Ring Array Processor

Programmer's Guide to the RAP Libraries

Michael C. Greenspon
September, 1992
TR-92-060

Realization Group
International Computer Science Institute

Programmer's Guide to the RAP Libraries V1.0

Greenspon, Michael C., *Programmer's Guide to the RAP Libraries*,
ICSI Technical Report TR-92-060

Acknowledgments

The International Computer Science Institute and its sponsors are gratefully acknowledged for supporting this work.

The Ring Array Processor is brought to you by: Jim Beck on hardware; Phil Kohn, Jeff Bilmes and Michael Greenspon on software and documentation; Steve Renals and Chuck Wooters on applications; and Drs. Nelson Morgan and Joachim Beer on algorithms and architecture.

This guide is set in the Garamond font and was produced interactively using Microsoft Word version 5.0 running on Apple Macintosh IIfx and Quadra 700 workstations.

Programmer's Guide to the RAP Libraries

Figures and Tables

Introduction

Welcome to the Programmer's Guide to the RAP Libraries! We hope this guide will make developing applications for the Ring Array Processor a painless experience. To make best use of the RAP, it's helpful to understand some of the basics of data-parallel and object-oriented programming. This section introduces the necessary concepts and provides examples of the conversion between scalar C code and data-parallel code for the RAP. It also provides an overview of the software development cycle for RAP applications and the RAPMC debugger.

Data-parallel programming

The RAP class libraries support *data-parallel* (SIMD) programming on the Ring Array Processor. Data-parallel means that each processor node simultaneously executes the same set of instructions, but on different portions of a large piece of data, for example, the rows of a matrix. The data-parallel approach is well suited for numerically intensive simulation and analysis applications, including many types of artificial neural network (ANN) applications.

Because each processor runs the same or nearly the same sequence of instructions, data-parallel programming is similar to programming a single processor machine. The RAP libraries further simplify the process by providing support for *distributed memory objects* which are automatically divided across the available processors. Each piece of the object's data has the same physical address on all processor nodes, but different contents.

Other styles of parallel programming are possible using the RAP, for example, multiprocessing (MIMD) with a processor farm or pipeline. These styles may be suitable for some types of asynchronous networks or dataflow simulations. However, the RAP libraries provide only low-level support for these models, such as communications ring functions and computational inner loops. Programming for a multiprocessing system is in general considerably more complex than for a data-parallel system and the expected performance gains, if any, are application specific. The remainder of this guide refers only to data-parallel (SIMD) programming using the RAP class libraries.

Programming without loops?

One way to view data-parallel programming is as the *removal of explicit inner loops on data elements.* The RAP libraries support applications that can be written in terms of matrix and vector algebraic operations. The libraries automatically map an atomic operation like a matrix-vector multiply onto a set of iterations on each processor's local subset of data. Figures 1–1 and 1–2 illustrate how typical scalar C code for the error back-propagation training of a multilayer perceptron is simplified when converted to data-parallel code using the RAP libraries:

■ **Figure 1–1** Scalar C code for back propagation algorithm

```
/* Propagate activity forward from layer in to layer out */
void forward(float* in, int n_in, float* out, int n_out,
       float* weights, float* bias) {
  int i, j;
  float sum;

  for (i = 0; i < n_out; i++)   {  /* for each output unit */
     sum = 0.0;
     for (j = 0; j < n_in; j++)    /* sum weighted inputs */
        sum += *weights++ * in[j]; /* weights[i][j] * in[j] */
     out[i] = bias[i] + sum;       /* activation=bias+input */
     out[i] = 1.0 / (1.0 + exp(-out[i]));
                                   /* output w/sigmoid gain */
  }
}

/* Propagate error backwards from layer out to in and */
/* adjust weights and biases for layer out */
void backward(float* in, int n_in, int n_out, float* weights,
       float* out_bias, float* out_err,
       float* in_err, float learning_rate) {
  int i, j; float sum, *w;

  /* Calculate error of previous layer */
  if (in_err)                      /* only if requested */
     for (j = 0; j < n_in; j++) {  /* for each element of */
     sum = 0.0;                    /* vector-matrix product */
     w = weights + j;              /* deal with array index */
     for (i = 0; i < n_out; i++) {
        sum += out_err[i] * *w;
        w += n_in;
     }                             /* take inverse sigmoid */
     in_err[j] = in[j]*(1-in[j])*sum;
  }

  /* Train weights and biases according to error */
  /* w[i][j] += -l * out_err[i] * in[j] */
  w = weights;
  for (i = 0; i < n_out; i++) {
     out_bias[i] += -learning_rate * out_err[i];
     for (j = 0; j < n_in; j++)  /* outer product of err*in */
        *w++ += -learning_rate * out_err[i] * in[j];
  }
}
```

■ **Figure 1–2** Data-parallel C++ code for back propagation algorithm

```
// Fvec and Fmat are RAP library vector and matrix objects
void forward(Fvec* in, Fvec* out, Fmat* weights, Fvec* bias) {
  out->copy(bias);           // out = bias
  out->muladd(weights,in);   // out += w*in
  out->sigmoid(out);
}

void backward(Fvec* in, Fmat* weights, Fvec* out_bias,
     Fvec* out_err, Fvec* in_err, float learning_rate) {
  // Calculate error of previous layer
  if (in_err) {
     in_err->mul(out_err,weights);
     in_err->d_sigmoid(in_err,in);
  }
  // Train in-out weights & out bias according to error
  weights->muladd(-learning_rate,out_err,in);
  out_bias->muladd(-learning_rate,out_err);
}
```

Object-oriented programming

Central to object-oriented programming are the concepts of *class* and *object*. A *class* is a functional type description, i.e. a packaged set of data declarations (like a C `struct`) and a set of function or *method* declarations that describe the operations that can be performed on the data. These data and method declarations collectively form the *class interface*, which is a protocol or contract between the class and its clients. The term *object* refers to an *instance* of the class, i.e. a particular copy of the `struct` at a particular location in memory. Methods are invoked on objects and apply to the data of the particular object.

Class declarations are maintained in a hierarchy (a tree or directed acyclic graph) so that descendants (called *derived classes* or *subclasses*) are said to *inherit* properties (data and methods) from their parents or *superclasses*. Derived classes can selectively redefine or *override* inherited methods so that their functional behavior is largely the same as that of their parent classes but with selected differences specified by the overridden methods.

Finally, *polymorphism* is the ability to operate on derived class objects knowing only the functional interface of the superclass. For example, matrices are implemented in terms of vectors; many methods that can be applied to vectors may also be applied to matrices.

Inheritance, the selective overriding of methods and polymorphism are the key features of object-oriented programming that support the re-use of code and facilitate structured software engineering.

RAP library classes

The RAP library classes can be thought of as packages or modules that encapsulate the code and structure data needed to perform the advertised functions. RAP library classes are implemented in the C++ language, which provides compiler support for inheritance, polymorphism and function overloading. Function overloading allows methods to be uniquely identified by both name and argument type; this is used extensively in the RAP libraries. For example, a multiply of two vectors `mul(Fvec*,Fvec*)` is a different operation than a multiply of a matrix and a vector `mul(Fmat*,Fvec*)`. Though both methods are named `mul`, the compiler can determine which method to invoke by the data types of the arguments.

Programming without `case` statements?

One way to view object-oriented programming is as *the removal of `case` statements on types.* In structured C code, you might define a `struct` with a field indicating the object type, for example the layer type in a multi-layer network, and then call particular functions based on this type field using a `switch`/`case` block. Figure 1-3 illustrates this structure. However, this approach makes the straight-line code dependent on the defined object types; adding a new layer type requires adding a new `case`, perhaps to many separate parts of the application.

■ **Figure 1–3** Structured approach to multiple layer types in C

```
typedef enum { input, hidden, output } LayerType;
typedef struct {
     LayerType      type;
     int            rows;
     int            cols;
     float*         data;
     struct Layer*  inputs[];
     struct Layer*  outputs[];
} Layer;
typedef Layer* Net[];

void run_forward(Net net) {
  while (*net)           /* for each layer */
     switch ((*net)->type) {
        case input:    forward_input(*net); break;
        case hidden:   forward_hidden1(*net); break;
        case output:   forward_output(*net); break;
     }
}
```

The object-oriented approach hides type-based dispatch within the method calling mechanism. The straight-line code invokes a single method on an abstract object type; the proper implementation of the method for the particular object is selected at run time by an implicit table dispatch. As a result of these mechanisms, all of the code for a particular subclass implementation can be centralized which increases maintainability and reusability. Figure 1–4 illustrates an implementation of this approach using C++. Note that the program fragment `run_forward()` is independent of the particular types of layers:

■ **Figure 1–4** Object-oriented approach to multiple layer types in C++

```
typedef List<class Layer*> Net;    // Net is a List of Layers
class Layer {                      // Abstract Layer class
     Fmat*       data;
     Net         inputs;
     Net         outputs;
  public:
     void forward() = 0;           // Real layers override and
     void backward() = 0;          // implement these methods
};

class InputLayer : public Layer { ... };
class HiddenLayer : public Layer { ... };
class OutputLayer : public Layer { ... };

void run_forward(Net& net) {
  ListIterator i(net);
  while(i)
     net[i++]->forward();
}
```

Software development cycle for RAP applications

The RAP can be viewed either as a single-user parallel machine with a Unix host acting as a "control processor" front-end, or as an array co-processor or computational server to the host. This guide documents use of the RAP libraries from the former perspective. That is, programs are written using the RAP libraries and initially compiled for testing on the Unix host. Once satisfactory results are obtained on the host processor, the application is recompiled for direct execution on the RAP machine. These native RAP applications are loaded, debugged and controlled using the RAPMC debugging shell on the Unix host. RAPMC allows the user to load, examine, breakpoint and run RAP programs and to redirect the input and output of each RAP processor node to files or pipes.

It is also possible to use the RAP as an embedded array co-processor for applications running on the host processor. In this mode, programs are compiled for the host and RAP library calls executed in the host program are mapped into remote procedure calls (RPCs) to the appropriate routine running on the RAP. The host application can then function as a computational server with the RAP as the back-end. This RPC implementation is presently undergoing revision and is not documented in this guide. However, the RPC interface is designed so as to minimize the source code changes needed to convert native RAP library applications to use the RPC interface. The RPC implementation will be provided in a future release of the RAP libraries.

About the Programmer's Guide

This guide provides a structure and content overview of the RAP class libraries and a road map for their use in building RAP applications. It also provides examples, tips and techniques. As a guide, it is intended to point you in the right direction. Frequently, that will be to the library code itself, which is the last word on the actual interfaces and algorithms.

Other references you should have

[SUM] RAP Software Users Manual V1.0, ICSI Technical Report TR-90-049.

[SAM] RAP Software Architecture Manual, ICSI Technical Report TR-90-050.

[NRC] Numerical Recipes in C, Press, Flannery, Teukolsky & Vetterling, Cambridge University Press, 1988.

[ARM] C++ Annotated Reference Manual, Ellis & Stroustrup, Addison-Wesley 1991.

Nomenclature and typographic conventions

Tables 2–1 and 2–2 show the symbols used in formulas for computational methods in this guide:

■ **Table 2–1** Data types of symbols

Symbol	Class	Data Type
$\mathbf{x}, \mathbf{v}, \mathbf{v1}, \mathbf{v2}$	`Fvec`	Column vector of floats
$\mathbf{x}^T, \mathbf{v}^T, \mathbf{v1}^T, \mathbf{v2}^T$	`Fvec`	Row vectors of floats
$\mathbf{i}$	`Ivec`	Column vector of ints
$\mathbf{b}$	`Ivec`	Index vector of column indices
$\mathbf{M}, \mathbf{m}, \mathbf{m1}, \mathbf{m2}$	`Fmat`	Matrix of floats

■ **Table 2–2** Formula interpretations of symbols

Symbol	Class	Interpretation in formulas
x	`Fvec`	Object for which method is invoked (e.g., `this`)
v, **v1**, **v2**	`Fvec`	Input arguments to method
M	`Fmat`	Object for which method is invoked (e.g., `this`)
m, **m1**, **m2**	`Fmat`	Input arguments to method
i	`int`	Row index
j	`int`	Column index
k	`int`	Constant row or column index
s	`float`	scalar constant
F	`Table`	Function lookup table
f	`float`	Function returning a scalar; scalar function result

Conventions used in code examples

The following declarations apply to code examples show in this guide:

```
Fvec *x, *v, *v1, *v2;
Ivec *iv;
Fmat *m, *m1, *m2;
int i, j, k, n, *ints;
float f, g, *floats;
```

Overview of the Libraries

The RAP class libraries provide a convenient applications program interface to the hand-optimized array processing methods running on the RAP machine. These methods are typically executed in parallel by all of the processors on the ring (SIMD operation.) The RAP libraries provide the following classes and functions:

■ **Table 3–1** Classes and functions in the RAP class libraries

Functions provided	Implementing Classes
Matrix-vector arithmetic	`Fvec, Fmat, Ivec`
IEEE floating point format conversion	`Fvec`
Table lookup of functions	`Table`
Random number generation	`AnyRandom`

In addition to computational methods, the RAP libraries provide input-output methods and general utility methods for programming and debugging.

RAP library classes are implemented in the C++ language with bottlenecks to assembly language inner loops. We encourage use of the C++ language for code built using the libraries because the language semantics can be used to simplify the coding of large applications. However, C language interfaces for most common operations are also provided.

Distributed memory abstraction

RAP library classes are built on a software *distributed memory* abstraction implemented in the `Dmem` class. The `Dmem` class manages the allocation of storage and communication of data across the available processors for a one-dimensional array of *elements* which can be scalars or aggregates. Elements are lumped into *groups* which are distributed but never broken across processors.

The matrix-vector classes `Fvec`, `Fmat` and `Ivec` derive from `Dmem`, which enables their methods to be implemented generally, without concern for the particular hardware configuration. Thus the distributed memory abstraction considerably simplifies the parallel implementation of many of these methods and provides inherent scalability as well as a degree of portability.

Matrix and vector classes

The three core library classes `Fvec`, `Fmat` and `Ivec` implement matrix-vector arithmetic on distributed data. The classes `Fvec` and `Fmat` provide floating point representations and methods. The `Ivec` class provides a simple integer representation with few computational methods, though `Ivec` objects can be used as arguments to methods on other classes. Data managed by `Fvec`, `Fmat`

and `Ivec` objects are automatically distributed across the available processors and arithmetic methods on these objects are executed in parallel when possible.

Table lookup classes

The library class `Table` provides discrete approximations of arbitrary functions by table lookup. Where appropriate, this can offer a significant speed advantage over computing the actual function. Both single values and arrays of values may be looked up with a single method call. The `Fvec` class uses methods in `Table` to look up an entire distributed vector in parallel.

Random number classes

The library class `AnyRandom` provides floating point and integer random number generation by several popular algorithms by Knuth [NRC]. You can easily add your own algorithm of choice for your particular application within the framework of `AnyRandom`.

Standard C library functions

Most functions of the standard C libraries are supported on the RAP. For a complete listing, refer to [SUM] §14.

C stdio

Reading and writing data to host files via the C stdio facility involves some special considerations because of the RAP's parallel hardware. In general, input-output is performed sequentially through processor node 0 and data are communicated to other nodes using the ring. This is handled internally by stdio and should be transparent to most programs. In addition to the stdio facility, the RAP library classes provide tagged binary and formatted input-output methods for their objects. This is covered in more detail in Section 5.

Extending the libraries

If you find a need for a special computational operation not in the libraries, you can extend the libraries by deriving a new class from those provided. The `Fvec` and `Fmat` classes provide a framework for the data-parallel implementation of computational methods. Your derived classes can take advantage of this framework to minimize their code complexity. Also, by deriving your special operations from the library classes, you are helping to extend the libraries in a way that is potentially useful to others. We encourage sharing of library class derivatives amongst RAP users and submission of user-defined classes for redistribution via ICSI.

Why you should use the libraries

The RAP library classes provide optimized versions of most if not all of the core operations needed for implementing a wide variety of artificial neural network models as well as solutions to general computational problems. The library classes hide the numerous details of coding in parallel for multiple processors from direct view so that you can concentrate on your application with the assurance that it will take full advantage of the speed of the RAP. Because the inner loops of most library routines are implemented in hand-optimized assembly code, you are free to take advantage of the structural benefits provided by the C++ classes without sacrificing performance. Further, by implementing your C++ or C application in terms of the library methods, your application will automatically run on larger RAP machines, potentially with proportionately faster performance. And your application will be substantially easier to port to new machines as they become available.

Methods provided by the libraries

We divide methods on RAP library classes loosely into three groups: *computational methods; input-output methods;* and *programmatic methods*. Computational methods (listed in Tables 3-2 and 3-3) include the arithmetic and algorithmic operations defined for `Fvec` and `Fmat` library objects. Programmatic methods (covered in Section 5) include all code-level operations needed to create, initialize, access, copy, delete and debug library objects and their members. Input-output methods provide format conversion when reading and writing library objects via the C stdio facility.

■ **Table 3–2** Computational methods on class `Fvec`

(✓ indicates optimized methods with assembly language inner loops)

✓	Method name	Description of result	Formula
✓	`add`	sum of two vectors	$\mathbf{x}_i = \mathbf{v1}_i + \mathbf{v2}_i$
✓	`add`	sum of each vector element and scalar	$\mathbf{x}_i = \mathbf{x}_i + s$
	`add`	sum of one vector element and scalar	$\mathbf{x}_i \mathrel{+}= s$
✓	`sub`	difference of two vectors	$\mathbf{x}_i = \mathbf{v1}_i - \mathbf{v2}_i$
✓	`sub`	difference of each vector element and scalar	$\mathbf{x}_i = \mathbf{x}_i - s$
✓	`mul`	dot product of two vectors	$f = \sum_i \mathbf{v1}_i \cdot \mathbf{v2}_i$
✓	`mul`	matrix-vector product	$\mathbf{x}_i = \sum_j \mathbf{m}_{ij} \cdot \mathbf{v}_j^T$
✓	`mul`	vector-matrix product	$\mathbf{x}_j^T = \sum_i \mathbf{v}_i \cdot \mathbf{m}_{ij}$
✓	`mul`	scaled vector	$\mathbf{x}_i = s \cdot \mathbf{v}_i$
✓	`mul_ele`	element-wise product of two vectors	$\mathbf{x}_i = \mathbf{v1}_i \cdot \mathbf{v2}_i$
✓	`mul_ele`	scaled element-wise product	$\mathbf{x}_i = s \cdot \mathbf{v1}_i \cdot \mathbf{v2}_i$
✓	`muladd`	accumulated matrix-vector product	$\mathbf{x}_i \mathrel{+}= \sum_j \mathbf{m}_{ij} \cdot \mathbf{v}_j^T$
✓	`muladd`	accumulated matrix-vector product of float matrix and sparse binary vector	$\mathbf{x}_i \mathrel{+}= \sum_j \mathbf{m}_{i\mathbf{b}_j}$
✓	`muladd`	accumulated vector-matrix product	$\mathbf{x}_j^T \mathrel{+}= \sum_i \mathbf{v}_i \cdot \mathbf{m}_{ij}$
✓	`muladd`	accumulated scaled vector	$\mathbf{x}_i \mathrel{+}= s \cdot \mathbf{v}_i$
✓	`muladd_ele`	accumulated element-wise product	$\mathbf{x}_i \mathrel{+}= \mathbf{v1}_i \cdot \mathbf{v2}_i$
	`muladd_ele`	accumulated scaled element-wise product	$\mathbf{x}_i \mathrel{+}= s \cdot \mathbf{v1}_i \cdot \mathbf{v2}_i$
	`recip`	element-wise reciprocal	$\mathbf{x}_i = 1 / \mathbf{v}_i \quad \} \mathbf{v}_i \neq 0$
✓	`sum`	scalar sum of vector elements	$f = \sum_i \mathbf{x}_i$
✓	`sum_row`	sum of matrix rows	$\mathbf{x}_j^T = \sum_i \mathbf{m}_{ij}$
✓	`l2`	norm of difference of two vectors	$f = \sum_i (\mathbf{v1}_i - \mathbf{v2}_i)^2$
✓	`wl2`	weighted norm of difference of two vectors	$f = \sum_i \mathbf{v}_i \cdot (\mathbf{v1}_i - \mathbf{v2}_i)^2$
✓	`l2_row`	norms of differences of corresponding row vectors	$\mathbf{x}_i = \sum_j (\mathbf{m1}_{ij} - \mathbf{m2}_{ij})^2$
✓	`max_ele`	maximum element of vector	$f = \max_i \mathbf{x}_i$
✓	`min_ele`	minimum element of vector	$f = \min_i \mathbf{x}_i$

■ **Table 3–2** Computational methods on class `Fvec` (continued)

✓	Method name	Description of result	Formula
	`apply`	scalar function applied to each vector element	$\mathbf{x}_i = \mathrm{f}(\mathbf{x}_i)$
	`softmax`	soft maximum of each vector element	$\mathbf{x}_i = e^{\mathbf{v}_i} / \sum_j e^{\mathbf{v}_j}$
✓	`d_sigmoid`	inverse of sigmoid (error backprop function)	$\mathbf{x}_i = \mathbf{v1}_i \cdot \mathbf{v2}_i \cdot (1 - \mathbf{v2}_i)$
✓	`lookup`	scalar table lookup applied to each vector element	$\mathbf{x}_i = \mathrm{F}(\mathbf{v}_i)$
✓	`sigmoid`	sigmoid table applied to each vector element	$\mathbf{x}_i \cong (1 + e^{-\mathbf{v}_i})^{-1}$
✓	`exponential`	exponential table applied to each vector element	$\mathbf{x}_i \cong e^{\mathrm{S}\mathbf{v}_i}$
✓	`log`	log table applied to each vector element	$\mathbf{x}_i \cong \log \mathbf{x}_i$
✓	`tanh`	tanh table applied to each vector element	$\mathbf{x}_i \cong \tanh \mathbf{x}_i$

■ **Table 3–3** Computational methods on class `Fmat`

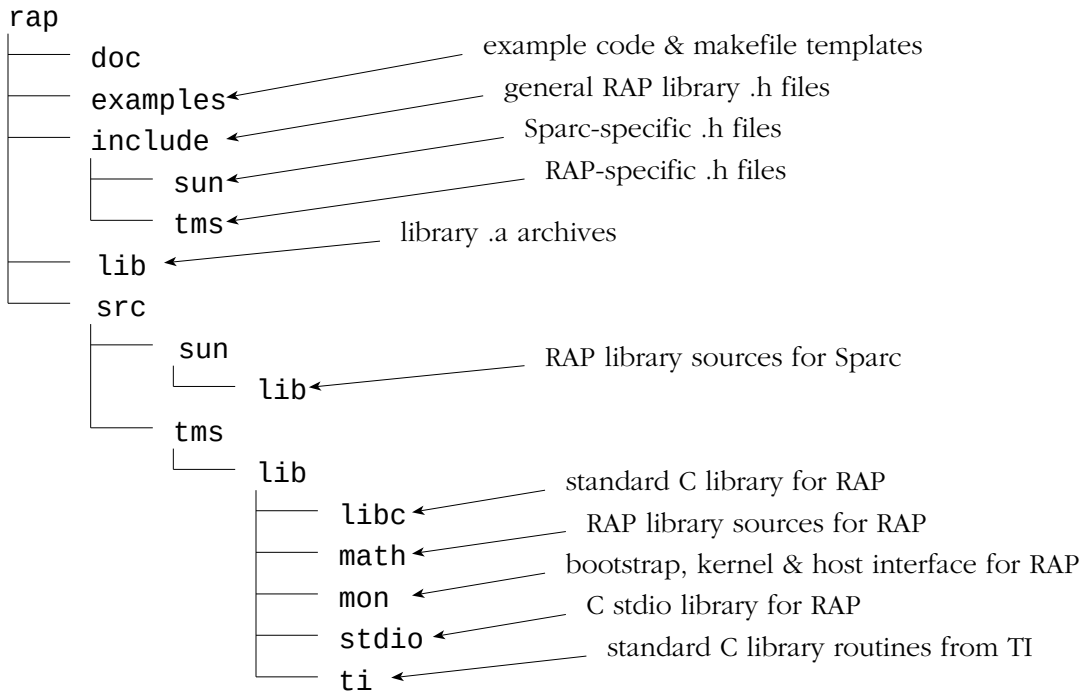
(✓ indicates optimized methods with assembly language inner loops)

✓	Method name	Description of result	Formula
✓	`add`	sum of one matrix element and scalar	$\mathbf{M}_{ij}$ += S
	`add_col`	sum of matrix column and column vector	$\mathbf{M}_{ij}$ += $\mathbf{v}_i$
✓	`sub`	difference of two matrices	$\mathbf{M}_{ij}$ −= $\mathbf{m}_{ij}$
	`sub`	matrix difference of two vectors	$\mathbf{M}_{ij} = \mathbf{v1}_j^{\mathrm{T}} - \mathbf{v2}_i$
✓	`sub_row`	difference of row vector and each matrix row	$\mathbf{M}_{ij} = \mathbf{v}_j^{\mathrm{T}} - \mathbf{m}_{ij}$
✓	`mul`	scaled matrix	$\mathbf{M}_{ij} = \mathrm{S} \cdot \mathbf{M}_{ij}$
✓	`mul`	scaled matrix product of two vectors (outer product)	$\mathbf{M}_{ij} = \mathrm{S} \cdot \mathbf{v1}_j^{\mathrm{T}} \cdot \mathbf{v2}_i$
✓	`mul_row`	row-wise product of matrix and vector	$\mathbf{M}_{ij} = \mathbf{m}_{ij} \cdot \mathbf{v}_i$
✓	`muladd`	accumulated scaled outer product	$\mathbf{M}_{ij}$ += $\mathrm{S} \cdot \mathbf{v1}_j^{\mathrm{T}} \cdot \mathbf{v2}_i$
✓	`muladd`	accumulated scaled outer product of float and sparse binary vectors	$\mathbf{M}_{i\mathbf{b}_j}$ += $\mathrm{S} \cdot \mathbf{v}_i$
	`muladd_row`	accumulated scaled row-wise product	$\mathbf{M}_{ij}$ += $\mathrm{S} \cdot \mathbf{m}_{ij} \cdot \mathbf{v}_i$
	`transpose`	transpose of matrix	$\mathbf{M} = \mathbf{m}^{\mathrm{T}}$
	`inverse`	inverse of matrix by LU decomposition; returns determinant	$\mathbf{M} = \mathbf{m}^{-1}$, $\mathrm{f} = \det \mathbf{m}$
✓	`closest_row`	minimum of norms of differences between given row and all other rows	$\mathrm{f} = \min_i \sum_j (\mathbf{m1}_{kj} - \mathbf{m2}_{ij})^2$

Source code organization

Figure 3–1 shows the directory tree organization for the files that make up the RAP libraries. Table 3–4 describes the contents of the source files that implement the RAP library classes documented in this guide.

■ **Figure 3–1** RAP library source code organization



■ **Table 3–4** RAP library source files

File name	Description of contents
`rap.h`	Single global include file for all RAP library methods
`matvec.h`	Matrix-vector & distributed memory class declarations
`c_matvec.h`	C interface to RAP library declarations
`mat3.cc`	Matrix-vector & distributed memory class implementations
`mat4_common.cc`	C interface to RAP library methods
`random.h, .cc`	Random number class declarations & implementation
`table.h, .cc`	Table lookup class declarations & implementation
`io.h, .cc`	Tagged binary and formatted ASCII I/O methods used by `Fvec`
`raplib.h`	General declarations and low-level computational method declarations
`mat2.cc`	Matrix-vector low-level computational methods and inner loops
`ring.h, .cc`	Communications ring function declarations & implementation
`convert.cc`	IEEE to TMS floating point conversion routines (for host only)

Using the Libraries

RAP library objects and methods can be used from both the C++ and C languages interchangeably. All library declarations are included from the single root file `rap.h`, located in `rap/include`.

Including <rap.h>

To declare the RAP library classes, methods and interfaces, be sure to include `rap.h` from your `.cc` or `.c` source file:

```
#include <rap.h>
```

Conventions used in code examples

In the C and C++ code examples of the sections that follow, these declarations apply:

```
Ivec* iv;
Fvec *vec, *v, *v1, *v2;
Fmat *mat, *m, *m1, *m2;
Table *table;
int *ints, i, j, k, n;
float *floats, f, g;
```

C++ language interface conventions

The RAP library classes were implemented with consistency and code legibility in mind. The following common-sense conventions generally apply when invoking methods on RAP library objects using the C++ language:

- Computational methods that produce matrix or vector results are invoked on the object that will hold the result:

```
vec->mul(m,v);              // vec = m * vt
mat->sub(m);                // mat -= m
vec->copy(v);               // vec = v
```

- Since `Fmat` is derived from `Fvec`, methods defined on vectors can often be sensibly applied to matrices as well:

```
vec->mul_ele(v1,v2);        // vec[i] = v1[i]*v2[i]
mat->mul_ele(m1,m2);        // mat[ij] = m1[ij]*m2[ij]
f = mat->sum();             // f = sum(mat[ij])
m2->copy(m1);               // m2[i2j2] = m1[i1j1]
```

- Default arguments are defined for many methods where appropriate:

```
vec->get(floats);          // vec[0..len] = floats[0..len]
vec->get(floats,n);        // vec[0..n] = floats[0..n]
vec->get(floats,n,5);      // vec[5..n] = floats[5..n]
```

- Objects are passed as pointers (e.g., `Fvec*`) rather than as references (e.g., `Fvec&`.) This is more compatible with dynamic allocation of computational objects. A reference-based interface can be derived from the library classes to enable operator notation for matrix and vector objects and to facilitate use of library objects as members of other structures. See `rap/examples` for an example of such subclasses.

C language interface

RAP library objects can be created and operated on using a set of C language interfaces. Most, but not all, library methods have a C language interface defined. These interface routines simply invoke the appropriate C++ methods on the objects. Thus the C language interface adds the overhead of an additional subroutine call for each method invocation. Typically this overhead is negligible but it may be significant for short (e.g., single-element) operations executed in a loop. Some of this overhead can be avoided by invoking the C++ methods directly.

C language calling conventions

The following common-sense conventions apply when invoking methods on RAP library objects using the C language:

- The C language interfaces are named according to the general form "`op_xx_y`" where "`op`" is the operation, "`x`" indicates the data type of one or more source objects and "`y`" optionally indicates the data type of the destination or resultant object. Scalar function results and parameters are implicit with respect to naming. The data type indicators are listed in Table 4–1. Objects are passed by reference with arguments in the order indicated in the name:

```
mul_MV_V(m,v,vec);         /* vec = m * vt */
copy_V_V(v1,v2);           /* v2[i] = v1[i] */
f = sum_V(vec);            /* f = sum(v[i]) */
```

- The C language does not support default arguments for function declarations. Thus all parameters to the method must be specified:

```
get_V(v,0,floats,size_V(v));
                      /* vec[0..len] = floats[0..len] */
```

■ **Table 4–1** Data type indicators for C language interfaces

Indicator	Data Type	Description
S	`float`	scalar float
V	`Fvec*`	Reference to vector of floats
I	`Ivec*`	Reference to vector of ints
B	`Ivec*`	Reference to sparse binary vector
M	`Fmat*`	Reference to matrix of floats
T	`Table*`	Reference to function lookup table

■ The C language does not support polymorphism. Thus `Fmat*` must be explicitly cast to `Fvec*` in order to use vector methods applicable to matrices when there is no explicit interface for `Fmat`:

```
mul_ele_VV_V(v1,v2,vec);   /* vec[i] = v1[i]*v2[i] */
mul_ele_VV_V((Fvec*)m1,(Fvec*)m2,(Fvec*)mat);
                           /* mat[ij] = m1[ij]*m2[ij] */
f = sum_V((Fvec*)mat);     /* f = sum(mat[ij]) */
copy_M_M(m1,m2);           /* explicit interface provided */
```

Working with Matrices and Vectors

This section covers the basic programmatic methods you'll need to create, delete, initialize, set, copy, read, write and access matrix and vector objects.

Creating objects

In C++, matrix and vector objects may be created dynamically on the heap using the `new` operator or automatically on the stack by declaration. (Note that the space for the object's data, i.e. the contents of the matrix or vector, as opposed to the object that describes it, is always allocated on the heap.) The optional parameter `memtype` specifies the desired memory partition for the object's data. Memory partitions and optimization for the RAP are discussed in Section 10. The default value for `memtype` is `FASTEST` which allocates the object's data in the fastest memory partition that has enough free space:

```
Ivec local_inputs(100,FASTEST);  // a new 100 element vector of ints
Fvec* v = new Fvec(10000);       // a 10000 element vector of floats
Fmat* m = new Fmat(200,100);     // a 200 row x 100 column matrix
```

In C, matrix and vector objects must be allocated on the heap, and the `memtype` parameter must be passed explicitly:

```
Ivec* iv = new_Ivec(100,FASTEST); /* Must specify memtype */
Fvec* v = new_Fvec(10000,FASTEST);
Fmat* m = new_Fmat(200,100,FASTEST);
```

Deleting objects

When you're finished with an object you can destroy it and free the memory allocated for its data with the `delete` operator:

```
delete iv; delete v; delete m;                      // C++ flavor
delete_Ivec(iv); delete_Fvec(v); delete_Fmat(m);    /* C flavor */
```

Putting and getting data

Data are moved into and out of matrix and vector objects using the methods `put` and `get`. The method `put` copies a data element or a linear array of data elements from local memory into the object's distributed storage. The symmetrical method `get` does the opposite, retrieving data from a distributed object into local memory. Typically these methods are used in conjunction with reading or writing data from files.

```
vec->put(floats);        // put local data into entire Fvec
vec->get(floats);        // get it back; floats must be big enough!
vec->put(floats,100);    // get only 100 elements starting at vec[0]
vec->put(floats,100,5);  // get only 100 elements starting at vec[5]
f = vec->get(5);         // f = vec[5]
vec->put(5,f);           // vec[5] = f    (these also work on Ivecs)

mat->put(floats);        // put local data in row-major order
f = mat->get(10,20);     // f = mat[10][20]
m->get(3,3,floats,5);    // get 5 elements starting at m[3][3]
```

For matrices, variants of `put` and `get` also move data between individual matrix rows and columns and either local memory or distributed vectors:

```
m->get_col(0,floats);    // get column m[*][0] into local memory
m->get_col(0,vec);       // get column m[*][0] into an Fvec
m->put_row(5,floats);    // put row m[5][*] from local memory
m->put_row(5,vec);       // put row m[5][*] from an Fvec
```

The C interface lacks overloading and default arguments. The method `size` is used to determine the number of matrix or vector elements:

```
put_I(iv,0,ints,size_I(iv));     /* put the whole Ivec */
put1_V(v,5,f);                   /* v[5] = f */
put_M(m,3,3,floats,100);   /* put 100 elements starting at m[3][3] */
put_row_M(m,3,floats,1);         /* put 1 row into m[3][*] */
put_row_MV(m,3,vec,1);           /* put Fvec into 1 row m[3][*] */
get_col_M(m,7,floats,3);         /* get 3 columns m[7..9][*] */
f = get1_M(m,10,20);             /* f = m[10][20] */
```

Converting IEEE floating point data

Single precision floating point data are normally stored in 32-bit IEEE standard format. However, the RAP uses a different binary format which must be converted to and from IEEE format when data are written or read from the host. The tagged binary and formatted input-output methods on `Fvec` and `Fmat` take care of the conversion automatically; this is covered below. If you read or write binary data directly with stdio (bypassing the `Fvec` and `Fmat` methods) you must perform the conversion explicitly. Note that the IEEE data are treated as type `unsigned int` by the RAP:

```
m->put_binary(0,ints,m->size());       // convert IEEE to Fmat
v->get_binary(10,ints,v->size()-10);   // convert v[10..n] to IEEE
```

In C:

```
put_ieee_V((Fvec*)m,0,ints,size_V((Fvec*)m));      /* C flavor */
get_ieee_V(v,10,ints,size_V(v)-10);
```

Converting ints to floats

The `cast` method converts the integer elements of an `Ivec` into floats and copies them into an `Fvec` of the same size:

```
vec->cast(iv);                  // vec[i] = (float) iv[i]

cast_I_V(iv,vec);               /* C */
```

Copying data

The `copy` method duplicates the data elements from one matrix or vector into another of identical size:

```
v2->copy(v1);                   // v2 = v1
m2->copy(m1);                   // m2 = m1

copy_I_I(iv1, iv2);             /* iv2 = iv1 */
copy_V_V(v1,v2);                /* v2 = v1  */
copy_M_M(m1,m2);                /* m2 = m1  */
```

Initializing objects

The `set` method initializes all elements of a matrix or vector to a constant scalar value:

```
v->set(0.0); m->set(1.0); iv->set(-1);            // C++

set_S_V(0.0,v); set_S_M(1.0,m); set_S_I(-1,iv);   /* C */
```

Accessing members

The methods `size`, `n_rows`, and `n_cols` are accessors used to determine the size of vectors (number of elements) and matrices:

```
int num_ele = vec->size();       // number of elements
int s = mat->size();             // same as n_rows*n_cols for Fmats
int nc = mat->n_cols();          // number of columns
for (int i=mat->n_rows(); i--;)  // loop backwards over rows of mat


int s = size_V(vec); s = size_I(iv);   /* C flavors */
int nr = n_rows_M(mat);
int nc = n_cols_M(mat);
s = size_V((Fvec*)mat);                /* same as nr*nc */
```

Computational methods

The available computational methods on matrix and vector class objects are given in Tables 3–2 and 3–3. Typically several overloaded versions of each method may be defined. For example, the `Fvec` method to add two vectors is defined both as `add(Fvec* v1, Fvec* v2)` and as `add(Fvec* v)`. The first definition adds the vectors `v1` and `v2` and places the result in `this`. The second adds the vector `v` to `this`. In fact, the second can be defined in terms of the first as simply `add(v,this)`. The corresponding C interfaces use the methods `add_VV_V` and `add_V_V`. Methods for the other available computational operations are defined similarly.

The complete C++ interface declarations can be found in `matvec.h` and the C interface declarations in `c_matvec.h`, both in `rap/include`. When coding, it's best to locate the method you need according to its formula by using Tables 3–2 and 3–3. Then refer to the appropriate `.h` file for the actual interface.

Sparse binary vectors

Limited support for sparse binary vector operations is provided by the RAP libraries. The libraries provide two methods `Fvec::muladd(Fmat*,Ivec*)` and `Fmat::muladd(float,Fvec*,IVec*)` (accumulated matrix-vector product and accumulated scaled outer product.) These operations correspond to the computationally intensive parts of the forward activity and backward error propagation steps and may be of use in certain artificial neural network applications that use binary inputs and unit activations.

Sparse binary vectors are `Ivec` objects that contain a list of the indices of the non-zero elements. For example, the binary vector [0 1 0 0 0 1 0] would be represented in an `Ivec` of two elements [1 5] corresponding to the elements of the original vector with the value 1. The creation and maintenance of these index vectors is up to the application.

Reading and writing objects

The method `print` is used to output a simple formatted ASCII representation of floating point matrix and vector objects to a host stdio stream. By default, matrices are written one row per line and vectors ten elements per line. The output is optionally preceded by a title string. Output is to `stdout` by default:

```
FILE* f = fopen("filename","w"); // open stream for writing
vec->print("my vector=");        // print to stdout, default format
m->print("weights\n","%4.1f\t"); // Custom format for floats
v->print(f,NULL,"%3.2f ");       // print v to file f, no title

print_V(stdout,"v\n","%f ");     /* Must specify all args in C */
print_M(f,"weights\n","%4.1f\t");
```

Tagged files

The method `io` is used to read and write matrix and vector objects to tagged host files. Both binary and formatted ASCII modes are supported and the data transfer is performed on a stdio stream. The tag format is an identifier padded to 4 characters ("`vec `" for matrices and vectors) followed by the data. For matrices and vectors the data begins with an integer element count. The mode can be either "`r`" to read data from the stream or "`w`" to write data, optionally followed with "`b`" to indicate binary mode. Note that the mode for `io` should be the same as the `fopen` mode:

```
FILE* f;                    // file must be fopen'd in the right mode
vec->io(f,"wb");            // write entire vector in binary to f
v->io(f,"rb");              // read entire vector in binary from f
mat->io(f,"w");             // write entire mat to f using fprintf
m->io(f,"w","%4.2f\n");     // write entire mat with custom format
v->io(f,"r");               // read entire mat from f using fscanf

io_V(vec,f,"w","%f\n");     /* C version */
io_I(iv,f,"r","%d");        /* Read entire integer vector */
io_M(m,f,"wb",0);           /* No format spec for binary mode */
```

Changing input-output formats

Note that when using the method `io` in formatted ASCII mode, elements are read and written one at a time using the default format `"%f\n"` for floats and `"%d\n"` for integers. You can use a custom format string (for example, fixed point or exponential.)

The method `io` calls the method `io_format_ele` for each element when writing and `io_scan_ele` for each element when reading. These methods are passed the format string as an argument along with the file and data element. The default definition of `io_format_ele` simply calls `fprintf` and `io_scan_ele` calls `fscanf`. These methods can be overridden in classes you derive to provide custom formats.

For binary mode, elements are written in blocks which tends to be much more efficient than sequential ASCII formatting. Floating point data are converted to IEEE format when writing to host files and converted back to RAP format when reading on the RAP.

Debugging distributed objects

C++ programs can call the methods `dump` and `info` on matrix and vector objects to produce a formatted display of the object's contents and fields related to allocation, which may be of use in program debugging. The output is written to `stdout`:

```
vec->info("suspect vector");     // Write info on vec to stdout
mat->dump("weights");            // Dump the whole weights matrix
```

Using C stdio

In addition to using the `Fvec` methods `print` and `io`, you can also read and write data to host streams directly using the C stdio facility. Note that the RAP is a word-addressed and not byte-addressed machine so characters are normally stored one per 32-bit word. The C stdio routines are by default character-oriented and thus each character read from the host will be stored in a separate word on the RAP.

To specify word-oriented instead of character-oriented data transfers, the `fopen` routine accepts the additional mode option "`b`" to indicate binary mode. That is, in binary mode, each word of the host file corresponds to one word of RAP memory. Note that floating point data are *not* automatically converted to and from IEEE format when using stdio routines directly:

```
FILE* f;
Fmat* m = new_Fmat(100,100,FASTEST);
float* buf = MALLOC(float,100*100,FASTEST,"i/o buf"); // [SUM] §6.2.2
fopen("filename","rb");                 // open for binary read
fread(buf,sizeof(float),100*100,f);     // read IEEE floats from host
put_ieee_V((Fvec*)m,0,buf,size_V((Fvec*)m));
                                        // convert & put into Fmat
```

Parallel use of stdio

Reading and writing data to host files via the C stdio facility involves some special considerations because of the RAP's parallel hardware. In general, input-output is performed sequentially through processor node 0 and data are communicated to other nodes using the ring. This is now handled internally by stdio and should be transparent to most programs. For example, though all nodes execute the `fopen` routine, only node 0 actually does the kernel call to open the host file; the other nodes get the file descriptor from node 0.

Similarly, for `fwrite`, data in the stream buffer are assumed to be the same on all nodes and only node 0 actually makes the `write` call to transfer data. The exception to this is the stream `stderr` which is normally written from all processor nodes so that error aborts in a particular node can be detected.

Note the stream `stdout` is by default written only from node 0. This can be overridden by setting the global `STDOUT_FROM_NODE0_ONLY` to 0, which enables output to `stdout` from all nodes. Note that RAP client processes on the host like RAPMC may redirect or ignore `stdout` from selected nodes.

Normally for `fread`, only node 0 reads data from the stream, and then broadcasts the data to all other nodes using the ring. An additional mode option "`m`" can be specified with `fopen` to enable parallel (MIMD) mode file reading. When a stream is opened using the mode option "`m`", the `fread` call results in a host `read` request for each processor node. This may be desirable for certain types of host streams. For more information, refer to [SUM] §6.2.3.

Working with Function Lookup Tables

The library class `Table` provides discrete approximations of arbitrary functions of a single variable by table lookup. Where appropriate, this can offer a significant speed advantage over computing the actual function.

Standard function lookup tables

The RAP libraries provide several standard predefined table lookup classes derived from `Table` for functions that are commonly used in artificial neural network applications. These include sigmoid, exponential, logarithmic and hyperbolic tangent functions. For convenience, the libraries provide a global instance of each of these commonly used tables which you must explicitly initialize. The `Fvec` methods `sigmoid`, `exponential`, `log`, and `tanh` use these global tables.

You can define your own function lookup tables for arbitrary functions of a single variable by deriving a C++ class from `Table` or by passing a scalar function to the method `setup_table_T`.

Initializing the standard tables

Before using the standard function lookup tables or any `Fvec` methods that use these tables, you must explicitly initialize each table by calling the appropriate `domain_init` method. This allocates space for the table and indicates the domain over which the function is to be evaluated in creating the table, as well as the step size between table entries (i.e. the input resolution.) Input values outside the domain of the table are pinned to the endpoints of the table. These initialization methods for the standard global tables can be called from either C++ or C:

```
int size = 1000;                 // Number of table entries
float start = -5.0;              // Minimum input value
float end = 5.0;                 // Maximum input value
float step = (end-start) / (float)size; // Input resolution
float k = 2.0;                   // Exponential scale factor
float eps = 1.0E-6;              // A small value

sigmoid_domain_init(size, start, step);      // domain is [-5,5]
exponential_domain_init(size,start,step,k); // f=exp(k*x)
tanh_domain_init(size,start,step);
log_domain_init(size,eps,10.0/(float)size);  // domain is (0,10]
```

Using the standard tables

Once the global tables have been initialized with one or more of the above calls, you can perform table lookups on matrices and vectors:

```
vec->sigmoid(v);                 // vec[i] = sigmoid(v[i])
```

```
mat->log(m);                    // mat[ij] = log(mat[ij])
```

In C:

```
exponential_V_V(v,vec);         /* vec[i] = exp(k*v[i]) */
tanh_V_V((Fvec*)m,(Fvec*)mat);  /* mat[ij] = tanh(mat[ij]) */
```

You can also look up single scalar values if necessary using the following methods from either C++ or C:

```
f = log_s(g); f = tanh_s(g);    /* table lookup versions */
f = sigmoid_s(g); f = exponential_s(g);
```

Creating custom function lookup tables

You can create your own custom function lookup tables for arbitrary functions that you define. In C++, this is best done by deriving a class from the library class `Table` and overriding the method `float func(float)` to define the function represented by the lookup table. You can pass any additional parameters that you need to your derived class constructor. Your constructor calls the inherited method `Table::setup` to create the data for the table:

```
class HatTable : public Table { // Define a table for hat function
   public:
      HatTable (int size, float start, float step, float f) {
         w = f;                 // Save the freq we're given
         setup(size,start,step); // Call inherited:: to setup table
      }
      float func(float x) {     // This defines the table's function
         return x * sin(w*x);   // in this case, a hat function
      }

   private:
      float w;    // Any other params you keep for your func
};
```

That's all there is to defining your custom table. To use create the custom table in your code, just use the `new` operator:

```
int size = 2000;                // Number of table entries
float start = -2*PI;            // Minimum input value
float end = 2*PI;               // Maximum input value
float step = (end-start) / (float)size; // Input resolution

HatTable* hat_table = new HatTable(size,start,step,1.0);
```

Creating custom function lookup tables in C

If you're using the C language you can still define a custom function lookup table by using the methods `new_Table` and `setup_table_T`. You define a scalar function of a single variable and pass a pointer to it to set up the table. Additional parameters to your table function must be handled in file or global scope:

```
static float hat_freq;           /* Declare func & its params  */
float HatFunc(float x) { return x * sin(hat_freq*x); }

Table* new_HatTable(int size, float start, float step, float f) {
   Table* t = new_Table();       /* Create an empty table object */
   hat_freq = f;                 /* Save function parameters */
   if (t)
      setup_table_T(t,size,start,step,HatFunc);
                                 /* Initialize the table like this */
   return t;                     /* Return the new table */
}

Table* hat_table = new_HatTable(size,start,step,1.0);
```

Using custom function lookup tables

Once you've created and initialized your custom table from either C++ or C, you can use it to perform table lookups on matrices, vectors and scalars:

```
vec->lookup(hat_table,v);        // vec[i] = hat_table[v[i]]
mat->lookup(hat_table,m);        // mat[ij] = hat_table[m[ij]]
f = hat_table->lookup(g);        // f = hat_table[g]
```

In C:

```
lookup_V_V(hat_table,v,vec);     /* vec[i] = hat_table[v[i]] */
lookup_V_V(hat_table,(Fvec*)m,(Fvec*)mat); /* use Fmat as an Fvec */
f = lookup_S(hat_table,g);       /* f = hat_table[g] */
```

Generating Random Numbers

The library class `AnyRandom` provides uniform spectrum floating point and integer random number generation by several popular algorithms of Knuth [NRC]. You can easily add your own algorithm of choice for your particular application within the C++ framework of `AnyRandom`.

Standard random generator

A default random number generator is defined at compile time in `random.cc`. As shipped, this is the `ran3` generator from Numerical Recipes in C [NRC].

Before using the standard generator, you must first initialize it by calling `seed` from either C or C++. This global method seeds each processor with a different value (i.e. it calls the method `distributed_seed`.)

```
seed(13);                  // initialize standard random generator
seed(time(nil) & 127);     // different seed every time
```

Once the standard generator has been seeded, the `Fvec` method `random` fills a distributed vector with random elements in a specified floating point range:

```
vec->random(-10.0,10.0);          // vec[i] = random[-10..10]
random_V(0.0,1.0,vec);            /* C flavor */
```

The `Fvec` method `irandom` fills a distributed floating point vector with integer-valued random elements in a specified integer range:

```
vec->irandom(-10,10);             // vec[i] = irandom[-10..10]
irandom_V(-10,10,vec);            /* C flavor */
```

Using custom random generators

Two classes of generators are provided by the libraries: `Random1` and `Random3` which implement, respectively, the algorithms `ran1` and `ran3` from [NRC]. These classes are derivatives of the abstract base `AnyRandom`. Each object holds its own state information and you can have multiple generators if you like:

```
AnyRandom* KnuthRandom1 = new Random1();
AnyRandom* KnuthRandom3 = new Random3();
```

You must seed the generator using either of the methods `replicated_seed` or `distributed_seed`. The method `replicated_seed` gives the same value to each processing node. The method `distributed_seed` gives each node a different value; this is normally what you want in order to produce spectrally uniform randoms over a distributed vector or matrix. Note that each generator may be seeded differently if you wish:

```
KnuthRandom1->distributed_seed(3);     // normal, each node different
KnuthRandom3->replicated_seed(3);      // each node produces the same
```

Any generator can be used to fill a vector or matrix with random elements:

```
vec->random(0.0,1.0,KnuthRandom1);
mat->random(-10.0,10.0,KnuthRandom3);
```

Defining custom random generators

Using C++, you can implement your own random number generation algorithm of choice by deriving a class from `AnyRandom`. You must override the method `uniform` to provide a spectrally uniform sequence of floating point random numbers in the range [0.0,1.0]. You can keep any state information you need as private members. You can also override the method `seed(int)` to provide an integer seed for your algorithm:

```
class MyRandom : public AnyRandom {
   public:
            MyRandom() { seed(0); }
      void seed(int);            // if you need to be seeded
      float uniform();           // your generator goes here

   private:
//    int state[16];             // you could keep state info here
};

void MyRandom::seed(int) { }
float MyRandom::uniform() { return 0; } // your generator goes here
```

Once defined, you can use your generator class with matrices and vectors:

```
AnyRandom* myRandom = new MyRandom();
myRandom->distributed_seed(time(nil) & 1023);

vec->random(0.0,10.0,myRandom);
```

Building Your Application

Programs written using the RAP libraries can be easily compiled for either the host machine or the RAP itself. Before running your application on the RAP machine, it's a good idea to build and test it on the Unix host. Once you've verified that your application is producing the desired results on the host machine, you can recompile it for the RAP and run it under the RAPMC debugging shell.

Building RAP applications for the Unix host

If you've written your application using the RAP libraries, you can build it on the host by linking with the library `librapsun.a`. You can use the following template `makefile`, found in `rap/examples/sun`:

```
# sample makefile for building RAP applications on Unix host
OBJECTS = myapp.o # your other objects go here
LIBS = -lrapsun      # include librapsun.a for building on host
MORELIBS = # other libraries you need go here, eg. -lm
CPLUS = CC

# root directory for rap software - change for your installation
RAP_ROOT = /usr/local/rap
CC_LIB_DIR = /usr/local/lib

# subdirectories of rap software root directory
RAP_LIB = $(RAP_ROOT)/lib/sun
RAP_INCLUDE_DIR = $(RAP_ROOT)/include/sun
CC_INCLUDE_DIR = /usr/CC/incl

myapp:         $(OBJECTS) $(RAP_LIB)/librapsun.a makefile
               $(CPLUS) $(OBJECTS) $(LDFLAGS) -L$(CC_LIB_DIR) \
                     -L$(RAP_LIB) $(LIBS) $(MORELIBS) \
                     -o myapp

# generic c++ compile
.cc.o:
        $(CPLUS) $(CC_FLAGS) -I$(RAP_INCLUDE_DIR) \
               -I$(CC_INCLUDE_DIR) -c  $<

# generic c compile
.c.o:
        $(CC) $(C_FLAGS) -I$(RAP_INCLUDE_DIR) -c $<
```

Unix host environment

Note that programs built for the Unix host have the symbols `N_NODE` set to 1 and `NODE_ID` set to 0. Your program must be able to run on a single processing node in order for it to work properly on the Unix host. All RAP library routines will run on a single processing node. For more information on the use of `N_NODE` and `NODE_ID`, see Section 11.

Building applications for the RAP

So you've debugged your program by compiling it on the host, and now you're ready to build it for execution on the RAP! A good place to start is with the sample `makefile` in `rap/examples/tms`. Edit this file to specify your program's components.

You'll also want to copy the `link.cmd` file in `rap/examples/tms` and edit it for your program's components. This file tells the RAP linker how allocate your program's object code in memory. Initially you can follow the comments in this file and not worry too much about placement of your objects. Optimizing performance by selective code and data placement is covered in the next section.

(You may notice that the make procedure is a little more complex than normal due to several aspects of the cross-compilation from host to RAP. For example, the RAP C compiler limits identifiers to 32 characters in length. This is fine for some programs, but C++ is fond of generating long identifiers for type-safe linkage. Thus the C output of your C++ compilation is run through a "name hack" to generate unique (and meaningless) 32 character identifiers for any externals longer than the C compiler's limit. The make procedure creates a directory called `.hack` which stores the intermediate results of this process if you need to debug the translated C++ code. Note that only the hacked names appear in the link map.)

Running your RAP application with RAPMC

Once your application is successfully linked, you can execute it on the RAP using the RAPMC debugging shell. For example, the following commands will execute your application with RAPMC (your typing is in *italics*:)

```
# rapmc                          # invoke RAPMC from your shell
RAP Master Commander...          # RAPMC prints its banner
0> node *                        # command all nodes
0> load yoapp                    # load your app into all nodes
0> run                           # and set 'em all running
```

For more information on using RAPMC and its many commands, refer to [SUM] §7.

Optimizing for the RAP

The RAP machine is based on the Texas Instruments TMS320C30 digital signal processor (DSP.) This chip is capable of parallel instruction execution which when used correctly can effectively double program execution speed. The RAP libraries provide hand-coded assembly language inner loops to realize this peak hardware performance where possible at no added cost to the applications programmer. The next level of optimization depends on how the programmer chooses to allocate data in the multi-tier memory hierarchy. This section explains several simple strategies and techniques for using RAP memory effectively.

Fast RAP facts

- Texas Instruments TMS320C30 digital signal processor running at 16MHz
- 32MFLOPS peak for pipelined, parallel multiply-accumulate operations
- 4 nodes per RAP board
- 32-bit word addressed (1KW = 4KB)
- Each node has 2 x 4KB on-chip memory, 256KB or 1MB static RAM, and 4MB or 16MB dynamic RAM
- RAP memory mapped into host's virtual address space

Using memory effectively

Each RAP processing node has memory organized into four banks as indicated below in Table 10–1. Generally, library code, user code and the run-time stack are located in static memory (SRAM), large data arrays are allocated in dynamic memory (DRAM) and selected assembly inner loops and other time-critical code may be located in on-chip memory (RAM0 or RAM1.)

■ **Table 10–1** RAP node memory layout

Name	Description	Relative Speed	Size	Primary Use
RAM0	on-chip	12 (fastest)	1KW	time-critical loops
RAM1	on-chip	12 (fastest)	1KW	and data (eg, tables)
SRAM	Static, on-board	4 (rd) 2 (wr)	64-256KW	code, data, stack
DRAM	Dynamic, on-board	1 (slowest)	1-4MW	large data arrays

Because of the difference in memory access speeds between the banks, application performance can benefit significantly from optimal memory placement of code and especially of data.

Code placement

Control over the memory location of user code on the RAP is provided by the linker via the `link.cmd` located in rap/`examples/tms`. In general, seldom-called code can safely be placed in DRAM (for example, initialization code) whereas more commonly used code can be placed in SRAM. Note that there is a tradeoff in using SRAM for large code blocks which could otherwise be used for frequently accessed data. Furthermore there may be little penalty in placing certain code blocks containing tight (several instruction) loops in DRAM because the processor also maintains an instruction cache. Small assembly language inner loops known to be time-critical may benefit from being moved to on-chip memory (RAM0 or RAM1.) See comments in `link.cmd` and [SUM] §3.4 and §6.2.5 for more information on placing code in specific memory banks.

Given programs which perform most of their computation in tight inner loops (such as those provided by the RAP libraries), the greatest performance benefits will likely come from optimizing the placement of data rather than code.

Data placement and memory type designators

Control over placement of dynamically allocated user data including library objects (vectors, matrices and function lookup tables) is provided by the RAP libraries using a *memory type designator* at allocation time. Separate control is provided for raw heap allocation (via the `MALLOC` macro or `rap_malloc` method; see [SUM] §6.2.2), data allocation for library objects (via their constructors) and object allocation (via the C++ `operator new`.)

Memory type designators for the RAP can have the values given above in Table 10–1, i.e. one of `{ RAM0, RAM1, SRAM, DRAM }`. The special designator `FASTEST` and modifier `FASTER` are also provided. `FASTEST` will satisfy the allocation request in the first bank with sufficient space, searching strictly in the order RAM1, RAM0, SRAM and finally DRAM. Note that earlier allocations will thus be satisfied with faster on-chip memory whereas later requests will get space in remaining off-chip static or dynamic memory. Is is therefore important to make allocations for time-critical data first in program execution order.

The modifier `FASTER` is intended to be used as an inclusive flag, eg. `SRAM|FASTER` would attempt to satisfy a request first from on-chip memory (RAM1 and RAM0) and, failing that, from static memory. For more information see [SUM] §6.2.2.

Specifying placement for matrices and vectors

For C++ programs, the constructors for `Fvec` and `Fmat` objects accept an optional parameter `memtype` which designates a particular memory bank to hold the object's data (i.e. the elements of the vector or matrix):

```
vec = new Fvec(1000);            // The default: FASTEST
```

```
vec = new Fvec(1000,SRAM);       // Specifically in SRAM
vec = new Fvec(200,SRAM|FASTER); // Tries RAM1, RAM0, SRAM
Fmat m(100,100,SRAM);            // auto declaration of m; data uses
                                 //    10KW of SRAM
```

In C, the `memtype` specifier is not optional; you can use `FASTEST` as the default with the understanding that the first requests will get the fastest memory and later requests will get whatever is left over:

```
vec = new_Fvec(1000,FASTEST);    /* C flavor */
mat = new_Fmat(1000,1000,DRAM);  /* Specifically in DRAM */
vec = new_Fvec(1000,SRAM|FASTER); /* etc...*/
```

Whereas the space allocated for the object's data elements is determined by the designator, the space for the object itself is allocated either on the stack (for C++ auto declarations) or via `operator new`. Note that the C interface to the libraries also calls `operator new`. Allocations performed by `operator new` can be controlled as described below.

Specifying placement for function lookup tables

Function lookup tables implemented with the library class `Table` can provide significant performance benefits for certain applications. These benefits can be fully realized by ensuring the lookup table is placed in on-chip memory. The placement of data for objects derived from `Table` is designated by the method `table_memory()`. This is by default set to `FASTEST` which attempts to guarantee that small tables allocated at the beginning of an application will be placed in on-chip memory. You can change this as needed, eg. calling `table_memory(RAM1)` will guarantee that an error abort is generated if succeeding `Table` objects cannot be allocated in RAM1 as expected:

```
table_memory(RAM1);              // FastTable must be on-chip
FastTable T(...);                // Create the fast table
table_memory(SRAM|FASTER);       // SRAM is ok for the rest
OtherTable OT(...);
```

Specifying placement with `operator new`

The RAP version of the default `operator new` uses `rap_malloc` with the memory designator set in the global variable `OBJ_RAM`. By default, `OBJ_RAM` is set to `FASTEST`. You can set `OBJ_RAM` to whatever is appropriate for a sequence of object allocations in your application:

```
OBJ_RAM = SRAM;                  // Only use SRAM for these objects
vec = new Fvec(1000,DRAM);       // Put object data in DRAM
...
OBJ_RAM = FASTEST;               // Now allow objects on-chip
vec = new Fvec(1000,SRAM|FASTER); // and data on-chip or in SRAM
```

You can use this technique to prevent object allocations at the beginning of your application from using all of the on-chip memory, preserving it for time-critical allocations made later in the execution sequence. Note that if you want to use this technique, you should set `OBJ_RAM` and possible call `table_memory()` before making any calls which may allocate objects (eg., `seed()`, `sigmoid_domain_init()`, etc.)

Extending the Libraries

The RAP libraries are designed to allow for the easy addition of user-defined methods by deriving user-defined classes from the library classes. For example, new matrix-vector computational methods can be added to subclasses of `Fvec`, `Fmat` and `Ivec`. In order to take advantage of the library framework for parallel computation of matrix-vector operations, it is important to understand how library objects are distributed across processors.

Distributed memory abstraction

RAP library classes are built on a software *distributed memory* abstraction implemented in the `Dmem` class. The `Dmem` class manages the allocation of storage and communication of data across the available processors for a one-dimensional array of *elements* which can be scalars or aggregates. Elements are lumped into *groups* which are distributed but never broken across processors.

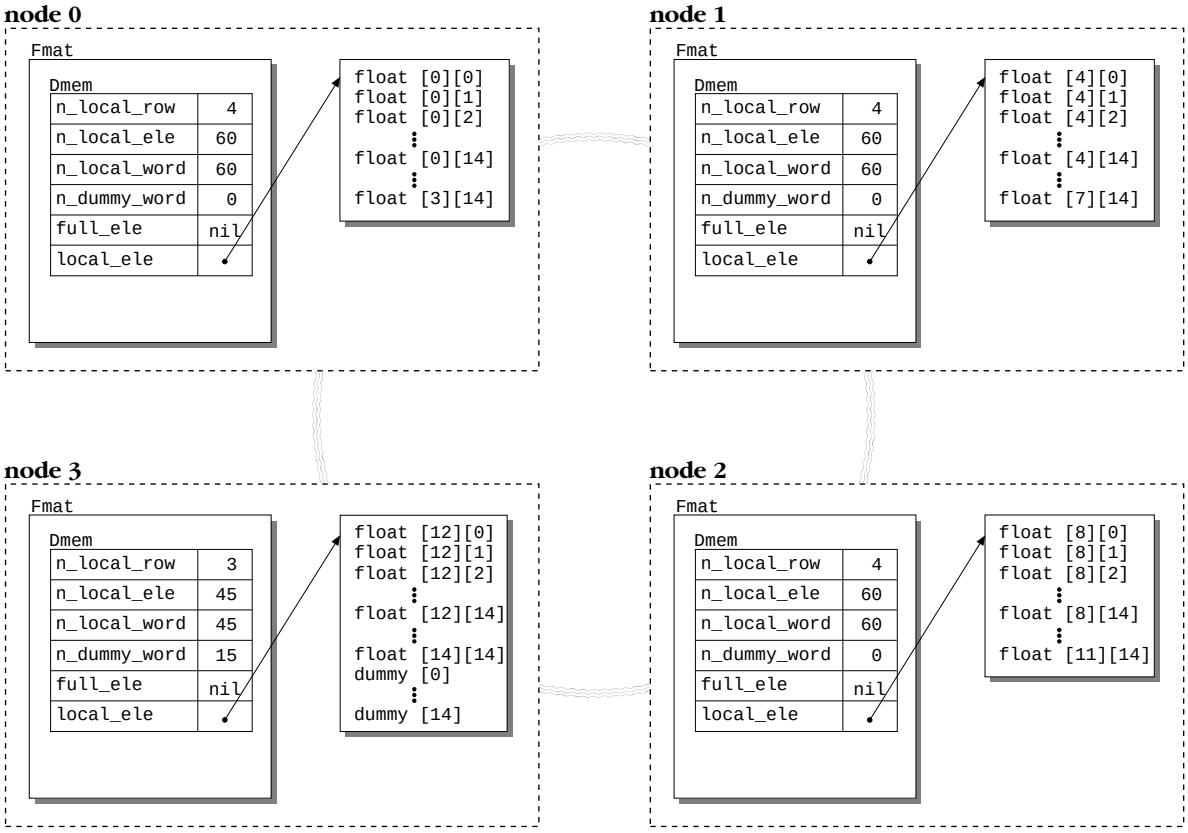
The matrix-vector classes `Fvec`, `Fmat` and `Ivec` derive from `Dmem`. Generally their methods are implemented in terms of operations on the local subset of grouped data, without concern for the particular hardware configuration. The distributed memory abstraction considerably simplifies the parallel implementation of many of these methods and provides inherent scalability as well as a degree of portability.

Working with distributed memory objects

`Dmem` objects maintain distributed data in one of two states: *distributed* or *replicated*. When initially created, `Dmem` objects are distributed, meaning that each available processor has only a subset of the object's data in its local memory. `Dmem` objects distribute their data groups (e.g., matrix rows) over the available processors so that each node, 0 through N_NODE-2, has an integral number of groups given by *ceil(n_groups /* N_NODE) except the last node, N_NODE-1, which will have possibly fewer. The last node will have *n_groups* mod *ceil(n_groups /* N_NODE) groups. The field `Dmem::local_ele` points to the data groups local to each node.

For example, Figure 11–1 shows how a 15 x 15 `Fmat` is distributed across a 4-node RAP. Notice that each processor has allocated space for 4 of the 15 rows of the matrix except the last which has 3 rows of data followed by one row of placeholders.

■ **Figure 11–1** Distributed 15 x 15 matrix on a 4-node RAP

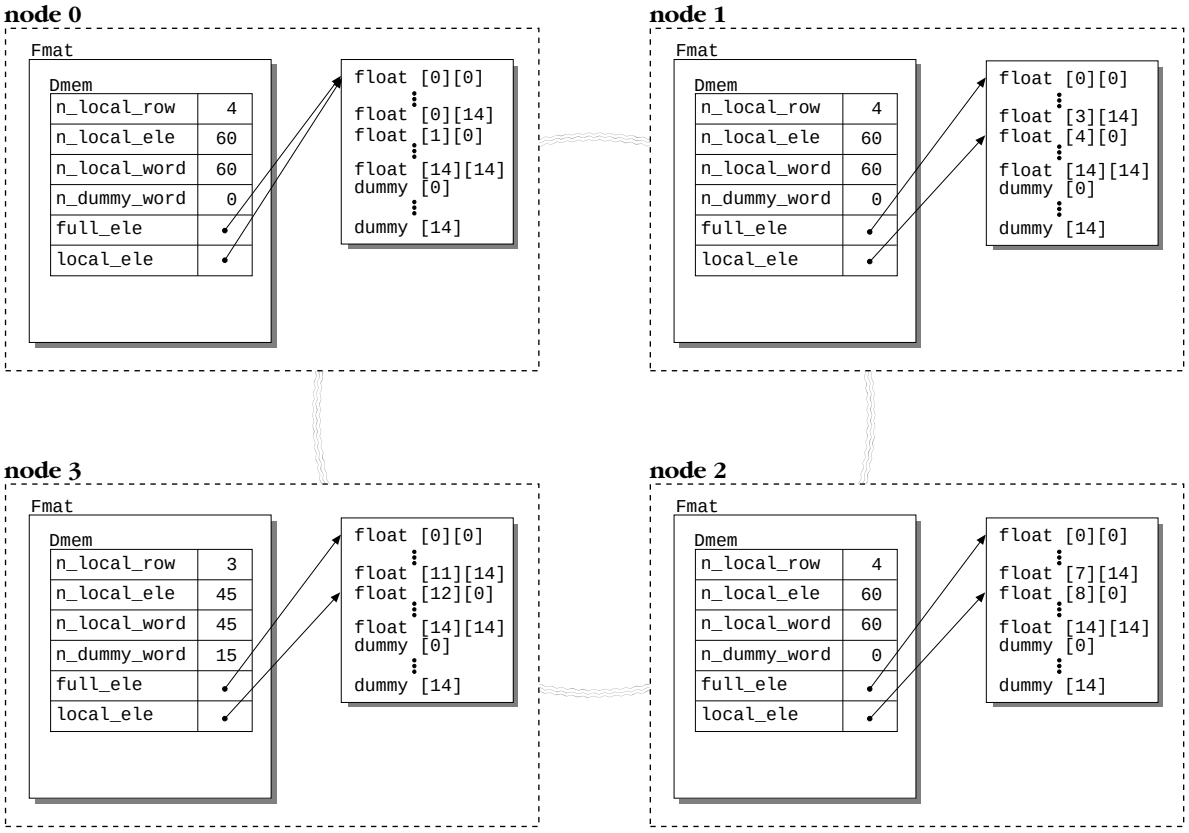


Replicating objects across processors

Normally `Dmem` objects remain distributed throughout their lifetimes. However, some methods require that all data are available in the local memory of each processor. That is, all of the object's data are replicated on each processor. The method `Dmem::replicate` calls `ring_distribute` which takes care of the inter-processor communication necessary to replicate all the data on each node. The `replicate` method also calls `Dmem::alloc_full` to allocate enough space for the entire object. The field `Dmem::full_ele` is set to point to this space. The field `local_ele` is adjusted to point within it to the subset of elements that the node had managed when the object was distributed.

Figure 11–2 shows the same 15 x 15 `Fmat` replicated across a 4-node RAP; each processor has a complete copy of the matrix. Notice that each node has a pointer within the matrix data to its local subset of data, and that the last local subset may be padded with placeholders so that each subset is of equal length.

■ **Figure 11–2** Replicated 15 x 15 matrix on a 4-node RAP



Adding computational methods

Computational methods can be added to the RAP libraries by deriving a new user-defined class from the library classes `Fvec`, `Fmat` or `Ivec`. New computational methods can then be added to the user-defined class. This approach avoids modification of the RAP library source code and will make it easier to upgrade the libraries for future releases without modifying your source code. Deriving separate classes also makes it easier to share new computational methods with other RAP users.

Deriving from the `Fvec` class

Your derivative must minimally have a constructor and destructor as well as any new methods you wish to add. As a derivative of `Fvec` or `Fmat` you can use the private accessors of `Dmem` and `Fvec` in your implementation. (Note that deriving from `Fvec` precludes inheriting methods from `Fmat`; you might want to derive from `Fmat` instead.)

```
typedef rapFvec Fvec;
class FVec : public rapFvec {
   public:
         // note size is n_groups == n_ele iff group_size == 1
      FVec(int size, int mem_type=FASTEST, int group_size=1) :
         rapFvec(size, mem_type, group_size) {}
      ~FVec() {}     // Declare destructor even if null to avoid bugs

      void sgn(FVec* in, float thresh=0.0); // new method: signum fn
};
```

Defining a new simple method

Given the above declarations for a new `FVec : rapFvec` class, you can define your new computational method following the pattern of library routines found in `tms/lib/math/mat3.cc` for routines that perform only local computations. To increase your chances of remaining compatible with future releases, use `Dmem` accessors rather than specifying the protected members directly (eg., use `size()` rather than `n_ele`):

```
void FVec::sgn(FVec* in, register float thresh) {
   assert (this->size() == in->size());  // vectors must be == length
   register int n = get_n_local_ele(); // count of our local elements
   register float* o = local_ptr();    // pointer to our locals
   register float* i = in->local_ptr();
   // note: avoid using x[i] -- C30 doesn't have hw integer multiply
   for (; n--; i++)              // all nodes do this on their locals
      if (*i > thresh) *o++ = 1.0;
      else if (*i < thresh) *o++ = -1.0;
      else *o++ = 0.0;
}
```

Advanced topics

Some users may wish to go further in extending the RAP libraries or integrating existing DSP code with the RAP libraries. A detailed discussion of the issues involved is beyond the scope of this guide; however, here are some orienting pointers to get you going in the right direction.

Defining methods with global computations

The sample method defined above performs only local computations; that is, each node need consider only the data in its portion of the vector in performing the computation. Other methods may need to use the data or intermediate results of other nodes in performing their computations. On a distributed architecture like the RAP, this necessitates some communication between nodes. The RAP libraries accomplish this using the ring functions provided in `ring.cc`. For more information on using the communications ring,

refer to [SUM] §6.2.4.3 and to the examples provided by the RAP library routines in `mat3.cc`.

Use of `N_NODE` and `NODE_ID`

The RAP libraries define two globals at system initialization time, `N_NODE` and `NODE_ID` which are, respectively, the number of nodes in the RAP system and the ID of each node ($0 \leq$ `NODE_ID` $<$ `N_NODE`.) In a data-parallel environment, there are few legitimate uses of these globals; programs should be written to scale to any number of nodes and in particular should work on a single node, i.e. a uniprocessor machine, which is the case when the program is compiled for the host. The RAP libraries meet these criteria. Still, there are several situations where it is necessary to refer to these globals, notably for ring communications. The code in `mat3` provides adequate examples; or refer to [SUM] and [SAM.] Note that it is easy to lose the simplicity of lock-step data parallelism by executing a conditional on `NODE_ID`. You are then writing essentially multithreaded code — caveat programmer.

Use of low-level routines

Further inspection of RAP library routines in `mat3.cc` reveals that most are implemented by calls to lower-level routines in `tms/lib/math/mat2.c`. Whereas the routines in `mat3` know about distributed objects, local elements and the ring, the routines in `mat2` are implemented in terms of an integer count and a pointer to a linear array of elements (i.e. a `float*`.) Most routines in `mat2` either in turn call their assembly-language equivalents if running on the RAP, or perform the actual inner loop of the computation in C if running on the host.

Routines in `mat2` are named according to a convention similar to that used for the high-level C interface to the libraries (which is defined in `mat4_common.cc`). That is, `mat2` contains routines such as `mul_vv` (dot product) whereas the high-level C interface would be called `mul_VV` (note difference in case.) The `mat2` routines are declared in `raplib.h` and you may choose to use them when implementing new methods on `Fvec` and `Fmat` derivatives.

When an assembly-language implementation is available, it will be named as the `mat2` routine with a preceding underscore, eg., `_mul_vv`. These inner loops are implemented for the RAP in `tms/lib/math/mat1.s`. We suggest you follow the convention of calling the `mat2` bottleneck routine rather than calling the assembly routine directly; also you should define a C implementation and bottleneck for any assembly routines you add. This will allow your application code to be debugged on the host and to port more easily to other processors. Also the C bottleneck provides some degree of type safety when calling the assembly routines (or at least it isolates the hazard.)

Example: Reference-based class interfaces

The `Fvec` interface provided by the RAP libraries is *pointer-based*; i.e., arguments to methods take pointers to objects. Some programs may be more conveniently written in terms of member objects, in which case a *reference-based* class interface can be easily derived as follows. Note that the present limitations of C++ necessitate the addition of garbage collection to efficiently implement binary operators and preclude the definition of compound operators like `*+` altogether. However, the assignment operators are trivially and perhaps usefully overloaded in a reference-based interface:

```
class FVec : rapFvec {
   public:
      FVec(int size, int mem_type=FASTEST, int group_size=1) :
         rapFvec(size, mem_type, group_size) {}
      ~FVec() {}

      // copy constructor just duplicates data (no aliasing or GC)
      FVec(const FVec& src) :
         rapFvec(src.n_rows(), FASTEST, src.group_size()) {
               *this = src;      // copy the data of the source
      }                          // vector to this using op=

      // Some assignment operators
      FVec& operator=(const float* vals) { // put data from a float*
         return (put(0,vals,n_ele),*this); }

      FVec& operator=(float value) { // set all v[i] = scalar
         return (set(value),*this); }

      FVec& operator=(const FVec& src) { // copy an FVec to *this
         assert(size() == src.size());
         copy((rapFvec*)&src);   // Just blindly copy data for now...
         return *this;
      }

      // some in-place unary operators
      // add in_vector to this vector (element by element)
      FVec& operator+=(const FVec& in_vector) {
         return(rapFvec::add((rapFvec*)&in_vector),*this); }

      // subtract in_vector from this vector (element by element)
      FVec& operator-=(const FVec& in_vector) {
         return(rapFvec::sub(this,(rapFvec*)&in_vector),*this); }

      // multiply in_vector to this vector (element by element)
      FVec& operator*=(const FVec& in_vector) {
         return(rapFvec::mul_ele((rapFvec*)&in_vector),*this); }

      // ... other methods -- no GC to support binary operators
      // add vector1 and vector2 and put result in this vector
      void add(FVec& vector1, FVec& vector2) {
         rapFvec::add(&vector1, &vector2); }
      // etc...
};
```

❑

Notes