated unless it was the argument to an explicit `shift()` command.

The offspring object created by `shift()` has a name formed by appending to the parent name an underscore (_) and a lower-case letter (or letters) derived from the locations of the shifted indices. This name is reported in the `computing <name>_<letter>` message that is displayed when `shift()` is called either explicitly or implicitly. The new object is given the appropriate `indextype` entry, and the same coordinates, print-name and type as the parent. The symmetries are derived from the parent's, accounting for the shifting of the indices, more is said on this below. Both the parent and the offspring are write-protected, so that they cannot be inadvertently altered and render the relation inconsistent. The offspring object has the `parent` property set, and the description string indicates its origin. The `shift` property of the parent is updated to show the new object belonging to the parent's family.

As the user may call `shift()` at any time, the environment of the system is changed to that of the parent before the operation proceeds. This is accomplished by looking at the `altmetric` property of the parent to find the tensor metric that existed at the time the parent was created. This is temporarily installed as the current-metric and the system coordinates are changed to match. After the offspring is created (having been given the correct `altmetric` and `coords` properties), they system environment is restored. If the user aborts the shift operation then the system environment can be reset by calling `setmetric()` with the name of the metric which should be the current metric. The `altmetric` property may be examined or changed with the `altmetric()` function.

When `shift()` is called, it first determines the index structure of the offspring object and checks in the list of existing offspring (on the property `shift` of the parent) for a match. If the object exists shift returns quickly. Otherwise, `shift()` finds the closest offspring and forms the metric contractions with it, which presumably requires the least amount of work. As the parent object is always first in the `shift` list (after a flag which if `nil` indicates that shifting is allowed on this object), it may be that the parent will be used. If the user types the name of an offspring object and an index indicating a shift, the parent object is determined from the `parent` property, and then the shift relative to the parent is computed, allowing `shift()` to proceed as before.

As noted above, the symmetries of the offspring are determined from those of the parent. Any independent symmetry of the parent which, in the offspring, relates indices that are all covariant or contravariant (but not mixed) is retained. Thus, if the first index of the Riemann tensor is raised, only the anti-symmetry in the third and fourth indices survives, but if all the indices are raised, the offspring has the full Riemann symmetry. To ensure the most efficient operation of the system, the parent object should always be the most symmetric of the entire family of related objects, since `shift()` cannot determine if any new symmetries will appear.

Normally the application of the intrinsic symmetries of an object to canonicalize the index results in the indices being sorted into alphabetical order. If an index element is to be shifted, then it takes precedence when being sorted, so that indices shifting up are moved to the left, and indices shifting down are moved to the right. Thus, the canonical form of the shifted object takes priority over the canonical form of the index alone. For example,

```
#: hG[b,a];
  hG
    a b
#: hG[@b,a];
    b
  hG
      a
```

Certain objects are shifted by contracting with things other than the metric; for these the user can change the `altmetric` property with the `altmetric()` command. This function takes the name of the object whose properties are being modified and the name of a rank-2 object that will be the alternate metric. With only one argument, `altmetric()` shows what metrics are defined for the given object.

The tensor metric itself has its indices raised to form the metric inverse via a matrix inverse computed by `invert()` ; this function sets up the `shift` property on the metric, and also adds the proper delta function to represent a metric with mixed indices. The delta functions are created by the function `delta()` , but ordinarily this function need never be called by the user since the delta functions are created automatically for each kind of index in the system by `defindextype()` . The delta functions themselves form a family of three objects, one with both indices up, one with both down, and a mixed object. The names of the delta functions are by default of the form `d<n>_c`, `d<n>_d` and `d<n>`, respectively, where `<n>` is the type of index the delta function is defined for. Unlike the usual naming convention for the delta functions, `d<n>_c` is the parent, because it has the symmetries, but the mixed object, being most commonly used, is given the simpler name `d<n>`; this is the delta function used for the mixed metric.

Chapter 4

The General Relativity Package

This chapter describes the functions in REDTEN that comprise the General Relativity package for managing the family of related objects that are constructed from the metric tensor (assumed symmetric). The user has already encountered the `metric()` function and seen how it can be used to make a metric tensor either from a rank-2 indexed object or directly from the line-element. We shall repeat the examples of the RW metric to illustrate the use of the GR package.

Constructing the metric tensor from the RW line-element can be done as follows:

```
#: ds2 := d(t)^2-rt^2*(d(om)^2+s^2*(d(th)^2+sin(th)^2*d(ph)^2));
                  2    2         2         2   2  2        2        2   2  2
ds2 :=  - d(om)  rt  - d(ph)  sin(th)  rt  s  + d(t)  - d(th)  rt  s
#: metric (ds2);
 computing g1  INVERT finished.

  g1
```

There are several functions to compute the various objects of interest starting from a metric. These are `christoffel1()` , `christoffel2()` , `riemann()` , `ricci()` , `riccisc()` , `riccisc()` , `einstein()` , and `weyl()` . The workings of these are so similar that they can be described together, special features of each function will be noted.

Each function takes one optional argument, the name to use when creating its output. The actual object name is made by concatinating the current-metric name and the user name (or the default name), separated by a _ symbol (these names may seem unweildly, see §4.3 below for a simplifying interface). If the user does not supply a name, the default name is used which is stored on the function name used as a lisp variable. For example, the default name for the Christoffel symbol of the first kind is found on the variable `lisp:christoffel1` . The defaults are `c1`, `c2`, `R`, `ric`, `ricsc`, `G`, and `C` for the functions listed above (these are set in the `sys.env` source file).

Each function stores the name of the object it creates on the property list of the current-metric under a key which is the functions' own name. Thus, the name of the object created by `christoffel1()` is stored on the metric under the key `christoffel1` . When these functions are called, they look first at the current-metric to see if the desired object already exists, if it does its name is returned immediately. If it does not exist, then it is created. If the object exists but the user provided a name, then it is recomputed with the new name.

These functions construct a family of related objects derived from the metric tensor, and some REDTEN functions can operate on the family as a whole. The names of family members for any object can be seen by calling the symbolic-mode function `getfam()` .

Since each function except `christoffel1()` calls the others to get the objects it needs, the user does not necessarily need to build the objects one at a time (but see §5.1 for an important consideration). For example, we can compute all the objects in a direct line from the metric tensor to the Weyl tensor with the single command:

```
#: weyl();
computing g1_C
  computing g1_ricsc
    computing g1_ric
      computing g1_R
        computing g1_c1
        christoffel1 finished.
        computing g1_c2
        christoffel2 finished.
      riemann finished.
    ricci finished.
  riccisc finished.
** this space is conformally flat
 g1_C
```

The Ricci scalar is created as an indexed scalar by `riccisc()` . If the Weyl tensor can be seen to be zero then `weyl()` will report that the space is conformally flat. If the Ricci tensor is zero then `ricci()` will report that the metric is a vacuum solution. For complicated metrics it may take some simplification before these objects reduce to zero, so the messages may not appear. Each function write-protects its output, so that the user cannot accidentally make inconsistent changes; see also `mapfi()` and `protect()` .

The output of each function is the name of an indexed object. If desired, the function can be wrapped in the function calling operator `fn()` allowing it to be used with an index directly. This lets the user to make indexed expressions that do not rely on the exact names of the various objects; a better method is described in the section on generic names below. For example, to compute the Einstein tensor and examine a component in the same command can be done by:

```
#: fn(einstein(),[1,1]);
computing g1_G
einstein finished.
                  2
 2 rt     rt + rt    + k
     t,2         t
```

The `fn()` operator accepts a function and the index to be attached to the output of the function (which must be an indexed object name) as its two arguments.

4.1 Ordinary Differentiation

As of REDTEN version v3.3 the ordinary differentiation code was entirely rewritten to make use of automatically created objects to store the derivatives of other objects. This change was intended to improve the performance of the system by avoiding the unnecessary re-computation of derivatives, and to simplify the code. When an objects' index indicates an ordinary differentiation and the object is involved in an indexed assignment, the system will either store a newly computed derivative or retrieve an existing value from the derivative object. Higher order derivatives cause multiple objects to be created, each containing the derivative of the previous object and having a rank one greater.

The automatically created derivative objects have names of the form `<name>_DF<#>` where `<name>` is the name of the object whose derivative is being computed, and `<#>` is an integer, starting from 2, that distinguishes higher order objects. The indextype is formed by adding enough covariant tensor indices to those already on the parent object to match the order of the derivative. This is strictly incorrect, since ordinary derivative indices do not transform as tensor indices, but it is much more convenient to regard them as such. The user is responsible for ensuring that combinations of objects having ordinary derivative indices are in fact tensors. For derivatives of order 2 or higher, the derivative indices are symmetric because of the commutative property of differentiation; the derivative objects also inherit the intrinsic symmetries of the parent object.

When created on the fly the derivative objects are filled with the hitherto unmentioned value `<undefined>` (which cannot be entered by the user) so as to distinguish between components that have been evaluated and those which have not. Thus, a derivative object may be "incomplete" after being created automatically, since the indexed algebra may not have touched on all the possible components. The user can fully evaluate any order derivative of an object via the function `odf()` , which takes the object name as the first argument, and the order of the derivative as the optional second argument. Objects so created are visible in a directory listing, but the automatic objects are flagged `nodir` and are hidden unless the `all` keyword (or the `*` pattern) is used. Since the number of real components is uncertain for the automatic objects, a directory listing will usually display a large number of components (representing the full number of possible components each with the value `<undefined>`), followed by a `?` to indicate the uncertainty. Using `odf()` also causes the derivatives and the parent to become write protected.

The derivative objects are given the `itype odf` and form a chain starting from the parent; each object stores its order and the objects that are its anti-derivative and derivative under the `odf` property. See the function `getfam()` for information on how to simplify both the parent object and its derivatives at the same time.

The derivative objects are set up with a `printname` that is the parent objects' `printname` , and their indices are printed with the appropriate differentiation symbols in place. For example, if `q` is a rank-1 indexed object, the user can create its 3rd-order derivative via

```
#:  odf(q,3);
computing q_DF3
```

```
q_DF3
```

and a display of the components of this object would appear as follows:

```
#: q_DF3[];
  0
 q          = <value>
    |0 0 0
  0
 q          = <value>
    |0 0 1
etc.
```

The user may also cause the automatic creation of the derivative objects by entering a fixed index containing a derivative operator. The required derivatives are computed and immediately returned, and also stored in the derivative objects. For example,

```
#: R[1,2,1,2,|0];
                              2                   2
   - (2 rt  sin(sqrt(k) om)  rt) (rt   rt + rt   + k)
          t                          2t       t
 ----------------------------------------------------
                              k
```

4.1.1 Partial Differentiation

Partial differentiation of algebraic expressions with respect to the current coordinates can be computed with the `pdf()` function. This function allows the user to attach an index (representing derivative indices) to an expression. Nothing is evaluated until the function call is involved in an indexed assignment, where the indices must obey the usual rules of indexed algebra, and are regarded as covariant tensor indices. For example,

```
#: pdf(sin(t)*r, [a,b]);
 (sin(t) r)
           |a b
```

The `pdf()` function has been superseded by the REDUCE `df()` operator as described below.

The partial derivative of an expression can be obtained with the `df()` operator directly and using the coordinate vector created by `coords()` (the default name for this vector is `x`):

```
#: df (sin(t)*r, x[a],x[b]);
```

```
  2
 d  sin(t)        d sin(t)    d r      d sin(t)    d r
------------- r + ---------- ------- + ---------- -------
   a     b           a         b          b         a
 d x   d x         d x       d x        d x       d x
      2
     d  r
 + ------------- sin(t)
     a     b
   d x   d x
```

In this situation the contravariant index of `x` will be regarded as covariant, so that the index structure of this expression has two covariant indices. The indexed object used as the indeterminate to the `df()` operator must be a coordinate vector, otherwise an error message will appear (possibly several times, due to the way in which the modifications to the REDUCE source code were accomplished). See §7.3 for more information about the appearance of the output of this expression.

If the index of the coordinate vector is fixed, the partial derivative will be immediately evaluated. If the expression does not depend on any of the coordinates, 0 will be returned. Such is not the case with the `pdf()` function, which will only imediately evaluate if the entire derivative index if fixed, and which does not check in advance for dependencies. The `pdf()` function may be removed from REDTEN in future versions.

4.2 Covariant Differentiation

Unlike ordinary differentiation, which requires only the coordinate names, covariant differentiation requires the existence of Christoffel symbols of the second kind, and hence of a metric. Covariant differentiation is accomplised by first taking the ordinary derivative of the object in question, and then forming contractions of each of the objects' indices with those of the Christoffel symbol:

$$\begin{aligned} T^{r_1\cdots r_m}_{s_1\cdots s_n\|i} = T^{r_1\cdots r_m}_{s_1\cdots s_n|i} &+ \sum_{\alpha}^{1\cdots m} T^{r_1\cdots r_{\alpha-1} j r_{\alpha+1}\cdots r_m}_{s_1\cdots s_n} \left\{ \begin{matrix} r_\alpha \\ j\ i \end{matrix} \right\} \\ &- \sum_{\beta}^{1\cdots n} T^{r_1\cdots r_m}_{s_1\cdots s_{\beta-1} l s_{\beta+1}\cdots s_n} \left\{ \begin{matrix} l \\ s_\beta\ i \end{matrix} \right\} \end{aligned}$$

It is apparent from this definition that covariant derivatives are non-trivial to compute. Although for a simple metric like RW it takes only a few seconds to compute the covariant derivative of the Riemann tensor, for more complicated metrics it can take quite some time; and higher level derivatives can take several minutes to compute even for simple metrics.

The `||` operator placed in an objects' index causes the covariant derivative to be computed when the object is involved in an indexed assignment. The function `cov()` , which is also callable by the user and takes an indexed object as its single argument, is used to do the computation. The output is the name of an indexed object in the form <input>_CD<#>, where <#> is an integer that distinguishes higher order derivatives. The covariant derivative will have the same symmetries as the input, and the same index structure with the addition of a single covariant tensor index. The covariant derivative is stored on the input object's property list under the key cov , in the same format as used for ordinary derivatives; whenever cov() is called it looks there first to determine if the covariant derivative object already exists.

Although the user can indicate multiple covariant derivatives, as in T[a,b,||e,f], cov() computes them individually; higher order derivatives have the order appended to the name, as noted above. The argument to cov() can include an index, but it is not used except to determine if a shift is also being requested (see below). Note that any index used in the argument to cov() should *not* contain the covariant derivative operator || or an extra index, since the index applies to the object whose derivative is being computed. The printed output of a covariant derivative is similar to that for ordinary derivatives: the printname is that of the parent and the index is printed with the covariant derivative operator in place.

As with the ordinary derivative, a fixed index including a covariant derivative operator may be applied to an object to access the specified value, but only after the covariant derivative has been computed. If it has not, the operation returns unevaluated.

```
#: cov(g1_R);
  computing g1_R_CD
 g1_R_CD
#: g1_R[1,0,1,0,||0];
 rt     rt - rt     rt
   t,3         t,2    t
```

Covariant differentiation is not restricted to tensors alone, objects with other types of indices may also be arguments to cov() . In each case the single new index will be a covariant tensor index. The Christoffel symbols used are computed by functions attached to each index-type by defindextype() ; for tensor indices the Christoffel symbols are computed by christoffel2() , for spinor indices by the function spchristoffel() , neither frame nor dyad indices have Christoffel symbols. In these cases the covariant derivative reduces to the ordinary derivative. Covariant differentiation is not defined for array indices.

To ensure the correct Christoffel symbols (and the correct coordinates for ordinary differentiation) are used, cov() will change the current-metric to that specified on the altmetric property of the input. When christoffel2() is called to compute or find the second Christoffel symbols, it will look at that metric under the christoffel2 property for the name of an indexed object with the proper index structure (i.e. '(1 -1 -1)). In the example of the last chapter, since christoffel2() was not called by the user it would not have found a pre-made object, and would have attempted to compute it. This could have been prevented by the command

```
#: lisp put ('h, 'christoffel2, 'hc2);
```

```
hc2
```

Some objects, specifically metrics and the Einstein tensor, have known covariant derivatives, namely 0. The functions that create metrics put the value 0 under the `cov` property so that `cov()` will immediately return 0.

4.2.1 Covariant derivatives of shifted objects

Because the covariant derivative of the metric is 0 and covariant differentiation distributes over products of tensors the covariant derivative of a shifted object is the equivalent shift of the covariant derivative of the parent object:

$$(R^a{}_{bcd})_{||e} = \left(g^{al}R_{lbcd}\right)_{||e} = g^{al}{}_{||e}R_{lbcd} + g^{al}R_{lbcd||e} = R^a{}_{bcd||e}$$

This fact makes it possible to compute covariant derivatives of shifted objects resonably efficiently, given that actually computing a covariant derivative is very costly compared to shifting an object. If `cov()` is given an object that is the offspring of a parent, the parent is examined for a covariant derivative, and, if none is found, it is computed. That covariant derivative is then shifted so that its indices match the original offspring, with the addition of one covariant tensor index. Thus,

```
#: cov(R[@a,b,c,d]);
computing g1_R_CD
computing g1_R_CD_b
SHIFT finished.

 g1_R_CD_b
```

In this way two corresponding families of objects are built, one having the original object as the parent, and the other having the covariant derivative as the parent. Rather than having to type the full object name, the user can read individual components by typing, for example, `g1_R[@1,2,1,2,||0]`. An assignment to such a component requires the full object name, however.

4.3 Generic Names

As noted above, each GR function creates an object with a name of the form `<metric>_<name>`, so that any number of metrics and their families may exist in the system simultaneously. Each object has a unique name, and each function returns the object appropriate for the metric currently in use. However, these names are not particularly convenient for general usage such as building indexed expressions. Thus, a set of "generic" names has been created that allow the user to easily access the objects in the family of the current metric.

A generic name is a psuedo-indexed object: it has some properties of indexed objects that allow it to be parsed with an index, but it never has explicit or implicit values, and it has some extra properties that mark it as generic. Specifically, the `generic` property of a generic name contains the type of metric to look at, and the name of the property key on that metric to examine. That key is the name of the function that created the actual object of interest, and is the place where that function stored the name of its output. For example, the generic name `R` has a `generic` property that refers to the tensor metric and the key `riemann`, which is where the function `riemann()` stores the name of its output.

The system, when it encounters a generic name, immediately examines the `generic` property to locate the target object. If the metric is changed, the system will look at the new metric and the generic name will refer to some other object. Thus, the same generic name is used for the same class of object for many different metrics. If the target object does not exist and the generic reference is not part of an indexed assignment, it will return unevaluated and will do so until the object is made, or the metric is changed to one for which the desired object has been created. The user can therefore set up expressions involving generic objects, assign them to REDUCE algebraic variables before any metrics are created, and then use them after some metrics are made to evaluate the same expression for each metric. For example,

```
#: kr := R[a,b,c,d]*R[@a,@b,@c,@d];   % R has no target yet
         a b c d
  kr := R        R
                  a b c d
#: % create some metrics and compute their Riemann tensors ...
#: setmetric(g1)$
#: seval(kr);
  <output>
#: setmetric(g2)$
#: seval(kr);
  <output>
```

will evaluate the Kretschmann scalar for the metrics `g1` and `g2`. Of course, if the output of the expression was also indexed, then a new object would have to be created each time the expression was evaluated.

If, when an expression containing unresolved generic references is included in an indexed assignment (such as the call to `seval()` above), and the target objects have not yet been created, they will be automatically computed since each generic object also carries information on how to make its target. However, it is generally better if the user calls the generating functions directly, since the target objects can be simpified and examined before being involved in a potentially expensive computation (see §5.1 for more information).

There are a large number of generic names pre-defined in REDTEN; for the GR package the commonly used ones are `g` for the metric, `c1` and `c2` for the Christoffel symbols, `R` for the Riemann tensor, `ric` and `ricsc` for the Ricci tensor and scalar respectively, `C` for the Weyl tensor and `G` for the Einstein tensor. More than one generic name can point to the same target, thus `ri` also refers to the Riemann tensor, and `ei` refers to the Einstein tensor. The function `generics()` will show the user what generic names exist, and whether there is a target object present.

The only generic names actually created in REDTEN are those whose targets are a parent object. Shifted objects and any other objects whose names include a _ symbol (eg. covariant derivatives) are handled by examining the fragments of their names. If the portion of the name before the _ symbol is a generic name, the target name replaces the initial portion of the name. Hence, if the current-metric is `g1` and the target of `R` is `g1_R` then the target of `R_b` is `g1_R_b`. All functions that take an indexed object as input will accept a generic name (actually, the name is resolved before the function gets it), and the output of these functions will involve the target name.

4.4 Other Functions

In addition to those functions listed above that calculate from a metric commonly used tensors, there are a number of other computations in GR that are conveniently packaged as functions. These compute the divergence of a tensor, the D'Alembertian of a scalar, Lie derivatives, the Killing equations, and the geodesic equations. These functions either operate on another object or produce output which is not usually regarded as a tensor, so they are not grouped with the functions above that compute the family members of a metric.

The divergence function `div()` computes the divergence of its first argument via the formula:

$$\operatorname{div}\left(T_{r_1 r_2 \ldots r_k}\right) = T^{r_1}_{\ r_2 \ldots || r_1}$$

and note that the first index will be raised if need be (however, this will be done to the covariant derivative, *not* to the parent object, see §4.2.1). The output is an object of rank $k-1$, with either the optional second argument as its name, or a default name of the form `<input>_D`. The divergence of a vector quantity is an indexed scalar. The name of the divergence object is stored on the property list of the input object under the key `div` , and `div()` will immediately return this name if the divergence has already been computed. The symmetries of the output will be those that survive the loss of the first index of the parent object.

The function `dalembert()` computes the D'Alembertian of a scalar u from the equation

$$\frac{\left(\sqrt{-|g|}\, g^{ab} u_{|a}\right)_{|b}}{\sqrt{-|g|}},$$

and the output of this function is also a (non-indexed) scalar. The determinant function `det()` (described below) is used to compute the determinant of the metric tensor.

The function `lie()` computes the Lie derivative of its first argument in the direction of the vector that is the second argument. The output object (whose name is formed by concatenating the name of the tensor and vector and adding `_lie` to the end[1]) has the same index structure as the input object. If the vector given is covariant it will be shifted to give a contravariant representation. The formula used is

$$\begin{aligned}\mathcal{L}_a T_n &= T_{n,i} a^i + T_i a^i_{,n} \\ \mathcal{L}_a T^n &= T^n_{,k} a^k - T^k a^n_{,k}\end{aligned}$$

which is expanded as required to cover each of the objects' indices.

The function `killing()` computes the Killing equations for the current-metric; this function cannot, however, solve these equations for the Killing vectors. The Killing equations are given by

$$\xi_{i||j} + \xi_{j||i} = \begin{cases} 0 & \mathrm{non-conformal} \\ \xi^a_{||a} & \mathrm{conformal} \end{cases} .$$

The output of this object is a rank-2 symmetric tensor whose name will be the first argument to `killing()` (there is no default name). The optional second argument, if non-`nil`, indicates that the conformal Killing equations are to be computed. The Killing vector is, by default, given the name `k` (the default name is stored on `lisp:killing`), it is created as an implicit covariant vector that depends on all the coordinates. The components of the output object are then a set of $n\,(n+1)/2$ first order differential equations that are to be solved for the components of `k`, where n is the dimension of the space.

It should be noted that computing the Killing equations for the RW metric as set up in the examples above will cause difficulties, since the Killing vector `k` will be made to depend on the coordinates but in the RW metric `k` is a constant. Any subsequent computation involving differentiation will be incorrect, since the dependencies for an indexed object are created and stored in the same fashion as those for non-indexed objects. This may be fixed in future, but involves rewritting some REDUCE code. If the switch `mkobjsafe` is on, then `mkobj()` called from `killing()` will err-out when it attempts to create the Killing vector for this metric.

In a similar fashion, `geodesic()` computes the geodesic equations from

$$\frac{d^2 x^l}{d\,s^2} + \left\{ \begin{matrix} l \\ j\ k \end{matrix} \right\} \frac{d\,x^j}{d\,s} \frac{d\,x^k}{d\,s} = 0.$$

The first argument is the name of the output object, which will be a covariant vector. The optional second argument is the name of the affine parameter, the default (stored on `lisp:geodesic`) is `s`. Each of the coordinate names (in `lisp:coords!*`) is made to depend on the affine parameter.

[1]Note that if the Lie derivative of a generic object is being taken, the actual target name, not the generic name, is used.

Another occasionally useful function is `mkcoords()` , which makes a contravariant coordinate vector of its single argument, or of the default name which is the value of `lisp:mkcoords` [x]. The `coords()` function calls `mkcoords()` whenever the user changes the coordinates so that the vector `x` always contains the current coordinates.

4.5 The Matrix Functions

The matrix functions all operate on rank-2 objects of any type. The function `idet()` computes the determinant of its argument via the cofactor method. The result is an algebraic value that is also stored on the object under the key `det` ; future calls to `idet()` will return this value immediately. The user can change the stored value of the determinant by giving `idet()` a second argument.

The functions `cofactor()` and `determ()` compute the cofactor matrix and the determinant from that matrix. The second argument to `cofactor()` is the name to use for the cofactor matrix, and that name is to be used as the second argument to `determ()` *after* `cofactor` has been called. Again, the determinant is stored on the object, and can be read-out with either `det()` or `determ()` .

The `invert()` function computes the matrix inverse of its argument, the inverse is given a name of the form `<name>_inv`; this is how metric inverses are computed. The `shift` property of the object is updated to include the inverse, and the appropriate delta-functions are also added.

Finally, `trace()` will compute the trace of a rank-2 object in the obvious way, and return a scalar value.

Chapter 5

Utilities

This chapter describes several REDTEN functions of a utilitarian nature, which aid the user in managing the process of computing objects of interest.

5.1 Expression Management

Although General Relativity and most tensor analysis requires only that algebra be performed and derivatives taken (both of which are easily implemented on a computer), it is still necessary for the user to aid the machine in finding a workable path to the desired goal. The brute force approach (i.e. start it up and let it take a whack at the problem) will usually not succeed except for very simple problems or on very fast machines with lots of memory. Despite its high-sounding name, computer algebra remains as much an art as it is a science. Hence, although it is possible for the user to, with a single command, compute all the objects in a direct line from the metric to the Weyl tensor, unless the metric is very simple the calculation is not likely to succeed. The user must take a hand in simplifying the intermediate results before the more complicated objects are computed, otherwise the calculation will either take an inordinate amount of time, or, in the worst case, run out of memory. The techniques for managing large expressions are general ones and apply to REDUCE as much as to REDTEN; here we shall point out some particular points to observe that relate specifically to the sorts of problems encountered by the REDTEN user.

The term "management" is chosen to reflect the process whereby the user interacts with the system to maintain expressions in a compact and reasonably simple form, and to avoid what is commonly referred to as "intermediate expression swell". Without attention it is quite possible that the components of some objects such as the Riemann or Ricci tensors could become so large as to jeopardize the calculation, even though the final answer is relatively simple (eg. the Weyl tensor is zero).

The most common culprit which results in unweildly expressions is unsimplified denominators, specifically those which involve sums, and which must be expanded over the numerators of other terms when the system makes common denominators. One way to avoid this is to turn off the REDUCE switch `mcd` , but this usually just postpones the difficulties, since no real simplification and cancellations likely have occured and a large expression involving many terms must later be placed in common denominator form if it is to be finally simplified.

In this regard, metrics come in two varieties: those which are diagonal and hence are trivial to invert; and those which are not diagonal, and generally yield a complicated denominator upon inversion. It is most important that the user address this issue before proceeding to the computation of the Christoffel symbols, and the `christoffel1()` function will print a warning message if the metric inverse seems not have not been simplified (This only means that the multiplier property must be combined with the components, as is done by `mapfi()` (qv)).

A method that is sometimes useful in dealing with sums in denominators is substituting another symbol so that the sum is reduced to a simple product. However, this can yield explosive expressions elsewhere and, additionally, the user must remember to declare the dependencies of the new symbols, or to define their derivatives explicitly. Needless to say, if these are forgotten, the calculations may proceed quite smoothly but they no longer represent the intended metric. The decision of whether and when (and what) to substitute for various terms or when to introduce new ones is one of the only controls the user has over how the calculation proceeds. The user may also turn on or off the various REDUCE switches that control the way in which REDUCE combines expressions; of these `exp` , `factor` , `gcd` , and `mcd` are the most useful. However, as noted above these may often simply postpone the difficulties, since at some point the expression must be expanded for it to be simplified.

An example involving the Kerr metric (a metric much-maligned for being complicated, when in fact it is still relatively simple) is used to illustrate some principles of expression management in REDTEN. The Kerr metric is well known in General Relativity and is identified with the vacuum solution of the Einstein Field Equations outside a spinning mass. In Boyer-Lindquist coordinates (t, r, θ, ϕ) the line-element has the form:

$$ds^2 = -\frac{\Delta}{\rho^2}\cdot\Big(dt - a\sin^2\theta d(\phi)\Big)^2 + \frac{\sin^2\theta}{\rho^2}\cdot\Big((r^2+a^2)d\phi - adt\Big)^2 + \frac{\rho^2}{\Delta}dr^2 + \rho^2 d\theta^2.$$

where

$\Delta = r^2 + a^2 - 2mr$,

$\rho^2 = r^2 + a^2\cos^2(\theta)$,

a is the angular momentum,

and m is the geometrized mass of the object.

In terms of a metric tensor, the Kerr metric is written as

$$g_{ab} = \begin{pmatrix} \frac{a^2 \sin^2\theta - \Delta}{\rho^2} & 0 & 0 & \frac{a \sin^2\theta \cdot \left(\Delta - a^2 \sin^2\theta - r^2\right)}{\rho^2} \\ 0 & \frac{\rho^2}{\Delta} & 0 & 0 \\ 0 & 0 & \rho^2 & 0 \\ 0 & 0 & 0 & \frac{\sin^2\theta \cdot \left(\sin^4\theta - a^2\Delta\sin^2\theta + 2\sin^2\theta a^2\rho^2 + \rho^4\right)}{\rho^2} \end{pmatrix}.$$

We shall enter this metric in line-element form and create the metric:

```
#: coords '(t r th ph)$
#: ds2 := - delta/rho2*(d(t)-a*sin(th)^2*d(ph))^2
   + sin(th)^2/rho2*((r^2+a^2)*d(ph)-a * d(t))^2
   + rho2/delta * d(r)^2+rho2*d(th)^2$
#: metric(ds2);
computing g2 cofactor finished.  determ finished.  invert finished.

 g2
```

We first observe that there are more messages during the computation of this metric, since it takes more work to invert it. At this point the metric inverse can be simplified in several ways. When the system inverts a non-diagonal metric, the reciprocal of the determinant of the metric is placed as a multiplicative factor on the `multiplier` property of the inverse. This can be examined (and altered) with the function `multiplier()` [1]:

```
#: multiplier(g_inv); % note we use the generic name
                    2          4  4              2  4             2  2  2
  ( - 1)/(sin(th)  (sin(th)   a   - 2 sin(th)   a   - 2 sin(th)   a  r
                    4       2  2    4
               + a   + 2 a  r   + r ))
```

If there is a second argument to `multiplier()` , the multiplier of the object is replaced by that value. Hence, the user can first save the multiplier, and then replace it with some other expression that presumably will help the calculation proceed. One could also examine the multiplier for a sub-expression that could be replaced by a symbol, as mentioned above.

For the Kerr metric, another approach has proven to work well. It will be observed that substitutions for ρ^2 and Δ (`rho2` and `delta`) have not yet been made. One can invert the relation for ρ^2 and solve for r^2 to yield the substitution

$$r^2 = 2sin^2\theta - a^2 + \rho^2$$

which is best entered as a REDUCE `let` rule[2]:

[1] This function may not be available in future versions of REDTEN

[2] It will be noticed that these expressions do not involve $\cos^2\theta$, this is because REDTEN introduces a `let` rule of the form `forall x let cos(x)^2 = 1 - sin(x)^2`. This and other `let` rules are in the source file `lets`.

```
#: let r^2=a^2*sin(th)^2-a^2+rho2;
```

Whenever r^2 is formed during the calculations that will follow it will immediately be replaced by this substitution. However, the metric and its inverse still need to have this rule applied. First we shall show that in fact this substitution does provide a useful simplification by re-evaluating the multiplier of the metric inverse:

```
#: multiplier(g_inv);
                                          - 1
 ----------------
         2     2
  sin(th)  rho2
```

Obviously this expression is much more compact than the previous one, and is no longer a sum. Note however, that the actual value of the multiplier property is as yet unchanged. Also recall that new `let` rules and assignments can be evaluated in a given expression by applying the REDUCE functions `sub()` or `reval()` to the expression.

5.1.1 mapfi()

To re-evaluate each of the components of the metric and its inverse and actually change the stored components the function `mapfi()` [3] is used. In its simplest form, `mapfi()` takes an indexed object name and re-evaluates in-place each component of the object. We can apply the new `let` rule to the metric and its inverse by

```
#: mapfi(g);
 g2
#: mapfi(g_inv);
 g2_inv
```

If the switch `showindices` is on, the index of each component is displayed during the computation. A side effect of `mapfi()` is that the multiplier of an object is combined with each of its components, and the multiplier property is reset to 1. The old values of the object are saved, and can be recovered if the user immediately executes the `restoreobj()` command. Only one object is saved by the system at a time, hence the need to immediately recover. If in doubt, the user can also make a backup by copying the object with the `icopy()` function (see below).

The `mapfi()` function is not limited to such simple things as re-evaluating an indexed object in a new environment, it can also be used to apply specific functions to specific portions of the object. The most general format of the call to `mapfi()` is

```
mapfi(function(name{index},{args}));
```

[3] Its name is derived from MAP a Function onto an Indexed object. It is about as pnemonic as the names of certain similar lisp functions.

where `<function>` is some function that operates on algebraic expressions. The indexed object can either be just a name (so that the function is applied to the whole object) or an index that isolates part of the object, or even a fixed index, so that a single element is affected. `mapfi()` works by inserting each component of the indexed object into the input expression and evaluating it. The input can in fact be any expression but is most often a `sub()` command or occasionally a Taylor series function such as `ps()` .

For example, if the user entered a general static spherically symmetric metric, and then wanted to substitute $f = 1 - 2m/r$, it could be done in two ways. First, an actual assignment to the variable `f` could be made, followed by an application of `mapfi()` similar to that demonstrated above. However, other objects involving `f` would also be affected if they were involved in indexed assignments, or if `mapfi()` were applied. To make the substitution into only a single object can be done as follows:

```
#: mapfi(sub(f=1-2*m/r,g3));  % g3 is a metric defined in terms of f
   g3
```

If it were necessary to use a series approximation to the metric coefficients about $r = 2m$, this could be done via

```
#: mapfi(ps(g3,r,2*m));
   g3
```

Continuing with the Kerr metric example, we can define the derivatives of the as yet unevaluated symbols `rho2` and `delta`, and compute the Ricci tensor:

```
#: depend delta,r;
#: depend rho2,r,th;
#: let df(rho2,r) = 2*r;
#: let df(delta,r) = 2*r-2*m;
#: let df(rho2,th) = -2*a^2*sin(th)*cos(th);
#: ricci();
computing g2_ric
  computing g2_R
    computing g2_c1
    christoffel1 finished.
    computing g2_c2
    christoffel2 finished.
  riemann finished.
 g2_ric
```

At this point in the calculation, the Ricci tensor `g2_ric` will still appear to have non-zero components, since `rho2` and `delta` remain unevaluated. For other metrics it may be found that it is better to not evaluate the derivatives, or it may be best to fully evaluate everything from the start. It is a matter of trying different possibilities to determine which allows the successful completion of the calculation. If certain terms remain unevaluated, then sooner or later they must be given their actual values; the issue is when to do this so as to balance the growth of expressions. Evaluating too soon may mean the expressions swell (especially if a sum in a denominator is formed in the metric inverse), whereas delayed evaluation may mean that many cancellations have not occurred, and a large amount of work must now be done to insert the terms. For another example of this sort of work, see the Sato metric in Appendix B.

It is in all cases, however, best to avoid the occurrence of radicals such as would be formed if the substitution in the Kerr metric were for ρ rather than ρ^2. Many algebraic systems seem to have difficulties where radicals are concerned, and since the computation of the higher order tensors involves differentiation more radicals will be formed. It is almost always best to make a substitution for the radical since its derivatives can be defined in terms of itself, and this often reduces the complexity of expressions.

The final simplication of the Ricci tensor for the Kerr metric can be done with a single `mapfi()` command:

```
#: clear r^2;
#: mapfi(sub(delta = r^2+a^2-2*m*r,rho2=r^2+a^2*cos(th)^2,ric));
 g2_ric
```

Note that is was necessary to clear the substitution rule for `r^2`, since otherwise this would result in a circular evaluation for `rho2`. A package for managing the various let rules and assignments in the system is described in §7.1.

It may be observed that attempting to evaluate the Ricci tensor for this metric when `delta` and `rho2` are given their actual values at the outset results in a `Heap space exhausted` fatal error on some systems. The value of carefully controlling evaluations is therefore apparent. It is useful to turn on the `showindices` switch when working with an unfamiliar metric, because a judgement can then be made as to whether the calculations are proceeding at an acceptable rate, and whether a different approach might be more profitable. The switch `peek` when on acts like `showindices`, but causes the system to indicate whether a component has evaluated to zero or not. The user can then immediately see if the calculation is proceeding correctly (assuming, as is often the case, that zero is the expected answer).

5.2 Other Utilities

There are several functions in REDTEN whose intention is to make the usage of the system more convenient for the user. Of these functions, `dir()` has been previously introduced.

`dir()`

The function `dir()` displays the indexed objects in the system, showing the full name of the object, its type and number of explicit elements (components), the protection flags, the coordinates and the index structure. For example, after the Ricci tensor of the Kerr metric is simplified, a directory shows the following:

```
#:  dir(!*);
  name          type          comp    prot  coordinates      indextype
  x             coordinates    4       w    (t r th ph)       (1)
* g1            metric         5       w    (t r th ph)       (-1 -1)
* g1_inv        metric         5       w    (t r th ph)       (1 1)
  g1_ric        ricci          5       w    (t r th ph)       (-1 -1)
  g1_R          riemann        13      w    (t r th ph)       (-1 -1 -1 -1)
  g1_c1         christoffel1   20      w    (t r th ph)       (-1 -1 -1)
  g1_DF         odf            10           (t r th ph)       (-1 -1 -1)
  g1_c2         christoffel2   20      w    (t r th ph)       (1 -1 -1)
  g1_c1_DF      odf            133?         (t r th ph)       (-1 -1 -1 -1)
  9 objects,    Total components: 215
 t
```

Notice that the Ricci tensor has no components, as it should for a vacuum solution. The current metric and its inverse are flagged by a leading `*` on the display line. Objects that are implicit have a `+` symbol following the number of components to indicate that there are more values than are explicitly stored. Ordinary derivatives (which are not normally displayed, see §4.1) may have a `?` following the number of components to indicate that there are undefined components. Objects that are write-protected will have a `W` in the protection column, if they are kill-protected there will be a `K`, and if they are write- and kill-protected then there will be a `KW`. If there are no objects in the system, the message `no objects` is displayed.

With no arguments `dir()` shows all objects in order of creation, any arguments are taken as the names of objects to display alone. If the switch `reversedir` is turned off, the display proceeds in the reverse order, with the most recently created objects displayed first. The function `nodir()` flags objects so that they do not appear in a directory listing, the flag can be cleared by calling `dir()` with the object name as an argument. Some objects created by the system, such as the delta-functions, are hidden from `dir()` and are never displayed. If the keyword `all` is the single argument to `dir()`, then all objects are displayed regardless of the `nodir` flags. Scalar objects are not displayed in line with the other objects, their names are displayed at the end of the listing.

A rudimentary pattern matcher is used by `dir()` to expand names containing certain special characters. A `!*` matches any number of characters (including none), while a `!?` matches exactly one character. Thus, the pattern `!*` matches all names and is equivalent to the keyword `all`. The user can display the family of a given object with the pattern `<name>!*` or find all objects of a certain type, such as the Riemann tensor, with `!*R`. Objects selected in this way are displayed regardless of the `nodir` flags.

For metrics, the `setmetric()` function uses `nodir()` to hide the metric that was previously the current-metric, `dir()` will not display that metric or any of its family until it is again the current-metric or it appears as an explicit argument to `dir()`. This is intended to keep the directory listing short and concise, rather than being cluttered by objects that are not currently in use.

`iprop()`

Each indexed object carries a large number of properties describing the objects' structure and characteristics, these are described in §2.2.6. The function `iprop()` will display those properties named in the `lisp:iprop!*!*` variable, the defaults are specially marked in §2.2.6. If the switch `iprop` is on, then `iprop()` is called whenever the user has requested a display of the object's values.

`rem(), remi()`

The `rem()` functions are used to remove indexed objects from the system, deleting all of their properties and values. With `rem()` the user can delete specified objects, `remi()` deletes objects interactively. Both functions will accept the same patterns as `dir()` , as well as the keyword `all` which is a synonym for the pattern `!*`. Once an object is deleted there is no way to recover its values, hence the need for the `protect()` function (qv).

To use `rem()` , the user simply types the names of the objects to delete as the arguments, so, for example,

```
#: rem(c1,c2,R);
  (g2_R g2_c2 g2_c1)
```

will remove the Christoffel symbols and the Riemann tensor. Notice that generic names were used as arguments to `rem()` but the correct target objects were deleted, as shown by the displayed list of deleted objects. If a name that was requested be deleted does not appear in this list, then the object is kill-protected and cannot be removed until the protection is cleared with `protect()` . If a name not of an indexed object is included in the arguments to `rem()` it will be ignored.

In the interactive mode, `remi()` presents each indexed object specified to the user an asks if is to be deleted. Valid responses are `y` to delete and move on to the next object, `n` to keep the object, `g` to go on and delete all remaining objects and `q` to quit out of `r` em(). Each reponse must be followed by a carriage return; the `g` response will cause a further query to ensure the user really wants to delete everything.

Removing certain objects causes the `rem()` functions to remove others as well, so as not to leave orphaned objects in the system. Specifically, removing a metric also removes its entire family of objects, while removing a parent object causes all of its offspring to be removed. If an indexed scalar is removed and it had a value, that value is placed as a normal algebraic value on the now non-indexed name.

`protect()`

The function `protect()` sets or clears an objects' protection flags. The only protections available are write-protection, kill-protection, or both. The first argument to `protect()` is the name the object whose protections are being modifed, and the second argument sets the new protections (a missing second argument is equivalent to `nil`). The values for the second argument may be one of the following:

`kw` or `wk`	set kill- and write-protection
`k`	set kill-protection
`w`	set write-protection
`nil`	no protection

The protection flags are displayed by `dir()` and by `iprop()` , however, the output from `iprop()` will show the internal format of the protection flags, which are simply the numbers 2, 3, or 6 for write-protection, kill-protection, or both, respectively.

`icopy()`

The function `icopy()` copies its first argument to a new object whose name is the second argument. The new objects' name should be unused; however, if it is an unprotected indexed object it will be removed, but its properties are saved and can be restored by an immediate call to `restoreobj()` . The `shift` property of the output will no longer refer to the offspring of the input, but all other properties are copied exactly. A more complete copy is offered by the function `copyfam()` , described below.

`copyfam(), getfam()`

The `copyfam()` function allows the user to copy an object and its entire family (either metric or shifted offspring) to another parent. This routine rebuilds the family structures of each member so that they all correctly refer to new objects. Usually the family of the first named argument is copied to the second argument, but, in the case of a metric, the second argument may be omitted and the default is the next metric name in the sequence used by `metric()` . The copied metric is not made the current-metric.

A family of objects is primarily a relation such that the offspring objects are derived in some manner from the parent, either by shifting indices, taking derivatives, or more complicated relations such as among the tensor metric and various GR quantities. In REDTEN a family is defined by the construction of object names. A family member has the parent name, a separating underscore "_", and some further identifying name. These constructions can be carried out repeatedly, yielding a family tree of related objects. For example, in the `dir()` listing on page 54 one can see that `g1` is the parent of many objects, while `c1` is itself the parent of an object, namely `g1_c1_DF`. The function `getfam()` will return a lisp list of all objects in the family of its argument, including the parent. Thus,

```
#: getfam('g);
 (g1_inv g1_ric g1_R g1_c1 g1_DF g1_c2 g1_c1_DF g1)
#: getfam('c1);
 (g1_c1_DF g1_c1)
```

and observe that the argument must be quoted to ensure the command line is parsed in symbolic mode. Also note that the argument may be either a generic name or the true name.

A common use of `copyfam()` and `getfam()` is to copy a metric family and then apply a consitent set of substitutions to the family without disturbing the original set of objects. For example, if `g1` is a general spherically symmetric metric with a parameter f, and with a family of computed objects, then the following will copy the family and substitute for f:

```
#: %create g1 and family
#: copyfam(g1);
 g2
#: lisp foreach x in getfam(g2) do write algebraic mapfi(sub(f=1-2*m/r,x));
```

```
<output>
#: lisp foreach x in getfam(g2) do if get (x, 'itype) eq 'odf then
        write algebraic mapfi (x);
<output>
```

The first `foreach` can be used to apply a substitution to all objects in the family, note the use of the `write` command to ensure that the `mapfi()` output is seen, and the use of the `algebraic` command to change mode partway along the command line. The second `foreach` command demonstrates the application of `mapfi()` to a specific class of objects, in this case ordinary derivatives. It is also possible to use these constructs with other list generating functions or with the `lisp:indexed!-names` list directly, to access all objects in the system.

`describe()`

The `describe()` function is used to retrieve or set a string containing a description an object. Most generating functions put a description string on their output specifying what kind of object has been created, and from what object it was derived. A string supplied as the second argument will replace the description on the object whose name is the first argument to `describe()`.

`help()`

The REDTEN `help()` function implements a rudimentary help utility to remind the user of the correct number and types of arguments to various REDTEN functions. `help()` takes any number of arguments, which may be patterns of the sort described on page 55; a `!*` pattern will show all help strings, while, for example, the pattern `!*env` will give help for all the functions that deal with the REDUCE environment. With no arguments, `help()` prints a list of the names of functions and switches for which help is available.

`defindextype()`

The `defindextype()` function allows the user to define new types of indices or to modify the range of the pre-defined index-types. See §2.2.2 for a description of the pre-defined index-types and table 2.1 for their properties. The call to `defindextype()` takes the form:

`defindextype ('range:`*int-list*`, index-type-val:`*int*`, {'name:`*id*`}, {'format:`*string*`}, {'chsymgen:`*func*`});`

where `range` is a list of two ordered non-negative integers representing the range of the indices of type `index-type-val` (the range must consist of non-negative numbers ≤ 32). For example, the default range is `'(0 3)` for the tensor indices, which have `index-type-val` 1. For the pre-defined index-types only the range may be changed, other arguments after `index-type-val` are ignored. For new index-types (`index-type-val` ≥ 7), the remaining arguments are: the name by which the new index-type is to be known (eg. tensor, spinor, etc.), the format in which indices are to be printed, and a function that will generate a Christoffel symbol for this index-type for use by `cov()`.

The `name` argument is used by `mkobj()` to determine the `indexed` property of a new object, different index-types may have the same name (such as the spinor or dyad types) so that an apparently mixed object will be given a meaningful property. The format is a string which must include a '%' mark to indicate the placement of the corresponding index element, any other characters are printed literally, so for example, the format for the frame indices is "(%)". The Christoffel symbol generation function is used by `cov()` to determine the appropriate Christoffel symbol to use, some index-types do not have a Christoffel symbol, in which case `chsymgen` is `'nil`. In other cases the function is of the form

```
'(lambda nil (car christoffel2!* '(nil)))
```

where `christoffel2!*` may be replaced by whatever is the correct lisp function. This complicated form is required since the function, following the usual convention, has a return value of the form `'(name . 1)`. It is not really intended that the casual user be concerned with this.

`defindextype()` calls the delta function generator `delta()` to create or modify the system delta functions for the new or updated index-type. If no arguments are given, `defindextype()` prints out the currently defined index-types, which are stored in the variable `lisp:defindextype!*` .

`restrict()`

The `indices` property of an indexed object are derived from the range parameter to `defindextype()` and specify the inclusive group range of the entire index of the object. The `restrict()` function allows the user to narrow this range, so that indexed references apply to only a part of the full range. A new property, `restricted` , is placed on the indexed object that is the first argument to `restrict()` (the name must be quoted to ensure parsing in symbolic mode). The next two arguments are the lower and upper bounds of the group range, these are lists of integers. If either of these arguments is left out, the default is the original group range obtained from the `indices` property on the object. Leaving both bounds off returns the group range to that present when the object was created, but the `restricted` property is not removed, just set to the new values.

For most object types, the restricted ranges must be contained within the original range. For `array` type objects, the user can reset the bounds to any desired values. Attempting to access components of the object outside the currently defined range, even if they are present, will yield 0 from a read or write operation. Indexed assignments will apply only to the current range, as will applications of `mapfi()` . The user can therefore work with a restricted section of an indexed object, without disturbing the remaining portion of the object.

`saveobj()`

If the user is about to modify an object, the value of that object can be saved by a call to `saveobj()` . If something goes wrong, the object's value can be recovered by calling `restoreobj()` . Only one object at a time can be saved, and the system also saves objects during indexed asignments and when `mapfi()` is used.

`mclear()`

The REDTEN hooks in the REDUCE parser in some places make use of global variables that control the parsing action. If an error has occured or the user has aborted an operation, these variables may be left in an inconsistent state. It may then be impossible to correctly enter indexed objects and output may be distorted. The function `mclear()` should always be called if the user suspects a parsing problem or has forced an abort.

If a directory listing shows objects with names of the form `#tmp1` etc, then an abort has been made, but `mclear()` still needs to be called. These object are the temporaries used by the system during the evaluation of an indexed expression.

Chapter 6

Other Packages

NOTE: Much of this chapter material is no longer valid. I will update it as soon as possible. In the meantime, see the demo files such as `kerrnp` for examples.

This chapter presents the other packages in REDTEN that are concerned with the various formalisms used to derive information about a metric: the frame package (a tetrad formalism), a spinor package and the Newmann-Penrose package. Incidental to this discussion is the complex arithmetic package. These packages have not reached the same fullness of developement as the GR package described in chapter 4.

6.1 The Frame Package

The Frame package implements a standard tetrad formalism in which one sets up a basis of four contravariant vectors that form a "connection" between the tensor and frame indices (following Chandra, ?? ref):

$$z^i_{(a)}.$$

The index enclosed in parenthesis distinguishes tetrad indices from normal tensor indices. As well, we may define the covariant vectors

$$z_{(a)i} = g_{ik} z^k_{(a)}$$

and the matrix inverse of e such that:

$$z^i_{(a)} z^{(b)}_i = \delta^{(b)}_{(a)} \qquad \text{and} \qquad z^i_{(a)} z^{(a)}_j = \delta^i_j$$

and we may also define a constant symmetric matrix η so that

$$z^i_{(a)} z_{(b)i} = \eta_{(a)(b)}$$

and

$$\eta^{(a)(b)} \eta_{(b)(c)} = \delta^{(a)}_{(c)}.$$

This matrix functions as a metric for the tetrad indices, and can be used to raise and lower indices. This "metric" is often assumed to be diagonal and Minkowskian so that the vectors $z_{(a)}$ form an orthonormal basis, but other forms can also be assumed, as shown below.

An important relation between the tensor metric and the connections exists:

$$z_{(a)i} z_j^{(a)} = g_{ij}.$$

The tetrad components of any tensor can be found via

$$T_{(a)(b)} = z^i_{(a)} z^j_{(b)} T_{ij} \tag{6.1}$$

and the tensor components of a tetrad can be found via the inverse relation:

$$T_{ij} = z_i^{(a)} z_j^{(b)} T_{(a)(b)}. \tag{6.2}$$

In REDTEN the tetrad metric is created via the function `frmetric()` , which by default creates a metric of the form

$$\eta_{(a)(b)} = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

and with a non-`nil` second argument creates a Minkowskian metric. For example,

```
#: frmetric()
computing eta1
cofactor finished.
determ finished.
invert finished.
  eta1
```

creates a frame metric of the first type, while

```
#: frmetric(nil, t) computing eta2
invert finished.
  eta2
```

creates a Mikowskian frame metric. Note that the second argument to `frmetric()` should *not* be quoted. `frmetric()` makes default metric names in the same way as `metric()` does for tensor metrics: a base name followed by a sequence number. The default base name is `eta` which is the value of `lisp:frmetric` . The optional first argument to `frmetric()` can be used to create a metric with another name. In all cases the target of the generic name `eta` is set to be the newly created metric.

To use the frame package the user must determine in advance the structure of the particular tetrad $z^i_{(a)}$ for the chosen metric. These are entered into the system and, along with the frame metric, the tensor metric can be constructed from them with the function `tenmetric()` (see below). In the current implementation the user must enter components for the $z_{(a)i}$ and declare this as a connection via the `setcon()` command. Again using the Kerr metric as an example, we have

```
#: coords '(t r th ph);
#: frmetric();
computing eta1
cofactor finished.
determ finished.
invert finished.
  eta1
#: mkobj ('(l n m), '(-1));
  l
   a
     n
   a
  m
   a
#: ias (l);
 l[0] = 1;
 l[1] = -rho2/delta;
 l[2] = 0;
 l[3] = -a*sin(th)^2;
 l
#: ias (n);
 n[0] = delta;
 n[1] = rho2;
 n[2] = 0;
 n[3] = -a*delta*sin(th)^2;
 n
#: ias(m);
 m[0] = i*a*sin(th);
 m[1] = 0;
 m[2] = -rho2;
 m[3] = -i*(r^2+a^2)*sin(th);
 m
#: complex (rhob)$
#: let r^2=sin(th)^2*a^2-a^2+rho2$
#: let rhob*cnj(rhob)=rho2$
#: mkobj('z,'(2 -1),'(),'(),'connection);
  a
 z
    b
#: z[0,i]==l[i];
  0
 z      = l
    i     i
#: z[1,i]==n[i];
```

```
  1
 z     = n
   i     i
#: z[2,i]==m[i];
  3
 z     = m
   i     i
#: z[3,i]==cnj(m[i]);
  3
 z     = cnj (m )
   i          i
#: setcon (z);
 z
```

This is the Kinnersley frame used to set up a null-tetrad as the basis of the Newmann-Penrose formalism (see §6.3). The frame is specified by the vectors $(\mathbf{l}, \mathbf{n}, \mathbf{m}, \bar{\mathbf{m}})$, where the bar over **m** indicates complex conjugation. These vectors are copied into the mixed form of z, which then has the first tetrad index lowered via the frame metric to form the connection. Note that the tensor index cannot be raised yet, since the tensor metric does not exist.

At this point it is convenient to discuss the complex arithmetic package in REDTEN. Other complex packages may exist and could be used in place of the one supplied with REDTEN, but they must use the function `cnj()` to carry out conjugation, since this name is coded into the REDTEN source.

6.1.1 The Complex Arithmetic Package

In REDUCE `i` is the pure imaginary number such that `i^2 = -1`[1]. Complex quantities can be constructed using `i` explicitly and the function `cnj()` will form the complex conjugate by changing the sign of `i`. This applies to simple expressions such as sums, products, and quotients. Functions can be declared as complex by giving them a `conjfn` property whose value is the name of a lisp-function that knows how to compute the conjugate of the given function.

A simple variable can be declared complex via the function `complex()` , and this declaration can be removed with `nocomplex()` . When `cnj()` is applied to a complex variable, it returns unevaluated, but the display of the output is of the variable name overtopped with a bar.

Other functions that apply to complex quantities are `re()` which returns the real part of an expression, `im()` which return the imaginary part, `cmod()` which returns the modulus, and `rat()` which attempts to rationalize a complex quotient. This last function will not be successful unless the product formed by the conjugate of the denominator with

[1]This is the one REDUCE name that is not mapped in case when the base system is upper case. Therefore, the user may have to use `I` as the pure imaginary number.

the numerator and the denominator do not contain common factors that will cancel out again.

Continuing with the Kerr metric in a null-tetrad, we can compute the tensor metric from the connection and the frame metric with the function `tenmetric()` . The tensor metric name is the next in the default sequence of names as used by `metric()` , or the name given as the single argument.

```
#:  tenmetrc();
computing g1
  computing z_b
  shift finished.
cofactor finished.
determ finished.
invert finished.
metric finished.
 g1
#: mapfi(g)$
#: mapfi(g_inv)$
```

As in the coordinate basis example, we find it is more convenient to not evaluate the metric components fully, so that `rhob`, `rho2`, and `delta` remain unevaluated; `rhob` has been defined as complex such that `rhob*cnj(rhob) = rho2`. It is, however, convenient to evaluate the derivatives of these quantities for this metric:

```
#: let df (rhob,r) = 1;
#: let df (cnj(rhob),r) = 1;
#: let df (rhob, th) = -i*a*sin(th);
#: let df (cnj(rhob), th) = i*a*sin(th);
#: let df(delta, r) = 2*r-2*m;
#: let df (rho2, r) = 2*r;
#: let df(rho2,th)=-2*a^2*sin(th)*cos(th);
```

Next, we compute the Ricci rotation-coefficients, γ, with the function `gamma()` . The default name of this object is of the form `<frame-metric>_gam`, while the generic name `gam` has this as its target. Note that the name of the output is stored on the `gamma` property of the *tensor* metric. The user may supply a name to be used in place of the default name as the only argument to this function.

The rotation coeffients are rank-3 objects that are anti-symmetric in their first two indices, and are computed from:

$$\begin{aligned} \gamma_{(a)(b)(c)} &= \left(\lambda_{(a)(b)(c)} + \lambda_{(c)(a)(b)} - \lambda_{(b)(c)(a)}\right)/2 \\ &\text{where} \\ \lambda_{(a)(b)(c)} &= \left[z_{(b)i|j} - z_{(b)j|i}\right] z^i_{(a)} z^j_{(b)}. \end{aligned}$$

The quantity λ is used internally by `gamma()` but is not saved.

Each of the interesting tensors (and scalars) of GR can be computed in the frame with the functions `frriemann()` , `frricci()` , `frriccisc()` , `freinstein()` , and `frweyl()` . As with the GR conterparts, each function can take optional argument that is the name to use in place of the default name. The default name is constructed from the *tensor* metric name appended with an underscore and a trailing segment which is the generic name of the each object and which is itself stored on the name of each function. These functions either compute the required object for the current metric if it does not exist, or they return its name.

```
#: frricci();
 computing eta1_frric
   computing eta1_frri
     computing eta1_gam
       computing z_d
       shift finished.
     gamma finished.
     computing eta1_gam_b
    shift finished.
  frriemann finished.
 eta1_frric
```

For this example the final simplification is carried out in a manner similar to that in the coordinate basis:

```
#: clear r^2;
#: let rhob = r+i*a*cos(th);
#: let rho2 = r^2 + a^2 * cos(th)^2;
#: let delta = r^2-2*m*r+a^2;
#: mapfi(frric);
 eta1_frric
#: frric[];
 eta1_frric
```

Again, the Ricci tensor is zero as it should be. In this example we did not examine or simplify the Riemann tensor or even the Rotation coefficients, but it is to be emphasized that for complicated or unknown metrics, this must be done, or the user risks having the calculation fail.

6.1.2 Converting basis

There are no functions in REDTEN that handle the conversion of a tensor in a coordinate basis to its representation in a frame. This can be done by the user directly using the equations 6.1 and 6.2 presented above. Generally, it is easier to start at the beginning with the metric and compute all intermediate objects in either representations, rather than to try to convert from one form to the other.

6.1.3 Covariant differentiation

Currently covariant differentiation reduces to ordinary differentiation and yields an object with a new *tensor* index. The more natural notion of an intrinsic derivative is not defined by any REDTEN function, but can be evaluated directly by the user if required.

6.2 The Spinor Package

In REDTEN a spinor consists of two objects, the named primary object, and a conjugate whose index structure is obtained by swapping each unprimed spinor index for a primed spinor index, and the reverse. The conjugate is named by taking the primary's name and appending `_cnj` to it. Normally conjugate objects are also flagged `nodir` so that they do not appear in directory listings. The conjugate object never has explicit components, any reference to it results in a reference to the primary object and a conjugation of the input or output value. Note also that the index-runs of spinor indices are generally half that of the tensor indices, from 1 to 2 by default.

A very rudimentary package for spinors is included in REDTEN consisting of the following functions: `spmetric()` , `spinmat()` , and `spchristoffel()` . The spinor metric created by `spmetric()` comes in two parts: one metric for unprimed indices (index-type 3), and another for primed indices (index-type 4). Thus a total of four objects are created, as each metric is accompanied by its conjugate. The default names are `e3` and `e4` . Currently, the spinor metrics are not the targets of generic names. This may change in future versions as the package develops. The spinor metrics have the form

$$\begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}$$

i.e. they are anti-symmetric rank-2 objects.

The function `spinmat()` creates the spin-matrices associated with the current tensor metric; currently it can only do this for diagonal metrics. The spin matrices follow the usual naming convention: the tensor metric name, an underscore, and the generic name stored on `lisp:spinmat` . These objects are of type `connection`. The primary object has an index structure of `(-1 3 4)` and has a conjugate symmetry between the primed and unprimed indices.

The function `spchristoffel()` computes the Christoffel symbols via the equation:

$$\left\{ {C \atop a\ B} \right\} = \frac{1}{2} \sigma_{\mathrm{b}}{}^{C\,Y'} \left(\sigma^{\mathrm{b}}{}_{B\,Y'} \left\{ {d \atop a\ b} \right\} + \sigma^{\mathrm{d}}{}_{B\,Y'\,|\mathrm{a}} \right).$$

The Christoffel symbol for the primed indices is obtained from the above through conjugation.

6.3 The Newman-Penrose Package

The Newman-Penrose formalism differs from the tetrad (frame) formalism mostly in the choice of basis, which is now taken to be null. In addition, the usual expression of this formalism draws away from the use of indexed objects and instead defines a new set of complex quantities. The implementation of the Newman-Penrose formalism in REDTEN is based on an article by Esteban and Ramos in *Computers in Physics*, May/June 1990, p. 285.

BUT: there are many errors in the article, not the least of which is a non-zero Ricci tensor for a vacuum metric!. This is due to a transcription error, and involves taking a conjugate when one shouldn't. To be fair, even the original N-P paper has several printing errors!

In the Newman-Penrose fromalism, the tetrad is null, i.e.

$$l_a l^a = n_a n^a = m_a m^a = \overline{m}_a \overline{m}^a = 0,$$

and also satisfies the conditions $l^a n_a = 1$, $m_a \overline{m}^a = -1$, $l^a m_a = 0$, $n^a m_a = 0$.

A connection can be defined in the same fashion as described above for the tetrad package, and the tensor metric can be defined in term of the basis vectors from

$$g_{a\,b} = l_a n_b + n_a l_b - m_a \overline{m}_b - \overline{m}_a m_b.$$

From the basis vectors one computes 12 spin coefficients that are commonly known by their greek names. The Newman-Penrose equations can then be solved for the 5 components of the tetrad projection of the Weyl tensr (Ψ_0, Ψ_1, Ψ_2, Ψ_3, Ψ_4), and the 6 components of the Ricci tensor (Φ_{00}, Φ_{01}, Φ_{02}, Φ_{11}, Φ_{12}, Φ_{22}), following the reference noted above. The Newman-Penrose equations are rather complicated and numerous (there being 18 equations), and are not repeated here.

The REDTEN function `npmetric()` will take either the name of a connection, or the names of the three basis vectors l, n, and m, and compute from them the tensor metric, and, in the latter form of input, also create the connection object (the default name is stored on `lisp:npcon`). The tensor metric is named according to the usual rules of metric names previously described.

The function `npspin()` computes the spin coefficients, and places them in an array whose name is stored on the variable `lisp:npspin` . The array has indices that run from 1 to 12. In order, these array components are α, β, γ, ϵ, κ, λ, μ, ν, π, ρ, σ, and τ. For convenience, the REDUCE names **alpha**, **beta**, **gamma**, **eps**, **kappa**, **lambda**, **mu**, **nu**, **pi**, **rho**, **sigma**, and **tau** can be made equivalent to the array references with the function `npnames()` . This function has no arguments, and causes the names to be mapped by the REDUCE parser. Since some of these names may be useful in other situations the mapping behaviour is not on by default. Calling `npnames()` again removes the mapping.

The functions `npweyl()` and `npricci()` compute the tetrad projections of their respective tensors. However, in the Newman-Penrose formalism, these projections are not normally regarded as being indexed objects, the subscripts serving only to distinguish the components. In REDTEN the Weyl components are stored in a one-dimensional array whose name is stored on `lisp:npweyl` . The Ricci components are stored in a symmetric two-dimensional array whose name in on `lisp:npricci` .

A demonstration of the Newman-Penrose package using the Kerr metric as an example follows.

We first ask that the spin-coefficient names be mapped by a call to `npnames()` . Then we create and assign the basis vectors (which are of course different than in the previous example).

```
#: coords '(t r th ph);
#: npnames();
#: mkobj ('(l n m), '(-1));
  l
   a
  n
   a
  m
   a
#: complex rhob;
#: rho2 := rhob * cnj(rhob);
#: ias (l);
 l[0] = sqrt(2);
 l[1] = -sqrt(2)*rho2/delta;
 l[2] = 0;
 l[3] = -sqrt(2)*a*sin(th)^2;
 l
#: ias (n);
 n[0] = sqrt(2)*delta/(2*rho2);
 n[1] = 1/2;
 n[2] = sqrt(2)*delta/(2*rho2);
 n[3] = -sqrt(2)*a*delta*sin(th)^2/(2*rho2);
 n
#: ias(m);
 m[0] = i*a*sin(th)/rhob;
 m[1] = 0;
 m[2] = -rho2/rhob;
 m[3] = -i*(r^2+a^2)*sin(th)/rhob;
 m
#: let r^2=sin(th)^2*a^2-a^2+rho2$
#: let rhob*cnj(rhob)=rho2$
#: let df (rhob,r) = 1;
#: let df (cnj(rhob),r) = 1;
#: let df (rhob, th) = -i*a*sin(th);
#: let df (cnj(rhob), th) = i*a*sin(th);
#: let df(delta, r) = 2*r-2*m;
```

We can now compute the tensor metric, the spin coefficients and the projections of the Weyl and Ricci tensors.

```
#: npmetric(l,n,m);
cofactor finished.
determ finished.
invert finished.
metric finished.
 g1
#: mapfi(g);
 g1
#: mapfi(g_inv);
 g1_inv
#: on showindices;
#: npspin();
computing g1_npspc
  computing npz_CD
    computing g1_c2
      computing g1_c1
      christoffel1 finished.
    christoffel2 finished.
  cov finished.
computing alpha
  computing npz_c
  shift finished.
computing beta
computing gamma
computing eps
computing kappa
computing lambda
computing mu
computing nu
computing pi
computing rho
computing sigma
computing tau
 g1_npspc
#: npweyl();
computing g1_npC
computing g1_npC[0]
computing g1_npC[1]
computing g1_npC[2]
computing g1_npC[3]
computing g1_npC[4]
 g1_npC
#: npricci();
```

```
computing g1_npric
computing g1_npric[0 0]
computing g1_npric[0 1]
computing g1_npric[0 2]
computing g1_npric[1 1]
computing g1_npric[1 2]
computing g1_npric[2 2]
 g1_npric
#: off showindices;
```

The final values for `rhob` and `delta` are entered, and the objects are resimplified. The simplification is done in two steps. First the new definitions for `rhob` and `delta` are combined in each component (which in most cases results in 0), then the remaining components are rationalized via the `rat()` function. If the initial simplification is not done, then a large amount of work involved in rationalizing expressions that eventually collapse to zero would be wasted.

```
#: clear r^2;
#: let rhob = r+i*a*cos(th);
#: let delta = r^2-2*m*r+a^2;
#: mapfi (npC);
 g1_npC
#: mapfi (rat (npC));
 g1_npC
#: mapfi (npric);
 g1_npric
#: mapfi (rat (npric));
 g1_npric
#: on factor;
#: npC[2];
           2  2    2      2                          2  2        2
 (((sin(th)  a  - a  + 3 r ) cos(th) a i + 3 sin(th)  a  r - 3 a  r
       3             2  2    2    2 3
    + r ) m)/(sin(th)  a  - a  - r )
```

In these results one will notice that $\cos^2\theta$ has been factored in favour of $\sin^2\theta$, due to the existence of a `let`-rule defining the relation. For this metric, a better choice would have been the reverse rule.

6.3.1 Newman-Penrose Operators

The four Newman-Penrose operators require the existence of the connection object and operate on algebraic expressions. They are defined as follows:

$$D = l^\mu \frac{\partial}{\partial x^\mu}, \qquad \Delta = n^\mu \frac{\partial}{\partial x^\mu}, \qquad \delta = m^\mu \frac{\partial}{\partial x^\mu}, \qquad \overline{\delta} = \overline{m}^\mu \frac{\partial}{\partial x^\mu},$$

The REDTEN functions `npD()` , `npDEL()` , `npdel()` , and `npdelc()` implement each of the operators, respectively. For example,

```
#: npD(th*sin(r));
  sqrt(2) cos(r) th
 -------------------
         2
#: npDEL(th*sin(r));
   - sqrt(2) cos(r) delta th
 ----------------------------

           ----
         4 rhob rhob
#: npdel(th*sin(r));
  sin(r)
 --------
  2 rhob
#: npdelc(th*sin(r));
  sin(r)
 --------

    ----
  2 rhob
```

Chapter 7

Enhancements

Among the various REDTEN utilities not intimately associated with indexed objects are some enhancements to REDUCE that have been made in order to make REDTEN easier to use. These deal primarily with the issue of the REDUCE environment: how to save sessions to disk for later use, and how to conveniently switch between several environments in the same system. Some of these enhancements may be available from other authors as separate library packages, but they were not available at the time the ones for REDTEN were produced.

7.1 The REDUCE Environment

The REDUCE environment consists of all `let` rules, assignments to algebraic variables, operators and switches. In working with a complicated metric it may be the case that a number of different, sometimes contradictory[1] `let` rules and assignments may be used. Recalling which variables and `let` rules have been used, and properly clearing them can require more concentration than the task at hand. Thus, REDTEN has an enhancement to help the user in saving, restoring, and altering the REDUCE environment.

Each time the user makes an algebraic assignment, or declares a new operator, the name of the variable or operator is added to a global lisp list (`lisp:!*reduce!-environment`); this is done through a redefinition of certain REDUCE functions. The function `addtoenv()` also does this, adding each argument to the REDUCE environment list; with no arguments `addtoenv()` shows what names are included in the environment list. REDUCE stores `let` rules on certain lisp variables that are already in the environment list. Indexed objects are *not* considered to be part of the REDUCE environment, since it would be pointless to have their values restored to an original state after switching to an old environment. The user can, however, use `addtoenv()` to add an indexed object to the environment, presumably after it has reached some checkpoint state in simplification. Similarly, other objects created in REDUCE by means other than an assignment or `operator` command

[1] In the sense that the simplifier may enter a loop if both exist simultaneously, for example, from the Kerr metric `let r^2=sin(th)^2*a^2-a^2+rho2;` and later `let rho2=r^2+a^2*cos(th)^2` cannot both be in the system together.

(eg. arrays) will not automatically be included in the environment, but can be added with `addtoenv()` .

As a general rule, the portion of the environment that changes with calls to `restorenv()` should involve secondary variables: those which are part of some larger expression plus `let` rules, and not the primary expression(s) being worked upon. Otherwise, as with indexed objects, it would defeat the purpose of having different environments. For reasons explained below, rather than providing a means to remove a symbol from the environment list, REDTEN allows the user to flag a variable so that its value will remain intact during environment switches. The flag `keep` prevents the system from saving or altering the value of a variable, it can be set via the command

`#:` *`lisp flag ('(var1, {var2}, ...), 'keep);`*

Certain variables are already flagged `keep` and retain their values and property lists despite changing environments. Among these are `lisp:currentmetric` and `lisp:reduce!--environment` . Aside from the problem of losing a value if the environment is switched and the work variable is not flagged `keep`, it may well be that storing the values for this variable several times for the saved environments could lead to memory problems.

To save an environment requires that the property lists and lisp values of each symbol in the reduce environment be copied to some location, so that they can later be restored. The location used is based on the user-given name for the environment being saved. The restoration of a named environment copies back this information and overwrites the current property lists and values, thus changing the current set of assignments, operators and `let` rules. If the current environment is required for future use it must be saved first. If an environment is being restored in which a new operator etc. was not defined, that operator etc. will dissappear from the restored environment, but of course can returned if the current environment was saved. A named environment cannot be changed in any way, but it can always be restored, changed, and re-saved.

The primary functions that handle the saving and restoring of the REDUCE environment are `savenv()` and `restorenv()` . With no arguments, `savenv()` prints a list of the currently saved environments. When REDTEN is started, it saves the current environment as "initial"[2] (note that the environment name is an id, not a string). With a name as its single argument, `savenv()` saves the current environment with that name. The same name may be used to have `restorenv()` restore the environment. `restorenv()` will also save the current environment with the name "current", but note that this will be overwritten with each new restore. With no argument, `restorenv()` restores the last saved environment. If the argument given to `restorenv()` is not that of a valid environment, an error results.

Other useful functions are `newenv()` , `swapenv()` , `delenv()` , and `renamenv()` . `newenv()` saves the current environment as either "current" or named by its single argument, then restores the "initial" environment. It is equivalent to `savenv(<name>);`

[2]Unfortunately, the system cannot know about user defined operators or assignments made prior to loading REDTEN, since the required system modification have not been made. `let` rules will be saved, however, and operators and assignments can be accounted for by using `addtoenv()`.

restorenv(initial). swapenv() allows the user to quickly flip back and forth between two environments. Calling swapenv() with a single argument switches to that environment. Calling swapenv() again, with a different argument switches to the new environment. Thereafter, calling swapenv() with no arguments switches between the two environments. At any time, the user can begin switching to a different environment by calling swapenv() with a name. The swapping pair can be set up more quickly by giving two arguments, the first is made the current environment. Note that any changes made to the current environment will be temporarily saved under "current" when the environment changes, but will be lost after repeated switches. delenv() deletes the environment associated with the given argument, its main use is to clean up the system. renamenv() can be used to grab the current environment as saved by restorenv() and move it to a safe name.

When the switch promptenv is turned on, the REDUCE prompt is changed to indicate the currently named environment. If changes have been made then the prompt also includes a '+' symbol. At system start-up the current environment is "initial":

```
#: on promptenv;
(initial) #: a:=34;
 a := 34
(initial)+ #: savenv(adef);
 adef
(adef) #:
```

7.1.1 Saving to Disk

In addition to saving environments within the system, REDTEN includes a package to allow the user to save an environment to disk for later use. The same data structures are used to find the names of things to save, and all stored environments are also saved. This package was intended primarily for the saving of indexed objects and the environment in which they were created, so that they could be re-introduced into a subsequent REDUCE session.

There are two routines in the disk-saving package: savei() and savec() . savei() is used to save indexed objects in internal form, so that they may be used again in normal fashion. savec() is similar to savei() in terms of saving the environment, but saves indexed objects in component form: new variables of the form <name><index> are created, each holding a component of the object <name> with fixed index <index>. When reloaded, the components of these objects must be used as normal algebraic variables, the indexed-object structure is no longer present.

Both of these routines save the entire current REDUCE environment, and any saved environments. The required first argument of each is a file name conforming to the local conventions about such, usually it will be a string enclosed in double quotes. Any remaining arguments are taken to be the names of indexed objects, and patterns may be used. The output file contains lisp code and can be re-read via the usual REDUCE in() command. During both the save and reload operations, the name of each saved or restored symbol is echoed to the screen.

7.2 The Pager

Since the display of even a relatively simple indexed object can occupy several screens it became necessary to introduce a pager to handle the scrolling of output. By redefining a REDUCE primitive REDTEN is able to page the output, allowing the user a chance to see and examine each page at leisure without needing to be quick with the no-scroll key; and to abort the output if it is too long-winded, without needing to use the sometimes-dangerous abort sequence.

If the switch `dopaging` is on, any output of an algebraic nature[3] will pause after a pre-determined number of lines. This number is stored in `lisp:lisp:screenlines!*` [22] and can be set by the user directly, from a startup file (assuming this is supported) or from the environment variable SCREENLINES (again, assuming this is supported). At each pause the prompt `Continue?` appears, asking the user for input before continuing. Valid inputs are (case is ignored):

- `q` or `n` — A throw to the top-level input loop is made, thus aborting the output
- c — continue the output with no more pauses
- d — make the next page half-height
- other — any other response continues the output, pausing after the next page.

All of the above responses must be followed by a carriage return to become effective; a carriage return alone is ignored.

For the pager to operate properly the up-cursor escape sequence `lisp:upcursor!*` must be correctly defined. This allows the pager code to overwrite the prompt and produce a seamless output.

7.3 Miscellaneous

In pre-3.4 versions of REDUCE the display of derivatives of functional forms (i.e. things declared with the `depend()` command) was somewhat dull. A quick fix was made in REDTEN to allow these forms to display as, for example,

```
#: depend f,r$
#: df(f,r);
   f
    r
```

rather than

```
#: df(f,r);
   df(f,r)
```

[3]The pager does not interact with other forms of output, such as that produced by loading new function definitions etc.

In REDUCE 3.4 a very similar package exists, controlled by the switch `dfprint` , which has been turned on in REDTEN.

In addition, to handle the display of certain constructs that can appear in REDTEN (derivatives with respect to coordinate vectors), another display package has been introduced for this case. If the switch `fancydf[off]` is on, then this package is used for all derivatives of functional forms. The format is of a fraction involving differentials, with the appropriate degrees located in the conventional places.

```
#: df (f,x[a]); % note f must depend on at least one coordinate.
    d f
  -------
      a
   d x
#: on fancydf$
#: df(f,r,r);
    2
   d  f
  ------
      2
   d r
```

A minor enhancement is obtained when the switch `bell` is turned on, in this case the terminal bell is sounded after every prompt. The only use for this is to notify the user at the end of a long calculation, at other times the bell is annoying.

To enable easier timing of the entire process of evaluating a metric family, two new timing functions have been added to REDTEN. To start the timer the function `stime()` can be called; the elapsed CPU time since the last call to this function is returned by the function `etime()` . The timer is not reset by `etime()` , so it can be called many times in succession, and will show progressively larger elapsed times until a new call to `stime()` is made.

7.4 Case Mapping

REDTEN is a two-case system, with the primary input being in lower-case. However, the underlying lisp system in which REDUCE is implemented may be either upper- or lower-case. If it is upper-case, all REDUCE and REDTEN functions will be defined in that case, despite the fact that the sources are in lower case, except in special protected situations. For these systems a small package is provided to map the names of commonly used functions from the lower-case input to the upper-case definitions. The user may thus type either `sin()` or `SIN()` to use the sine function; and similarly for a large number of other functions, switches and variables. Any identifier that is not case mapped may be made so in the following fashion:

```
#: PUT ('mapthisname, 'NEWNAM, 'MAPTHISNAME);
```

which will cause the parser to replace any occurance of the first name with the second. This will only work correctly if the default REDTEN behaviour of not mapping any characters to an alternate case remains enabled (i.e. the REDUCE switches `raise` and `lower` are both off).

The mapping method shown above cannot be used for the special symbol `nil`, since `NIL` as an input to `put()` removes the property on the key `NEWNAM`. For this identifier only, and the, the user must use the correct case.

Appendix A

Extending REDTEN

From the user perspective REDTEN consists of two levels: a lower level of indexed algebra in which the user can enter indexed expressions and have them evaluated; and a higher level in which the user applies functions that compute objects of interest and which maintain the book-keeping in the system. The higher level functions are "layered" onto the lower level in the same way that REDTEN itself is layered onto REDUCE. The packages described in Chapters 4 and 6 are high level functions, but it is important to realize that the basic REDTEN indexed algebra engine could be used to compute interesting quantites even without these packages, as was done in Chapter 2.

Although the development of the GR package is essentially complete, the spinor, frame (tetrad), and Newman-Penrose packages require more work. As well, there are many other computations that would be useful to have automated in high level functions, such as Petrov classification of metrics, etc [1]. In this sense REDTEN is incomplete: there are always other interesting objects to compute, in other formalisms etc., which would be convenient (but not absolutely necessary) to have high level functions to deal with.

This appendix describes how the user can extend REDTEN by writing functions to automate the computation of specific indexed objects. It is assumed that the user has a basic understanding of RLISP and lisp in general. NOTE: Although the user is free to modify or extend REDTEN as required, no such modified copy may be redistributed without the modifications being plainly marked, and the copy must not be misrepresented as the original. If you have a brilliant addition, send it to the authors to be included in the distribution.

From the programmers perspective, there are actually three levels in REDTEN: a "basement" level of basic functions lies under the low level (the division is somewhat blurry). Some of these functions are required when writing high level code, as they deal with the mundane details of maintaining indexed objects in the system.

As an example of writing high level function, we shall use the `riemann()` function of the GR package as a starting point. All of the high level functions work in the same basic fashion: they write an indexed expression the equivalent of what the user would type and cause it to be evaluated. The complications come with all of the other book-keeping functionality required. Before actually detailing the structure of a typical high

[1] Some of these are under development

User Op	Internal form	User Op	Internal form
|	!*br	||	!*dbr
@	!*at!*		
@-	!*at!-	@+	!*at!+
:[	!*lsqb	:]	!*rsqb
:{	!*lcub	:}	!*rcub
:&	!*bach		

Table A.1: Internal forms of index operators

level function, consideration must first be given to the internal representation of an indexed object.

The user interface to an indexed object consists of a name followed by an index enclosed in square brackets. The indexed object name has another property not mentioned in §2.2.6: the `simpfn` propert gives the name of an RLISP function to execute when the algebraic simplifier encounters the name. All indexed objects are handled by the same function (`mkrdr()`) which receives both the object name and the separately parsed index as arguments. This function then constructs the internal "`rdr`-form" of an indexed object; it also handles some error checking and display requests (see §2.4.3).

Internally, an indexed object (after parsing) is a lisp list whose first element (the `car()`) is the "function" `rdr`[2] which itself has a `simpfn` property pointing to the RLISP function `simprdr()` . The `simprdr()` function supervises the read-out of object componenets if the index is fixed, and returns an unevaluated `rdr`-form otherwise. The second element (`cadr()`) of the `rdr`-form is the indexed object name, and the third (`caddr()`) is the index, containing the internal forms of any index operators. A fourth element is no longer used, but some REDTEN code may still make a harmless reference to it.

The various index operators are all translated into an internal form when they are parsed as shown in table A.1. The index itself becomes an ordinary lisp list, most operators exist as elements at the top level of this list, except for the shift operations which become sublists with the operator first, and the index element second. Knowing these internal forms the programmer can write any indexed expression.

To write a high level function first requires an equation defining the output indexed object; the definition of the Riemann tensor (equation 3.3) is repeated here for reference with different indices,

$$R_{hijk} = \frac{d}{dx^c}[bd,a] - \frac{d}{dx^d}[bc,a] + \left\{ \begin{matrix} e \\ b\ c \end{matrix} \right\} [ad,e] - \left\{ \begin{matrix} e \\ b\ d \end{matrix} \right\} [ac,e]. \tag{A.1}$$

Below is the source code for the `riemann()` function. The function performs several basic tasks: determining the name of its output object, checking to see if the object already

[2]The symbol `rdr` is historical; it might once have been intended to be a short form of "reader" in the original muTENSOR version.

exists, creating it if not, setting up and evaluating the indexed expression which computes the output, and some book-keeping and cleanup. Each line of the function is described separately after.

```
riemann := '!R; ........................................ 1
put ('riemann, 'simpfn, 'riemann!*); ................... 2

symbolic procedure riemann!* (u); ...................... 3
begin scalar tnsr, lex; ................................ 4
  tnsr := mycar getnme (mycar (u), '(nil . nil), 'riemann); ....... 5
  lex := get (getmetric (1), 'riemann);  dotfill 6
  if not tnsr and indexed (lex) then return (lex .  1); ........... 7
  tnsr := newnme (tnsr, riemann); ....................... 8
  mktnsr!* (tnsr, '(-1 -1 -1 -1), '(((-1) 1 2)((-1) 3 4)((2) 1 3)),
           '(), 'riemann,
           mkmsg list("Riemann tensor created from metric %",       ..... } 9
           getmetric(1)));
  put (tnsr, 'printname, riemann); ...................... 10
  lex := list ('plus,
    list ('rdr, mycar (christoffel1!* ('nil)),
                               '(b!# d!# a!# !*br c!#)),
     list ('minus,
    list ('rdr, mycar (christoffel1!* ('nil)),
                               '(b!# c!# a!# !*br d!#))),
     list ('times,                                                  .... } 11
    list ('rdr, mycar (christoffel2!* ('nil)), '(e!# b!# c!#)),
    list ('rdr, mycar (christoffel1!* ('nil)), '(a!# d!# e!#))),
     list ('minus,
     list ('times,
    list ('rdr, mycar (christoffel2!* ('nil)), '(e!# b!# d!#)),
    list ('rdr, mycar (christoffel1!* ('nil)), '(a!# c!# e!#)))));
  evaltnsr1 (tnsr, '(a!# b!# c!# d!#), lex, 'nil); ...... 12
  protect!* (tnsr, 'w); ................................. 13
  put (getmetric (1), 'riemann, tnsr); .................. 14
  if not get (tnsr, 'tvalue) then << .................... 15a
    tabthenprint ("** this space is flat"); ............. 15b
    terpri ()............................................ 15c
    >>;
  cleaner ('riemann); ................................... 16
  return (tnsr . 1); .................................... 17
end; .................................................... 18

makegeneric (riemann, '(1 . riemann), '(-1 -1 -1 -1),       ............... } 19
  '(((-1) 1 2)((-1) 3 4)((2) 1 3)), 'riemann!*)$
```

1. We first define the default name of the object that this function will create. It is conventional in REDTEN to assign this name to the lisp varibale with the same name as the user-name of the function, in this example `riemann`.

2. When the user executes the `riemann()` function, the REDUCE parser examines the `simpfn` property of `riemann` for the name of the actual RLISP function to execute. It is conventional in REDTEN to name this function by appending a `!*` to the user-name (there are plenty of violations of this convention, however). Thus, the actual work of computing the Riemann tensor is done by the RLISP function `riemann!*()` (but see §E.1 for a caveat).

3. A function called in this way receives a single argument which is a list of all actual arguments given in the function call (note the arguments should not be quoted). This statement begins the actual definition of the function (or "procedure" in REDUCE terminology).

4. Various local variables will be required, which should be declared here, otherwise they will become "fluid" when this function is compiled.

5.
   - The `mycar()` function is similar to the standard lisp `car()` function, except that it works with `nil` as well, i.e. `mycar(nil)` → `nil` (in many lisps `car ('nil)` is an error because `nil` is an atom). The `mycar()` function (and its siblings) are used throughout the REDTEN code, even where their enhanced functionality is not required. Thus, `mycar(u)` is the argument to `riemann()`, or `nil` if none was given.
   - The `getnme()` function is used to determine the name (and index) of a given object from a number of possible formats. The input can be just a name, as will be the case here, and either indexed or not; or the input can be an `rdr`-form, where the index may indicate shift operations and the offspring object will be created if it does not yet exist. The second argument is a dotted-pair the first element of which indicates whether the indexed object must already be indexed (i.e. the object is being used for a calculation), or can be non-indexed (i.e. it is the name to use when creating the output object). The second element of the pair indicates if a name is even required (i.e. there is a default available). In this example a name is not required, but if one is given it need not be indexed. The last argument to `getnme()` is simply the name of the calling function, for use in error messages.
   - The return from `getnme()` is a list of two elements: the first is the object name, the second is its index. Thus, `tnsr` is assigned the name the user typed (if any), and will get the value `nil` if no name was given. Any index is ignored. If an index was attached to the input name, that name is already indexed, and, if not protected, will be destroyed and remade.

6. For the current metric the Riemann function stores the name of the object it creates on the metric under the key `riemann` (see line 14). Any future calls to `riemann()`

will find and return this name without making new calculations *if* the user did not supply a name (see line 7). The current tensor metric (returned by `getmetric()` with an argument of 1) is checked for a `riemann` property, any value found is assigned to `lex`; `nil` is assigned if no property exists.

7. If the user does not supply a name to use for the output object, and the value found at line 6 is that of an indexed object, that object is returned immediately (see line 17 for an explanation of the format of the return value). If the user did supply a name for the output object, then a new object will be created and the calculations performed, even if the same calculation has already been done.

8. The actual name of the output object is determined from the following rules: If a name was supplied by the user, then that is the name of the new object (the generic Riemann name will also point to it, see `generics()`). If no name was supplied, then the default name (stored on `lisp:riemann`) is made into the object name by prepending the name of the current tensor metric and an _ character. Thus, if the metric has the name `g1`, the default name of the Riemann tensor created for this metric is `g1_R`.

   `newnme()` requires two arguments: the user-input name (which may be `nil`), and the default name (stored in this case on `lisp:riemann` , note there is no quote before `riemann`: its value is passed to the function). `newnme()` also prints the `Computing ...` message to indicate to the user what the real name of the object will be. The generic name for the Riemann tensor will point to this object.

9. With the name of the output object stored in `tnsr` the internal function to create indexed object is called (still named `mktnsr!*()` although it can create any kind of indexed object). Its arguments are nearly as described in §2.2, but the symmetry lists are entered in internal format where the block size is placed in a sublist in each independent symmetry. The object has four covariant tensor indices, a Riemann symmetry, is not implicit, the `itype` is `riemann`, and the description string is set to something informative.

   `mkmsg()` takes one list, the first is a string containing `%` symbols. These are replaced in sequence by the remaining elements of the list. To print an indexed object wrap its `rdr`-form with the `rdr!-string()` function; to print a list as an index wrap it with `index!-string()` ; `\%` produces a literal `%`.

10. The `printname` of the object is changed so that no matter what its real name is it is *displayed* in the same fashion (by default using the name `R` in this example).

11. Equation A.1 is now used to generate an indexed expression that will compute the Riemann tensor. This is written in an internal REDUCE format called "prefix-form". The result is assigned to `lex` for use in line 12 (this assignment could have been bypassed, writing the expression directly into line 12, but the code would be harder to read).

The prefix form involves lisp lists whose first element is an algebraic function, which will be applied to the remaining arguments. The following functions may be useful in constructing an indexed expression:

Function	Type	Remarks
rdr	binary	standard internal form of indexed object
quotient	binary	first arg divided by second arg
plus	n-ary	minimum of two arguments required
minus	unary	negation of single arg
times	n-ary	minimum of two arguments required
df	1 + n-ary	first arg differentiated against remaining args.
pdf	binary	first arg an algebraic expression, second arg a list representing an index.
expt	binary	first arg raised to power of second arg

There are two ways to construct a list in lisp: using the quote function (the input shorthand is `'`) which *does not* evaluate its arguments, and using the function `list()` which *does* evaluate its arguments. If a list is to be written whose contents may vary between one evaluation and the next, the `list()` function must be used. Generally, only numbers will be constant as such (for example, 1/2 can be constructed as `'(quotient 1 2)`); even if the name of an indexed object is assumed to be known it is better to make use of a variable which holds the name, so that the name can easily be changed later without affecting the code. Hence, writing indexed expressions requires many calls to `list()` . Notice that the arguments to `list()` must themselves be quoted if they are to be used literally. If any sublist is constructed using the `list()` function, all enclosing lists must be constructed the same way (not using the quote function), or the sublists will not be evaluated.

Consider the code fragment

```
list ('rdr, mycar (christoffel1!* ('nil)), '(b!# d!# a!# !*br c!#))
```

which creates an `rdr`-form representing the first term of equation A.1. The name to be used for the indexed object is found by calling the `christoffel1!*()` function, which works in the same way as the `riemann!*()` function considered here. That is, `christoffel1!*()` either returns the name of the previously computed object, or does the computation now. In either case the return value is of the form described under line 17, so that `mycar()` is used to isolate the name. This method of having one function call others that it depends on allows the user to create all the objects of interest from the metric with one command (but see §5.1 for reasons why this is not usually a good idea).

The index can of course be written with specific indices and is thus constructed using the quote function. The actual indices are selected from those stored in the variable `lisp:alphalist!*` (which is simply a list of indices that the user will not use, since algebraic values assigned to these symbols could potentially have side-effects). The index is written with the internal forms of the index operators as shown in table A.1;

in the example above an ordinary differentiation operator has been placed in the index.

Shift operations are indicated by making the index-element into a sublist with the operator as the first element. For example, an index might be written `'(a!# (!*at!* b!#))` to indicate shifting the second index-element. If the exact `indextype` of an available object is unknown, the absolute shift operators can be used to ensure a canonical form is used in the expression. This technique is used in the frame package where it is not certain in advance whether the user has defined a covariant or contravariant connection.

Examination of the rest of line 11 will show the correspondence with the terms of equation A.1. The repeated index `e!#` represents a contraction in the two products; and note the use of the unary minus to negate the last term.

12. The indexed expression in prefix-form is now passwd to the evaluator `evaltnsr1()` . This function is called almost directly when the user makes an indexed assignment with `==`, and receives the name and index of the object on the left-hand side, and the indexed expression on the right-hand side. A fourth argument is no longer used. It is in this function that the two levels of REDTEN alluded to above meet: the `riemann!*()` function as examined so far exists simply to automate the construction of an expression the user could have typed in by hand.

    When `evaltnsr1()` finsishes, the output object (whose name is the value of `tnsr`) will have the appropriate values. If `showindices` is on, then the indices will be displayed as the calculation proceeds.

13. It is usual to apply a write-protection to the output object to prevent the user from accidentally changing components. If modification is desired, then the user can clear the protection with `protect()` .

14. The name of the output object is then placed on the property list of the current tensor metric under the key `riemann`. This is checked in line 6 to see if the object has previously been computed for this metric. This property is also used by the generic object system.

15. Lines 15 do a special check; other functions may have similar checks or none at all.

    (a) The components of the object are stored on the `tvalue` property. If there is no value here, then the object has no explicit components, i.e. it is 0. The body of the `i f` statement is enclosed by the `<<` and `>>` symbols.

    (b) In this case an informative message is printed. The `tabthenprint()` function keeps track of the indenting of messages delivered during calculations.

    (c) `terpri()` simply terminates a line, moving the cursor to the beginning of the next.

16. A call to `cleaner()` prints the `... finished` messages for functions that have been called upon to perform work (but not directly by the user), and it cleans up temporary objects if this function was called directly by the user. Its single argument is the user-name of the function in which it is used.

17. Finally, the name of the object is returned. To return a single name to the algebraic simplifier it should be in the form of a dotted-pair whose first element is the name, and whose second element is 1. Any function which calls this one must use `car()` to isolate the actual object name in the return value.

18. The function definition ends here.

19. To interface with the generic name system, a psuedo-object is created by `makegeneric()` which has many of the properties of the actual object. The first argument to `makegeneric()` is the name of the generic object, in this case the value of `lisp:riemann` (`R`). Recall that the name of the actual object is of the form `g1_R`. The second argument specifies on which metric the relation between the generic and actual object will be stored (type 1 in this example, i.e. the tensor metric), and under what property to look for the relation. In this case, when the generic object `R` is used, the system will examine the tensor metric for the `riemann` property, and use the object so specified (this is the same property used by the `riemann!*` function itself). The third and fourth arguments gives the `indextype` and symmetries of the generic object; the last argument gives the name of the lisp function that can be used to compute the actual object.

Appendix B

The Sato Metric

This appendix provides an example of a very complicated metric that pushes the limits of the capabilities of REDUCE and REDTEN. A description of the metric may be found in ?? The Sato metric is axially-symmetric and .. The metric coefficients are comprised of rational polynomials This metric provides a prime example of "intermediate expression swell", since the combination of the rational coefficients grows explosively, before the final collapse to zero.

Until the introduction of computers with large amounts of real and virtual memory, this metric could not be shown to be vacuum by REDTEN. The example below causes REDUCE to swell in size to somewhat more than 12 Megabytes, an alternative approach to the problem once yielded a REDUCE of more than 50 Meg. in size.

This example can be found in the `demo` directory. The output below was generated from REDTEN running in REDUCE 3.4 in CSL on a MIPS M/120.

```
#: coords '(t x y ph)$
#: on time$
#: depend a,x,y$
#: depend b,x,y$
#: depend c,x,y$
#: let x^2 = u + 1$
#: let y^2 = 1 - v$
#: let df(u,x) = 2*x$
#: let df(v,y) = -2*y$
#: z := (m*p/2)*x*y;
       m p x y
 z := ---------
          2
#: rho := (m*p/2)*sqrt(x^2-1)*sqrt(1-y^2);
         sqrt(v) sqrt(u) m p
 rho := ---------------------
                  2
#: om:=2*m*q*c*(1-y^2)/a;
```

```
        2 c m q v
 om := -----------
            a


#: egam:=a/(p^4*tmp);
            a
 egam := --------
           4
          p  tmp
#: f := a/b;
       a
 f := ---
       b
#: ds := 1/f*(egam*(d(z)^2+d(rho)^2)+rho^2*d(ph)^2)-f*(d(t)-om*d(ph))^2;
             2  2  2  4      2  2           2  2  2  2  2        3
 ds := (d(ph)  b  m  p  tmp u  v  - 16 d(ph)  c  m  p  q  tmp u v
                                2          2         2  2  2
         + 16 d(ph) d(t) a c m p  q tmp u v  - 4 d(t)  a  p  tmp u v
               2    2  2           2    2  2  2       2    2  2  2
         + d(x)  a b  m  u v + d(x)  a b  m  v  + d(y)  a b  m  u
               2    2  2              2
         + d(y)  a b  m  u v)/(4 a b p  tmp u v)
#: metric (ds);
computing g1
cofactor finished.
determ finished.
invert finished.
 g1
Time: 1150 ms
#: off exp$
#: tmp := (x^2-y^2)^4;
               4
 tmp := (u + v)
#: on factor$
#: mapfi (g);
 g1


Time: 483 ms
#: mapfi(g_inv);
 g1_inv
Time: 334 ms
#: off factor,exp$
#: christoffel1();
```

```
computing g1_c1
 g1_c1
Time: 2133 ms
#: christoffel2();
computing g1_c2
 g1_c2
Time: 1900 ms
#: riemann();
computing g1_R
 g1_R
Time: 11317 ms  plus GC time: 1333 ms
#: ricci();
computing g1_ric
 g1_ric
Time: 7183 ms  plus GC time: 750 ms


#: let q^2=1-p^2$
#: A:=(p^2*(x^2-1)^2+q^2*(1-y^2)^2)^2-4*p^2*q^2*(x^2-1)*(1-y^2)*
     (x^2-y^2)^2;
        2  2    2  2    2 2       2             2  2
 A := (p  u  - p  v  + v )  + 4 (p  - 1) (u + v)  p  u v
#: B:=(p^2*x^4+q^2*y^4-1+2*p*x^3-2*p*x)^2+4*q^2*y^2*
     (p*x^3-p*x*y^2+1-y^2)^2;
         2             2          2  2               2
 B := ((p  - 1) (v - 1)  - (u + 1)  p  - 2 p u x + 1)
                            2   2
       + 4 ((u + v) p x + v)  (p  - 1) (v - 1)
#: CC:=p^2*(x^2-1)*((x^2-1)*(1-y^2)-4*x^2*(x^2-y^2))-p^3*x*(x^2-1)*
     (2*(x^4-1)+(x^2+3)*(1-y^2))+q^2*(1+p*x)*(1-y^2)^3;
                                          2
 CC :=  - ((4 (u + v) (u + 1) - u v) p  u
                                        2       3
            + ((u + 4) v + 2 (u + 1)  - 2) p  u x
                 2                 3
            + (p  - 1) (p x + 1) v )
#: on peek$
#: mapfi(sub(a=A,b=B,c=CC,ric));
```

```
(0 0) is zero.
(0 1) is zero.
(0 2) is zero.
(0 3) is zero.
(1 1) is zero.
(1 2) is zero.
(1 3) is zero.
(2 2) is zero.
(2 3) is zero.
(3 3) is zero.
 g1_ric

Time: 590550 ms  plus GC time: 102667 ms
```

Appendix C

REDTEN Functions, Variables, and Switches

C.1 Functions

This section gives the details of the calling sequence and return values of each of the REDTEN user-callable functions. For a detailed description of what each function does, see the relevant section of the main manual.

The argument type will be one of

id — an identifier is expected.

list — the argument is a list, this implies the use of ' (see 1.2.3). Following the Standard Lisp Report (Marti et.al. 1979) a list of homogeneous entities is denoted by *xxx*-list, where *xxx* is the class of items in the list. For example, an id-list is a list of identifiers.

string — a string, enclosed in double-quotes is expected.

bool — either 't or 'nil.

lit — the argument is the literal value shown.

any — any kind of argument may be given.

int — the argument is an integer.

plist — the argument is a property list.

alist — the argument is an association list.

aexp — the argument is an algebraic expression.

iexp — the argument is an indexed expression.

iobj[n] — the argument is an indexed object (either the name alone, or with an index), of rank n if specified.

index — the index of an indexed object, including the enclosing square brackets.

Any argument name beginning with a ' indicates that the lisp `quote` function should be applied (i.e. type **'** before the argument). This will most often apply to arguments of type **list**, and on occasion to **id**'s as well.

Argument types preceeded by a quote character (**'**) indicate that the argument *must* be quoted when calling the function, failing to do so will likely result in an error. Quoting an argument that is not supposed to be quoted will likewise generate an error.

altmetric (name:*iobj*, metric:*iobj*):*id*=¡name

christoffel1 ({name:*id*}):*id*=¡christoffel1—¡name

christoffel2({name:*id*}):*id*=¡christoffel2—¡name

cleartmp ():*t*

cmod (val:*aexp*):*aexp*

cofactor (name$_1$:*iobj*, name$_2$:*id*, {transpose_flag:*bool*}):*id*=¡name$_2$

complex (val:*aexp*):*aexp*

copyfam (metric:*iobj*, new_metric:*id*):*id*=¡new_metric

cnj (val:*aexp*):*aexp*

cov (name:*iobj*):*id*=¡name_CD

d (val:*aexp*):*aexp*

dalembert (val:*aexp*):*aexp*

defindextype ('range :*list*, n :*int*, {name :*id*}, {format :*string*},
{flg :*bool*}) :*id*=¡name—user¡n

delta (name:*id*, type:*int*):*id*=¡name

describe (name:*iobj*, {description:*string*}):*id*=¡name
This function has a side effect.

det (name:*iobj[2]*,{val:*aexp*}):*aexp*=?—¡val

determ (name:*iobj[2]*, cofname:*id*,{trans_flag:*bool*}):*id*=¡cofname

dir ():*bool*=t

div (name:*iobj*,output:*id*):*id*=¡output—¡name_D

einstein ({name:*id*}):*id*=¡einstein—¡name

fn (func:*function name*, index:*index*):*aexp*
need to declare fname

freinstein ({name:*id*}):*id*¡freinstein—¡name

frmetric ({name:*id*}):*id*¡frmetric—¡name

fricci ({name:*id*}):*id*¡fricci—¡name

friccisc ({name:*id*}):*id*¡friccisc—¡name

friemann ({name:*id*}):*id*¡friemann—¡name

frweyl ({name:*id*}):*id*¡frweyl—¡name

gamma ({name:*id*}):*id*¡gamma—¡name

generics({name$_1$:*id*, ...}):*bool*=t
side effect: prints generic links of specified names, or all of them

geodesic (name:*id*, {affine:*kernel*):*id*=¡name

getcon (n:*int*, m:*int*):*id*=name of connection

getmetric (n:*int*):*id*=name of metric of type n

help ({name:*id*}):*bool*=t

ias (name:*iobj*, {flag:*bool*}):*id*=¡name

iclear(name$_1$:*iobj*{,name$_2$:*iobj*, ...}):*bool*

icopy (in:*iobj*, out:*id*):*id*=¡out

im (val:*aexp*):*aexp*

indexed (name:*iobj*):*bool*

invert (name:*iobj*):*id*=¡name_inv

iprop (name:*iobj*):*bool*

killing (name:*id*,{conf:*bool*}):*id*=¡name

lie (name:*iobj*, vec:*iobj[1]*):*id*=¡name_¡vec

mapfi (name:*iobj*):*bool*

mclear ()

metric (line:*aexp*—name:*iobj[2]*, {name:*id*}):*id*=¡name—¡metric¡metricseq

mkcoords (name:*id*):*id*=¡name

mkobj ('name:*id*—*id-list*, 'indextype :*int-list*, {'symmetries :*list-list*}(), {'implicit :*bool*}, {'itype :*id*}) :*iobj*

multiplier (name:*iobj*, {val:*aexp*}):*id*=?—¡val

nocomplex (z$_1$:*id*, {z$_2$:*id*, ...}):*bool*

nodir (name$_1$:*iobj*, {name$_2$:*iobj*, ...}):*bool*

npmetric(con:*ibobj[2]*—l:*iobj[1]*, n:*iobj[1]*, m:*iobj[1]*):*id*=¡metric_¡metricseq

npnames()

npspin({name:*id*})¡name—¡npspin

npD(val:*aexp*):*aexp*

npDEL(val:*aexp*):*aexp*

npdel(val:*aexp*):*aexp*

npdelc(val:*aexp*):*aexp*

npricci({name:*id*}):*id*¡npricci—¡name

npweyl({name:*id*}):*id*¡npweyl—¡name

nulltetrad ({name:*id*}):*id*¡nulltetrad—¡name

odf(name:*iobj*, {order:*int*}):*id*=¡name_DF¡order

pdf (val:*aexp*, index:*index*):*indexed expression*

protect (name:*iobj*, {k—w —kw:*lit*}):*id*=¡name

rat (val:*aexp*):*aexp*

re (val:*aexp*):*aexp*

rem (name$_1$:*pattern*, {name$_2$:*pattern*}):*bool*

remi (name$_1$:*pattern*, {name$_2$:*pattern*}):*bool*

restoreobj():*id*=?

restrict(name:*iobj*, lb:*int*, ub:*int*):*id*=¡name

ricci ({name:*id*}):*id*¡ricci—¡name

riccisc ({name:*id*}):*id*¡riccisc—¡name

riemann ({name:*id*}):*id*¡riemann—¡name

setcon (name:*iobj[2]*):*id*=¡name

setmetric (name:*iobj[2]*):*id*=¡name

seval (val:*indexed expression*):*aexp*

shift (name:*iobj*):*id*=¡name

spchristoffel ({name:*id*}):*id*¡spchristoffel—¡name

spinmat ({name:*id*}):*id*¡spinmat—¡name

spmetric ({name:*id*}):*id*¡spmetric—¡name

symz(exp:*iexp*):*iexp*

tenmetric():*id*=?

trace (name:*iobj[2]*):*aexp*

weyl ({name:*id*}):*id*¡weyl—¡name

Additional Functions

addtoenv (name$_1$:*id*, {name$_2$:*id*,. . . }):*bool*=t

delenv(name:*id*)

etime()

newenv ({name:*id*}):*id*=previous environment

restorenv(name:*id*):*id*

savec (file:*string*, name_1:*id*, {name_2:*id*}):*bool*

savei (file:*string*, name_1:*id*, {name_2:*id*}):*bool*

savenv(name:*id*):*id*= ?

stime()

swapenv(name:*id*):*bool*

Lisp Functions

put(name:*id*, key:*id*, value:*any*):*any*

get(name:*idm* key:*id*):*any*

flag(names:*id*-list, flag:*id*):*nil*

flagp(name:*id*, flag:*id*):*bool*

prop(name:*id*):*plist*

remprop(name:*id*, key:*id*):*any*

setprop(name:*id*, val:*plist*):*plist*

C.2 Switches

Switch Name	Initial Setting	Purpose
allerr	off	When on causes all REDTENwarning messages to be treated as fatal.
beep	off	When on causes terminal bell to sound after the evaluation of input is complete.
evalindex	off	When on causes the elements of an index to be evaluated, so that the user can make variable substitutions. The default behaviour is to ignore the values of symbols used in an index.
extendedsum	off	Enables the extended Einstein summation: repeated indices at any level indicate summation. The default behaviour is to accept only repeated indices of which one is covariant and the other is contravariant.
fancydf	off	When on causes derivatives of id's with dependencies to print in a fancy quotient-like form. Otherwise, an unevaluated `df()` call is returned.
hide	off	When on causes either display format for indexed objects to print `¡value¿` rather than the tru value, so the user can determine which components are non-zero. Default behaviour is to display the value of each component.
iprop	off	When on causes either of the two display formats for indexed objects to also show the various properties associated with the object, as can also be done with `iprop()` .
mkobjsafe	on	When on will cause `mkobj()` to fail to create any object whose name appears to have been used for any purpose.
paging	on	When on causes the pager code to interrupt the output from a calculation after a screen-full of lines.
peek	off	Similar to `showindices`, except that an indication of whether the component is zero is also given.
promptenv	off	When on causes the name of the currently saved or last restored environment to be displayed as part of the prompt.
reversedir	on	When on causes `dir()` to display objects in oldest-first order. When off causes the objects to be shown in most-recently-created first order.

rewrite	on	Causes the system to echo the input indexed expression in a pretty-printed form, so the user can verify the correct structure of the indices.
showindices	off	When on causes the display of running indices, indicating the progress of calculations.
symzexp	on	Causes symmetrization operations to be expanded immediately when on; they remain unevaluated until an indexed assignment when off.
xeval	off	When on causes an extra simplification of indexed object components to ensure that all current assignments, let-rules and switch settings are applied.

Appendix D

Error Messages

"bad block size: % in %."

The indicated block size in the indicated symmetry list (shown in internal format) is not an integer.

"bad connection %."

The object given to `setcon()` does not have an `itype` of `connection`.

"bad index % for %"

This error may indicate an internal error in the system.

"bad indextype element: %"

The indicated element from an `indextype` list is either not an integer, or does not correspond to a defined type.

"bad input: % (from %)."

The input to the indicated routine is not a valid indexed object or name.

"bad metric: %."

The metric used as input to either `setmetric()` or `altmetric()` is either does not have an `itype` of `metric`, or is not the covariant metric.

"bad pointer: % in %"

The indicated pointer of the independent symmetry list is either not an integer, or is not positive.

"can't make spin matrices – sorry."

The `spinmat()` function only works at present for diagonal metrics.

"cannot assign to %."

The left-hand side of an indexed assignment is either not a single indexed object, or the index of that object indicates a differentiation operation.

"cannot compute cov deriv of generic object: %."

If this message appears the user has applied `cov()` to a generic name for which no target has as yet been created. Create the target object, then retry the operation.

"% cannot be assigned."
: The indicated name is an indexed object and cannot have an algebraic value assigned to it.

"cannot create conjugate %."
: The conjugate object for a spinor cannot be created because it already exists and cannot be removed (presumably it is kill-protected).

"cannot create object: %"
: `mkobj()` cannot create the object because it either already exists and is kill-protected, or the name is a reserved symbol.

"connection type % % does not exist."
: The connection object relating the two types of indices does not exist.

"coord-indextype mis-match %, %."
: The length of the coordinate list (as given to `coords()`) is not the same length as indicated by the run of tensor indices.

"description is not a string or atom: %"
: The `describe()` function will only accept strings or single atoms for use as a description of an indexed object.

"dif op without an index: %."
: The user has terminated an index with a derivative operation, but no following index-element.

"% does not have metric structure."
: The named indexed object given to `metric()` is either not of rank-2, or is not covariant.

"empty index!"
: May indicate an internal error in the system.

"environments do not match: %"

"format not a string."

"free index element % in %."

"improper blocks for hermitian symmetry: %."

"improper contraction: %, %"

"improper line-element: %."

"improper structure for hermitian symmetry: %."

"inconsistent indices: %, %"

"index % cannot be shared.(???)"

"index % wrong length for %."

"index without a name."

"indices do not match ."

Mixed cov and contra indices in expression

"input name required (from %)"

"Internal error: name not passed to mkrdr."

This message indicates a failure in the system and should be reported to the authors along with a sample input and output. It may also reflect a flaw in REDTEN as compiled on the users' system.

"internal error (continuing with no contraction symmetry)"

"internal error."

"internal error, blocks mis-matched (continuing)"

"% invalid as affine connection."

The optional second parameter to `geodesic()` , which is the name of the affine parameter to be used in building the geodesic equations, is not an id.

"% invalid as index."

The second argument to `pdf()` must be a valid object index, including the square brackets.

"invalid index ops"

"Invalid index ops: %

"invalid indextype % for %."

"invalid input: % % % '%' %."

"invalid input: %."

"invalid line element: %."

"invalid pointers for hermitian symmetry: %."

"invalid symmetry: %"

"% is a singular matrix."
: This message indicates that the named rank-2 object is singular.

"% is already in use."
: The name which the user is trying to make into an indexed object is already being used in some fashion. It either has dependencies, an algebraic value, has been entered in another expression, or is already indexed. This error only occurs is the switch `mkobjsafe` is on. The user can overide the error by immediately repeating the failed call to `mkobj()` .

"% is no longer a generic object."
: The named input to `mkobj()` was a generic object. In making the new object the generic properties have been lost. The user should not steal the generic object names.

"% is not a vector."
: The second argument to `lie()` must be a rank-1 indexed object.

"% is write-protected"
: This message means that the indicated object cannot have new components written to it, since it is protected. The user can remove the protection with `protect()` and try again.

"LINES env not valid."
: The LINES environment variable is not an integer, or is negative.

"metric % not simplified"

"metric inverse % not simplified"

"metric type % does not exist."

"mis-matched index ops"

"missing conjugate for % being created."

"missing differential: %."

"missing inverse: %."

"missing tetrad member"

"no such offspring for %"

"non-atomic index element in % for %"

"% not indexed (from %)."
The indicated input to the indicated routine is not an indexed object.

"% not of type coordinates."
The indexed object used as the indeterminate in a `df()` command is not of `itype coordinates`. This object must either be the default coordinates (whose name is stored in `lisp:mkcoords`), or an object created by the user with a call to `mkcoords()` .

"object % must have two indices"

"output % is protected"

"overlapping blocks in %."

"pointer out of order in %."

"restricted indices invalid: % %"

"subscript out of range: %"

"symmetry out of order: %"

"symmetry too long: %"

"this shifted object does not exist: %"

"too few pointers in %"

"too many contraction indices: %, %"

"too many indexed objects in arg: % %"

"Update or remove derivatives after changing %."

"unmatched coordinates: %, %."

Appendix E

Getting Started

Since REDTEN is a large package layered onto REDUCE, the first prerequisite to running REDTEN is to have a computer that can successfully run REDUCE, and to obtain REDUCE itself[1]. The range of machines which can be used is now very large, from relatively small 286's to large super-minis and mainframes. In all cases, at least 4 Megabytes of RAM are recomended if any reasonable amount of work is to be done. With the ever-decreasing cost of memory, this requirement is no longer particularly difficult or expensive to meet.

The REDTEN distribution comes as a compressed tar file of a little more than 500 Kbytes. It uncompresses to about 1.4 Mbytes. One needs twice this amount of disk space to hold the tar file and the extracted sources. The size of the compiled REDTEN varies depeding on the lisp system used, but can be a few hundred Kbytes.

Assuming that a REDUCE package has been obtained for a particular machine, bringing up REDTEN is a straightforward process of compiling the source modules together into a loadable file. The user should make note of the following information:

- The REDUCE version; currently versions 3.5, 3.4 (and subversions thereof) and version 3.3 are supported. Support for REDUCE version 3.2 has fallen off.
- The kind of lisp system used to support REDUCE. REDTEN has been compiled in PSL, CSL, Culisp. Other lisp bases for REDUCE may require some minor amount of adjustment to REDTEN before it will run correctly (see below). Ask your system manager if you do not know which lisp is used.
- The default case of the base system. This should be evident after using REDUCE once: if input typed in lower case is reprinted in upper case, the base system is upper case.

This information is used to set up the following files in the REDTEN source directory (see also the file `README` in the source directory).

- In the `src` directory there are some local files: `local3?.red`. Each of these contains the redefinitions of some REDUCE functions for a particular version of REDUCE.

[1] Where to get REDUCE

The appropriate file must be linked or copied to `local.red`, which is the file that is loaded with the rest of the REDTEN system.

- There are also some decl files: `decl3?.red`. If your REDUCE version is 3.2 or 3.3, copy or link `decl33.red` to `decl.red`; if the version is 3.4 or higher copy `decl34.red`.
- If the base case of your REDUCE is upper, copy (or link) the file `bcaseup.red` to `basecase.red`, otherwise copy `bcasedn.red` to `basecase.red`.
- REDTEN makes use of two functions that are not part of Standard Lisp, but which are included in every reasonable lisp. In PSL these functions are named `prop()` and `unboundp()` . Other lisps may use different names. The `src/sysfn` sub-directory contains files for the definition of these functions for other lisps. Copy or link the appropriate system file into the source directory under the name `sysfn.red`. For example, for csl lisp copy `src/sysfn/csl.red` to `src/sysfn.red`.

Under a UNIX system (or one that is capable of running shell scripts), the shell script `setup` in the root directory of the distribution will determine the above information and set-up the first three system files correctly. This script assumes a link to the REDUCE executable has been made in the root directory of the REDTEN distribution. The determination of the lisp type and the copying of `sysfn.red` is up to the user.

Finally, REDTEN makes use of a control sequence to move the cursor up a line on the screen, this is stored as a list on the variable `lisp:upcursor!*` . The default value is set near the top of the `sys.env` file, and is appropriate to a VT100 terminal or similar. If different kind of display screen is used, the correct control sequence can be defined here (the list consists of literal control character and strings). The control variable can also be set from the users environment via the UPCUROSR environment variable. This is still a bit flakey.

Once the source files are properly set the user must define the `redten` environment variable. Under DOS try

```
set redten=<path to root of REDTEN distribution>
```

and under UNIX (and csh) type

```
setenv redten <path to root of REDTEN distribution>
```

You are now ready to compile REDTEN. The exact details vary depending on the underlying lisp system. For PSL and similar lisps, move in to the `fasl` directory and start REDUCE. Type the following:

```
#: in "$redten/src/redten.gen"$
```

This will start the REDUCE compiler, load all the source files, and leave a file called `redten.b` in the current directory. The process may take several minutes to complete, and REDUCE will exit at the end of it.

For lisps like CSL, the same commands apply, but a separate compiled `redten.b` file is not produced, rather the compiled code is added to the REDUCE "image", and can be loaded on demand. You may need special privileges in order to modify the REDUCE image.

If REDUCE fails to compile the entire set of sources in one go (exiting with a "Heap space exhausted" message), then REDTEN will need to be compiled one source file at a time. This can be done as above by using the file `redten1.gen` instead of `redten.gen`. If this is done, you must also set the `redten` environment variable each time you want to run REDTEN.

To use REDTEN, first start REDUCE, then load the compiled REDTEN code. In PSL this is done with the `load()` function:

```
#: load "redten";

 "Reduce-tensor, Version v4.0, November 23, 1994"$
```

while in CSL the equivalent command is called `load!-module()`.

After REDTEN is loaded for the first time, it should be tested with some of the examples found in the `demo` directory. Simply use the REDUCE `in()` command to input any of the demo files.

E.1 Renaming Internal Functions

To avoid the possibility of collisions with the names of other functions in other packages that may be used with REDTEN in REDUCE, nearly all of the function and variable names in the REDTEN sources are in fact prepended with the string `r10!-`, so that, for example, the actual name of the lisp function executed when computing the Riemann tensor is `r10!-riemann!*`. Only the names actually required by the user are left unchanged. This manual uses the source code names, thus to access a variable described in the manual the `r10!-` should be prepended.

The sources themselves *do not* contain these names; it was considered too much trouble to rewrite everything. Instead, the REDUCE parser is modified while it loads the REDTEN source, and converts nearly all the names into the format described above. The code for this is in the file `rename.red`, which is loaded before any other files by the `redten.gen` build file. It must be emphasized that it is the *compiling* REDUCE whose parser is changed, the modifications are not part of the final REDTEN system (it also means that this file should not be loaded if the lisp system must be dumped, as opposed to creating fasl files).

Appendix F

Modifications to REDUCE

A few special modifications were made to the REDUCE source code in order to provide the hooks required for the interaction of the REDUCE algebra engine with some REDTEN constructs. In most cases the modifications consist of no more than a few added lines to a REDUCE function. The modified sources are in the files `local3{3,4,5}.red`, the filename relects the version of REDUCE for which it is useful. Other functions have been replaced by renaming them, and interposing a new function under the old name. This appendix briefly lists these modifications; for a complete description see the relevant source files.

1. A minor cosmetic modification involves changing the print character for multiplication from "*" to a blank, so that, for example, `a * b` is now displayed as `a b`. There is some justification for this, since a complicated expression can be hard to see when it is full of * symbols (and indexed expressions can be worse), but this modification also has a detrimental effect when the REDUCE switch `nat` is off. The user can either restore the old behaviour with the command

   ```
   #:  lisp put ('times, 'prtch, "*");
   ```

   or comment the relevant line out of the local source file.

2. The function `inprint()` has been modified to add the required hook to permit the printing of indexed objects.

3. The function `diffp()` has been modified to add the required hook for applying the `df()` operator to an indexed object.

4. The function `xread1()` has a modification to allow parsing of "incomplete" index-operations such as those used by `symz()` .

5. The function `prin2!*()` has a modification to permit names containing special characters to print without embedded `!` symbols.

6. The function `terpri!*()` has been modified to support the output pager code.

7. The function `begin1()` has been modified to support environments, the pager, and the `beep` switch.

8. The function `setk1()` has been redefined to support environments, and to catch attempts at algebraic assignments to indexed object names (see `save.red`).

9. The primative function `scan()` has been redefined to save the previous two tokens encountered in the input stream. This allows the system to find the un-indexed name to which an index has been attached and prompt the user for its proper creation (see `io.red`).

10. The `token()` function is redefined as described in §E.1 to implement the renaming of REDTEN functions and variables during compilation. This modification is not present in the final REDTEN system (see `rename.red`).

Appendix G

New Features

New in Version 4.0:

- Completely rewritten symmetrization code using new operators `:[` etc. Also can now do symmetrizations across a product of objects using `symz()`.
- Many internal changes.

  New in Version 3.5:

- New code to handle connections between various index-types, much more flexible than prevously.
- As of this version, the default build scripts rename nearly all internal function names and variables by prepending them with `r10!-`. Only names that are used directly by the user are unchanged. This manual still refers to the original names, as these are what actually appear in the source code.

New in Version 3.4:

- Various bug fixes.
- `mkobjsafe` switch; including objects in arguments to `mkobj()`.
-

New in Version 3.12:

- Functions `savenv()`, `restorenv()` and related functions that allow the user to save and restore named REDUCE environments. A REDUCE environment consists of all algebraic assignments, `let` and `match` rules, and all defined operators.
- A Newman-Penrose package, still under development.

New (changed) since the last wide-spread release (v2.80):

- `mkobj()` is the preferred way to create a new object, `mktnsr()` remains for backward compatibility.

- relaced use of # with _ in all names.
- function *coords()* now sets the coordinates.
- generic names, allowing the user to conveniently refer to the unweildly names required to support multiple metrics.
- `mkscalar()` routine removed, its funtionality is taken over by `mkobj()`.

Index