Gandalf: The Fast Computer Vision and Numerical Library

Philip F McLauchlan
Imagineer Systems Ltd.

Contents

1 Introduction **6**

1.1 Conventions and style . . . 6

1.1.1 Function/name prefixes and case . . . 6

1.1.2 "Quick", "slow" and "in-place" Gandalf routines . . . 6

1.1.3 Variable argument list functions . . . 7

1.1.4 Dynamic self-modification . . . 7

1.1.5 Coordinate pair ordering . . . 7

1.1.6 Pixel values . . . 7

1.1.7 Image orientation . . . 7

1.1.8 Error checking . . . 8

2 The Common Package **9**

2.1 Miscellaneous definitions . . . 9

2.1.1 Simple types . . . 9

2.1.2 Types with specific bit sizes . . . 10

2.1.3 Debugging tools . . . 10

2.2 Linked lists . . . 10

2.3 Bit arrays . . . 12

2.4 Memory allocation . . . 12

2.5 Array fill/copy . . . 13

2.6 Complex numbers . . . 13

2.7 Numerical routines . . . 13

2.8 Comparison routines . . . 14

2.9 Error handling . . . 14

2.9.1 More on the error trace . . . 15

2.10 Error tests and codes . . . 16

3 The Linear Algebra Package **17**

3.1 Fixed size matrices and vectors . . . 18

3.1.1 Fixed size vectors . . . 18
3.1.1.1 Creating and accessing fixed size vectors . . . 18
3.1.1.2 Fixed size vector addition . . . 19
3.1.1.3 Fixed size vector subtraction . . . 20
3.1.1.4 Rescaling a fixed size vector . . . 20
3.1.1.5 Fixed size vector products . . . 21
3.1.1.6 Fixed size vector file I/O . . . 21
3.1.1.7 Conversion from general to fixed size vector . . . 22
3.1.1.8 Single precision fixed size vectors . . . 22
3.1.1.9 Other types of fixed size vector . . . 23
3.1.1.10 Other sizes of fixed size vector . . . 23
3.1.1.11 Setting/extracting parts of fixed size vectors . . . 23
3.1.2 Fixed size matrices . . . 25
3.1.2.1 Definitions of fixed size matrice . . . 25
3.1.2.2 Creating and accessing fixed size matrices . . . 26
3.1.2.3 Fixed size matrix addition . . . 28
3.1.2.4 Fixed size matrix subtraction . . . 29
3.1.2.5 Rescaling a fixed size matrix . . . 30
3.1.2.6 Transposing a fixed size matrix . . . 31
3.1.2.7 Fixed size vector outer products . . . 31
3.1.2.8 Fixed size matrix/vector multiplication . . . 32
3.1.2.9 Fixed size matrix/matrix multiplication . . . 33
3.1.2.10 Fixed size matrix inverse . . . 37
3.1.2.11 Determinant, trace, norms of fixed size matrix . . . 37
3.1.2.12 Fixed size matrix decompositions . . . 38
3.1.2.13 Fixed size matrix file I/O . . . 40
3.1.2.14 Conversion from general to fixed size matrix . . . 41
3.1.2.15 Single precision fixed size matrices . . . 42
3.2 General size matrices and vectors . . . 42
3.2.1 General size vectors . . . 42
3.2.1.1 Creating and freeing general size vectors . . . 43
3.2.1.2 Adjusting the size of a general size vector . . . 43
3.2.1.3 Filling a general size vector with values . . . 44
3.2.1.4 Accessing the elements of a general size vector . . . 44
3.2.1.5 Copying a general size vector . . . 45
3.2.1.6 General size vector addition . . . 45
3.2.1.7 General size vector subtraction . . . 46

3.2.1.8 Rescaling a general size vector . . . 46
3.2.2 General size matrices . . . 47
3.2.2.1 Creating and freeing general size matrices . . . 48
3.2.2.2 Adjusting the size of a general size matrix . . . 50
3.2.2.3 Filling a general size matrix with values . . . 51
3.2.2.4 Accessing the elements of a general size matrix . . . 54
3.2.2.5 Copying a general size matrix . . . 55
3.2.2.6 Transposing a general size matrix . . . 56
3.2.2.7 General size matrix addition . . . 56
3.2.2.8 General size matrix subtraction . . . 58
3.2.2.9 Rescaling a general size matrix . . . 59
3.2.2.10 General size matrix/vector multiplication . . . 60
3.2.2.11 General size matrix/matrix multiplication . . . 61
3.2.2.12 Inverting a general size matrix . . . 65
3.2.2.13 Cholesky factorising a general size symmetric matrix . . . 66
3.2.2.14 Symmetric matrix eigendecomposition . . . 66
3.2.2.15 Accumulated symmetric matrix eigendecomposition . . . 67
3.2.3 Single precision general size matrices & vectors . . . 68

4 The Image Package **70**

4.1 Image formats and types . . . 70
4.2 Simple image/pixel routines . . . 71
4.2.1 Image creation/destruction . . . 71
4.3 Image file I/O . . . 73
4.3.1 Setting an image to a new format, type and dimensions . . . 74
4.3.2 Accessing single pixels . . . 75
4.3.3 Filling an image with a constant value . . . 81
4.3.4 Converting a pixel to a given format/type . . . 82
4.4 Binary images . . . 83
4.5 Pointer images . . . 86
4.6 Copying/converting the whole or part of an image . . . 87
4.6.1 Accessing channels of an image . . . 88
4.7 Displaying images . . . 90
4.8 Image pyramids . . . 91
4.9 Inverting an image . . . 92
4.10 Image sequence I/O . . . 93

5 The Vision Package **95**

5.1 Cameras . . . 95
5.1.1 Building cameras . . . 100
5.1.2 Projecting points and lines . . . 101
5.1.3 Adding/removing camera distortion . . . 103
5.1.4 Building the camera calibration matrix . . . 104
5.1.5 Converting cameras between precisions . . . 105
5.2 Computing the fundamental/essential matrix . . . 105
5.3 Computing a homography between 2D scene and image . . . 107
5.3.1 Computing a 2D homography from an array of feature matches . . . 109
5.3.2 Computing a 2D affine homography . . . 110
5.4 Computing a homography between 3D scene and image . . . 111
5.5 Smoothing an image using a 1D convolution mask . . . 112
5.6 Smoothing an image using a 2D convolution mask . . . 114
5.7 Feature detection . . . 117
5.7.1 Image feature coordinate frames . . . 117
5.7.2 Edge detection . . . 117
5.7.3 Displaying an edge map . . . 120
5.7.4 The Canny edge detector . . . 121
5.7.5 Corner detection . . . 122
5.7.6 Displaying a corner map . . . 123
5.7.7 The Harris corner detector . . . 124
5.7.8 Line segment detection . . . 124
5.7.9 Displaying a line map . . . 126
5.7.10 The Gandalf line detector . . . 127
5.8 Representing 3D rotations . . . 127
5.8.1 Quaternion routines . . . 129
5.8.2 General rotation routines . . . 129
5.9 Representing 3D Euclidean transformations . . . 133
5.10 Levenberg-Marquardt minimisation . . . 135
5.10.1 Robust observations . . . 137
5.10.2 Generalised observations . . . 138
5.10.3 Levenberg-Marquardt software . . . 139
5.11 Fast Hough Transform . . . 144
5.11.1 The Fast Hough Transform (FHT) . . . 144
5.11.1.1 Notation . . . 144
5.11.1.2 The FHT Algorithm . . . 145
5.11.1.3 Example: Line Fitting . . . 145

5.11.2 Example: Plane Fitting . . . 145
5.11.2.1 Calculating the Intersection of a Plane and a Sphere . . . 147
5.11.2.2 Calculating the Plane Parameters of a Child Cube . . . 148
5.11.2.3 Formal Statement of the FHT Plane Fitting Algorithm . . . 148
5.11.3 Speed Improvement to FHT Line Finder . . . 151
5.11.3.1 Formal Statement of Algorithm . . . 153
5.11.3.2 Comparision of Speed and Memory Requirement with Standard FHT . . . 156
5.11.3.3 Complexity Comparison of FHT and MFHT . . . 157

6 The Test Framework **158**

6.1 Adding new tests . . . 158

Chapter 1

Introduction

Gandalf is a C library designed to support the development of computer vision applications. For installation instructions see the INSTALL file provided with the distribution. Gandalf is divided into the following packages:–

- The **Common** package contains general purpose tools used by other packages. It includes routines for memory allocation, linked lists, various numerical functions, array manipulations and error handling.
- The **Linear algebra** package includes vector/matrix manipulation routines both for small objects (sizes 2,3,4) and general size vectors & matrices. The latter optionally employs LAPACK for certain operations, where a fast implementation of LAPACK is available.
- The **Image** package contains low-level image creation and manipulation functions, supporting grey-level, RGB colour images with or without alpha channels, with various different pixel depths. 2D and 3D vector field images are also supported.
- The **Vision** package contains some useful vision utility modules, including edge/line/corner feature detection, and geometrical fitting routines.

In the following chapters we introduce each package in turn, describing the scope and applications of the package, followed by tutorial examples showing how to build a user application using Gandalf. This is followed by chapter 6 describing the testing framework used by Gandalf. Firstly we describe the conventions and style of Gandalf.

1.1 Conventions and style

1.1.1 Function/name prefixes and case

All Gandalf functions begin with the `gan_...` prefix, to minimise the possibility of name conflicts. Function names are written in lower case, with a few exceptions where upper case is used for single letters in a function name. Defined constants and enum values are written in upper case, for instance `GAN_TRUE` and `GAN_FALSE` for the boolean type `Gan_Bool`. Structure, union and enum type names are capitalised, for instance `Gan_List`, `Gan_Vector` and `Gan_Image`, the names of the Gandalf linked list, vector and image structures.

1.1.2 "Quick", "slow" and "in-place" Gandalf routines

A convention used extensively in the linear algebra and image packages is to provide "quick" and "slow" versions of the same operation, indicating the difference using the suffices `..._q` and `..._s` respectively. The meanings of the suffices vary, but are one of:

1. The "slow" version dynamically allocates the memory to hold the result, whereas the "quick" version uses a pre-allocated result passed in to the function, avoiding repetitively allocating and freeing memory when the function is called several times. This is the convention used in the image package and for general size matrix and vector functions in the linear algebra package.

2. The "slow" version returns a structure as its result, and the "quick" version expects a pointer to the result structure to be passed in. This is the convention used for fixed-size matrices and vectors in the linear algebra package.

There is also a conventional `..._i` suffix for operations that overwrite one of the input arguments with the result.

1.1.3 Variable argument list functions

Variable argument lists have been avoided where possible, because of the lack of argument type and number checking. All functions with a variable argument list have a `..._va` suffix.

1.1.4 Dynamic self-modification

Another important design feature of Gandalf is the ability of many Gandalf objects to dynamically modify themselves. Thus Gandalf matrices, vectors and images can be redefined at any time to a new type and/or size. This feature has many benefits, among which are:

1. Results of computations may be overwritten in-place on the input arguments, in many cases without any memory or computational overhead. For instance, the Cholesky factorisation of a positive-definite symmetric matrix is a lower triangular matrix factor. In Gandalf the operation can be performed in-place on the symmetric matrix input argument, producing a lower triangular matrix. See the function `gan_squmat_cholesky_i()`.

2. The same structure can be used to store the results of several computations, whether or not the sizes and types of the output results vary. This saves many calls to `malloc()` and `free()`.

1.1.5 Coordinate pair ordering

A large number of Gandalf routines require horizontal and vertical sizes or coordinates to be passed as arguments, whether representing the dimensions of a matrix or coordinates of a pixel in an image. The ordering of the arguments in such cases is problematic. For specifying a matrix element the most natural ordering is to have the row coordinate first, corresponding to the coordinate order i, j for matrix element A_{ij}. On the contrary, for image coordinates there is no obvious convention. In Gandalf a universal convention is applied that in all such cases the *vertical* or row coordinate is the first argument.

1.1.6 Pixel values

Gandalf uses the convention that image pixels represented in floating point are always in the range [0,1]. In integer formats the range is the whole range of the type, whether character, short integer, integer or long integer. This convention is applied for all conversions between pixels, as well as the image processing and display routines. When programming with Gandalf you should where possible stick to this convention.

1.1.7 Image orientation

The normal convention is that the first (zero) row of an image is at the top, and Gandalf uses this convention. It is only relevant in certain image processing and vision algorithms where the image orientation affects the results. OpenGL assumes the opposite convention, which Gandalf gets around by displaying images using a negative vertical

scaling of image coordinates. Currently the only image processing algorithm which assumes an image orientation is the Canny edge detector.

1.1.8 Error checking

Gandalf is written with a large amount of automatic error checking, for instance testing for accesses to illegal matrix or image data. When `NDEBUG` is defined, most of these tests are switched off, using a similar mechanism to `assert()`. It is up to the programmer to decide which tests are for bugs in the code, and can therefore be turned off when code is compiled for release by defining `NDEBUG`, and which are data-dependent errors and should therefore remain.

Chapter 2

The Common Package

The **common** package defines general purpose types and routines used extensively elsewhere in Gandalf, and also available to other application code using Gandalf. To use any function or structure in the common package use the declaration

```
#include <gandalf/common.h>
```

but including individual module header files instead will speed up program compilation. We describe each module in the common package below.

2.1 Miscellaneous definitions

```
#include <gandalf/common/misc_defs.h>
```

The module `misc_defs.[ch]` defines basic types used in Gandalf.

2.1.1 Simple types

A boolean type `GAN_BOOL` is defined:

```
typedef enum { GAN_FALSE=0, GAN_TRUE=1 } Gan_Bool;
```

The boolean type is the standard type returned by a Gandalf function, where a return type of `GAN_TRUE` indicates success, `GAN_FALSE` failure. Another use for the `Gan_Bool` type is with bit arrays and binary images. See Sections 2.3 and 4.4.

The `Gan_Type` enumerated type is used extensively to indicated different kinds of simple objects:

```
/// labels for simple types used throughout Gandalf
typedef enum
{
   GAN_CHAR,       /**< signed character */
   GAN_UCHAR,      /**< unsigned character */
   GAN_SHORT,      /**< signed short integer */
   GAN_USHORT,     /**< unsigned short integer */
   GAN_INT,        /**< signed integer */
   GAN_UINT,       /**< unsigned integer */
```

```
   GAN_LONG,       /**< signed long integer */
   GAN_ULONG,      /**< unsigned long integer */
#if (SIZEOF_LONG_LONG != 0)
   GAN_LONGLONG,   /**< signed extra-long integer */
#endif
   GAN_FLOAT,      /**< single precision floating point */
   GAN_DOUBLE,     /**< double precision floating point */
   GAN_LONGDOUBLE, /**< long double precision floating point */
   GAN_STRING,     /**< string (array of characters) */
   GAN_BOOL,       /**< boolean */
   GAN_POINTER     /**< generic pointer */
} Gan_Type;
```

Note that the GAN_LONGLONG value is only defined if the configure program finds the long long C type, and is able to determine its size. The array gan_type_sizes[] holds the sizes of each Gan_Type value:

```
/// array of sizeof()'s of each Gandalf type, one for each value in a Gan_Type
extern const size_t gan_type_sizes[];
```

gan_type_sizes and the gan_debug boolean flag (see below) are the only global variables in Gandalf. gan_type_sizes[] is a constant array, so it is thread-safe.

Gandalf also provides single and double precision floating point versions of the integer limit values found in <limits.h>. For instance GAN_INT_MAXF and GAN_INT_MAXD are the float and double versions of INT_MAX.

2.1.2 Types with specific bit sizes

Gandalf builds unsigned integer types with specific bit sizes, so that the types GAN_UINT8, GAN_UINT16, GAN_UINT32 and GAN_UINT64 are #define'd to the relevant Gan_Type value, using the information on the sizes of architecture-dependent C object sizes provided by configure. You can also define your own variables with specific sizes using the typedefs gan_ui8 etc. For instance a declaration

```
gan_ui16 val;
```

declares a 16-bit variable. gan_ui32 and gan_ui64 are also defined, the latter only on 64-bit architectures.

2.1.3 Debugging tools

The macro gan_assert() provides a mechanism similar to assert() except that it allows a user-defined message to be printed when the test fails. When NDEBUG is #define'd gan_assert() will have no effect. The global variable gan_debug determines whether to print certain debugging messages. It is only available when NDEBUG is not defined.

2.2 Linked lists

```
#include <gandalf/common/linked_list.h>
```

Gandalf linked lists are stored in Gan_List structures. The underlying structure is a doubly-linked list, so Gandalf lists can be traversed both forwards and backwards. A new empty list can be created using

```
Gan_List List;

gan_list_form(&List);
```

Note that to detect errors the return value of `gan_list_form()` should be compared with `NULL`, invoking the Gandalf error package (see Sections 2.10 and 2.9) and returning an error condition if `NULL` is returned. Here and elsewhere we omit these tests for the sake of clarity, but many examples of this testing can be found in the Gandalf source.

To insert a new list node at the start of the list, use

```
gan_list_insert_first ( &List, ptr );
```

where `ptr` is the data item (pointer) to be stored. By repetitively called this function, a list can be built up, with the last item added as the first node in the list. A Gandalf list stores pointers in an ordered way, while still transparently allowing pseudo-random access to the list nodes. As well as the stored data, the list maintains a state variable indicating a position within the list, from 0 to $N-1$ for a list of N nodes. The normal way to traverse a list is to use the following sequence:

```
int iCount;
Gan_Matrix *pMatrix;

gan_list_goto_head ( &List );
for ( iCount = gan_list_get_size(&List)-1; iCount >= 0; iCount-- )
{
   pMatrix = gan_list_get_next ( &List, Gan_Matrix );
   ... [ do something with pMatrix ] ...
}
```

`gan_list_goto_head()` sets the position state variable to -1, i.e. the position just before the start of the list. `gan_list_get_size()` returns the number of nodes in a list. So the above loop runs through each node of the list, calling `gan_list_get_next()` to provide each data item in turn, in this case matrix structure pointers. `gan_list_get_next()` increments the position state variable by one each time it is called, so on the first call in the above loop, the position is increment from -1 to 0, and the node at position 0 returned.

To free the list use

```
gan_list_free ( &List, NULL );
```

which frees the list nodes but not the data they point to. If you wanted to free the data as well, for the above list you would use

```
gan_list_free ( &List, (Gan_FreeFunc) gan_mat_free );
```

which calls `gan_mat_free()` on each matrix in the list.

Note that pointers to lists may be used instead of directly using the structures. To create a list using pointers use

```
Gan_List *pList;

pList = gan_list_new();
```

and at the end call

```
gan_list_free ( pList, NULL );
```

In this and other examples Gandalf keeps track of which parts of a structure were dynamically allocated, and only frees those which were.

Other ways to build and access linked lists are provided in the reference documentation.

2.3 Bit arrays

```
#include <gandalf/common/bit_array.h>
```

Gandalf bit arrays are a compact representation of binary flags. They are used both directly and as the foundation of binary images in Gandalf (see Section 4.4). They allow compact storage of an array of boolean values. The architecture of the computer determines how the boolean values are stored, so for instance on 32 bit machines, the boolean values are packed as 32 values in a single word. To create a bit array use the following code:

```
Gan_BitArray BitArray;

gan_bit_array_form ( &BitArray, 200 );
```

for an array of 200 bits.

A bit array may be initialised to zero by calling

```
gan_bit_array_fill ( &BitArray, GAN_FALSE );
```

or to one (all bits set) by passing `GAN_TRUE` instead of `GAN_FALSE`. To set a bit to one, use

```
gan_bit_array_set_bit ( &BitArray, pos );
```

where `pos` is the bit you want to set, from zero to $N-1$ for a bit array of N bits. Similarly use `gan_bit_array_clear_bit()` to clear a bit to zero. To return the value of a particular bit use

```
Gan_Bool bBit;

bBit = gan_bit_array_get_bit ( &BitArray, pos );
```

Several boolean operations are supported on bit arrays. Given two bit arrays, the operation

```
gan_bit_array_and_i ( &BitArray1, &BitArray2 );
```

performs the bitwise AND operation on each bit of the bit arrays, overwriting `BirArray1` with the result. Bitwise OR, exclusive-OR (EOR) and not-AND (NAND) are also supported, as well as inversion (NOT).

To free a bit array after you have finished using it, call

```
gan_bit_array_free ( &BitArray );
```

2.4 Memory allocation

```
#include <gandalf/common/allocate.h>
```

Gandalf provides some macros to simplify access to the standard `malloc()` and `realloc()` functions. So for instance to allocate an array of a hundred integers you can use

```
int *aiArray;

aiArray = gan_malloc_array ( int, 100 );
```

instead of the usual

```
int *aiArray;

aiArray = (int *) malloc ( 100*sizeof(int) );
```

2.5 Array fill/copy

```
#include <gandalf/common/array.h>
```

There are two sets of functions in this module, one set dealing with filling an array of numbers or pointers with a constant value, the other dealing with copying an array into another array. To fill an array of `float`s with the value five, for instance, use

```
float afArray[100];

gan_fill_array_f ( afArray, 100, 1, 5.0F );
```

The third `stride` argument will be one for filling a simple contiguous array. A different value indicates the number of elements of the array to skip when filling each value. A value of three, for instance, indicates that only every third element of the array is to be filled.

To copy one array into another, each having arbitrary stride, use

```
float afArray1[100], float afArray2[100];

/* fill array afArray1 with values */
...

gan_copy_array_f ( afArray1, 1, 100, afArray2, 1 );
```

This copies array `afArray1` into `afArray2`.

2.6 Complex numbers

```
#include <gandalf/common/complex.h>
```

Complex numbers are defined here, used

2.7 Numerical routines

```
#include <gandalf/common/numerics.h>
```

This contains a few useful numerical routines not often included in C libraries, such as square (e.g. `gan_sqr_d()` or the macro `gan_sqr()`), cube-root `gan_cbrt()` and various flavours of random number generators. There are functions `gan_solve_quadratic ()` and `gan_solve_cubic()` for finding the real and complex roots of quadratic and cubic equations, and also a function `gan_normal_sample()` for generating samples from a normal distribution.

2.8 Comparison routines

```
#include <gandalf/common/compare.h>
```

This is a set of functions to compute the maximum or minimum of up to six numbers, either as macros (e.g. `gan_min3()` for the minumum of three numbers) or as functions such as `gan_max4_d()` to return the maximum of four double precision floating point numbers.

2.9 Error handling

```
#include <gandalf/common/gan_err.h>
```

The purpose of the error module is to provide a mechanism by which generic reusable code (typically a software library) can report errors to a variety of applications without the need to modify the library code for each new application context. That is, the error reporting mechanism of the library is highly decoupled from that of the application. Communication of error information from library to application is performed using a small and well defined interface.

The role of the library is to communicate full and unprocessed error information to the application. The role of the application is to access the error information and act on it, whether reporting the error back to the user, ignore it or perform some other action. This demarcation of roles allows the application to use its own error reporting mechanism, without any need to embed application specific code in the library. The library achieves generality because it plays no role in reporting the error information, which usually requires system and application specific facilities. Specifically, the library writes (registers) error information into a LIFO stack (error trace) which is built up as the error unwinds through the nested calls. When the library function called by the application finally returns, with an error code, the application uses an error reporter to access the errors details and processes that information in any way it chooses (e.g. displays an error dialogue box, logs the error in a database).

The library function at which a new error occurs must first flush the error trace before registering the error.

When using the error handling code the following definitions are useful.

Error record: a struct holding error code, file name, line number, and text message for one error.

Error trace: a LIFO stack of error records, which allows temporary storage of error information until defered retrieval by application.

Top record: The most recent error stored in trace.

Error detection: Code that detects occurrence of an error.

Error handling: Action undertaken as a result of detecting an error. In library this typically involves registering the error and returning from current function with an error status. In application this typically involves invoking the reporter function.

Registering: The process of placing an error into the trace

Flushing: The clearing of the error trace

Reporter: A function provided by the application to access error stored in trace and then communicating that information to the user or to a log. The reporter function must then flush the error trace.

To illustrate the use of the error handling package, consider an example application which calls library (Gandalf or other) function A, which itself calls library function B, which has an error that is detected. Function B "flushes" the "error trace" (because it is the last function called that uses the facilities of the error package), and then "registers" the error details into the error trace and unwinds to function A with a return value that indicates an error has

occurred. Function A tests the return value and detects the error, and then registers an error into the trace and unwinds to the application with a return value which indicates that an error has occurred. The application tests the return value and detects that an error has occurred, so calls a facility in the error package to report the error. The error report function in turn calls an application-supplied error "reporter" function with a pointer to the error trace as an argument. The address of the error trace is stored as a module scoped variable in the error package The reporter accesses the information contained in the error trace using accessor functions, and communicates the error details to the user or to a log in some application specific way (or it may ignore the error or perform some other action).

Consequences and liabilities:

1. The application is able to:
   - control when errors are reported to the user interface (the library should not itself report errors to the user)
   - provide its own error reporting mechanism (to suit its own user interface).
   - extract sufficient information from the library to enable sufficient error reporting to be performed.
2. The library can be used with many applications, without modification.
3. Interactive resolution of errors occurring in library is problematic. Essentially the library is a black box to the application.

Usage notes for application writer: No code is needed to initialise the error trace. But a error reporting function is optionally installed in the error module using `gan_err_set_reporter()`. The reporter is an application function of type `Gan_ErrorReporterFunc`, which is defined in `gan_err.h`. The reporter must get the error count using `gan_err_get_error_count()` and then sequentially access the errors stored in the trace using `gan_err_get_error(n)`, where `n` is the `n`-th error, and `n=1` is the most recent error. If no error reporter is installed, then the error module provides a default reporter `gan_err_default_reporter()`, whose action is to print the error details into `stderr`. The function call `gan_err_set_reporter(GAN_ERR_DFL)` causes the default error reporter to be used, and the call `gan_err_set_reporter(GAN_ERR_IGN)` inhibits the error reporter from being called. `gan_err_set_reporter()` returns the address of the error reporter that was replaced so that it can be reinstalled later.

When the application tests the return value of a library function and detects that an error has occurred, it should call `gan_err_report()` which invokes the error reporter.

The application writer can choose not to buffer the error details in a trace, but instead have the library function report errors immediately, by automatically calling `gan_err_report()` inside `gan_err_register()`. No error trace is built up. If the application calls `gan_err_report()`, no errors are reported because the trace will be empty. Usage of the trace is controlled by `gan_err_set_trace()` with arguments `GAN_ERR_TRACE_ON` or `GAN_ERR_TRACE_OFF`.

Usage notes for library writer: When a error is detected at the deepest function call that uses the facilities of the error module, then `gan_err_flush_trace()` should be called, followed by `gan_err_register()`. As the subsequent library function unwinds, they should call `gan_err_register()` (but not `gan_err_flush_trace()`), and return with an error code. This continues until the call stack unwinds into the applicaton.

Multi-thread safe: A programmer attempting to use this module in a multithreaded system must heed all precautions attendent with using fully share memory address spaces. To make this module multithread safe, global locks must used to prevent concurrent access to the error trace.

2.9.1 More on the error trace

```
#include <gandalf/common/gan_err_trace.h>
```

This module implements the error trace used in `gan_err.[ch]`. The header file would not normally be included explicitly in library or application code. An error trace is a last-in first-out (LIFO) stack for temporarily holding details of multiple error events until an application is ready to read the stack.

The stack is usually flushed by the function in a sequence of nested calls that initially detects an error. As the call stack unwinds the successive functions also register errors, but they should not flush the error trace.

The stack is implemented as a linked list of error records. If in the process of allocating heap memory for a new error record a memory error occurs, then this is refered to as a deep error.

The stack always maintains two preallocated and unused error records for storing the details of the deep error and the error that was in the process of being registered when the deep error occured.

Even if the top of the stack holds a deep error record and the two preallocated records are used, new errors can still be registered into the trace. These attempts may lead to repeated deep errors, in which case the top deep error serves as an indicator of the deep continuing error state. However, if the registration process is successful (because in the intervening time, some external agent has `free`'d heap memory) the old deep error record is left on the stack and the new errors are registered on top of it.

To ensure that the stack has at least two preallocated records at process startup time, the bottom and second to bottom records of the stack use statically allocated memory. These can never be dynamically `free`'d.

To do this, an external module must define two static error records. In `gan_err.c`, this is implemented as:

```
/* The error trace */

/* Statically allocate last and 2nd to last records for error trace */
static Gan_ErrorTrace record_last = { NULL, GAN_ET_YES, GAN_ET_NO,
                                      GAN_ET_YES, NULL,
                                      GAN_EC_DFT_SPARE, NULL, 0, NULL };
static Gan_ErrorTrace record_2nd_last = { &record_last, GAN_ET_YES, GAN_ET_NO,
                                          GAN_ET_YES, NULL,
                                          GAN_EC_DFT_SPARE, NULL, 0, NULL};

/* Address of error trace (i.e. top of LIFO stack) */
static Gan_ErrorTrace * gan_err_trace_top = &record_2nd_last;
```

The symbol `gan_et_get_record_first()` refers to the current top of stack and is passed into the functions defined in this module as argument 1.

NB. A statically allocated string containing the deep error text message must also exist, but this is defined in `gan_err_trace.c`.

2.10 Error tests and codes

```
#include <gandalf/common/misc_error.h>
```

The Gandalf error package itself is described in Section 2.9. The `misc_error.[ch]` defines some Gandalf-specific error codes.

Chapter 3

The Linear Algebra Package

The **linear algebra** package covers matrix and vector manipulations, matrix decompositions and other operations. To be able to use any function or structure in the linear algebra package use the declaration

```
#include <gandalf/linalg.h>
```

but including individual module header files instead will speed up program compilation. There are two parts to the linear algebra package, one dealing with small fixed size vectors and matrices, between size two and four, and another for general size objects. This separation allows the most efficient implementation of linear algebra operations when the size of the objects is known and small. Being designed to support image- and goemetry-based applications, the size range from two to four allows 2D image and 3D camera/world objects to be manipulated, in homogeneous coordinates where required; thus four is the natural size limit for Gandalf.

A major design feature of the linear algebra package is the application of *implicit operations*. By this is meant, for example, that adding a matrix to the transpose of another matrix is a *one* step operation. Rather than transposing the matrix and then adding it, there is a specific Gandalf routine to apply the operation "add matrix to transpose of another matrix", implemented by indexing the elements of the second matrix in transposed order. This principle increases greatly the number of routines that Gandalf implements, but also greatly increases the efficiency of the package. It can also help to reduce errors, in the case of implicit matrix inverse. Let us say that we want to compute a matrix/vector product where the matrix is to be inverted:

$$\mathbf{y} = A^{-1}\mathbf{x}$$

If A happens to be a diagonal matrix, it makes sense to apply the inverse operation implicitly, inside the product operation. This is because if, for example, we are dealing with vectors & matrix of size 2, we have

$$\mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}, \quad \mathbf{y} = \begin{pmatrix} y_1 \\ y_2 \end{pmatrix}, \quad A = \begin{pmatrix} A_{11} & 0 \\ 0 & A_{22} \end{pmatrix},$$

and the operations required are

$$y_1 \leftarrow \frac{x_1}{A_{11}}, \quad y_2 \leftarrow \frac{x_2}{A_{22}}.$$

Applying the inverse firstly to A and then computing the product would involve the following operations

$$A_{11} \leftarrow \frac{1}{A_{11}}, \quad A_{22} \leftarrow \frac{1}{A_{22}}, \quad y_1 \leftarrow A_{11}x_1, \quad y_2 \leftarrow A_{22}x_2.$$

This has two drawbacks: the two stages of inverting followed by multiplication reduces the accuracy of the result compared to a single division operation, and the A matrix is overwritten with the inverse of A, which is not normally what is wanted (explicit matrix inverse is to be avoided wherever possible). As we shall see, Gandalf implements a comprehensive set of implicit transpose and inverse operations, which apply when the matrix involved is diagonal (as above) or triangular. For these types of matrix the inverse operation can be conjoined with multiplication, such that effectively only one operation is performed. Implicit inverse does not apply to symmetric or general square matrices, because there is no way of conjoining the inverse with multiplication in the same way.

3.1 Fixed size matrices and vectors

3.1.1 Fixed size vectors

Vectors of sizes two, three and four are represented by specific structures in Gandalf. The structure and function definitions are nearly identical, so we shall only describe the workings of 3-vectors. To use 3-vectors include the header file

```
#include <gandalf/linalg/3vector.h>
```

for double precision, or

```
#include <gandalf/linalg/3vectorf.h>
```

for single precision 3-vectors. A double precision 3-vector is defined as

```
typedef struct Gan_Vector3
{
   double x, y, z;
} Gan_Vector3, Gan_Vector3_d
```

and a single precision 3-vector as

```
typedef struct Gan_Vector3_f
{
   float x, y, z;
} Gan_Vector3_f;
```

Most of the routines below return a pointer to the result vector/matrix. This may be used as an argument to another routine, although care must be taken with macros as regards multiple evaluation. The routines are very safe, because everything using fixed size vectors & matrices can be written to involve only automatic variables with no dynamic allocation, and the only failure modes are arithmetic overflow (Gandalf does not check for this). The few exceptions are noted in the text.

3.1.1.1 Creating and accessing fixed size vectors

Single fixed size vectors are such simple objects, it makes sense to normally use declare structure variables directly, rather than use pointers to structures created by `malloc()`. So to create a double precision 3-vector, use the declaration

```
Gan_Vector3 v3x;
```

From now on, we shall describe the functions for double precision vectors only. Single precision functions are very similar and will be explained below. Setting the coordinates of a 3-vector can be achieved by one of

1. Initialising the 3-vector when it is created, as in

   ```
   Gan_Vector3 v3x = {1.0, 2.0, 3.0};
   ```

2. Accessing the structure elements directly:

   ```
   v3x.x = 1.0; v3x.y = 2.0; v3x.z = 3.0;
   ```

3. Using the macro call

```
gan_vec3_fill_q ( &v3x, 1.0, 2.0, 3.0 );
```

   Note that the Gnu C compiler prints a warning when the above call is compiled, and also for most other similar calls in the linear algebra package. This warning can be avoided by inserting an initial `(void)` cast:

```
(void)gan_vec3_fill_q ( &v3x, 1.0, 2.0, 3.0 );
```

   We omit this cast in the following to keep the exposition simple.

4. The equivalent function call

```
v3x = gan_vec3_fill_s ( 1.0, 2.0, 3.0 );
```

Setting a 3-vector to zero can be accomplished using one of the calls

```
gan_vec3_zero_q ( &v3x ); /* macro, OR */
v3x = gan_vec3_zero_s(); /* function call */
```

Copying 3-vectors can be accomplished either by direct assignment

```
Gan_Vector3 v3y;

v3y = v3x;
```

or by use of the one of the routines

```
gan_vec3_copy_q ( &v3x, &v3y ); /* macro, OR */
v3y = gan_vec3_copy_s ( &v3x ); /* function call */
```

3.1.1.2 Fixed size vector addition

To add two 3-vectors use either the macro

```
Gan_Vector3 v3z;

gan_vec3_add_q ( &v3x, &v3y, &v3z ); /* macro */
```

or the function

```
v3z = gan_vec3_add_s ( &v3x, &v3y ); /* function call */
```

See the discussion of "quick" and "slow" versions of the same operation, identified by the `..._q` and `..._s` suffices, in Section 1.1. In this case, the "slow" version `gan_vec3_add_s()` has the overhead of a function call relative to the "quick" version `gan_vec3_add_q()`, so the latter should be used unless the input vectors are not simple variables (i.e. they might be elements of arrays), in which case the repeated evaluation required by the macro version might be slower.

There are also in-place versions of the add operation, which overwrite one of the input vectors with the result. The macro operations

```
gan_vec3_add_i1 ( &v3x, &v3y ); /* result in-place in v3x */
```

and

```
gan_vec3_add_i2 ( &v3x, &v3y ); /* result in-place in v3y */
```

produce the same result but overwrite respectively the first `v3x` and the second `v3y` argument with the result. There is also a more explicit macro

```
gan_vec3_increment ( &v3x, &v3y ); /* result in-place in v3x */
```

which increments `v3x` by `v3y`, i.e. identical to `gan_vec3_add_i1()`. Note that if one of the input arguments is a non-trivial expression, and the result is being overwritten on the other, use the function `gan_vec3_add_s()`, as in

```
Gan_Vector3 av3x[100];

/* ... fill av3x array ... */
v3x = gan_vec3_add_s ( &v3x, &av3x[33] );
```

3.1.1.3 Fixed size vector subtraction

To subtract 3-vectors use the equivalent macros and functions

```
gan_vec3_sub_q ( &v3x, &v3y, &v3z ); /* macro */
v3z = gan_vec3_sub_s ( &v3x, &v3y ); /* function call */
gan_vec3_sub_i1 ( &v3x, &v3y ); /* result in-place in v3x */
gan_vec3_sub_i2 ( &v3x, &v3y ); /* result in-place in v3y */
gan_vec3_decrement ( &v3x, &v3y ); /* result in-place in v3x */
```

3.1.1.4 Rescaling a fixed size vector

There are similar functions to multiply a 3-vector by a scalar

```
double ds;

gan_vec3_scale_q ( &v3x, ds, &v3z ); /* macro */
v3z = gan_vec3_scale_s ( &v3x, ds ); /* function call */
gan_vec3_scale_i ( &v3x, ds ); /* macro, result in-place in v3x */
```

to divide a 3-vector by a (non-zero) scalar

```
gan_vec3_divide_q ( &v3x, ds, &v3z ); /* macro */
v3z = gan_vec3_divide_s ( &v3x, ds ); /* function call */
gan_vec3_divide_i ( &v3x, ds ); /* macro, result in-place in v3x */
```

to negate a 3-vector

```
gan_vec3_negate_q ( &v3x, &v3z ); /* macro */
v3z = gan_vec3_negate_s ( &v3x ); /* function call */
gan_vec3_negate_i ( &v3x ); /* macro, result in-place in v3x */
```

and to scale a 3-vector to unit length (2-norm)

```
gan_vec3_unit_q ( &v3x, &v3z ); /* macro */
v3z = gan_vec3_unit_s ( &v3x ); /* function call */
gan_vec3_unit_i ( &v3x ); /* macro, result in-place in v3x */
```

The last guarantees than the total squared element of the vector returned by `gan_vec3_sqrlen()` will be one; see below.

`Error detection`: If zero is passed as the scalar value to the `..._divide_[qi]()` routines, `NULL` will be returned, while the `..._divide_s()` routines will abort the program. You should add tests for division by zero *before* calling any of the `..._divide_[qsi]()` routines. Similarly, the `..._unit_[qsi]()` routines will fail if the input vector contains only zeros. If this is a possibility then the program should check beforehand.

3.1.1.5 Fixed size vector products

Vector dot product (scalar product) is compute using the alternatives

```
ds = gan_vec3_dot_q ( &v3x, &v3y ); /* macro, or */
ds = gan_vec3_dot_s ( &v3x, &v3y ); /* function call */
```

Similarly, to compute the squared length of a 3-vector, use

```
ds = gan_vec3_sqrlen_q ( &v3x ); /* macro, or */
ds = gan_vec3_sqrlen_s ( &v3x ); /* function call */
```

For 3-vectors only we also have the cross product (vector product)

```
gan_vec3_cross_q ( &v3x, &v3y, &v3z ); /* macro */
v3z = gan_vec3_cross_s ( &v3x, &v3y ); /* function call */
```

There are also outer products formed by two vectors, producing a matrix. These functions are described in Section 3.1.2.7.

3.1.1.6 Fixed size vector file I/O

Gandalf supports both ASCII and binary format file I/O of vectors and matrices. Both formats use standard `FILE *` file streams. ASCII format is obviously more convenient to use, while binary format is more compact and guarantees no loss of precision when the data is read. To print a 3-vector in ASCII format, use

```
Gan_Bool gan_vec3_fprint ( FILE *fp, Gan_Vector3 *p,
                           const char *prefix, int indent, const char *fmt );
```

`prefix` is a prefix string to print before the vector itself, `indent` is the number of spaces to indent the vector by, and `fmt` is a format string to use when printing the vector, e.g. `"%f"`. So for example

```
FILE *pfFile;

pfFile = fopen ( "/tmp/vectors", "w" );
gan_vec3_fill_q ( &v3x, 1.0, 2.0, 3.0 );
gan_vec3_fprint ( pfFile, &v3x, "Example vector", 3, "%f" );
```

will print the output

```
   Example vector: 1.000000 2.000000 3.000000
```

to the file `"/tmp/vectors"`. There is also a version `gan_vec3_print()` for printing to standard output:

```
Gan_Bool gan_vec3_print ( Gan_Vector3 *p,
                          const char *prefix, int indent, const char *fmt );
```

The corresponding input function is

```
Gan_Bool gan_vec3_fscanf ( FILE *fp, Gan_Vector3 *p,
                           char *prefix, int prefix_len )
```

which reads the vector from the file stream `fp` into the 3-vector pointer `p`. It also reads the prefix string (up to the specified maximum length `prefix_len`), which can be compared with the expected prefix string to check for consistency.

Binary file I/O is handled by the functions `gan_vec3_fwrite()` and `gan_vec3_fread()`. To write a 3-vector in binary format use

```
Gan_Bool gan_vec3_fwrite ( FILE *fp, Gan_Vector3 *p, gan_ui32 magic_number );
```

The `magic_number` takes the same role as the `prefix` string in `gan_vec3_fprint()`, and is written into the file so that it can be used later to identify the vector when it is read back using

```
Gan_Bool gan_vec3_fread ( FILE *fp, Gan_Vector3 *p, gan_ui32 *magic_number );
```

`Error detection`: The I/O routines return a boolean value, returning `GAN_TRUE` on success, `GAN_FALSE` on failure, invoking the Gandalf error handle in the latter case.

3.1.1.7 Conversion from general to fixed size vector

Functions are provided to convert a general vector `Gan_Vector` `Gan_Vector3`, provided that the general vector has actually been created with size three. So for instance

```
Gan_Vector *pvx;

pvx = gan_vec_alloc(3);
gan_vec_fill_va ( pvx, 3, 2.0, 3.0, 4.0 );
gan_vec3_from_vec_q ( pvx, &v3x );
```

or

```
v3x = gan_vec3_from_vec_s ( pvx );
```

fill `v3x` with the same values as the general vector. Calling these functions with a general vector `pvx` not having the same size as the fixed size vector is an error.

`Error detection`: The conversion routines return the pointer to the filled fixed size vector, or `NULL` on failure, invoking the Gandalf error handle in the latter case.

3.1.1.8 Single precision fixed size vectors

```
#include <gandalf/linalg/3vectorf.h>
```

There is an identical set of functions for handling single precision floating point vectors, the names of which are obtained by replacing `"gan_vec3_..."` in the above functions with `"gan_vec3f_..."`. For example, to add two single precision 3-vectors use

```
Gan_Vector3_f v3x, v3y, v3z;

/* ... fill v3x and v3y ... */
gan_vec3f_add_q ( &v3x, &v3y, &v3z );
```

3.1.1.9 Other types of fixed size vector

As well as floating point coordinates, integer vectors are often useful. Although Gandalf does not currently provide sets of functions handling vectors with integer coordinates, structures are defined as follows

```
/* structure definition for unsigned character 3-vector */
typedef struct Gan_Vector3_uc
{
   unsigned char x, y, z;
} Gan_Vector3_uc;

/* structure definition for short integer 3-vector */
typedef struct Gan_Vector3_s
{
   short x, y, z;
} Gan_Vector3_s;

/* structure definition for unsigned short integer 3-vector */
typedef struct Gan_Vector3_us
{
   unsigned short x, y, z;
} Gan_Vector3_us;

/* structure definition for integer 3-vector */
typedef struct Gan_Vector3_i
{
   int x, y, z;
} Gan_Vector3_i;

/* structure definition for unsigned integer 3-vector */
typedef struct Gan_Vector3_ui
{
   unsigned int x, y, z;
} Gan_Vector3_ui;
```

3.1.1.10 Other sizes of fixed size vector

Gandalf supports fixed size vectors with sizes two, three and four. The functions described above for size three vectors are repeated for sizes two and four, both single and double precision, in the header files

```
#include <gandalf/linalg/2vector.h>   /* double precision */
#include <gandalf/linalg/2vectorf.h>  /* single precision */
#include <gandalf/linalg/4vector.h>   /* double precision */
#include <gandalf/linalg/4vectorf.h>  /* single precision */
```

3.1.1.11 Setting/extracting parts of fixed size vectors

Apart from the cross product routines `gan_vec3_cross_[qs]()` defined only for 3-vectors, there are a few other miscellaneous routines which apply to a subset of the fixed size vectors. These routines enable setting or extracting parts of a fixed size vector using another fixed size with a different size. The most comprehensive set of such routines is for vectors of size four. So for instance to extract the first three elements of a 4-vector and write them into a 3-vector, use

```
Gan_Vector3 v3x;
Gan_Vector4 v4x;

gan_vec4_fill_q ( &v4x, 1.0, 2.0, 3.0, 4.0 );
gan_vec4_get_v3t_q ( &v4x, &v3x ); /* macro */
```

or alternatively

```
v3x = gan_vec4_get_v3t_s ( &v4x ); /* function */
```

both of which set `v3x` to {`1.0, 2.0, 3.0`}. To build a 4-vector from a 3-vector and a scalar use

```
gan_vec3_fill_q ( &v3x, 1.0, 2.0, 3.0 );
gan_vec4_set_parts_q ( &v4x, &v3x, 4.0 ); /* macro */
```

or alternatively

```
v4x = gan_vec4_set_parts_s ( &v3x, 4.0 ); /* function */
```

both of which set `v4x` to {`1.0, 2.0, 3.0, 4.0`}. To build a 4-vector from two 2-vectors use

```
Gan_Vector3 v2xt, v2xb;
Gan_Vector4 v4x;

gan_vec2_fill_q ( &v2xt,  1.0, 2.0 );
gan_vec2_fill_q ( &v2xb, 3.0, 4.0 );
gan_vec4_set_blocks_q ( &v4x, &v2xt, &v2xb ); /* macro */
```

(note that the "t" and "b" in `v2xt` and `v2xb` stand for the "top" and "bottom" parts of vector **x**), or alternatively

```
v4x = gan_vec4_set_blocks_s ( &v2xt, &v2xb ); /* function */
```

both of which again set `v4x` to {`1.0, 2.0, 3.0, 4.0`}.

For 3-vectors the equivalent set of functions involves splitting the 3-vector into the `x,y` coordinates as a 2-vector and `z` as the scalar. Then we have

```
Gan_Vector2 v2xt;
Gan_Vector3 v3x;

gan_vec3_fill_q ( &v3x, 1.0, 2.0, 3.0 );
gan_vec3_get_v2t_q ( &v3x, &v2xt ); /* macro, or */
v2xt = gan_vec3_get_v2t_s ( &v3x ); /* function */
```

the last two lines of which both set `v2xt` to {`1.0, 2.0`}. To build a 3-vector from a 2-vector and a scalar use

```
gan_vec2_fill_q ( &v2xt, 1.0, 2.0 );
gan_vec3_set_parts_q ( &v3x, &v2xt, 3.0 ); /* macro, or */
v3x = gan_vec3_set_parts_s ( &v2xt, 3.0 ); /* function */
```

both of which set `v3x` to {`1.0, 2.0, 3.0`}.

3.1.2 Fixed size matrices

Matrices of sizes 2×2, 2×3, 2×4, 3×3, 3×4 and 4×4 are represented by specific structures in Gandalf. There is a large set of functions which is repeated across every size of matrix. There is also a set of functions specific to square matrices. In both cases we will choose a single size of matrix and describe the functions available for that size. The sizes we will use are 3×4 matrices for the functions available to every size of matrix, and 3×3 matrices for the functions specific to square matrices.

3.1.2.1 Definitions of fixed size matrice

To use 3×4 matrices or 3×3 matrices include the header files

```
#include <gandalf/linalg/3x4matrix.h> /* OR */
#include <gandalf/linalg/3x3matrix.h>
```

respectively. This is for double precision matrix elements. The files to include for single precision elements are

```
#include <gandalf/linalg/3x4matrixf.h> /* OR */
#include <gandalf/linalg/3x3matrixf.h>
```

A double precision 3×4 matrix is defined as

```
typedef struct Gan_Matrix34
{
   double xx, xy, xz, xw,
          yx, yy, yz, yw,
          zx, zy, zz, zw;
} Gan_Matrix34;
```

A 3×3 matrix is similarly defined as

```
typedef struct Gan_Matrix33
{
   double xx, xy, xz,
          yx, yy, yz,
          zx, zy, zz;
} Gan_Matrix33;
```

For square matrices there is also a specific structure to handle symmetric and triangular matrix structures, as follows:

```
#ifndef NDEBUG
/* square matrix type, for setting and checking in debug mode */
typedef enum { GAN_SYMMETRIC_MATRIX33, GAN_LOWER_TRI_MATRIX33 }
 Gan_SquMatrix33Type;
#endif /* #ifndef NDEBUG */

/* structure definition for square 3x3 matrix */
typedef struct Gan_SquMatrix33
{
#ifndef NDEBUG
   /* square matrix type, for setting and checking in debug mode */
   Gan_SquMatrix33Type type;
```

```
#endif /* #ifndef NDEBUG */

   /* matrix data */
   double xx,
          yx, yy,
          zx, zy, zz;
} Gan_SquMatrix33;
```

Note that the matrix type field `Gan_SquMatrix33` is only used in debug compilation mode (`NDEBUG` not defined). The `type` field is actually invisible to the programmers' interface to the Gandalf functions, and is used merely for internal checking. Note also that Gandalf does not provide explicit support for *upper* triangular fixed size matrices (although it does for general size matrices; see Section 3.2). Any operations involving upper triangular matrices can be implemented using implicit transpose of a lower triangular matrix.

Single precision structures are defined similarly, with the names changed to `Gan_Matrix34_f`, `Gan_SquMatrix33_f` etc.

3.1.2.2 Creating and accessing fixed size matrices

Single fixed size matrices are such simple objects, it makes sense to normally use declare structure variables directly, rather than use pointers to structures created by `malloc()`. So to create a double precision 3×4 matrix, use the declaration

```
Gan_Matrix34 m34A;
```

From now on, we shall describe the routines for double precision matrices only. Single precision functions are very similar and will be explained below. Setting the coordinates of a 3×4 matrix can be achieved by one of

1. Initialising the 3×4 matrix when it is created, as in

   ```
   Gan_Matrix34 m34A = {1.0,  2.0,  3.0,  4.0,
                        5.0,  6.0,  7.0,  8.0,
                        9.0, 10.0, 11.0, 12.0};
   ```

2. Accessing the structure elements directly:

   ```
   m34A.xx = 1.0; m34A.xy = 2.0; m34A.xz = 3.0; /* etc. */
   ```

3. Using the macro call

   ```
   gan_mat34_fill_q ( &m34A, 1.0,  2.0,  3.0,  4.0,
                             5.0,  6.0,  7.0,  8.0,
                             9.0, 10.0, 11.0, 12.0 );
   ```

   Note that the Gnu C compiler prints a warning when the above call is compiled, and also for most other similar calls in the linear algebra package. This warning can be avoided by inserting an initial `(void)` cast:

   ```
   (void)gan_mat34_fill_q ( &m34A, 1.0,  2.0,  3.0,  4.0,
                                   5.0,  6.0,  7.0,  8.0,
                                   9.0, 10.0, 11.0, 12.0 );
   ```

   We omit this cast in the following to keep the exposition simple.

4. The equivalent function call

```
m34A = gan_mat34_fill_s ( 1.0,  2.0,  3.0,  4.0,
                          5.0,  6.0,  7.0,  8.0,
                          9.0, 10.0, 11.0, 12.0 );
```

The methods of initialising a 3×3 matrix follow those listed above for 3×4 matrices, for instance

```
Gan_Matrix33 m33A;

gan_mat33_fill_q ( &m33A, 1.0, 2.0, 3.0,
                          4.0, 5.0, 6.0,
                          7.0, 8.0, 9.0 ); /* OR */
m33A = gan_mat33_fill_s ( 1.0, 2.0, 3.0,
                          4.0, 5.0, 6.0,
                          7.0, 8.0, 9.0 );
```

For a symmetric or lower triangular `Gan_SquMatrix33` matrix, direct initialisation (options 1 and 2 above) is not advisable, because of the `type` field of the structure whose presence depends on `NDEBUG`, Instead use the macro calls

```
Gan_SquMatrix33 sm33S, sm33L;

/* symmetric matrix */
gan_symmat33_fill_q ( &sm33S, 1.0,
                              2.0, 3.0,
                              4.0, 5.0, 6.0 );

/* lower triangular matrix */
gan_ltmat33_fill_q ( &sm33L, 1.0,
                             2.0, 3.0,
                             4.0, 5.0, 6.0 );
```

The first of these fills the matrix without specifying the values above the diagonal, and actually builds the matrix

$$S = \begin{pmatrix} 1 & 2 & 4 \\ 2 & 3 & 5 \\ 4 & 5 & 6 \end{pmatrix}$$

The second builds the lower triangular matrix

$$L = \begin{pmatrix} 1 & 0 & 0 \\ 2 & 3 & 0 \\ 4 & 5 & 6 \end{pmatrix}$$

Setting a fixed-size matrix to zero can be accomplished using one of the calls

```
gan_mat34_zero_q ( &m34A ); /* OR */ m34A = gan_mat34_zero_s();
gan_mat33_zero_q ( &m33A ); /* OR */ m33A = gan_mat33_zero_s();
gan_symmat33_zero_q ( &sm33S ); /* OR */ sm33S = gan_symmat33_zero_s();
gan_ltmat33_zero_q ( &sm33L ); /* OR */ sm33L = gan_ltmat33_zero_s();
```

Setting a square matrix to identity is achieved using

```
gan_mat33_ident_q ( &m33A ); /* OR */ m33A = gan_mat33_ident_s();
gan_symmat33_ident_q ( &sm33S ); /* OR */ sm33S = gan_symmat33_ident_s();
gan_ltmat33_ident_q ( &sm33L ); /* OR */ sm33L = gan_ltmat33_ident_s();
```

Copying 3×4 matrices can be accomplished either by direct assignment

```
Gan_Matrix34 m34B;

m34B = m34A;
```

or by use of one of the routines

```
gan_mat34_copy_q ( &m34A, &m34B ); /* macro, OR */
m34B = gan_mat34_copy_s ( &m34A ); /* function call */
```

The methods of copying general, symmetric and lower triangular matrices follow that of 3×4 matrices.

3.1.2.3 Fixed size matrix addition

To add two 3×4 matrices use either the macro

```
Gan_Matrix34 m34C;

/* ... set up m34A, m34B using e.g. gan_mat34_fill_q() ... */
gan_mat34_add_q ( &m34A, &m34B, &m34C ); /* macro */
```

or the function

```
m34C = gan_mat34_add_s ( &m34A, &m34B ); /* function call */
```

See the discussion of "quick" and "slow" versions of the same operation, identified by the `..._q` and `..._s` suffices, in Section 1.1. In this case, the "slow" version `gan_mat34_add_s()` has the overhead of a function call relative to the "quick" version `gan_mat34_add_q()`, so the latter should be used unless the input matrices are not simple variables (i.e. they might be elements of arrays), in which case the repeated evaluation required by the macro version might be slower.

There are also in-place versions of the add operation, which overwrite one of the input matrices with the result. The macro operations

```
gan_mat34_add_i1 ( &m34A, &m34B ); /* result in-place in m34A */
```

and

```
gan_mat34_add_i2 ( &m34A, &m34B ); /* result in-place in m34B */
```

produce the same result but overwrite respectively the first `m34A` and the second `m34B` argument with the result. There is also a more explicit macro

```
gan_mat34_increment ( &m34A, &m34B ); /* result in-place in m34A */
```

which increments `m34A` by `m34B`, i.e. identical to `gan_mat34_add_i1()`. Note that if one of the input arguments is a non-trivial expression, and the result is being overwritten on the other, use the function `gan_mat34_add_s()`, as in

```
Gan_Matrix34 am34A[100];

/* ... fill am34A array ... */
m34A = gan_mat34_add_s ( &m34A, &am34A[33] );
```

For general, symmetric and lower triangular 3×3 matrices the addition routines follow those for 3×4 matrices. So for general 3×3 matrices we have the options

```
Gan_Matrix33 m33A, m33B, m33C;

/* ... set up m33A, m33B using e.g. gan_mat33_fill_q() ... */
gan_mat33_add_q ( &m33A, &m33B, &m33C ); /* macro, OR */
m33C = gan_mat33_add_s ( &m33A, &m33B ); /* function call */
gan_mat33_add_i1 ( &m33A, &m33B ); /* macro, result in-place in m33A */
gan_mat33_add_i2 ( &m33A, &m33B ); /* macro, result in-place in m33B */
gan_mat33_increment ( &m33A, &m33B ); /* equivalent to gan_mat33_add_i1() */
```

For symmetric 3×3 matrices we have

```
Gan_SquMatrix33 sm33Sa, sm33Sb, sm33Sc;

/* ... set up sm33Sa, sm33Sb using e.g. gan_symmat33_fill_q() ... */
gan_symmat33_add_q ( &sm33Sa, &sm33Sb, &sm33Sc ); /* macro, OR */
sm33Sc = gan_symmat33_add_s ( &sm33Sa, &sm33Sb ); /* function call */
gan_symmat33_add_i1 ( &sm33Sa, &sm33Sb ); /* macro, result in-place in sm33Sa */
gan_symmat33_add_i2 ( &sm33Sa, &sm33Sb ); /* macro, result in-place in sm33Sb */
gan_symmat33_increment ( &sm33Sa, &sm33Sb ); /* equivalent to gan_symmat33_add_i1() */
```

Finally for lower triangular 3×3 matrices we have

```
Gan_SquMatrix33 sm33La, sm33Lb, sm33Lc;

/* ... set up sm33La, sm33Lb using e.g. gan_ltmat33_fill_q() ... */
gan_ltmat33_add_q ( &sm33La, &sm33Lb, &sm33Lc ); /* macro, OR */
sm33Lc = gan_ltmat33_add_s ( &sm33La, &sm33Lb ); /* function call */
gan_ltmat33_add_i1 ( &sm33La, &sm33Lb ); /* macro, result in-place in sm33La */
gan_ltmat33_add_i2 ( &sm33La, &sm33Lb ); /* macro, result in-place in sm33Lb */
gan_ltmat33_increment ( &sm33La, &sm33Lb ); /* equivalent to gan_ltmat33_add_i1() */
```

For general square matrices there are routines to implement the operation

$$S = A + A^{\top},$$

obtaining a symmetric matrix S by adding together a general square matrix A and its transpose. The routines for 3×3 matrices are

```
Gan_Matrix33 m33A;
Gan_SquMatrix33 sm33S;

/* set up m33A using e.g. gan_mat33_fill_q() */
gan_mat33_saddT_q ( &m33A, &sm33S ); /* S = A+A^T, macro */
sm33S = gan_mat33_saddT_s ( &m33A ); /* S = A+A^T, function call */
```

3.1.2.4 Fixed size matrix subtraction

To subtract 3×4 matrices use the equivalent macros and functions

```
gan_mat34_sub_q ( &m34A, &m34B, &m34C ); /* macro */
```

```
m34C = gan_mat34_sub_s ( &m34A, &m34B ); /* function call */
gan_mat34_sub_i1 ( &m34A, &m34B ); /* result in-place in m34A */
gan_mat34_sub_i2 ( &m34A, &m34B ); /* result in-place in m34B */
gan_mat34_decrement ( &m34A, &m34B ); /* result in-place in m34A */
```

For general 3×3 matrices we have the options

```
Gan_Matrix33 m33A, m33B, m33C;

/* ... set up m33A, m33B using e.g. gan_mat33_fill_q() ... */
gan_mat33_sub_q ( &m33A, &m33B, &m33C ); /* macro, OR */
m33C = gan_mat33_sub_s ( &m33A, &m33B ); /* function call */
gan_mat33_sub_i1 ( &m33A, &m33B ); /* macro, result in-place in m33A */
gan_mat33_sub_i2 ( &m33A, &m33B ); /* macro, result in-place in m33B */
gan_mat33_decrement ( &m33A, &m33B ); /* equivalent to gan_mat33_sub_i1() */
```

For symmetric 3×3 matrices we have

```
Gan_SquMatrix33 sm33Sa, sm33Sb, sm33Sc;

/* ... set up sm33Sa, sm33Sb using e.g. gan_symmat33_fill_q() ... */
gan_symmat33_sub_q ( &sm33Sa, &sm33Sb, &sm33Sc ); /* macro, OR */
sm33Sc = gan_symmat33_sub_s ( &sm33Sa, &sm33Sb ); /* function call */
gan_symmat33_sub_i1 ( &sm33Sa, &sm33Sb ); /* macro, result in-place in sm33Sa */
gan_symmat33_sub_i2 ( &sm33Sa, &sm33Sb ); /* macro, result in-place in sm33Sb */
gan_symmat33_decrement ( &sm33Sa, &sm33Sb ); /* equivalent to gan_symmat33_sub_i1() */
```

Finally for lower triangular 3×3 matrices we have

```
Gan_SquMatrix33 sm33La, sm33Lb, sm33Lc;

/* ... set up sm33La, sm33Lb using e.g. gan_ltmat33_fill_q() ... */
gan_ltmat33_sub_q ( &sm33La, &sm33Lb, &sm33Lc ); /* macro, OR */
sm33Lc = gan_ltmat33_sub_s ( &sm33La, &sm33Lb ); /* function call */
gan_ltmat33_sub_i1 ( &sm33La, &sm33Lb ); /* macro, result in-place in sm33La */
gan_ltmat33_sub_i2 ( &sm33La, &sm33Lb ); /* macro, result in-place in sm33Lb */
gan_ltmat33_decrement ( &sm33La, &sm33Lb ); /* equivalent to gan_ltmat33_sub_i1() */
```

3.1.2.5 Rescaling a fixed size matrix

There are similar functions to multiply a 3×4 matrix by a scalar

```
double ds;

gan_mat34_scale_q ( &m34A, ds, &m34C ); /* macro */
m34C = gan_mat34_scale_s ( &m34A, ds ); /* function call */
gan_mat34_scale_i ( &m34A, ds ); /* macro, result in-place in m34A */
```

to divide a 3×4 matrix by a (non-zero) scalar

```
gan_mat34_divide_q ( &m34A, ds, &m34C ); /* macro */
m34C = gan_mat34_divide_s ( &m34A, ds ); /* function call */
gan_mat34_divide_i ( &m34A, ds ); /* macro, result in-place in m34A */
```

to negate a 3×4 matrix

```
gan_mat34_negate_q ( &m34A, &m34C ); /* macro */
m34C = gan_mat34_negate_s ( &m34A ); /* function call */
gan_mat34_negate_i ( &m34A ); /* macro, result in-place in m34A */
```

and to scale a 3×4 matrix to unit unit Frobenius norm

```
gan_mat34_unit_q ( &m34A, &m34C ); /* macro */
m34C = gan_mat34_unit_s ( &m34A ); /* function call */
gan_mat34_unit_i ( &m34A ); /* macro, result in-place in m34A */
```

The Frobenius norm of a matrix is the square-root of the sum of squares of the matrix elements. The Gandalf functions for computing it are described in 3.1.2.11.

Equivalent routiness to the above for multiplying/dividing a matrix by a scalar, negating a matrix and scaling a matrix to unit Frobenius norm are available for square fixed size matrices. Without listing the routines exhaustively, some examples are

```
gan_mat33_scale_q ( &m33A, ds, &m33C ); /* macro */
sm33Sc = gan_symmat33_divide_s ( &sm33Sa, ds ); /* function call */
gan_ltmat33_negate_i ( &sm33La ); /* macro, result in-place in sm33La */
m33C = gan_mat33_unit_s ( &m33A ); /* function call */
```

`Error detection`: If zero is passed as the scalar value to the `..._divide_[qi]()` routines, `NULL` will be returned, while the `..._divide_s()` routines will abort the program. You should add tests for division by zero *before* calling any of the `..._divide_[qsi]()` routines. Similarly, the `..._unit_[qsi]()` routines will fail if the input matrix contains only zeros. If this is a possibility then the program should check beforehand.

3.1.2.6 Transposing a fixed size matrix

Explicit matrix transposition is not often required in Gandalf, because of the extensive support for implicit transpose in other matrix operations. However where necessary, computing the transpose can be achieved using

```
Gan_Matrix33 m33A, m33B;

/* set up m33A using e.g. gan_mat33_fill_q() */
gan_mat33_tpose_q ( &m33A, &m33B ); /* B = A^T */
m33B = gan_mat33_tpose_s ( &m33A ); /* B = A^T */
gan_mat33_tpose_i ( &m33A ); /* A = A^T, result in-place in A */
```

3.1.2.7 Fixed size vector outer products

The outer product operation on two fixed size vectors $\mathbf{x}$ and $\mathbf{y}$ is the matrix

$$A = \mathbf{x}\mathbf{y}^\top$$

In Gandalf, any pair of vectors can be combined in this way[1]. The operation on a 3-vector and a 4-vector produces a 3×4 matrix, using either of the routines

```
Gan_Vector3 v3x;
Gan_Vector4 v4y;
```

[1] So long as $\mathbf{x}$ and $\mathbf{y}$ are ordered so that the size of $\mathbf{x}$ is less than or equal to the size of $\mathbf{y}$.

```
Gan_Matrix34 m34A;

/* ... set up v3x and v4y using e.g. gan_vec[34]_fill_q() ... */
gan_vec34_outer_q ( &v3x, &v4y, &m34A ); /* macro, or */
m34A = gan_vec34_outer_s ( &v3x, &v4y ); /* function call */
```

Similar routines `gan_vec33_outer_[qs]()` enable the computation of the outer product of two 3-vectors, resulting in a 3×3 matrix. If $\mathbf{x} = \mathbf{y}$, the result of the outer product is a symmetric matrix

$$S = \mathbf{x}\mathbf{x}^\top$$

Gandalf has special routines for this case:

```
Gan_Vector3 v3x;
Gan_SquMatrix33 sm33S;

/* ... set up v3x using e.g. gan_vec3_fill_q() ... */
gan_vec33_outer_sym_q ( &v3x, &sm33S ); /* macro, or */
sm33S = gan_vec33_outer_sym_s ( &v3x ); /* function call */
```

3.1.2.8 Fixed size matrix/vector multiplication

Gandalf supports matrix/vector multiplication with the matrix optionally being (implicitly) transposed. If the matrix is triangular, Gandalf also supports multiplication by the inverse of the matrix, computed implicitly as described in the introduction to this chapter. These operations can be written as

$$\begin{aligned} \mathbf{y} &= A\mathbf{x} \quad \text{OR} \quad \mathbf{y} = A^\top\mathbf{x} \quad \text{OR} \\ \mathbf{y} &= A^{-1}\mathbf{x} \quad \text{OR} \quad \mathbf{y} = A^{-\top}\mathbf{x} \quad \text{(triangular } A \text{ only)} \end{aligned}$$

The Gandalf routines involving a 3×4 matrix for the first of these (no transpose of A) is

```
Gan_Vector4 v4x;
Gan_Vector3 v3y;
Gan_Matrix34 m34A;

/* ... set up m34A and v4x ... */
gan_mat34_multv4_q ( &m34A, &v4x, &v3y ); /* macro, or */
v3y = gan_mat34_multv4_s ( &m34A, &v4x ); /* function call */
```

while if A is to be transposed then the routines are

```
Gan_Vector3 v3x;
Gan_Vector4 v4y;
Gan_Matrix34 m34A;

/* ... set up m34A and v3x ... */
gan_mat34T_multv3_q ( &m34A, &v3x, &v4y ); /* macro, or */
v4y = gan_mat34T_multv3_s ( &m34A, &v3x ); /* function call */
```

There are similar routines `gan_mat33_multv3_[qs]()` and `gan_mat33T_multv3_[qs]()` for A being a general 3×3 matrix. For symmetric matrices, there is only one pair of routines:

```
Gan_Vector3 v3x, v3y;
Gan_SquMatrix33 sm33S;
```

```
/* ... set up sm33S using e.g. gan_symmat33_fill_q(), and v3x ... */
gan_symmat33_multv3_q ( &sm33S, &v3x, &v3y ); /* macro, or */
v3y = gan_symmat33_multv3_s ( &sm33S, &v3x ); /* function call */
```

In the case that matrix A is triangular, Gandalf also supports multiplication of vectors by the inverse of A. In this case the result may be computed in-place in the input vector, So the complete set of matrix/vector multiplication routines for triangular matrices is

```
Gan_Vector3 v3x, v3y;
Gan_SquMatrix33 sm33L;

/* ... set up sm33L using e.g. gan_ltmat33_fill_q(), and v3x ... */

/* multiply vector by lower triangular matrix */
gan_ltmat33_multv3_q ( &sm33L, &v3x, &v3y ); /* macro, or */
v3y = gan_ltmat33_multv3_s ( &sm33L, &v3x ); /* function call, or */
gan_ltmat33_multv3_i ( &sm33L, &v3x ); /* macro, in-place in v3x */

/* multiply vector by upper triangular matrix */
gan_ltmat33T_multv3_q ( &sm33L, &v3x, &v3y ); /* macro, or */
v3y = gan_ltmat33T_multv3_s ( &sm33L, &v3x ); /* function call, or */
gan_ltmat33T_multv3_i ( &sm33L, &v3x ); /* macro, in-place in v3x */

/* multiply vector by inverse of lower triangular matrix */
gan_ltmat33I_multv3_q ( &sm33L, &v3x, &v3y ); /* macro, or */
v3y = gan_ltmat33I_multv3_s ( &sm33L, &v3x ); /* function call, or */
gan_ltmat33I_multv3_i ( &sm33L, &v3x ); /* macro, in-place in v3x */

/* multiply vector by inverse of upper triangular matrix */
gan_ltmat33IT_multv3_q ( &sm33L, &v3x, &v3y ); /* macro, or */
v3y = gan_ltmat33IT_multv3_s ( &sm33L, &v3x ); /* function call, or */
gan_ltmat33IT_multv3_i ( &sm33L, &v3x ); /* macro, in-place in v3x */
```

Error detection: If implicit inverse is used (the `...I_multv...()` or `...IT_multv...()` routines), the matrix must be non-singular, which for triangular matrices means that none of the diagonal elements should be zero. If the matrix was produced by successful Cholesky factorisation of a symmetric matrix (see Section 3.1.2.12) the matrix is guaranteed to be non-singular. This is the normal method of creating a triangular matrix, and Gandalf uses `assert()` to check for the singularity of the matrix.

3.1.2.9 Fixed size matrix/matrix multiplication

Most useful matrix product combinations are supported by Gandalf. Here we describe all the combinations involving 3×4 matrices. The first functions to describe are those which involve multiplication by a 3×3 on the left or 4×4 matrix on the right, the square matrix optionally being (implicitly) transposed, the product producing another 3×4 matrix. The operator combinations are

$$D_{3\times4} = B_{3\times3}A_{3\times4}, \quad D_{3\times4} = B_{3\times3}^{\top}A_{3\times4}, \quad D_{3\times4} = A_{3\times4}C_{4\times4}, \quad D_{3\times4} = A_{3\times4}C_{4\times4}^{\top}$$

which are implemented in Gandalf using the macros

```
Gan_Matrix34 m34A, m34D;
Gan_Matrix33 m33B;
Gan_Matrix44 m44C;
```

```
/* ... set up m34A, m33B and m44C ... */
gan_mat34_lmultm33_q  ( &m34A, &m33B, &m34D ); /* D = B*A   */
gan_mat34_lmultm33T_q ( &m34A, &m33B, &m34D ); /* D = B*A^T */
gan_mat34_rmultm44_q  ( &m34A, &m44C, &m34D ); /* D = A*C   */
gan_mat34_rmultm44T_q ( &m34A, &m44C, &m34D ); /* D = A*C^T */
```

Equivalent function calls are available:

```
m34D = gan_mat34_lmultm33_s  ( &m34A, &m33B ); /* D = B*A   */
m34D = gan_mat34_lmultm33T_s ( &m34A, &m33B ); /* D = B*A^T */
m34D = gan_mat34_rmultm44_s  ( &m34A, &m44C ); /* D = A*C   */
m34D = gan_mat34_rmultm44T_s ( &m34A, &m44C ); /* D = A*C^T */
```

Note that although by and large the functions described here for 3×4 matrices are repeated for square matrices, there is redundancy because in the case of 3×3 matrices the routines

```
m33D = gan_mat33_lmultm33_s  ( &m33A, &m33B ); /* D = B*A   */
m33D = gan_mat33_rmultm33_s  ( &m33B, &m33A ); /* D = B*A   */
```

would be equivalent, so in fact only the routines `gan_mat33_rmultm33_[qs]()` are defined.

The square matrix may be symmetric or triangular, for which cases there are specific Gandalf functions. Firstly for multiplying by symmetric matrices we have the routines

```
Gan_Matrix34 m34A, m34B;
Gan_SquMatrix33 sm33S;
Gan_SquMatrix44 sm44S;

/* ... set up m34A, symmetric sm33S and sm44S ... */
gan_mat34_lmults33_q ( &m34A, &sm33S, &m34B ); /* B = S*A, macro */
gan_mat34_rmults44_q ( &m34A, &sm44S, &m34B ); /* B = A*S, macro */
```

with equivalent function calls

```
m34B = gan_mat34_lmults33_q ( &m34A, &sm33S ); /* B = S*A, function call */
m34B = gan_mat34_rmults44_q ( &m34A, &sm44S ); /* B = A*S, function call */
```

When multiplying by a triangular matrix, there are also options of implicit transpose and inverse, as described in the introduction to this chapter. Gandalf also supports in-place operations in this case. So there is a whole family of functions covering multiplication of a matrix by a triangular matrix. Mathematically the operations are

$$B = LA, \quad B = L^\top A, \quad B = L^{-1}A, \quad B = L^{-\top}A$$

$$B = AL, \quad B = AL^\top, \quad B = AL^{-1}, \quad B = AL^{-\top}$$

Gandalf macro routines to implement these operations are

```
Gan_Matrix34 m34A, m34B;
Gan_SquMatrix33 sm33L;
Gan_SquMatrix44 sm44L;

/* ... set up m34A, lower triangular sm33L and sm44L ... */
gan_mat34_lmultl33_q   ( &m34A, &sm33L, &m34B ); /* B = L*A,    macro */
gan_mat34_lmultl33T_q  ( &m34A, &sm33L, &m34B ); /* B = L^T*A,  macro */
```

```
gan_mat34_lmultl33I_q  ( &m34A, &sm33L, &m34B ); /* B = L^-1*A, macro */
gan_mat34_lmultl33IT_q ( &m34A, &sm33L, &m34B ); /* B = L^-T*A, macro */
gan_mat34_rmultl44_q   ( &m34A, &sm44L, &m34B ); /* B = A*L,    macro */
gan_mat34_rmultl44T_q  ( &m34A, &sm44L, &m34B ); /* B = A*L^T,  macro */
gan_mat34_rmultl44I_q  ( &m34A, &sm44L, &m34B ); /* B = A*L^-1, macro */
gan_mat34_rmultl44IT_q ( &m34A, &sm44L, &m34B ); /* B = A*L^-T, macro */
```

There are also function calls to implement the same operations:

```
m34B = gan_mat34_lmultl33_s   ( &m34A, &sm33L ); /* B = L*A,    function call */
m34B = gan_mat34_lmultl33T_s  ( &m34A, &sm33L ); /* B = L^T*A,  function call */
m34B = gan_mat34_lmultl33I_s  ( &m34A, &sm33L ); /* B = L^-1*A, function call */
m34B = gan_mat34_lmultl33IT_s ( &m34A, &sm33L ); /* B = L^-T*A, function call */
m34B = gan_mat34_rmultl44_s   ( &m34A, &sm44L ); /* B = A*L,    function call */
m34B = gan_mat34_rmultl44T_s  ( &m34A, &sm44L ); /* B = A*L^T,  function call */
m34B = gan_mat34_rmultl44I_s  ( &m34A, &sm44L ); /* B = A*L^-1, function call */
m34B = gan_mat34_rmultl44IT_s ( &m34A, &sm44L ); /* B = A*L^-T, function call */
```

Finally there is a set of macros for writing the result in-place into the 3×4 matrix A.

```
gan_mat34_lmultl33_i   ( &m34A, &sm33L ); /* A = L*A,    macro */
gan_mat34_lmultl33T_i  ( &m34A, &sm33L ); /* A = L^T*A,  macro */
gan_mat34_lmultl33I_i  ( &m34A, &sm33L ); /* A = L^-1*A, macro */
gan_mat34_lmultl33IT_i ( &m34A, &sm33L ); /* A = L^-T*A, macro */
gan_mat34_rmultl44_i   ( &m34A, &sm44L ); /* A = A*L,    macro */
gan_mat34_rmultl44T_i  ( &m34A, &sm44L ); /* A = A*L^T,  macro */
gan_mat34_rmultl44I_i  ( &m34A, &sm44L ); /* A = A*L^-1, macro */
gan_mat34_rmultl44IT_i ( &m34A, &sm44L ); /* A = A*L^-T, macro */
```

The next set of functions deals with multiplying a matrix by itself in transpose, resulting in a symmetric matrix. These operations have the form

$$S_{4\times4} = A^{\top}_{3\times4} A_{3\times4} \quad \text{OR} \quad S_{3\times3} = A_{3\times4} A^{\top}_{3\times4}$$

The Gandalf macro routines to implement these operations are

```
Gan_Matrix34 m34A;
Gan_SquMatrix33 sm33S;
Gan_SquMatrix44 sm44S;

/* ... set up m34A using e.g. gan_mat34_fill_q() ... */
gan_mat34_slmultT_q ( &m34A, &sm44S ); /* S = A^T*A */
gan_mat34_srmultT_q ( &m34A, &sm33S ); /* S = A*A^T */
```

with equivalent function calls

```
sm44S = gan_mat34_slmultT_s ( &m34A ); /* S = A^T*A */
sm33S = gan_mat34_srmultT_s ( &m34A ); /* S = A*A^T */
```

There are similar routines for general 3×3 matrices. For triangular matrices the functions are

```
Gan_SquMatrix33 sm33L, sm33S;

/* ... set up sm33L using e.g. gan_ltmat33_fill_q()... */
```

```
gan_ltmat33_slmultT_q ( &sm33L, &sm33S ); /* S = L^T*L, macro */
sm33S = gan_ltmat33_slmultT_s ( &sm33L ); /* S = L^T*L, function call */
gan_ltmat33_srmultT_q ( &sm33L, &sm33S ); /* S = L*L^T, macro */
sm33S = gan_ltmat33_srmultT_s ( &sm33L ); /* S = L*L^T, function call */
```

In the case of triangular matrices the operation "multiply by transposed self" can also be done in-place, so we also have the macro routines

```
Gan_SquMatrix33 sm33A;

/* ... set up sm33A as triangular using e.g. gan_ltmat33_fill_q()... */
gan_ltmat33_slmultT_i ( &sm33A ); /* A = A^T*A, macro */
gan_ltmat33_srmultT_i ( &sm33A ); /* A = A*A^T, macro */
```

There are also routines to multiply a matrix by the transpose of another matrix of the same size, where the result is *assumed* to be a symmetric matrix. So mathematically the operations have the form

$$S_{4\times4} = A^{\top}_{3\times4} B_{3\times4} \quad \text{OR} \quad S_{3\times3} = A_{3\times4} B^{\top}_{3\times4}$$

The Gandalf macro routines to implement these operations are

```
Gan_Matrix34 m34A, m34B;
Gan_SquMatrix33 sm33S;
Gan_SquMatrix44 sm44S;

/* ... set up m34A, m34B using e.g. gan_mat34_fill_q() ... */
gan_mat34_rmultm34T_sym_q ( &m34A, &m34B, &sm33S ); /* S = A*B^T */
gan_mat34_lmultm34T_sym_q ( &m34B, &m34A, &sm44S ); /* S = A^T*B */
```

with equivalent function calls

```
sm33S = gan_mat34_rmultm34T_sym_s ( &m34A, &m34B ); /* S = A*B^T */
sm44S = gan_mat34_lmultm34T_sym_s ( &m34B, &m34A ); /* S = A^T*B */
```

A common operation is to multiply a symmetric matrix on left and right by a matrix and its transpose, producing another symmetric matrix. Gandalf supports all combinations of these operations. Those involving 3×4 matrices are

```
Gan_SquMatrix33 sm33Sa;
Gan_SquMatrix44 sm44Sb;
Gan_Matrix34 m34A;

gan_symmat33_lrmultm34T_q ( &sm33Sa, &m34A, &sm44Sb ); /* Sb = A^T*Sa*A, macro */
sm44Sb = gan_symmat33_lrmultm34T_s ( &sm33Sa, &m34A ); /* Sb = A^T*Sa*A, function call */
gan_symmat44_lrmultm34_q ( &sm44Sb, &m34A, &sm33Sa ); /* Sa = A*Sb*A^T, macro */
sm33Sa = gan_symmat44_lrmultm34_s ( &sm44Sb, &m34A ); /* Sa = A*Sb*A^T, function call */
```

Error detection: If implicit inverse is used (e.g. the `...mult133I_[qsi]()` or `...mult133IT_[qsi]()` routines), the matrix must be non-singular, which for triangular matrices means that none of the diagonal elements should be zero. If the matrix was produced by successful Cholesky factorisation of a symmetric matrix (see Section 3.1.2.12) the matrix is guaranteed to be non-singular. This is the normal method of creating a triangular matrix, and Gandalf uses `assert()` to check for the singularity of the matrix.

3.1.2.10 Fixed size matrix inverse

All types of square matrices can be inverted in Gandalf. If the matrix is singular, `NULL` is normally returned and an error condition set using `gan_err_register()` (see Section 2.9). The routines to invert a 3×3 matrix are

```
Gan_Matrix33 m33A, m33B;

/* ... set up m33A using e.g. gan_mat33_fill_q() ... */
gan_mat33_invert_q ( &m33A, &m33B ); /* B = A^-1, OR */
m33B = gan_mat33_invert_s ( &m33A ); /* B = A^-1, OR */
gan_mat33_invert_i ( &m33A ); /* A = A^-1, in-place in A */
```

Note that the routine `gan_mat33_invert_s()` returns the structure, rather than a pointer value, so cannot return an error condition. Do not use it unless you are SURE that your matrix is non-singular! The basic routines `gan_mat33_invert_[qs]()` are implemented as functions rather than macros, because they require temporary variables. There are similar routines for symmetric matrices

```
Gan_SquMatrix33 sm33Sa, sm33Sb;

/* ... set up sm33Sa using e.g. gan_symmat33_fill_q() ... */
gan_symmat33_invert_q ( &sm33Sa, &sm33Sb ); /* Sb = Sa^-1, OR */
sm33Sb = gan_symmat33_invert_s ( &sm33Sa ); /* Sb = Sa^-1, OR */
gan_symmat33_invert_i ( &sm33Sa ); /* Sa = Sa^-1, in-place in Sa */
```

and for triangular matrices

```
Gan_SquMatrix33 sm33La, sm33Lb;

/* ... set up sm33La using e.g. gan_ltmat33_fill_q() ... */
gan_ltmat33_invert_q ( &sm33La, &sm33Lb ); /* Lb = La^-1, OR */
sm33Lb = gan_ltmat33_invert_s ( &sm33La ); /* Lb = La^-1, OR */
gan_ltmat33_invert_i ( &sm33La ); /* La = La^-1, in-place in La */
```

If you don't want to invoke the error package when inversion is attempted on a singular matrix, there is a set of routines which allows to instead to return the error condition as part of the result. For instance the code fragment

```
Gan_Matrix33 m33A, m33B;
int error_code;

/* ... set up m33A using e.g. gan_mat33_fill_q() ... */
if ( gan_mat33_invert ( &m33A, &m33B, &error_code ) == NULL )
{
   /* error found, act on it ... */
}

/* no error found */
```

attempts to invert matrix A, and if an error is found, returns NULL, with the error condition returned in the `error_code` variable. For singular matrices the error condition is `GAN_ERROR_SINGULAR_MATRIX`. There are similar routines `gan_symmat33_invert()` and `gan_ltmat33_invert()` for symmetric and triangular matrices respectively.

3.1.2.11 Determinant, trace, norms of fixed size matrix

To compute the determinant of a square matrix use one of the routines

```
Gan_Matrix m33A;
Gan_SquMatrix33 sm33S, sm33L;
double dDetA, dDetS, dDetL;

/* set up m33A, sm33S as symmetric and sm33L as lower triangular */
dDetA = gan_mat33_det_q(&m33A); /* macro computing det(A) */
dDetA = gan_mat33_det_s(&m33A); /* function computing det(A) */
dDetS = gan_symmat33_det_q(&sm33S); /* macro computing det(S) */
dDetS = gan_symmat33_det_s(&sm33S); /* function computing det(S) */
dDetL = gan_ltmat33_det_q(&sm33L); /* macro computing det(L) */
dDetL = gan_ltmat33_det_s(&sm33L); /* function computing det(L) */
```

The routines to compute the trace of a square matrix are similar:

```
Gan_Matrix m33A;
Gan_SquMatrix33 sm33S, sm33L;
double dTraceA, dTraceS, dTraceL;

/* set up m33A, sm33S as symmetric and sm33L as lower triangular */
dTraceA = gan_mat33_trace_q(&m33A); /* macro computing trace(A) */
dTraceA = gan_mat33_trace_s(&m33A); /* function computing trace(A) */
dTraceS = gan_symmat33_trace_q(&sm33S); /* macro computing trace(S) */
dTraceS = gan_symmat33_trace_s(&sm33S); /* function computing trace(S) */
dTraceL = gan_ltmat33_trace_q(&sm33L); /* macro computing trace(L) */
dTraceL = gan_ltmat33_trace_s(&sm33L); /* function computing trace(L) */
```

For all types of fixed size matrix there are also routines to compute the sum of squares of the matrix elements, as well as the Frobenius norm, which is the square-root of the sum of squares. For 3×4 matrices the routines are

```
Gan_Matrix34 m34A;
double dSumSA, dFNormA;

/* ... set up m34A using e.g. gan_mat34_fill_q() ... */
dSumSA = gan_mat34_sumsqr_q(&m34A); /* macro computing sum(A_ij^2) */
dSumSA = gan_mat34_sumsqr_s(&m34A); /* function computing sum(A_ij^2) */
dFNormA = gan_mat34_Fnorm_q(&m34A); /* macro computing sqrt(sum(A_ij^2)) */
dFNormA = gan_mat34_Fnorm_s(&m34A); /* function computing sqrt(sum(A_ij^2)) */
```

There are equivalent routines for other types of matrix.

3.1.2.12 Fixed size matrix decompositions

Gandalf supports several of the standard matrix decompositions. Cholesky factorisation applies to any positive definite symmetric matrix S, producing the lower triangular matrix L so that

$$S = LL^\top$$

It can be computed for 3×3 symmetric matrices using the routines

```
Gan_SquMatrix33 sm33S, sm33L;

/* ... set up sm33S using e.g. gan_symmat33_fill_q() ... */
gan_symmat33_cholesky_q ( &sm33S, &sm33L ); /* L = chol(S) */
sm33L = gan_symmat33_cholesky_s ( &sm33S ); /* L = chol(S) */
```

There is also a routine for computing the Cholesky factorisation in-place in the input matrix, converting an input symmetric matrix A into a lower triangular matrix:

```
Gan_SquMatrix33 sm33A;

/* ... set up sm33A as symmetric using e.g. gan_symmat33_fill_q() ... */
gan_symmat33_cholesky_i ( &sm33A ); /* A = chol(A) */
```

The routines `gan_symmat33_cholesky_[qi]()` return `NULL` and invoke the Gandalf error handler `gan_err_register()` if the matrix is not positive definite (`gan_symmat33_cholesky_s()` aborts the program on error, so don't use it unless you're SURE your matrix is OK!). If you don't want to invoke the error package when factorisation is attempted on a non-positive-definite matrix, there is a set of routines which allows to instead to return the error condition as part of the result. For instance the code fragment

```
Gan_SquMatrix33 sm33S, sm33L;
int error_code;

/* ... set up sm33S using e.g. gan_symmat33_fill_q() ... */
if ( gan_symmat33_choleksy ( &sm33S, &sm33L, &error_code ) == NULL )
{
   /* error found, act on it ... */
}

/* no error found */
```

attempts to factorise matrix S, and if an error is found, returns NULL, with the error condition returned in the `error_code` variable. For non-positive-definite matrices the error condition is `GAN_ERROR_NOT_POSITIVE_DEFINITE`.

Other factorisations are available in Gandalf. Singular value decomposition (SVD) can be used to decompose almost any matrix A into factors as

$$A = UWV^\top$$

where U and V are orthogonal matrices and W is diagonal. Currently Gandalf supports SVD for 3×3 and 4×4 matrices. To use the functions for 3×3 matrices, include the header file

```
#include <gandalf/linalg/3x3matrix_svd.h>
```

There are routines for SVD of a matrix or its transpose, as follows

```
Gan_Matrix33 m33A, m33U, m33VT;
Gan_Vector3 v3W;

/* ... set up m33A using e.g. gan_mat33_fill_q() ... */
gan_mat33_svd ( &m33A, &m33U, &v3W, &m33VT ); /* A = U*W*V^T, OR */
gan_mat33T_svd ( &m33A, &m33U, &v3W, &m33VT ); /* A^T = U*W*V^T */
```

These routines return a `Gan_Bool` result, which is `GAN_TRUE` on success and `GAN_FALSE` on failure.

There are also a number of routines for computing the eigenvalues and eigenvectors of fixed size matrices. For 3×3 matrices only there is a routine to compute the eigenvectors and complex eigenvalues of a 3×3 matrix. To use the routine include the header file

```
#include <gandalf/linalg/3x3matrix_eigen.h>
```

The matrix A has "left" and "right" eigenvectors associated with the same eigenvalues λ_i, so that the equation

$$A\mathbf{v}_i = \lambda_i \mathbf{v}_i$$

defines the right eigenvectors, and

$$\mathbf{u}_i^H A = \lambda_i \mathbf{u}_i^H$$

defines the left eigenvectors, where $\mathbf{u}_i^H$ denotes the conjugate transpose of vector $\mathbf{u}_i$. The computed eigenvectors are normalized to have Euclidean norm equal to one and largest component real. The Gandalf routine that implements this is built on the equivalent CLAPACK routine `dgeev()`:

```
Gan_Matrix33 m33A; /* matrix to be decomposed */
Gan_Matrix33 m33UL, m33VR; /* left and right eigenvectors */
Gan_Vector3 v3lr, v3li; /* real and imaginary parts of eigenvalues */

/* ... set up m33A using e.g. gan_mat33_fill_q() ... */
gan_mat33_eigen ( &m33A, &v3lr, &v3li, &m33UL, &m33VR );
```

The eigenvalues of symmetric matrices are guaranteed to be real. Routines are available for computing the eigenvalues and eigenvectors of 3×3 and 4×4 symmetric matrices, based on either the CLAPACK routine `dspev()` or the CCMath routine `eigval()`. For 3×3 matrices the routine is declared in the header file

```
#include <gandalf/linalg/3x3matrix_eigsym.h>
```

Here is an example using the routine

```
Gan_SquMatrix33 sm33S; /* symmetric matrix to be decomposed */
Gan_Matrix33 m33Z; /* (right) eigenvectors of A */
Gan_Vector3 v3W; /* eigenvalues */

/* ... set up sm33S using e.g. gan_symmat33_fill_q() ... */
gan_symmat33_eigen ( &sm33S, &v3W, &m33Z );
```

3.1.2.13 Fixed size matrix file I/O

Gandalf supports both ASCII and binary format file I/O of matrices. Both formats use standard `FILE *` file streams. ASCII format is obviously more convenient to use, while binary format is more compact and guarantees no loss of precision when the data is read. To print a 3×4 matrix in ASCII format, use

```
Gan_Bool gan_mat34_fprint ( FILE *fp, Gan_Matrix34 *p,
                            const char *prefix, int indent, const char *fmt );
```

`prefix` is a prefix string to print before the matrix itself, `indent` is the number of spaces to indent the matrix by, and `fmt` is a format string to use when printing the matrix, e.g. `"%f"`. So for example

```
FILE *pfFile;

pfFile = fopen ( "/tmp/matrices", "w" );
gan_mat34_fill_q ( &m34A, 1.0,  2.0,  3.0,  4.0,
                          5.0,  6.0,  7.0,  8.0,
                          9.0, 10.0, 11.0, 12.0 );
gan_mat34_fprint ( pfFile, &m34A, "Example matrix", 3, "%f" );
```

will print the output

```
Example matrix
 1.000000 2.000000 3.000000 4.000000
 5.000000 6.000000 7.000000 8.000000
 9.000000 10.000000 11.000000 12.000000
```

to the file "/tmp/matrices". There is also a version gan_mat34_print() for printing to standard output:

```
Gan_Bool gan_mat34_print ( Gan_Matrix34 *p,
                           const char *prefix, int indent, const char *fmt );
```

The corresponding input function is

```
Gan_Bool gan_mat34_fscanf ( FILE *fp, Gan_Matrix34 *p,
                            char *prefix, int prefix_len )
```

which reads the matrix from the file stream fp into the 3×4 matrix pointer p. It also reads the prefix string (up to the specified maximum length prefix_len), which can be compared with the expected prefix string to check for consistency.

Binary file I/O is handled by the functions gan_mat34_fwrite() and gan_mat34_fread(). To write a 3×4 matrix in binary format use

```
Gan_Bool gan_mat34_fwrite ( FILE *fp, Gan_Matrix34 *p, gan_ui32 magic_number );
```

The magic_number takes the same role as the prefix string in gan_mat34_fprint(), and is written into the file so that it can be used later to identify the matrix when it is read back using

```
Gan_Bool gan_mat34_fread ( FILE *fp, Gan_Matrix34 *p, gan_ui32 *magic_number );
```

There are similar functions gan_mat33_fprint(), gan_mat33_print(), gan_mat33_fscanf() for ASCII I/O of 3×3 matrices, and gan_mat33_fwrite(), gan_mat33_fread() for binary I/O of 3×3 matrices. Functions for symmetric and triangular matrices follow the same pattern.

Error detection: The I/O routines return a boolean value, returning GAN_TRUE on success, GAN_FALSE on failure, invoking the Gandalf error handle in the latter case.

3.1.2.14 Conversion from general to fixed size matrix

Functions are provided to convert a general matrix Gan_Matrix Gan_Matrix34, provided that the general matrix has actually been created with size three. So for instance

```
Gan_Matrix *pmMatrix;

pmMatrix = gan_mat_alloc(3,4);
gan_mat_fill_va ( pmMatrix, 3, 4, 1.0,  2.0,  3.0,  4.0,
                                  5.0,  6.0,  7.0,  8.0,
                                  9.0, 10.0, 11.0, 12.0 );
gan_mat34_from_mat_q ( pmMatrix, &m34A );
```

or

```
m34A = gan_mat34_from_mat_s ( pmMatrix );
```

fill m34A with the same values as the general matrix. Calling these functions with a general matrix pmMatrix not having the same size as the fixed size matrix is an error.

Error detection: The conversion routines return the pointer to the filled fixed size matrix, or NULL on failure, invoking the Gandalf error handle in the latter case.

3.1.2.15 Single precision fixed size matrices

```
#include <gandalf/linalg/3matrixf.h>
```

There is an identical set of functions for handling single precision floating point matrices, the names of which are obtained by replacing "gan_mat34_..." in the above functions with "gan_mat34f_...". For example, to add two single precision 3×4 matrices use

```
Gan_Matrix34_f m34A, m34B, m34C;

/* ... fill m34A and m34B ... */
gan_mat34f_add_q ( &m34A, &m34B, &m34C );
```

3.2 General size matrices and vectors

When the vector/matrix object to be represented can have variable size, or number of rows/columns greater than four, Gandalf provides the structures and functions through the general size matrix/vector package. With this package similar operations to the fixed size package are supported. The general size package has addition features designed to ease the burden of the programmer, while still maintaining efficient run-time operation.

General size square matrices are handled in a subtly different way to their fixed-size equivalents. For fixed size symmetric and triangular matrices, there are specific functions dealing with each type of square matrix, currently the two types, symmetric and lower triangular. A fixed size square matrix does not "know" what type of matrix it is. The `type` field in this case is an inessential field of the matrix, and is indeed `#ifdef`'d out when Gandalf is compiled with `NDEBUG` set. The only reason for using the same structure for both symmetric and lower triangular matrices is to allow the in-place operations that convert a symmetric matrix to a triangular matrix or vice versa. This arrangement is optimal for speed, because no type checking needs to be done at run time.

In contrast, the general size square matrix structure has a `type` field that is meaningful as the current matrix type. Many Gandalf functions are written for general square matrices. This simplifies the programming interface, in that one function can be used to implement an operation (e.g. square matrix add) for every type of square matrix, at the expense of some loss of speed, since the general function has to call different subroutines depending on the type of matrix. The overhead in implementing this arrangement is reduced to the minimum by including the routines to implement each operation for a given matrix type in the matrix structure itself. This object-oriented design feature is hidden from the programmer through the use of macros, so the package appears to the programmer as a normal set of functions.

Another difference between fixed and general vectors and matrices is that whereas in general fixed size vectors & matrices should be declared as structures, avoiding the need for dynamic allocation, general size vectors & matrices require dynamic allocation of their internal data. There is still an advantage in declaring structures rather than structure pointers, in that with pointer variables you require a call to `malloc()` to create the structure they point to. Gandalf lets the programmer decide which style to use. In the following description examples of both styles are presented.

Once again, both double precision and single precision routines are available, and once again a parallel set of header files, structures and functions is provided. We shall concentrate on the double precision package, and the equivalent structure and function name conversions for single precision are given in Section 3.2.3.

Most of the routines return a pointer to the matrix/vector result structure. `NULL` is returned on failure, and the Gandalf error handler is invoked. This is reiterated in the text below, and exceptions are noted.

3.2.1 General size vectors

The structure and functions for general size vectors are declared in the header file

```
#include <gandalf/linalg/vec_gen.h>
```

The structure for general size vectors is the `Gan_Vector`.

3.2.1.1 Creating and freeing general size vectors

To create a general size vector use one of the routines

```
Gan_Vector *pvx;

pvx = gan_vec_alloc(5);
```

or

```
Gan_Vector vx;

gan_vec_form ( &vx, 5 );
```

Both these examples create a vector with five elements. The former allocates a structure and passes back a pointer to it, whereas the latter builds the vector using the provided structure `vx`. Whichever routine is used, the two vectors are equivalent in every way and can be used in all the Gandalf general size vector routines.

In the above calls Gandalf will invoke `malloc()` to create the data block to hold the vector elements. Sometimes you will want to provide the data block yourself, avoiding the `malloc()` call, if you know the size, or at least the maximum size, of the vector. Then you can use the following routine.

```
Gan_Vector vx;
double adXData[10];

gan_vec_form_data ( &vx, 5, adXData, 10 );
```

The last argument is the size of the array `adXData` passed in. This means that although the vector `vx` is created with size five, the size of the data block, 10, is also stored, and this allows the size of `vx` to change (see `gan_vec_set_size()` below) up to size 10.

Once you have finished with a vector use the routine

```
gan_vec_free ( pvx ); /* for a pointer variable, OR */
gan_vec_free ( &vx ); /* for a structure variable */
```

The `gan_vec_free()` routine applies without modification to all the methods of creating the vector. The vector structure maintains knowledge of which parts of it (the structure, the data block) were dynamically allocated, and only frees the bits that were allocated.

From now on the example code fragments we provide will use the convention that vectors are declared as structures rather than pointers, but bear in mind that either style may be used.

Error detection: All the above vector creation routines return a pointer to the created vector. If an error occurs, the Gandalf error handler is invoked and `NULL` is returned. The most likely error modes are failing to allocate the data required (i.e. internal `malloc()` or `realloc()` calls failing), or passing too small an array into the `gan_vec_form_data()` routine.

3.2.1.2 Adjusting the size of a general size vector

Once a vector has been created, its size may be adjusted dynamically as needs arise. Gandalf stores the currently allocated maximum size of a vector in the vector structure, so it can determine when the size of the result of a

computation will exceed the current size, and reallocate accordingly. This happens automatically when a vector is the result of a calculation, but sometimes it is necessary to explicitly set the size of a vector. This is done using the following routine.

```
gan_vec_set_size ( &vx, 3 );
```

This resets the size of the vector `vx` to 3. If the size of a vector created by `gan_vec_alloc()` or `gan_vec_form()` is increased in size in this way beyond its originally allocated size, `gan_vec_set_size()` will automatically reallocate the vector to the new size. On the other hand, if `gan_vec_form_data()` was used to create the vector, it cannot be increased in size beyond the size of the array passed as the last argument into `gan_vec_form_data()`.

Error detection: `NULL` is returned and the error handler is invoked on failure. The most likely failure mode is failing to reallocate the vector data, i.e. failure of a call to `realloc()`.

3.2.1.3 Filling a general size vector with values

To fill a vector with values, create the vector and then use the routine `gan_vec_fill_va()`. An example is

```
gan_vec_fill_va ( &vx, 6, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0 );
```

which sets vector `vx` to have size 6, and sets the six elements to the values one to six.

To fill a vector with a constant value, you can use

```
gan_vec_fill_const_q ( &vx, 4, 3.0 );
```

which sets the size of `vx` to four elements, all of which are set to three. This function is also available in a form which allocates and fills a vector from scratch:

```
Gan_Vector *pvx;

pvx = gan_vec_fill_const_s ( 4, 3.0 );
```

There are special macro routines for setting a vector to zero:

```
gan_vec_fill_zero_q ( &vx, 4 ); /* OR */
pvx = gan_vec_fill_zero_s(4);
```

Error detection: `NULL` is returned and the error handler is invoked on failure. The most likely failure mode is failing to reallocate the vector data when the size of the vector is changed, i.e. failure of a call to `realloc()`.

3.2.1.4 Accessing the elements of a general size vector

To read the value of a specific element of a vector use

```
double dEl;

dEl = gan_vec_get_el ( pvx, 1 ); /* returns x[1], x = (x[0] x[1] ... )^T */
```

This sets `dEl` to the second element of vector `pvx` (zero is the first). To set an element to a specific value use

```
gan_vec_set_el ( pvx, 2, 3.0 ); /* set x[2] to 3, x = (x[0] x[1] ... )^T */
```

This sets the third element of vector pvx to 3. There are also routines to increment or decrement an element of a vector:

```
gan_vec_inc_el ( pvx, 0, 2.5 ); /* x_0 += 2.5, x = (x[0] x[1] ... )^T */
gan_vec_dec_el ( pvx, 4, 5.0 ); /* x_4 -= 5.0, x = (x[0] x[1] ... )^T */
```

which respectively increment the first element by 2.5 and subtract 5 from the fifth element of pvx.

Error detection: gan_vec_set_el(), gan_vec_inc_el() and gan_vec_dec_el() all return boolean values, with GAN_FALSE returned on failure, in which case the Gandalf error handler is invoked. The most likely failure mode is accessing an element outside the bounds of the vector. If NDEBUG is set then no error checking is done. gan_vec_get_el() operates similarly, but returns DBL_MAX on error.

3.2.1.5 Copying a general size vector

To copy a vector **x** to another vector **y**, both vectors must have been created, and **x** should be filled with values. **y** can be created with arbitrary initial size (for instance zero), since Gandalf will if necessary reallocate **y** to the same size as **x** if necessary. So for instance the following code is perfectly valid:

```
Gan_Vector vx, vy; /* declare vectors x & y */

gan_vec_form ( &vx, 0 ); /* create vector x */
gan_vec_form ( &vy, 0 ); /* create vector y */
gan_vec_fill_va ( &vx, 5, 11.0, 9.0, 7.0, 5.0, 3.0 ); /* reallocate & initialise x */
gan_vec_copy_q ( &vx, &vy ); /* set y = x, reallocating y */
```

The last two lines reallocate first **x** and then **y**, because both were created with zero size. Note that **y** may have previously been filled with other values, which are now lost.

There is also a version that creates a copy of a vector from scratch:

```
Gan_Vector *pvy; /* declare vector y */

pvy = gan_vec_copy_s ( &vx ); /* create y and set y = x */
```

Error detection: The vector copy routines return NULL and invoke the Gandalf error handler upon failure.

3.2.1.6 General size vector addition

To add two vectors **x** and **y** together, obtaining the sum $\mathbf{z} = \mathbf{x} + \mathbf{y}$, use the routine

```
Gan_Vector vx, vy, vz; /* declare vectors x, y and z */

/* ... create and fill vx & vy, create vz ... */
gan_vec_add_q ( &vx, &vy, &vz ); /* compute z = x + y */
```

Again vector **z** is reallocated if necessary. Vectors **x** and **y** must of course be the same size, or the error handler is invoked and NULL is returned. The sum vector **z** may be create from scratch using

```
Gan_Vector *pvz; /* declare vector z as pointer */

/* ... create and fill vx & vy ... */
pvz = gan_vec_add_s ( &vx, &vy ); /* compute z = x + y */
```

Another way of computing vector addition is to replace one of the input vector **x** or **y** with the result, using one of the in-place routines

```
gan_vec_add_i1 ( &vx, &vy ); /* replace x = x + y */
gan_vec_add_i2 ( &vx, &vy ); /* replace y = x + y */
```

An alternative to `gan_vec_add_i1()` is the more explicit routine

```
gan_vec_increment ( &vx, &vy ); /* replace x = x + y */
```

Error detection: `NULL` is returned and the Gandalf error handler invoked if the vector addition fails. The most likely failure modes are failing to create/set the result vector, or size incompatibility between the input vectors.

3.2.1.7 General size vector subtraction

The routines for vector subtraction follow the scheme of those for vector addition, leading to the options

```
Gan_Vector vx, vy, vz; /* declare vectors x, y and z */
Gan_Vector *pvz; /* declare vector z alternatively as pointer */

/* ... create and fill vx & vy, create vz ... */
gan_vec_sub_q ( &vx, &vy, &vz ); /* compute z = x - y */
pvz = gan_vec_sub_s ( &vx, &vy ); /* compute z = x - y */
gan_vec_sub_i1 ( &vx, &vy ); /* replace x = x - y */
gan_vec_sub_i2 ( &vx, &vy ); /* replace y = x - y */
gan_vec_decrement ( &vx, &vy ); /* replace x = x - y */
```

Error detection: `NULL` is returned and the Gandalf error handler invoked if the vector subtraction fails. The most likely failure modes are failing to create/set the result vector, or size incompatibility between the input vectors.

3.2.1.8 Rescaling a general size vector

Multiplying or dividing a vector by a scalar value follows the scheme of the above copy, addition and subtraction operations. To multiply a vector **x** by a scalar s, $\mathbf{y} = s\mathbf{x}$, use for example

```
Gan_Vector vx, vy; /* declare vectors x & y */

/* ... create & fill vx, create (& optionally fill) vy ... */
gan_vec_scale_q ( &vx, 5.0, &vy ); /* y = 5*x */
```

to multiply all the elements in vector **x** by five, writing the result into vector **y**. Alternatively you can create the rescaled vector from scratch as in

```
Gan_Vector *pvy; /* declare vector y */

/* ... create & fill vx ... */
pvy = gan_vec_scale_s ( &vx, 5.0 ); /* y = 5*x */
```

or overwrite **x** with the result

```
gan_vec_scale_i ( &vx, 5.0 ); /* replace x = 5*x */
```

There are similar routines for dividing a general size vector by a scalar value:

```
gan_vec_divide_q ( &vx, 5.0, &vy ); /* y = x/5 */
pvy = gan_vec_divide_s ( &vx, 5.0 ); /* y = x/5 */
gan_vec_divide_i ( &vx, 5.0 ); /* replace x = x/5 */
```

Passing zero as the scalar value in this case invokes the error handler, with a division by zero error (error code `GAN_ERROR_DIVISION_BY_ZERO`), and `NULL` is returned.

There are specific routines to negate a vector, i.e. multiply it by -1, as follows:

```
gan_vec_negate_q ( &vx, &vy ); /* y = -x */
pvy = gan_vec_negate_s ( &vx ); /* y = -x */
gan_vec_negate_i ( &vx ); /* replace x = -x */
```

`Error detection`: The Gandalf error handler is invoked and `NULL` is returned if an error occurs. The most likely failure modes are (i) failing to create the result vector; (ii) division by zero.

3.2.2 General size matrices

The structure and functions for general size matrices are declared in the header file

```
#include <gandalf/linalg/mat_gen.h>
```

The structure for general size matrices is the `Gan_Matrix`. For special types of square matrix the structure is the `Gan_SquMatrix`, and is declared in the header file

```
#include <gandalf/linalg/mat_square.h>
```

The square matrix types are listed in `linalg_defs.h` (file not to be included explicitly in application programs):

```
/* types of square matrix */
typedef enum { GAN_SYMMETRIC_MATRIX,     /* symmetric */
               GAN_DIAGONAL_MATRIX,      /* diagonal */
               GAN_SCALED_IDENT_MATRIX,  /* identity times scalar */
               GAN_LOWER_TRI_MATRIX,     /* lower triangular */
               GAN_UPPER_TRI_MATRIX,     /* upper triangular */
               GAN_ZERO_SQUARE_MATRIX }  /* square matrix filled with zeros */
  Gan_SquMatrixType;
```

Use of the special matrix types producs savings both in memory and computation time, and should be exploited wherever appropriate. To use any functions specific to the above square matrix types (as opposed to general square matrix functions common to many types), you will need to include the header file for the specific type you need, for instance

```
#include <gandalf/linalg/mat_symmetric.h>
```

for symmetric matrices, and

```
#include <gandalf/linalg/mat_triangular.h>
```

for triangular matrices (covers lower and upper triangular matrices). The full list of header files for the specific square matrix types is

```
#include <gandalf/linalg/mat_symmetric.h>  /* for symmetric matrices */
#include <gandalf/linalg/mat_triangular.h> /* lower/upper triangular matrices */
#include <gandalf/linalg/mat_diagonal.h>   /* diagonal matrices */
#include <gandalf/linalg/mat_scaledI.h>    /* scaled identity matrices */
```

The square matrix routines are organised so that once the matrix has been set at a specific type and size, and filled with values, the functions that operate on it are general functions. The computational overhead is that whenever an operation is performed (such at matrix/matrix multiplication) there is an indirection to take the program into the correct routine for the current square matrix type. This design allows the programmer to easily design algorithms that will work correctly and efficiently for any square matrix type.

3.2.2.1 Creating and freeing general size matrices

To create a general size rectangular matrix use one of the routines

```
Gan_Matrix *pmA;

pmA = gan_mat_alloc ( 3, 5 );
```

or

```
Gan_Matrix mA;

gan_mat_form ( &mA, 3, 5 );
```

Both these examples create a matrix with three rows and five columns. The former allocates a structure and passes back a pointer to it, whereas the latter builds the matrix using the provided structure `mA`. Whichever routine is used, the two matrices are equivalent in every way and can be used in all the Gandalf general size matrix routines.

In the above calls Gandalf will invoke `malloc()` to create the data block to hold the matrix elements. Sometimes you will want to provide the data block yourself, avoiding the `malloc()` call, if you know the size, or at least the maximum size, of the matrix. Then you can use the following routine.

```
Gan_Matrix mA;
double adAData[30];

gan_mat_form_data ( &mA, 3, 5, adXData, 10 );
```

The last argument is the size of the array `adXData` passed in. This means that although the matrix `mA` is created with size five, the size of the data block, 10, is also stored, and this allows the size of `mA` to change (see `gan_mat_set_size()` below) up to size 10.

Once you have finished with a matrix, use the routine

```
gan_mat_free ( pmA ); /* for a pointer variable, OR */
gan_mat_free ( &mA ); /* for a structure variable */
```

The `gan_mat_free()` routine applies without modification to all the methods of creating the matrix. The matrix structure maintains knowledge of which parts of it (the structure, the data block) were dynamically allocated, and only frees the bits that were allocated. To free several rectangular matrices at once use the variable argument list routine

```
gan_mat_free_va ( pmA, pmB, pmC, NULL ); /* free matrices A, B & C */
```

which must be terminated by `NULL` to indicate the end of the list.

From now on, we use the convention that a "square" matrix refers to a square matrix with one of the special types listed above. A square general matrix falls into the category of rectangular matrices, for the purpose of the Gandalf linear algebra package. To create a square matrix with one of the special types use a function from one of the families

```
Gan_SquMatrix *psmS, *psmL, *psmU, *psmD, *psmsI;

psmS = gan_symmat_alloc ( 3 ); /* create a 3x3 symmetric matrix */
psmL = gan_ltmat_alloc ( 3 ); /* create a 3x3 lower triangular matrix */
psmU = gan_utmat_alloc ( 3 ); /* create a 3x3 upper triangular matrix */
psmD = gan_diagmat_alloc ( 3 ); /* create a 3x3 diagonal matrix */
psmD = gan_scalImat_alloc ( 3 ); /* create a 3x3 scaled Identity matrix */
```

or

```
Gan_SquMatrix smS, smL, smU, smD, smsI;

gan_symmat_form ( &smS, 3 ); /* create a 3x3 symmetric matrix */
gan_ltmat_form ( &smL, 3 ); /* create a 3x3 lower triangular matrix */
gan_utmat_form ( &smU, 3 ); /* create a 3x3 upper triangular matrix */
gan_diagmat_form ( &smD, 3 ); /* create a 3x3 diagonal matrix */
gan_scalImat_form ( &smsI, 3 ); /* create a 3x3 scaled Identity matrix */
```

There are also `..._form_data()` functions available for the case that an array to hold the matrix data is already available. The size of the data array depends on the type of matrix; for instance a 4×4 symmetric and triangular matrix has ten independent elements, while a 4×4 diagonal matrix has only four, and a scaled identity matrix only one. The general formula for the number of independent elements in a triangular or symmetric matrix is

$$\#\text{ independent elements} = \frac{n(n+1)}{2} \tag{3.1}$$

for an $n \times n$ matrix, compared with n for a diagonal matrix and 1 for a scaled identity matrix. So these are appropriate function calls.

```
Gan_SquMatrix smS, smL, smU, smD, smsI;
double adSdata[10], adLdata[10], adUdata[10], adDdata[4], dsIdata;

gan_symmat_form_data ( &smS, 4, adSdata, 10 ); /* create 4x4 symmetric matrix */
gan_ltmat_form_data ( &smL, 4, adLdata, 10 ); /* create 4x4 lower triangular matrix */
gan_utmat_form_data ( &smU, 4, adUdata, 10 ); /* create 4x4 upper triangular matrix */
gan_diagmat_form_data ( &smD, 4, adDdata, 4 ); /* create 4x4 diagonal matrix */
gan_scalImat_form_data ( &smsI, 4, &dsIdata, 1 ); /* create 4x4 scaled Ident. mat. */
```

The final way of creating a square matrix should be used only when the matrix type is a variable:

```
Gan_SquMatrixType type;
Gan_SquMatrix *psmA, smA;
double adAdata[10];

/* ... set matrix type e.g. GAN_SYMMETRIC_MATRIX ... */
psmA = gan_squmat_alloc ( type, 4 ); /* create 4x4 square matrix, pointer version, OR */
gan_squmat_form ( &smA, type, 4 ); /* create 4x4 square matrix, structure version, OR */
gan_squmat_form_data ( &smA, type, 4, adAdata, 10 ); /* create 4x4 square matrix */
```

These routines call the lower level routine specific to the provided `type`. Whichever type of matrix is created, use the function

```
gan_squmat_free ( psmA );
```

to free the memory associated with it. The `gan_squmat_free()` routine applies to all the square matrix types and all methods (`..._alloc()`, `..._form()` and `..._form_data()`) of creating the matrix. The matrix structure maintains knowledge of which parts of it (the structure, the data block) were dynamically allocated, and only frees the bits that were allocated. To free several square matrices at once use the variable argument list routine

```
gan_squmat_free_va ( &smsI, &smD, &smU, &smL, &smS, NULL ); /* free square matrices */
```

which must be terminated by `NULL` to indicate the end of the list.

From now on the example code fragments we provide will use the convention that matrices are declared as structures rather than pointers, but bear in mind that either style may be used.

Error detection: All the above matrix creation routines return a pointer to the created matrix. If an error occurs, the Gandalf error handler is invoked and `NULL` is returned. The most likely error modes are failing to allocate the data required (i.e. internal `malloc()` or `realloc()` calls failing), or passing too small an array into the `..._form_data()` routines.

3.2.2.2 Adjusting the size of a general size matrix

Once a matrix has been created, its size may be adjusted dynamically as needs arise. Gandalf stores the currently allocated maximum size of a matrix in the matrix structure, so it can determine when the size of the result of a computation will exceed the current size, and reallocate accordingly. This happens automatically when a matrix is the result of a calculation, but sometimes it is necessary to explicitly set the size of a matrix. This is done using the following routine.

```
Gan_Marix mA;

/* create matrix A using e.g. gan_mat_form() */
gan_mat_set_size ( &mA, 3 );
```

This resets the size of the matrix `mA` to 3. If the size of a matrix created by `gan_mat_alloc()` or `gan_mat_form()` is increased in size in this way beyond its originally allocated size, `gan_mat_set_size()` will automatically reallocate the matrix to the new size. On the other hand, if `gan_mat_form_data()` was used to create the matrix, it cannot be increased in size beyond the size of the array passed as the last argument into `gan_mat_form_data()`.

For square matrices there are similar routines for specific matrix types, for instance

```
Gan_SquMarix smA;

/* create matrix A using e.g. gan_squmat_form() */
gan_symmat_set_size ( &mA, 3 ); /* set A to be a symmetric matrix with size 3 */
```

An important feature here is that Gandalf allows both the size and type of the matrix to be changed. For instance, the following code is valid:

```
Gan_SquMarix smA;

gan_diagmat_form ( &smA, 2 ); /* create matrix A as diagonal with size 2 */
gan_symmat_set_size ( &mA, 3 ); /* set A to be a symmetric matrix with size 3 */
```

Gandalf will reallocate the matrix internally if necessary. The main proviso here is that if the matrix was created using a `..._form_data()` routine, setting it to a type and size which requires more independent elements than the size of the data array passed in is an error, so for instance

```
Gan_SquMarix smA;
double adAdata[10];

/* create matrix A as diagonal with size 2 */
gan_diagmat_form_data ( &smA, 2, adAdata, 10 );

/* set A to be a symmetric matrix with size 4 */
gan_symmat_set_size ( &mA, 4 );
```

is OK, because a 4×4 symmetric matrix has ten independent elements, but an additional line

```
gan_ltmat_set_size ( &smA, 5 ); /* set A to be a lower triangular matrix with size 5 */
```

will fail (return `NULL`), because a 5×5 triangular matrix has fifteen independent elements (see Equation 3.1), and the array `adAdata` passed into the matrix originally has only ten elements.

The complete list of routines for setting a square matrix to a specific size (here 5) and type is

```
gan_symmat_set_size ( &smA, 5 ); /* set A to be a 5x5 symmetric matrix */
gan_ltmat_set_size ( &smA, 5 ); /* set A to be a 5x5 lower triangular matrix */
gan_utmat_set_size ( &smA, 5 ); /* set A to be a 5x5 upper triangular matrix */
gan_diagmat_set_size ( &smA, 5 ); /* set A to be a 5x5 diagonal matrix */
gan_scalImat_set_size ( &smA, 5 ); /* set A to be a 5x5 scaled identity matrix */
```

and there is also a function for setting a matrix with a variable type:

```
/* set A to be a symmetric matrix with size 5 */
gan_squmat_set_type_size ( &smA, GAN_SYMMETRIC_MATRIX, 5 );
```

and routines to set only the type, or only the size, of the matrix:

```
/* set A to be a symmetric matrix, size unchanged */
gan_squmat_set_type ( &smA, GAN_SYMMETRIC_MATRIX );

/* set A to be size 4, type unchanged */
gan_squmat_set_size ( &smA, 4 );
```

Error detection: `NULL` is returned and the error handler is invoked on failure. The most likely failure mode is failing to reallocate the matrix data, i.e. failure of a call to `realloc()`.

3.2.2.3 Filling a general size matrix with values

To fill a matrix with values, create the matrix and then use the routine `gan_mat_fill_va()`. An example is

```
Gan_Matrix mA;

/* ... create mA using e.g. gan_mat_form() ... */
gan_mat_fill_va ( &mA, 2, 3, 1.0, 2.0, 3.0,
                             4.0, 5.0, 6.0 );
```

which sets matrix `mA` to have dimensions 2 rows by 3 columns, and sets the value to

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$

To fill a matrix with a constant value, you can use

```
gan_mat_fill_const_q ( &mA, 4, 2, 3.0 );
```

which sets the dimensions of `mA` to four rows by two columns, and sets all the elements to three. This gives rise to the matrix

$$A = \begin{pmatrix} 3 & 3 & 3 & 3 \\ 3 & 3 & 3 & 3 \end{pmatrix}$$

This function is also available in a form which allocates and fills a matrix from scratch:

```
Gan_Matrix *pmA;

pmA = gan_mat_fill_const_s ( 4, 2, 3.0 );
```

There are special macro routines for setting a matrix to zero:

```
gan_mat_fill_zero_q ( &mA, 4, 2 ); /* OR */
pmA = gan_mat_fill_zero_s ( 4, 2 );
```

For square matrices there are specific routines for each square matrix type. The order in which the elements are passed in the variable argument list corresponds to the matrix type. For symmetric matrice, only the *lower* triangle is passed (including the diagonal). So for instance to create a symmetric matrix

$$S = \begin{pmatrix} 1 & 2 & 4 \\ 2 & 3 & 5 \\ 4 & 5 & 6 \end{pmatrix}$$

use the code

```
Gan_SquMatrix smS;

/* ... create smS using e.g. gan_symmat_form() ... */
gan_symmat_fill_va ( &smS, 3, 1.0,
                              2.0, 3.0,
                              4.0, 5.0, 6.0 );
```

For lower and upper triangular matrices pass the elements in the relevant order for the corresponding triangle. So for instance

```
Gan_SquMatrix smL;

/* ... create smL using e.g. gan_ltmat_form() ... */
gan_ltmat_fill_va ( &smL, 3, 1.0,
                             2.0, 3.0,
                             4.0, 5.0, 6.0 );
```

creates the lower triangular matrix

$$L = \begin{pmatrix} 1 & 0 & 0 \\ 2 & 3 & 0 \\ 4 & 5 & 6 \end{pmatrix}$$

while

```
Gan_SquMatrix smU;

/* ... create smU using e.g. gan_utmat_form() ... */
gan_utmat_fill_va ( &smU, 3, 1.0, 2.0, 4.0,
                                  3.0, 5.0,
                                       6.0 );
```

creates the upper triangular matrix

$$U = \begin{pmatrix} 1 & 2 & 4 \\ 0 & 3 & 5 \\ 0 & 0 & 6 \end{pmatrix}$$

The routines for diagonal and scaled identity matrices follow a similar pattern, so you can use

```
Gan_SquMatrix smD;

/* ... create smD using e.g. gan_diagmat_form() ... */
gan_diagmat_fill_va ( &smD, 4, 1.0, 2.0, 3.0, 4.0 );
```

to create the diagonal matrix

$$D = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 4 \end{pmatrix}$$

or

```
Gan_SquMatrix smsI;

/* ... create smsI using e.g. gan_scalImat_form() ... */
gan_scalImat_fill_va ( &msI, 4, 2.0 );
```

to create the scaled identity matrix

$$D = \begin{pmatrix} 2 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 2 \end{pmatrix}$$

There are also routines to fill square matrices with a constant value, with a special routine for filling with zero:

```
Gan_SquMatrix smA;

/* ... create smA using e.g. gan_squmat_form() ... */
gan_symmat_fill_const_q ( &smA, 4, 3.0 ); /* set A as symmetric(4x4), each element 3 */
gan_ltmat_fill_const_q ( &smA, 4, 3.0 ); /* set A as l. triang.(4x4), each element 3 */
gan_utmat_fill_const_q ( &smA, 4, 3.0 ); /* set A as u. triang.(4x4), each element 3 */
gan_diagmat_fill_const_q ( &smA, 4, 3.0 ); /* set A as diagonal(4x4), each element 3 */
gan_scalImat_fill_const_q ( &smA, 4, 3.0 ); /* set A as scaled I(4x4), each element 3 */
gan_symmat_fill_zero_q ( &smA, 4 ); /* set A as symmetric(4x4), each element zero */
gan_ltmat_fill_zero_q ( &smA, 4 ); /* set A as l. triang.(4x4), each element zero */
gan_utmat_fill_zero_q ( &smA, 4 ); /* set A as u. triang.(4x4), each element zero */
gan_diagmat_fill_zero_q ( &smA, 4 ); /* set A as diagonal(4x4), each element zero */
gan_scalImat_fill_zero_q ( &smA, 4 ); /* set A as scaled I(4x4), each element zero */
```

set the type and size of an existing square matrix, and sets all its elements to the same value,

```
Gan_SquMatrix *psmS, *psmL, *psmU, *psmD, *psmsI;

psmS = gan_symmat_fill_const_s ( 4, 3.0 ); /* create 4x4 symmetric mat., each el. 3 */
psmL = gan_ltmat_fill_const_s ( 4, 3.0 ); /* create 4x4 u. tri. mat., each el. 3 */
psmU = gan_utmat_fill_const_s ( 4, 3.0 ); /* create 4x4 l. tri. mat., each el. 3 */
psmD = gan_diagmat_fill_const_s ( 4, 3.0 ); /* create 4x4 diagonal mat., each el. 3 */
psmsI = gan_scalImat_fill_const_s ( 4, 3.0 ); /* create 4x4 scaled I mat., each el. 3 */
psmS = gan_symmat_fill_zero_s ( 4 ); /* create 4x4 symmetric mat., each el. zero */
psmL = gan_ltmat_fill_zero_s ( 4 ); /* create 4x4 u. tri. mat., each el. zero */
psmU = gan_utmat_fill_zero_s ( 4 ); /* create 4x4 l. tri. mat., each el. zero */
psmD = gan_diagmat_fill_zero_s ( 4 ); /* create 4x4 diagonal mat., each el. zero */
psmsI = gan_scalImat_fill_zero_s ( 4 ); /* create 4x4 scaled I mat., each el. zero */
```

create new matrices with the given type and size, and set all the elements to the same value,

There are also equivalent routines that work with a variable square matrix type:

```
Gan_SquMatrix smA, *psmA;
Gan_SquMatrixType type;

/* ... create smA using e.g. gan_squmat_form() and set type to
       desired square matrix type, e.g. GAN_SYMMETRIC_MATRIX ...*/

/* set up an existing matrix, fill it with a constant value */
gan_squmat_fill_const_q ( &smA, type, 4, 3.0 ); /* constant element value 3 */
gan_squmat_fill_zero_q ( &smA, type, 4 ); /* fill with zero */

/* create a matrix from scratch */
psmA = gan_squmat_fill_const_s ( type, 4, 3.0 ); /* constant element value 3 */
psmA = gan_squmat_fill_const_s ( type, 4, 3.0 ); /* fill with zero */
```

Note that the dynamic reconfiguration feature of square matrices means again that the `..._fill_va()`, `..._fill_const_q()` and `..._fill_zero_q()` square matrix routines do not require the matrix to be set up initially with either the same type or size.

Error detection: `NULL` is returned and the error handler is invoked on failure. The most likely failure mode is failing to reallocate the matrix data when the size of the matrix is changed, i.e. failure of a call to `realloc()`.

3.2.2.4 Accessing the elements of a general size matrix

To read the value of a specific element of a matrix A use

```
Gan_Matrix mA; /* matrix A */
double dEl;

/* ... create and fill matrix A ... */
dEl = gan_mat_get_el ( &mA, 1, 2 ); /* returns A[1][2], A = (A[0][0] A[0][1] ... )
                                                           (A[1][0] A[1][1] ... )
                                                           (   :       :    ... ) */
```

This sets `dEl` to the third element of the second row of matrix A. To set an element to a specific value use

```
gan_mat_set_el ( &mA, 0, 3, 3.0 ); /* sets A[0][3] to 3.0 */
```

This sets the fourth element of the first row of A to 3. There are also routines to increment or decrement an element of a matrix:

```
gan_mat_inc_el ( &mA, 0, 1, 2.5 ); /* A[0][1] += 2.5 */
gan_mat_dec_el ( &mA, 3, 2, 5.0 ); /* A[3][2] -= 5.0 */
```

which respectively increment the second element of the first row of A by 2.5 and subtract 5 from the third element of the fourth row of A.

For special square matrices there are equivalent routines which can be illustrated by the following code fragment.

```
Gan_SquMatrix smA; /* matrix A */
double dEl;

/* ... create and fill matrix A ... */
dEl = gan_squmat_get_el ( &smA, 1, 2 ); /* returns A[1][2] */
gan_squmat_set_el ( &smA, 0, 3, 3.0 ); /* sets A[0][3] to 3.0 */
gan_squmat_inc_el ( &smA, 0, 1, 2.5 ); /* A[0][1] += 2.5 */
gan_squmat_dec_el ( &smA, 3, 2, 5.0 ); /* A[3][2] -= 5.0 */
```

Error detection: `gan_{squ}mat_set_el()`, `gan_{squ}mat_inc_el()` and `gan_{squ}mat_dec_el()` all return boolean values, with `GAN_FALSE` returned on failure, in which case the Gandalf error handler is invoked. The most likely failure modes is accessing an element outside the bounds of the matrix, or seting an illegal element of a square matrix (e.g. an off-diagonal element of a diagonal matrix). If `NDEBUG` is set then no error checking is done. `gan_{squ}mat_get_el()` operate similarly, but return `DBL_MAX` on error.

3.2.2.5 Copying a general size matrix

To copy a matrix A to another matrix B, both matrices must have been created, and A should be filled with values. B can be created with arbitrary initial dimensions (for instance zero), since Gandalf will if necessary reallocate B to the same size as A. So for instance the following code is perfectly valid:

```
Gan_Matrix mA, mB; /* declare matrices A & B */

gan_mat_form ( &mA, 0, 0 ); /* create matrix A */
gan_mat_form ( &mB, 0, 0 ); /* create matrix B */

/* reallocate & initialise A */
gan_mat_fill_va ( &mA, 2, 3, 11.0, 9.0, 7.0,
                             5.0, 3.0, 1.0 );
gan_mat_copy_q ( &mA, &mB ); /* set B = A, reallocating B */
```

The last two lines reallocate first A and then B, because both were created with zero size. Note that B may have previosly been filled with other values, which are now lost.

There is also a version that creates a copy of a matrix from scratch:

```
Gan_Matrix *pmB; /* declare matrix B */

pmB = gan_mat_copy_s ( &mA ); /* create B and set B = A */
```

For special square matrices, use one of the functions

```
Gan_SquMatrix smA, smB, *psmB; /* declare matrices A & B */

/* ... create A & B using e.g. gan_diagmat_form(), and initialise A using
       e.g. gan_diagmat_fill_va() ...  */
gan_squmat_copy_q ( &smA, &smB ); /* set B = A, reallocating B if necessary, OR */
psmB = gan_squmat_copy_s(&smA); /* set B = A, creating B */
```

Error detection: The matrix copy routines return `NULL` and invoke the Gandalf error handler upon failure.

3.2.2.6 Transposing a general size matrix

Gandalf supports implicit matrix transpose across all relevant routines, so it is not often necessary to explicitly transpose a matrix. Nonetheless, like matrix inverse it sometimes cannot be avoided. To transpose a matrix A into another matrix B, both matrices must have been created, and A should be filled with values. B can be created with arbitrary initial dimensions (for instance zero), since Gandalf will if necessary reallocate B to the same size as A. So for instance the following code is perfectly valid:

```
Gan_Matrix mA, mB; /* declare matrices A & B */

gan_mat_form ( &mA, 0, 0 ); /* create matrix A */
gan_mat_form ( &mB, 0, 0 ); /* create matrix B */

/* reallocate & initialise A */
gan_mat_fill_va ( &mA, 2, 3, 11.0, 9.0, 7.0,
                             5.0, 3.0, 1.0 );
gan_mat_tpose_q ( &mA, &mB ); /* set B = A^T, reallocating B */
```

The last two lines reallocate first A and then B, because both were created with zero size. Note that B may have previosly been filled with other values, which are now lost.

There is also a version that creates the transpose of a matrix from scratch:

```
Gan_Matrix *pmB; /* declare matrix B */

pmB = gan_mat_tpose_s ( &mA ); /* create B and set B = A */
```

If in this case matrix A happens to be square, Gandalf supports in-place transpose:

```
/* A have the same number of rows and columns */
gan_mat_tpose_i ( &mA ); /* replace A = A^T */
```

There is no explicit transpose implemented in Gandalf for special square matrices. With the current matrix types supported by Gandalf, it would only be relevant anyway for triangular matrices. Implicit transpose can handle every practical situation.

Error detection: The matrix transpose routines return `NULL` and invoke the Gandalf error handler upon failure.

3.2.2.7 General size matrix addition

To add two matrices A and B together, obtaining the sum $C = A + B$, use the routine

```
Gan_Matrix mA, mB, mC; /* declare matrices A, B and C */

/* ... create and fill mA & mB, create mC ... */
gan_mat_add_q ( &mA, &mB, &mC ); /* compute C = A + B */
```

Again matrix C is reallocated if necessary. Matrices A and B must of course be the same size, or the error handler is invoked and `NULL` is returned. The sum matrix C may be create from scratch using

```
Gan_Matrix *pmC; /* declare matrix C as pointer */
```

```
/* ... create and fill mA & mB ... */
pmC = gan_mat_add_s ( &mA, &mB ); /* compute C = A + B */
```

Another way of computing matrix addition is to replace one of the input matrix A or B with the result, using one of the in-place routines

```
gan_mat_add_i1 ( &mA, &mB ); /* replace A = A + B */
gan_mat_add_i2 ( &mA, &mB ); /* replace B = A + B */
```

An alternative to `gan_mat_add_i1()` is the more explicit routine

```
gan_mat_increment ( &mA, &mB ); /* replace A = A + B */
```

There is also a set of routines for adding a general size matrix to the transpose of another:

```
Gan_Matrix mA, mB, mC, *pmC; /* declare matrices A, B and C */

/* ... create and fill A & B, create C ... */

/* B must have the same number of columns as A has rows, and vice versa */
gan_mat_addT_q ( &mA, &mB, &mC ); /* compute C = A + B^T, OR */
pmC = gan_mat_addT_s ( &mA, &mB ); /* compute C = A + B^T, OR */
gan_mat_incrementT ( &mA, &mB ); /* replace A = A + B^T */
```

Another set of routines allows you to add two matrices and generate a symmetric matrix, on the assumption that the result is indeed symmetric. Either matrix may be implicitly transposed for the purpose of the operation:

```
Gan_Matrix mA, mB; /* declare matrices A, B */
Gan_SquMatrix smS, *psmS; /* declare result matrix S */

/* ... create and fill A & B, create S ... */

/* for these functions, B must have the same number of columns and rows as A */
gan_mat_add_sym_q ( &mA, &mB, &smS ); /* S = A + B, OR */
psmS = gan_mat_add_sym_s ( &mA, &mB ); /* S = A + B */
gan_matT_addT_sym_q ( &mA, &mB, &smS ); /* S = A^T + B^T, OR */
psmS = gan_matT_addT_sym_s ( &mA, &mB ); /* S = A^T + B^T */

/* here B must have the same number of columns as A has rows, and vice versa */
gan_mat_addT_sym_q ( &mA, &mB, &smS ); /* S = A + B^T, OR */
psmS = gan_mat_addT_sym_s ( &mA, &mB ); /* S = A + B^T */
gan_matT_add_sym_q ( &mA, &mB, &smS ); /* S = A^T + B, OR */
psmS = gan_matT_add_sym_s ( &mA, &mB ); /* S = A^T + B */
```

Finally we have some routines for adding a matrix to its own transpose, producing a symmetric matrix:

```
Gan_Matrix mA; /* declare matrix A */
Gan_SquMatrix smS, *psmS; /* declare result matrix S */

/* ... create and fill A, create S ... */

gan_mat_saddT_sym_q ( &mA, &smS ); /* S = A + A^T, OR */
psmS = gan_mat_saddT_sym_s ( &mA ); /* S = A + A^T */
```

There are equivalent functions for square matrices. Firstly the simple routines for adding two matrices:

```
Gan_SquMatrix smA, smB, smC, *psmC; /* declare matrices A, B & C */

/* ... create and fill smA & smB, create smC ... */
gan_squmat_add_q ( &smA, &smB, &smC ); /* compute C = A + B, OR */
gan_squmat_add_i1 ( &smA, &smB ); /* replace A = A + B, OR */
gan_squmat_add_i2 ( &smA, &smB ); /* replace B = A + B, OR */
gan_squmat_increment ( &smA, &smB ); /* replace A = A + B, OR */
psmC = gan_squmat_add_s ( &smA, &smB ); /* compute C = A + B as new matrix */
```

Other routines implicitly transpose one of the input matrices:

```
Gan_SquMatrix smA, smB, smC, *psmC; /* declare matrices A, B & C */

/* ... create and fill smA & smB, create smC ... */
gan_squmat_addT_q ( &smA, &smB, &smC ); /* compute C = A + B^T, OR */
gan_squmat_incrementT ( &smA, &smB ); /* replace A = A + B^T, OR */
psmC = gan_squmat_addT_s ( &smA, &smB ); /* compute C = A + B^T as new matrix */
```

Error detection: `NULL` is returned and the Gandalf error handler invoked if the matrix addition fails. The most likely failure modes are failing to create/set the result matrix, or size/type incompatibility between the input matrices.

3.2.2.8 General size matrix subtraction

The routines for matrix subtraction follow the scheme of those for matrix addition, leading to the options

```
Gan_Matrix mA, mB, mC; /* declare matrices x, y and z */
Gan_Matrix *pmC; /* declare matrix z alternatively as pointer */

/* ... create and fill mA & mB, create mC ... */
gan_mat_sub_q ( &mA, &mB, &mC ); /* compute C = A - B */
pmC = gan_mat_sub_s ( &mA, &mB ); /* compute C = A - B */
gan_mat_sub_i1 ( &mA, &mB ); /* replace A = A - B */
gan_mat_sub_i2 ( &mA, &mB ); /* replace B = A - B */
gan_mat_decrement ( &mA, &mB ); /* replace A = A - B */
```

If one of the input matrices is to be implicitly transposed, use instead

```
Gan_Matrix mA, mB, mC; /* declare matrices x, y and z */
Gan_Matrix *pmC; /* declare matrix z alternatively as pointer */

/* ... create and fill mA & mB, create mC ... */

/* here B must have the same number of columns as A has rows, and vice versa */
gan_mat_subT_q ( &mA, &mB, &mC ); /* compute C = A - B^T */
pmC = gan_mat_subT_s ( &mA, &mB ); /* compute C = A - B^T */
gan_mat_decrementT ( &mA, &mB ); /* replace A = A - B^T */
```

There are equivalent functions for square matrices. Firstly the simple routines for subtracting two matrices:

```
Gan_SquMatrix smA, smB, smC, *psmC; /* declare matrices A, B & C */
```

```
/* ... create and fill smA & smB, create smC ... */
gan_squmat_sub_q ( &smA, &smB, &smC ); /* compute C = A - B, OR */
gan_squmat_sub_i1 ( &smA, &smB ); /* replace A = A - B, OR */
gan_squmat_sub_i2 ( &smA, &smB ); /* replace B = A - B, OR */
gan_squmat_decrement ( &smA, &smB ); /* replace A = A - B, OR */
psmC = gan_squmat_sub_s ( &smA, &smB ); /* compute C = A - B as new matrix */
```

Other routines implicitly transpose one of the input matrices:

```
Gan_SquMatrix smA, smB, smC, *psmC; /* declare matrices A, B & C */

/* ... create and fill smA & smB, create smC ... */
gan_squmat_subT_q ( &smA, &smB, &smC ); /* compute C = A - B^T, OR */
gan_squmat_decrementT ( &smA, &smB ); /* replace A = A - B^T, OR */
psmC = gan_squmat_subT_s ( &smA, &smB ); /* compute C = A - B^T as new matrix */
```

Error detection: `NULL` is returned and the Gandalf error handler invoked if the matrix addition fails. The most likely failure modes are failing to create/set the result matrix, or size/type incompatibility between the input matrices.

3.2.2.9 Rescaling a general size matrix

Multiplying or dividing a matrix by a scalar value follows the scheme of the above copy, addition and subtraction operations. To multiply a matrix A by a scalar s, $B = sA$, use for example

```
Gan_Matrix mA, mB; /* declare matrices A & B */

/* ... create & fill mA, create (& optionally fill) mB ... */
gan_mat_scale_q ( &mA, 5.0, &mB ); /* B = 5*A */
```

to multiply all the elements in matrix A by five, writing the result into matrix B. Alternatively you can create the rescaled matrix from scratch as in

```
Gan_Matrix *pmB; /* declare matrix B */

/* ... create & fill mA ... */
pmB = gan_mat_scale_s ( &mA, 5.0 ); /* B = 5*A */
```

or overwrite A with the result

```
gan_mat_scale_i ( &mA, 5.0 ); /* replace A = 5*A */
```

There are similar routines for dividing a general size matrix by a scalar value:

```
gan_mat_divide_q ( &mA, 5.0, &mB ); /* B = A/5 */
pmB = gan_mat_divide_s ( &mA, 5.0 ); /* B = A/5 */
gan_mat_divide_i ( &mA, 5.0 ); /* replace A = A/5 */
```

There are specific routines to negate a matrix, i.e. multiply it by -1, as follows:

```
gan_mat_negate_q ( &mA, &mB ); /* B = -A */
pmB = gan_mat_negate_s ( &mA ); /* B = -A */
gan_mat_negate_i ( &mA ); /* replace A = -A */
```

The equivalent routines for square matrices are

```
Gan_SquMatrix smA, smB, *psmB; /* declare matrices A & B */

/* ... create & fill smA, create (& optionally fill) smB ... */

/* scale a square matrix */
gan_squmat_scale_q ( &smA, 5.0, &smB ); /* B = 5*A, B an existing matrixOR */
psmB = gan_squmat_scale_s ( &smA, 5.0 ); /* B = 5*A, B a new matrix, OR */
gan_squmat_scale_i ( &smA, 5.0 ); /* replace A = 5*A */

/* divide a square matrix by a scalar */
gan_squmat_divide_q ( &smA, 5.0, &smB ); /* B = A/5, B an existing matrix, OR */
psmB = gan_squmat_divide_s ( &smA, 5.0 ); /* B = A/5, B a new matrix, OR */
gan_squmat_divide_i ( &smA, 5.0 ); /* replace A = A/5 */

/* negate a square matrix */
gan_squmat_negate_q ( &smA, &smB ); /* B = -A, B an existing matrix, OR */
psmB = gan_squmat_negate_s ( &smA ); /* B = -A, B a new matrix, OR */
gan_squmat_negate_i ( &smA ); /* replace A = -A */
```

Passing zero as the scalar value to the `gan_mat_divide_[qsi]()` or `gan_squmat_divide_[qsi]()` routines invokes the error handler, with a division by zero error (error code `GAN_ERROR_DIVISION_BY_ZERO`), and `NULL` is returned.

`Error detection`: The Gandalf error handler is invoked and `NULL` is returned if an error occurs. The most likely failure modes are (i) failing to create the result matrix; (ii) division by zero.

3.2.2.10 General size matrix/vector multiplication

The general size matrix/vector multiplication, with optional implicit transpose of the matrix, computes one of the operations

$$\mathbf{y} = A\mathbf{x} \quad \text{OR} \quad \mathbf{y} = A^\top \mathbf{x}$$

for vectors $\mathbf{x}$, $\mathbf{y}$ and matrix A. They are implemented in Gandalf as follows.

```
Gan_Matrix mA; /* matrix A */
Gan_Vector vx, vy; /* vectors x & y */

/* ... create and fill matrix A and vector x, create (and optionally
       fill) vector y ... */
gan_mat_multv_q ( &mA, &vx, &vy ); /* set y = A*x, OR */
gan_matT_multv_q ( &mA, &vx, &vy ); /* set y = A^T*x */
```

with the alternative forms

```
Gan_Matrix mA; /* matrix A */
Gan_Vector vx, *pvy; /* vectors x & y */

/* ... create and fill matrix A and vector x ... */
pvy = gan_mat_multv_s ( &mA, &vx ); /* set y = A*x, y a new vector, OR */
pvy = gan_matT_multv_s ( &mA, &vx ); /* set y = A^T*x, y a new vector */
```

If A is a special square matrix, more options are available. If A is a triangular matrix, multiplication with a vector can be implemented as an in-place operation, whether or not A is (implicitly) inverted or transposed, in any combination. This gives rise to the following Gandalf routines.

```
Gan_SquMatrix smA; /* matrix A */
Gan_Vector vx, vy; /* vectors x & y */

/* ... create and fill matrix A and vector x, create (and optionally
       fill) vector y ... */
gan_squmat_multv_q ( &smA, &vx, &vy ); /* set y = A*x, OR */
gan_squmatT_multv_q ( &smA, &vx, &vy ); /* set y = A^T*x, OR */
gan_squmatI_multv_q ( &smA, &vx, &vy ); /* set y = A^-1*x, OR */
gan_squmatIT_multv_q ( &smA, &vx, &vy ); /* set y = A^-T*x */
```

with in-place versions

```
Gan_SquMatrix smA; /* matrix A */
Gan_Vector vx; /* vector x */

/* ... create and fill matrix A and vector x ... */
gan_squmat_multv_i ( &smA, &vxy ); /* replace x = A*x, OR */
gan_squmatT_multv_i ( &smA, &vx ); /* replace x = A^T*x, OR */
gan_squmatI_multv_i ( &smA, &vx ); /* replace x = A^-1*x, OR */
gan_squmatIT_multv_i ( &smA, &vx ); /* replace x = A^-T*x */
```

and also the routines to create the result vector from scratch:

```
Gan_SquMatrix smA; /* matrix A */
Gan_Vector vx, *pvy; /* vectors x & y */

/* ... create and fill matrix A and vector x ... */
pvy = gan_squmat_multv_s ( &smA, &vxy ); /* set y = A*x, OR */
pvy = gan_squmatT_multv_s ( &smA, &vx ); /* set y = A^T*x, OR */
pvy = gan_squmatI_multv_s ( &smA, &vx ); /* set y = A^-1*x, OR */
pvy = gan_squmatIT_multv_s ( &smA, &vx ); /* set y = A^-T*x */
```

Note that the implicit inverse and in-place features are not available when A is of symmetric type; Gandalf will invoke the error handler and return an error condition `NULL` if `mA` has type `GAN_SYMMETRIC_MATRIX`.

Error detection: If implicit inverse is used (the `gan_squmatI_multv_[qsi]()` or `gan_squmatIT_multv_[qsi]()` routines), the matrix must be non-singular. If the matrix is singular then `NULL` is returned and the Gandalf error handler is invoked. Other failure modes are failing to create the result vector and incompatibility between the sizes of the input matrix and vector.

3.2.2.11 General size matrix/matrix multiplication

Similar options are available to matrix/matrix multiplication as with matrix/vector multiplication, with the added complication that either or both if the matrices may have an implicit transpose or (for square matrices) inverse applied to them. Firstly we list the routines available when both input matrices are general rectangular matrices. In this case only implicit transpose is of relevance to us, and we can write the operations we need to implement as

$$C = AB, \quad C = A^\top B, \quad C = AB^\top, \quad C = A^\top B^\top$$

To right-multiply a matrix A by another matrix B, with all the above transpose combinations, the Gandalf routines are

```
Gan_Matrix mA, mB, mC; /* matrices A, B & C */
```

```
/* ... create and fill matrices A, B, create matrix C ... */
gan_mat_rmult_q ( &mA, &mB, &mC ); /* C = A*B, OR */
gan_mat_rmultT_q ( &mA, &mB, &mC ); /* C = A*B^T, OR */
gan_matT_rmult_q ( &mA, &mB, &mC ); /* C = A^T*B, OR */
gan_matT_rmultT_q ( &mA, &mB, &mC ); /* C = A^T*B^T, OR */
```

with similar routines to create the result matrix C from scratch:

```
Gan_Matrix mA, mB, *pmC; /* matrices A, B & C */

/* ... create and fill matrices A, B ... */
pmC = gan_mat_rmult_s ( &mA, &mB ); /* C = A*B, OR */
pmC = gan_mat_rmultT_s ( &mA, &mB ); /* C = A*B^T, OR */
pmC = gan_matT_rmult_s ( &mA, &mB ); /* C = A^T*B, OR */
pmC = gan_matT_rmultT_s ( &mA, &mB ); /* C = A^T*B^T, OR */
```

The next set of routines deals with the case where it is known that the result of multiplying matrices A and B is symmetric. In that case we can use the routines

```
Gan_Matrix mA, mB; /* matrices A & B */
Gan_SquMatrix smC; /* matrix C */

/* ... create and fill matrices A, B, create matrix C ... */
gan_mat_rmult_sym_q ( &mA, &mB, &mC ); /* C = A*B, OR */
gan_mat_rmultT_sym_q ( &mA, &mB, &mC ); /* C = A*B^T, OR */
gan_matT_rmult_sym_q ( &mA, &mB, &mC ); /* C = A^T*B, OR */
gan_matT_rmultT_sym_q ( &mA, &mB, &mC ); /* C = A^T*B^T */
```

with the alternatives

```
Gan_Matrix mA, mB; /* matrices A & B */
Gan_SquMatrix *psmC; /* matrix C */

/* ... create and fill matrices A, B ... */
psmC = gan_mat_rmult_sym_s ( &mA, &mB ); /* C = A*B, OR */
psmC = gan_mat_rmultT_sym_s ( &mA, &mB ); /* C = A*B^T, OR */
psmC = gan_matT_rmult_sym_s ( &mA, &mB ); /* C = A^T*B, OR */
psmC = gan_matT_rmultT_sym_s ( &mA, &mB ); /* C = A^T*B^T */
```

In the case that A and B are the same matrix, we can be sure that both $AA^\top$ and $A^\top A$ are symmetric, and there are special Gandalf routines for these cases, distinguished according to whether A is multiplied by its own transpose on the **r**ight or **l**eft:

```
Gan_Matrix mA; /* matrix A */
Gan_SquMatrix smC; /* matrix C */

/* ... create and fill matrix A, create matrix C ... */
gan_mat_srmultT_q ( &mA, &mC ); /* C = A*A^T, OR */
gan_mat_slmultT_q ( &mA, &mC ); /* C = A^T*A */
```

with the alternatives

```
Gan_Matrix mA; /* matrix A */
Gan_SquMatrix *psmC; /* matrix C */
```

```
/* ... create and fill matrix A ... */
psmC = gan_mat_srmultT_s ( &mA ); /* C = A*A^T, OR */
psmC = gan_mat_slmultT_s ( &mA ); /* C = A^T*A */
```

If one or both of the input matrices is a special square matrix, there are many more combinations available. First consider a square matrix A being multiplied on left or right by a general rectangular matrix B, giving a result matrix C. Given the possibility of both implicit transpose and inverse of the square matrix, we need to consider the operations

$$C = AB, \quad C = AB^\top, \quad C = A^\top B, \quad C = A^\top B^\top,$$
$$C = A^{-1}B, \quad C = A^{-1}B^\top, \quad C = A^{-\top}B, \quad C = A^{-\top}B^\top$$
$$C = BA, \quad C = B^\top A, \quad C = BA^\top, \quad C = B^\top A^\top,$$
$$C = BA^{-1}, \quad C = B^\top A^{-1}, \quad C = BA^{-\top}, \quad C = B^\top A^{-\top}$$

These operations are implemented by the following Gandalf routines:

```
Gan_SquMatrix smA; /* square matrix A */
Gan_Matrix mB, mC; /* matrices B & C */

/* ... create and fill matrices A, B, create matrix C ... */

/* routines right-multipling A by B */
gan_squmat_rmult_q ( &smA, &mB, &mC ); /* C = A*B, OR */
gan_squmat_rmultT_q ( &smA, &mB, &mC ); /* C = A*B^T, OR */
gan_squmatT_rmult_q ( &smA, &mB, &mC ); /* C = A^T*B, OR */
gan_squmatT_rmultT_q ( &smA, &mB, &mC ); /* C = A^T*B^T, OR */
gan_squmatI_rmult_q ( &smA, &mB, &mC ); /* C = A^-1*B, OR */
gan_squmatI_rmultT_q ( &smA, &mB, &mC ); /* C = A^-1*B^T, OR */
gan_squmatIT_rmult_q ( &smA, &mB, &mC ); /* C = A^-T*B, OR */
gan_squmatIT_rmultT_q ( &smA, &mB, &mC ); /* C = A^-T*B^T */

/* routines left-multipling A by B */
gan_squmat_lmult_q ( &smA, &mB, &mC ); /* C = B*A, OR */
gan_squmat_lmultT_q ( &smA, &mB, &mC ); /* C = B^T*A, OR */
gan_squmatT_lmult_q ( &smA, &mB, &mC ); /* C = B*A^T, OR */
gan_squmatT_lmultT_q ( &smA, &mB, &mC ); /* C = B^T*A^T, OR */
gan_squmatI_lmult_q ( &smA, &mB, &mC ); /* C = B*A^-1, OR */
gan_squmatI_lmultT_q ( &smA, &mB, &mC ); /* C = B^T*A^-1, OR */
gan_squmatIT_lmult_q ( &smA, &mB, &mC ); /* C = B*A^-T, OR */
gan_squmatIT_lmultT_q ( &smA, &mB, &mC ); /* C = B^T*A^-T */
```

These routines have the alternative form

```
Gan_SquMatrix smA; /* square matrix A */
Gan_Matrix mB, *pmC; /* matrices B & C */

/* ... create and fill matrices A, B ... */

/* routines right-multipling A by B */
pmC = gan_squmat_rmult_s ( &smA, &mB ); /* C = A*B, OR */
pmC = gan_squmat_rmultT_s ( &smA, &mB ); /* C = A*B^T, OR */
pmC = gan_squmatT_rmult_s ( &smA, &mB ); /* C = A^T*B, OR */
pmC = gan_squmatT_rmultT_s ( &smA, &mB ); /* C = A^T*B^T, OR */
```

```
pmC = gan_squmatI_rmult_s ( &smA, &mB ); /* C = A^-1*B, OR */
pmC = gan_squmatI_rmultT_s ( &smA, &mB ); /* C = A^-1*B^T, OR */
pmC = gan_squmatIT_rmult_s ( &smA, &mB ); /* C = A^-T*B, OR */
pmC = gan_squmatIT_rmultT_s ( &smA, &mB ); /* C = A^-T*B^T */

/* routines left-multipling A by B */
pmC = gan_squmat_lmult_s ( &smA, &mB ); /* C = B*A, OR */
pmC = gan_squmat_lmultT_s ( &smA, &mB ); /* C = B^T*A, OR */
pmC = gan_squmatT_lmult_s ( &smA, &mB ); /* C = B*A^T, OR */
pmC = gan_squmatT_lmultT_s ( &smA, &mB ); /* C = B^T*A^T, OR */
pmC = gan_squmatI_lmult_s ( &smA, &mB ); /* C = B*A^-1, OR */
pmC = gan_squmatI_lmultT_s ( &smA, &mB ); /* C = B^T*A^-1, OR */
pmC = gan_squmatIT_lmult_s ( &smA, &mB ); /* C = B*A^-T, OR */
pmC = gan_squmatIT_lmultT_s ( &smA, &mB ); /* C = B^T*A^-T */
```

The in-place versions will overwrite the contents of matrix B with the result, and work fine unless A is of symmetric type (in which case the error handler is invoked and `NULL` returned):

```
Gan_SquMatrix smA; /* square matrix A */
Gan_Matrix mB; /* matrix B */

/* ... create and fill matrices A, B ... */

/* routines right-multipling A by B */
gan_squmat_rmult_i ( &smA, &mB ); /* replace B = A*B, OR */
gan_squmat_rmultT_i ( &smA, &mB ); /* replace B = A*B^T, OR */
gan_squmatT_rmult_i ( &smA, &mB ); /* replace B = A^T*B, OR */
gan_squmatT_rmultT_i ( &smA, &mB ); /* replace B = A^T*B^T, OR */
gan_squmatI_rmult_i ( &smA, &mB ); /* replace B = A^-1*B, OR */
gan_squmatI_rmultT_i ( &smA, &mB ); /* replace B = A^-1*B^T, OR */
gan_squmatIT_rmult_i ( &smA, &mB ); /* replace B = A^-T*B, OR */
gan_squmatIT_rmultT_i ( &smA, &mB ); /* replace B = A^-T*B^T */

/* routines left-multipling A by B */
gan_squmat_lmult_i ( &smA, &mB ); /* replace B = B*A, OR */
gan_squmat_lmultT_i ( &smA, &mB ); /* replace B = B^T*A, OR */
gan_squmatT_lmult_i ( &smA, &mB ); /* replace B = B*A^T, OR */
gan_squmatT_lmultT_i ( &smA, &mB ); /* replace B = B^T*A^T, OR */
gan_squmatI_lmult_i ( &smA, &mB ); /* replace B = B*A^-1, OR */
gan_squmatI_lmultT_i ( &smA, &mB ); /* replace B = B^T*A^-1, OR */
gan_squmatIT_lmult_i ( &smA, &mB ); /* replace B = B*A^-T, OR */
gan_squmatIT_lmultT_i ( &smA, &mB ); /* replace B = B^T*A^-T */
```

Now we consider multiplying a square matrix A by its own transpose, producing a symmetric matrix B. This operation will most often be applied to triangular matrices, and in Gandalf implicit transpose and inverse of the input matrix are supported, giving rise to the operations

$$B = AA^{\top}, \quad B = A^{\top}A, \quad B = A^{-1}A^{-\top}, \quad B = A^{-\top}A^{-1}$$

which are implemented by the routines

```
Gan_SquMatrix smA, smB; /* declare matrices A & B */

/* ... create & fill matrix A, create (& optionally fill) matrix B ... */
gan_squmat_srmultT_squ_q ( &smA, &smB ); /* set B = A*A^T, OR */
```

```
gan_squmatT_srmult_squ_q ( &smA, &smB ); /* set B = A^T*A, OR */
gan_squmatI_srmultIT_squ_q ( &smA, &smB ); /* set B = A^-1*A^-T, OR */
gan_squmatIT_srmultI_squ_q ( &smA, &smB ); /* set B = A^-T*A^-1 */
```

There are also routines to build the result matrix B from scratch:

```
Gan_SquMatrix smA, *psmB; /* declare matrices A & B */

/* ... create & fill matrix A ... */
psmB = gan_squmat_srmultT_squ_s ( &smA ); /* create B = A*A^T, OR */
psmB = gan_squmatT_srmult_squ_s ( &smA ); /* create B = A^T*A, OR */
psmB = gan_squmatI_srmultIT_squ_s ( &smA ); /* create B = A^-1*A^-T, OR */
psmB = gan_squmatIT_srmultI_squ_s ( &smA ); /* create B = A^-T*A^-1 */
```

and in-place versions of these operations are also available:

```
Gan_SquMatrix smA; /* declare matrix A */

/* ... create & fill matrix A ... */
gan_squmat_srmultT_squ_i ( &smA ); /* replace A = A*A^T, OR */
gan_squmatT_srmult_squ_i ( &smA ); /* replace A = A^T*A, OR */
gan_squmatI_srmultIT_squ_i ( &smA ); /* replace A = A^-1*A^-T, OR */
gan_squmatIT_srmultI_squ_i ( &smA ); /* replace A = A^-T*A^-1 */
```

Finally, there is a set of routines that multiply a symmetric matrix on left and right by a rectangular matrix and its transpose, producing another symmetric matrix. The operations implemented are

$$S' = ASA^\top, \qquad S' = A^\top SA.$$

This triple product is implemented as two matrix multiplications, and the matrix to hold the intermediate result is also passed in to the routines, so that it is also available on output. The routines are

```
Gan_SquMatrix smS, smSp; /* declare matrices S & S' */
Gan_Matrix mA, mB; /* declare matrices A & B */

/* ... create & fill matrices S & A, create (& optionally fill) matrices B & Sp ... */
gan_symmat_lrmult_q ( &smS, &mA, &mB, &smSp ); /* set B = S*A^T and Sp = A*S*A^T, OR */
gan_symmat_lrmultT_q ( &smS, &mA, &mB, &smSp ); /* set B = S*A and Sp = A^T*S*A */
```

with alternative versions that create the result matrix S' from scratch:

```
Gan_SquMatrix smS, *psmSp; /* declare matrices S & S' */
Gan_Matrix mA, mB; /* declare matrices A & B */

/* ... create & fill matrices S & A, create (& optionally fill) matrix B ... */
psmSp = gan_symmat_lrmult_s ( &smS, &mA, &mB ); /* set B = S*A^T and Sp = A*S*A^T, OR */
psmSp = gan_symmat_lrmultT_s ( &smS, &mA, &mB ); /* set B = S*A and Sp = A^T*S*A */
```

It is allowable to pass `NULL` for the B matrix (`&mB` in the above function calls). In that case the intermediate result is computed and thrown away.

3.2.2.12 Inverting a general size matrix

If a general rectangular matrix A happens to be square, it can be inverted using the routine

```
Gan_Matrix mA, mB; /* declare matrices A & B */

/* ... create and fill matrix A, which must be square, create B ... */
gan_mat_invert_q ( &mA, &mB ); /* B = A^-1 */
```

There is also a routine to create the inverse matrix A^{-1} from scratch:

```
Gan_Matrix mA, *pmB; /* declare matrix A */

/* ... create and fill matrix A, which must be square ... */
pmB = gan_mat_invert_s ( &mA ); /* B = A^-1 */
```

The routines for special square matrices are similar

```
Gan_Matrix mA, mB, *pmB; /* declare matrices A & B */

/* ... create and fill matrix A, which must be square, create B ... */
gan_squmat_invert_q ( &mA, &mB ); /* B = A^-1, OR */
pmB = gan_squmat_invert_s ( &mA ); /* B = A^-1 */
```

The type of the output B is in this case set to the appropriate type given the input. For all the square matrix types supported by Gandalf (symmetric, triangular, diagonal, scaled identity), the matrix type of the inverse B is the same as that of the input matrix A.

Error detection: If implicit inverse is used (the `..._squmatI_...()` or `..._squmatIT_...()` routines), the square matrix involved must be non-singular. If the matrix is singular then `NULL` is returned and the Gandalf error handler is invoked. Other failure modes are failing to create the result matrix and incompatibility between the sizes of the input matrices.

3.2.2.13 Cholesky factorising a general size symmetric matrix

The Cholesky factor of a symmetric positive definite matrix S is a lower triangular matrix L such that

$$S = LL^\top$$

and we write $L = \text{chol}(S)$. The Gandalf Cholesky factorisation routines apply to all symmetric types of matrix, i.e. `GAN_SYMMETRIC_MATRIX` itself, `GAN_DIAGONAL_MATRIX` and `GAN_SCALED_IDENT_MATRIX`. Routines are available to compute the factorisation, with the usual options, as follows:

```
Gan_SquMatrix smS, smL, *psmL; /* declare matrices S & L */

/* ... create and fill matrix S, which must be symmetric and positive definite,
       create L ... */
gan_symmat_cholesky_q ( &smS, &smL ); /* L = chol(S), OR */
psmL = gan_symmat_cholesky_s ( &smS ); /* L = chol(S) */
gan_symmat_cholesky_i ( &smS ); /* replace S = chol(S) */
```

The last option `gan_symmat_cholesky_i()` replaces S in-place by $\text{chol}(S)$.

Error detection: If S is not either symmetric or positive definite in the above routines, `NULL` is returned and the Gandalf error handler is invoked. Another failure mode is failing to create the result matrix.

3.2.2.14 Symmetric matrix eigendecomposition

```
#include <gandalf/linalg/mat_symmetric.h>
```

Gandalf has a routine for computing the real eigenvalues and eigenvectors of a general size symmetric matrix, based on either the CLAPACK routine `dspev()` or the CCMath routine `eigval()`. A symmetric matrix S can be written as

$$SZ = ZW$$

where W is a diagonal matrix of real eigenvalues and Z is a square matrix of orthognal eigenvectors, unique if the eigenvalues are distinct. If the matrix is positive definite (or semi-definite) then all the eigenvalues will be > 0 (or ≥ 0). Here is an example code fragment using the Gandalf routine to compute W and (optionally) Z.

```
Gan_SquMatrix smS; /* declare symmetric matrix */
Gan_SquMatrix smW; /* declare matrix of eigenvalues W */
Gan_Matrix mZ; /* declare matrix of eigenvectors */

/* create and fill S */
gan_symmat_form ( &smS, 5 );
gan_symmat_fill_va ( &smS, 5,
                      1.0,
                      2.0,  3.0,
                      4.0,  5.0,  6.0,
                      7.0,  8.0,  9.0, 10.0,
                     11.0, 12.0, 13.0, 14.0, 15.0 );

/* create Z and W */
gan_mat_form ( &mZ, 5, 5 );
gan_diagmat_form ( &smW, 0 );

/* compute sigenvalues and eigenvectors of S */
gan_symmat_eigen ( &smS, &smW, &mZ, GAN_TRUE, NULL, 0 );
```

After calling this routine `smW` will contain the computed eigenvalues, and `mZ` the eigenvectors. If the eigenvector matrix is passed as `NULL`, the eigenvectors are not computed. The boolean fourth argument indicates whether the eigenvectors should be sorted into ascending order. The fifth and sixth arguments define a workspace array of `doubles`, and the size of the array, which can be used by LAPACK. If passed as `NULL, 0` as above, the workspace is allocated inside the function.

3.2.2.15 Accumulated symmetric matrix eigendecomposition

```
#include <gandalf/linalg/symmat_eigen.h>
```

There is also a specific module in Gandalf to compute the eigenvalues and eigenvectors of a positive semi-definite matrix accumulated as a sum of outer products of vectors. This is useful for instance when solving homogeneous linear equations; for an example see the computation of the fundamental & essential matrix in Section 5.2. The symmetric matrix S is constructed as

$$S = \sum_{i=1}^{n} \sigma_i \mathbf{x}_i \mathbf{x}_i^\top \tag{3.2}$$

given the n vectors $\mathbf{x}_i$ and weighting factors s_i, $i = 1, \ldots, n$.

There is a structure to hold the accumulated matrix and the resulting eigendecomposition matrices:

```
/* structure to hold accumulated symmetric matrix and resulting
 * eigendecomposition of a sum of vector outer products
 */
typedef struct
```

```
{
   Gan_SquMatrix SxxT; /* accumulated sum of vector outer products */
   Gan_SquMatrix W;    /* diagonal matrix of eigenvalues */
   Gan_Matrix    Z;    /* matrix of eigenvectors */
   Gan_Vector    work; /* workspace vector for computing eigendecomposition */

   /* whether this structure was dynamically allocated */
   Gan_Bool alloc;
} Gan_SymMatEigenStruct;
```

To start the computation, create an instance of the structure using

```
Gan_SymMatEigenStruct SymEigen;

/* create structure for computing eigenvalues and eigenvectors,
   initialising accumulated matrix S (here 5x5) to zero */
gan_symeigen_form ( &SymEigen, 5 );
```

This routine creates the structure and its constituent matrices, and initialises the outer product sum matrix, in this case a 5×5 matrix, to zero. You can then add a term to the sum in Equation 3.2 using the routine

```
/* increment matrix S by
gan_symeigen_increment ( &SymEigen, 2.0, /* weighting factor s */
                         3.4, 1.0, 9.7, 3.4, 2.1 ); /* elements of vector x */
```

Here the first value 2.0 is the weighting factor σ for this vector, and the elements of the vector $\mathbf{x}$ are passed into a variable argument list (and hence have to be explicitly `double` type). You should call `gan_symeigen_increment()` once for each of the n vectors. Then solve for the eigenvalues & eigenvectors using the routine

```
/* solve for eigenvalues and eigenvectors */
gan_symeigen_solve ( &SymEigen );
```

after which the eigenvalues can be read back from `SymEigen.W` and the eigenvectors from `SymEigen.Z`. If you want to reuse the structure on a new eigendecomposition computation, call the routine

```
/* reset accumulated symmetric to zero, optionally changing size of matrix */
gan_symeigen_reset ( &SymEigen, 5 );
```

where the last argument allows you to change the size of the matrix if desired. Finally to free the structure use

```
/* free eigensystem structure and constituent matrices */
gan_symeigen_free ( &SymEigen );
```

Error detection: `sym_symeigen_form()` returns a pointer to the eigensystem structure (`&SymEigen`), and so returns `NULL` on error. `sym_symeigen_increment()`, `sym_symeigen_solve()` and `sym_symeigen_reset()` return boolean values, so `GAN_FALSE` indicates an error. In all cases the Gandalf error handler is invoked.

3.2.3 Single precision general size matrices & vectors

Note that for the routines to fill a matrix/vector with values, described in Sections 3.2.1.3 and 3.2.2.3, the values provided in the single precision case must actually still be double precision, because C has the restriction that floating point variable argument list values must be double precision. So for instance, this won't work:

```
Gan_Vector_f vx;

gan_vecf_form ( &vx, 6 );
gan_vecf_fill_va ( &vx, 6, 1.0F, 2.0F, 3.0F, 4.0F, 5.0F, 6.0F ); /* WRONG */
```

Instead use

```
gan_vecf_fill_va ( &vx, 6, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0 ); /* RIGHT! */
```

If necessary use a `(double)` cast in front of each value.

Chapter 4

The Image Package

The **image** package covers images of all formats and types, and defines low-level image manipulation routines. To be able to use any routine or structure in the image package use the declaration

```
#include <gandalf/image.h>
```

but including individual module header files instead will speed up program compilation.

4.1 Image formats and types

Gandalf distinguishes between the external `file` format of an image and the an internal image format used by Gandalf to represent the image data. The former is often used to select the latter, but it is important to separate the two. The available internal Gandalf image formats are defined by the `Gan_ImageFormat` enumerated type, found in `<gandalf/image/pixel.h>`:

```
typedef enum { GAN_GREY_LEVEL_IMAGE,       /* grey-level images */
               GAN_GREY_LEVEL_ALPHA_IMAGE, /* grey-level images with alpha
                                              channel */
               GAN_RGB_COLOUR_IMAGE,       /* RGB colour images */
               GAN_RGB_COLOUR_ALPHA_IMAGE, /* RGB colour images with alpha
                                              channel */
               GAN_VECTOR_FIELD_2D,        /* 2D image of 2D vectors */
               GAN_VECTOR_FIELD_3D }       /* 2D image of 3D vectors */
  Gan_ImageFormat;
```

The formats are (hopefully) self-explanatory, and allow Gandalf to represent most useful kinds of image data. Along with the format there is also an image **type**, which determines what type of data is stored in each pixel of the image. The `Gan_Type` enumerated type is used to distinguish the image type. It allows for instance boolean, unsigned character, short integer or floating point types to be defined, and is described in Section 2.1.1. Note that not all types are supported by each format. For instance boolean images (`GAN_BOOL` type) are only supported as grey-level format images (format `GAN_GREY_LEVEL_IMAGE`), and the vector field formats `GAN_VECTOR_FIELD_2D` and `GAN_VECTOR_FIELD_3D` are currently supported only for signed types. Support for extra types can be added when required.

There is also a structure to represent a rectangular sub-window of an image, useful in many situations:

```
/* Definition of a rectangular sub-part of an image.
 */
```

```
typedef struct Gan_ImageWindow
{
   /* offset of window relative to top-left corner of the image */
   unsigned c0, r0;

   /* dimensions of window */
   unsigned width, height;
} Gan_ImageWindow;
```

Apart from the standard simple C types, boolean and pointer image types are supported. These are described in Sections 4.4 and 4.5.

4.2 Simple image/pixel routines

To minimise computational overheads, Gandalf provides a common set of low level image and pixel handling routines specific to each supported format and type of image. We will illustrate the routines with reference to two representative examples:

1. Grey-level signed short integer images. The format here is `GAN_GREY_LEVEL_IMAGE`, and the type is `GAN_SHORT`. To use the routines specific to this format and type use the header file

   ```
   #include <gandalf/image/image_gl_short.h>
   ```

2. RGB colour unsigned character images. The format here is `GAN_RGB_COLOUR_IMAGE`, and the type is `GAN_UCHAR`. To use the routines specific to this format and type use the header file

   ```
   #include <gandalf/image/image_rgb_uchar.h>
   ```

We need to use two examples because the way that pixels are handled for simple grey-level images is different to RGB colour and other image formats, since in the latter and related formats (every format except grey-level) a pixel is represented as a structure rather than a simple C object.

There are also higher level routines that work with all the formats and types, accessible through the header file.

```
#include <gandalf/image/image_rgb_uchar.h>
```

Examples of these are also provided in the following sections.

4.2.1 Image creation/destruction

In the same manner as the general size matrix/vector package, Gandalf images can be represented either using structures or pointers to structures. The normal Gandalf convention is to use pointers, because images are relatively large objects, and the extra overhead of having to use `malloc()` to create the image structure is insignificant relative to the computation time needed to process the image. We will follow this convention, but bear in mind that either convention is possible. To create a grey-level short integer image, use the routine

```
Gan_Image *pImage;

pImage = gan_image_alloc_gl_s ( 150, 100 );
```

This creates an image with dimensions 150 (height) by 100 (width). The same operation using an image structure rather than a pointer would be

```
Gan_Image Image;

gan_image_form_gl_s ( &Image, 150, 100 );
```

To free the image, use

```
gan_image_free ( pImage ); /* OR */
gan_image_free ( &Image );
```

This function can be used universally to free a Gandalf image created in any of the ways described here.

Sometimes the image data array is already present in memory, and we want to create a Gandalf image that points to the data. Let us assume that the data for a 256×256 pixel grey-level short integer image is available in an array `asData`. Assume for now that it is a contiguous array, stored with rows following rows consecutively without any gaps, i.e. an array of 65536 elements. Then to build a Gandalf image that points into this data we might use the code

```
short asData[65536];

/* ... set up array asData with image data ... */
pImage = gan_image_alloc_data_gl_s ( 256, 256, 256*sizeof(short),
                                     asData, 65536, NULL, 0 ); /* OR */
gan_image_form_data_gl_s ( &Image, 256, 256, 256*sizeof(short),
                           asData, 65536, NULL, 0 );
```

After the height and width arguments is a "stride" argument, which indicates the separation in memory between adjacent rows of the image, as stored in the `asData` array. Here it is 256 pixels (the image width), but since stride is measured in bytes, we need to multiply by the pixel size, as here. The data array `asData` is passed in along with its size in pixels (65536). The size is passed mainly as a means of error checking: if the requested Gandalf image as defined by the height, width and stride were to exceed the size of the data array, it would be an error, the Gandalf error handler would be invoked, and `NULL` would be returned. Here the data array size and the image size match exactly. The final two arguments allow the programmer also to pass in an array of *row pointers* which point into the start of each row of the image. Here we pass `NULL` for the row pointers, which means that they will be allocated inside the function.

Note that the `Gan_Image` structure stores the information concerning which parts of the structure were dynamically allocated: the structure itself, the image data array and the row pointer array. `gan_image_free()` then knows which bits to free.

A slightly more complex example is when the rows of the image as stored in the data array are not contiguous in memory. This might happen for instance in frame-grabber (video) memory, where the hardware might restrict the stride to a fixed number of bytes, say 1024. We shall also provide an array of row pointers to the image creation function. Then we would have to call the above functions as follows:

```
short *psData, *apRowPointer[256];

/* ... set psData to point to video memory ... */
pImage = gan_image_alloc_data_gl_s ( 256, 256, 1024,
                                     asData, 65536, apRowPointer, 256 ); /* OR */
gan_image_form_data_gl_s ( &Image, 256, 256, 256*sizeof(short),
                           asData, 131072, apRowPointer, 256 );
```

Here we assume that `shorts` are 2 bytes. These function calls will set the Gandalf image to point directly into the video memory, so that if desired the image stored may be copied for further processing (see below) or processed directly.

For RGB unsigned character images, the function calls would be similar:

```
Gan_Image *pImage, Image;
Gan_RGBPixel_uc argbucData[65536];

pImage = gan_image_alloc_rgb_uc ( 150, 100 ); /* OR */
gan_image_form_rgb_uc ( &Image, 150, 100 ); /* OR */

/* ... set up array aucData with image data ... */
pImage = gan_image_alloc_data_rgb_uc ( 256, 256, 256*sizeof(unsigned char),
                                       argbucData, 65536, NULL, 0 ); /* OR */
gan_image_form_data_rgb_uc ( &Image, 256, 256, 256*sizeof(unsigned char),
                             argbucData, 65536, NULL, 0 );
```

For RGB and other similar formats, Gandalf assumes that the channels for each pixel are grouped in memory, so that a pixel can be represented as a structure, rather than the channels being stored in separate arrays. For RGB unsigned character images, the pixel structure is `Gan_RGBPixel_uc`, as defined in `<gandalf/image/pixel.h>`:

```
/* Structure defining RGB colour unsigned character pixel
 */
typedef struct Gan_RGBPixel_uc
{
   unsigned char R, G, B;
} Gan_RGBPixel_uc;
```

A different structure type is defined for each image format (apart from grey-level) and type.

There are also higher level functions which create a Gandalf images using arguments to determine the format and type. Use these functions only if the format/type is determined at run-time. An example emulating the above examples for grey-level images is

```
Gan_Image *pImage, Image;

pImage = gan_image_alloc ( GAN_GREY_LEVEL_IMAGE, GAN_SHORT, 150, 100 ); /* OR */
gan_image_form ( &Image, GAN_GREY_LEVEL_IMAGE, GAN_SHORT, 150, 100 );
```

Error detection: All the above routines return `NULL` and invoke the Gandalf error handler if they fail. The most likely failure modes are failing to allocate the data required (i.e. internal `malloc()` or `realloc()` calls failing), or passing too small an array into the `..._alloc_data...()` or `..._form_data...()` routines.

4.3 Image file I/O

```
#include <gandalf/image/io/image_io.h>
```

Currently Gandalf supports six image file formats: PNG, PBM, PGM, PPM, TIFF and JPEG. These are described by the
`Gan_ImageFormat` enumerated type:

```
/* image file formats supported by Gandalf */
typedef enum
{
   GAN_PNG_FORMAT,  /**< PNG image format */
   GAN_PBM_FORMAT,  /**< Portable bitmap image format */
   GAN_PGM_FORMAT,  /**< Portable greymap image format */
   GAN_PPM_FORMAT,  /**< Portable pixmap image format */
```

```
   GAN_TIFF_FORMAT, /**< TIFF image format */
   GAN_JPEG_FORMAT, /**< JPEG image format */
   GAN_UNKNOWN_FORMAT /**< Unknown Image Format */
} Gan_ImageFileFormat;
```

PBM, PGM and PPM are very simple formats, for boolean, grey-level and RGB colour image formats respectively, and the code to implement I/O in those formats is built into Gandalf, although currently only binary file formats are supported. PNG, TIFF and JPEG formats are considerably more complex, and require specific libraries to be installed. The Gandalf `configure` script detects the presence of the PNG, TIFF and JPEG libraries, and only compiles in the I/O code for those formats when the relevant libraries are detected on the host system.

The `image_io.h` header file contains declaration of the basic image I/O functions. To read a image from a PNG image file, for instance, you can use the code

```
Gan_Image *pImage;

/* read the image from a file in PNG format */
pImage = gan_image_read ( "image1.png", GAN_PNG_FORMAT, NULL );
```

The first argument is the file name, the second the file format (Gandalf doesn't currently support automatic file format determination via magic numbers. Who wants to volunteer?). The last argument is either a pointer to an already created image structure or `NULL`, as here. In the latter case the image is created inside the `gan_image_read()` function and returned.

To write an RGB unsigned character image to a PNG file you could write

```
Gan_Image *pRGBImage;

/* ... create and fill RGB unsigned character image ... */

/* output the image to a file in PNG format */
gan_image_write ( "image1.png", GAN_PNG_FORMAT, pRGBImage, 0.0 );
```

We recommend that where possible you should use the PNG format. It is the most flexible of the formats supported by Gandalf, allowing alpha channels to be stored with the image, and also supporting binary images. PPM images are restricted to unsigned character type (`GAN_UCHAR`), while PGM format supports unsigned character and binary (`GAN_BOOL`) type. However the binary support in PGM files is very inefficient, storing one byte per pixel, so again PNG is the better format.

4.3.1 Setting an image to a new format, type and dimensions

Once an image has been created with a certain format, type, width and height, that is not the end of the matter. In a similar manner to the Gandalf general size vectors and matrices, the format, type and dimensions of a Gandalf image may be changed an arbitrary number of times. Gandalf will reallocate the image data array and row pointer array as necessary, as the internal attributes of the image are changed. However note that if the image data array or row pointer array is passed in by the user program, as in `gan_image_alloc_data_gl_short()` above, The provided array(s) cannot be reallocated, so care should be taken never to attempt to set format/type/dimensions that cause the array bounds to be exceeded.

Let us take one of the above examples and modify it a bit.

```
Gan_Image *pImage;

/* create image */
pImage = gan_image_alloc_gl_s ( 150, 100 );
```

```
/* convert an existing image to new format, type & dimensions */
gan_image_set_rgb_uc ( pImage, 300, 200 );
```

This code fragment allocates an image as a 150×100 grey-level short integer image, and then converts it into a 300×200 RGB unsigned character image. This feature allows the same image to be used as the result of several different computations, easing the burden of keeping track of a large number of images, as well as potentially saving memory.

There are also higher level functions that set an image to a format, type and dimensions all selected by variables. For instance

```
Gan_Image *pImage;

/* create image */
pImage = gan_image_alloc_gl_s ( 150, 100 );

/* convert an existing image to new format, type & dimensions */
gan_image_set_format_type_dims ( pImage, GAN_VECTOR_FIELD_2D, GAN_FLOAT,
                                 200, 50 );
```

sets the image `pImage` to be a 2D vector field (2D image of 2-vectors), float type and dimensions 200 by 50. There are also routines for setting the format, type or dimensions, leaving the other attributes fixed. So for instance

```
gan_image_set_format_type ( pImage, GAN_VECTOR_FIELD_2D, GAN_FLOAT );
```

sets just the format and type of the image, leaving the dimensions unchanged, while

```
gan_image_set_type ( pImage, GAN_FLOAT );
```

sets only the image type, leaving the format and dimensions unchanged. Finally

```
gan_image_set_dims ( pImage, 200, 50 );
```

changes only the image dimensions.

Error detection: All the above routines return `NULL` and invoke the Gandalf error handler if they fail. The most likely failure mode is failing to reallocate the image data, i.e. an internal `realloc()` call failing.

4.3.2 Accessing single pixels

To return the value of a single image pixel use

```
Gan_Image *pImage;
short sPixel;

/* ... create and fill grey-level short integer image pImage ... */
sPixel = gan_image_get_pix_gl_s ( pImage, 33, 40 );
```

This returns the pixel value at row position 33 and column position 40 (starting from zero). The RGB colour version would be

```
Gan_Image *pImage;
```

```
Gan_RGBPixel_uc rgbucPixel;

/* ... create and fill RGB colour unsigned character image pImage ... */
rgbucPixel = gan_image_get_pix_rgb_uc ( pImage, 33, 40 );
```

An alternative is to return a pointer to a pixel. This operation is available for every type and format of image except binary images, which are stored packed with 32 or 64 pixels to a memory word. To return a pointer to the above pixels you would use

```
Gan_Image *pImage;
short *psPixel;

/* ... create and fill grey-level short integer image pImage ... */
psPixel = gan_image_get_pixptr_gl_s ( pImage, 33, 40 );
```

for a grey-level image, or

```
Gan_Image *pImage;
Gan_RGBPixel_uc *prgbucPixel;

/* ... create and fill RGB colour unsigned character image pImage ... */
prgbucPixel = gan_image_get_pixptr_rgb_uc ( pImage, 33, 40 );
```

for an RGB colour image. This type of image access is useful when you want to read or set a lot of consecutive pixels on a row of an image, since you can use the returned pointer as a starting point. For instance the code fragment

```
Gan_Image *pImage;
Gan_RGBPixel_uc *prgbucPixel, rgbucZeroPixel = {0,0,0};
int iCount;

/* ... create and fill RGB colour unsigned character image pImage ... */
prgbucPixel = gan_image_get_pixptr_rgb_uc ( pImage, 33, 40 );
for ( iCount = 4; iCount >= 0; iCount-- )
   *prgbucPixel++ = rgbucZeroPixel;
```

sets the five RGB pixels at positions (33,40-44) to zero.

To set a pixel in a grey-level short integer image to a particular value, use the routine

```
Gan_Image *pImage;

/* ... create grey-level short integer image pImage ... */
gan_image_set_pix_gl_s ( pImage, 33, 40, 123 );
```

This sets the pixel value at position 33, 40 to value 123. For an RGB unsigned character image you would use the code

```
Gan_Image *pImage;
Gan_RGBPixel_uc rgbucPixel = {12, 13, 14};

/* ... create RGB colour unsigned character image pImage ... */
gan_image_set_pix_rgb_uc ( pImage, 33, 40, &rgbucPixel );
```

This builds a pixel with RGB values 12 (red), 13 (green), 14 (blue) and sets the pixel at position 33, 40 to that RGB value.

With `NDEBUG` set these routines evaluate to macros which implement direct memory access, so there is no efficiency advantage to be gained from using other methods of accessing individual image pixels.

The higher level routines for accessing single pixels use the `Gan_Pixel` structure, which can be used to store data for a single pixel of any format and type. The `Gan_Pixel` structure stores the format and type of the pixel internally, and is defined in `<gandalf/image/pixel.h>`:

```
/* structure definition for image pixel of any format or type */
typedef struct Gan_Pixel
{
   /// format of image: grey-level, RGB colour etc.
   Gan_ImageFormat format;

   /// type of pixel values: unsigned char, float etc.
   Gan_Type type;

   /// nested union defining pixel types
   union
   {
      /// grey level
      union
      {
         unsigned char  uc;
         short          s;
         unsigned short us;
         int            i;
         unsigned int   ui;
         double         d;
         float          f;
         Gan_Bool       b;
         void          *p;

#ifdef GAN_UINT8
         gan_ui8 ui8;
#endif
#ifdef GAN_UINT16
         gan_ui16 ui16;
#endif
#ifdef GAN_UINT32
         gan_ui32 ui32;
#endif
      } gl;

      /// grey level with alpha channel
      union
      {
         Gan_GLAPixel_uc uc;
         Gan_GLAPixel_s  s;
         Gan_GLAPixel_us us;
         Gan_GLAPixel_i  i;
         Gan_GLAPixel_ui ui;
         Gan_GLAPixel_d  d;
```

```
         Gan_GLAPixel_f  f;

#ifdef GAN_UINT8
         Gan_GLAPixel_ui8 ui8;
#endif
#ifdef GAN_UINT16
         Gan_GLAPixel_ui16 ui16;
#endif
#ifdef GAN_UINT32
         Gan_GLAPixel_ui32 ui32;
#endif
      } gla;

      /// RGB colour
      union
      {
         Gan_RGBPixel_uc uc;
         Gan_RGBPixel_s  s;
         Gan_RGBPixel_us us;
         Gan_RGBPixel_i  i;
         Gan_RGBPixel_ui ui;
         Gan_RGBPixel_d  d;
         Gan_RGBPixel_f  f;

#ifdef GAN_UINT8
         Gan_RGBPixel_ui8 ui8;
#endif
#ifdef GAN_UINT16
         Gan_RGBPixel_ui16 ui16;
#endif
#ifdef GAN_UINT32
         Gan_RGBPixel_ui32 ui32;
#endif
      } rgb;

      /// RGB colour with alpha channel
      union
      {
         Gan_RGBAPixel_uc uc;
         Gan_RGBAPixel_s  s;
         Gan_RGBAPixel_us us;
         Gan_RGBAPixel_i  i;
         Gan_RGBAPixel_ui ui;
         Gan_RGBAPixel_d  d;
         Gan_RGBAPixel_f  f;

#ifdef GAN_UINT8
         Gan_RGBAPixel_ui8 ui8;
#endif
#ifdef GAN_UINT16
         Gan_RGBAPixel_ui16 ui16;
#endif
#ifdef GAN_UINT32
         Gan_RGBAPixel_ui32 ui32;
```

```
#endif
      } rgba;

      /// 2D vector field
      union
      {
         Gan_Vector2_f f;
         Gan_Vector2   d;
         Gan_Vector2_s s;
         Gan_Vector2_i i;
      } vfield2D;

      /// 3D vector field
      union
      {
         Gan_Vector3_f f;
         Gan_Vector3   d;
         Gan_Vector3_s s;
         Gan_Vector3_i i;
      } vfield3D;
   } data;
} Gan_Pixel;
```

The `Gan_Pixel` structure should be accessed directly. There are no Gandalf access routines for it. The doubly nested union contains a structure for each Gandalf image format and type. These structures are also defined in the `pixel.h` header file. We have seen the definition of the `Gan_RGBPixel_uc` structure above, and the other structures are defined similarly. For instance the pixel to represent a single-precision floating point RGB pixel with alpha channel is

```
/**
 * \brief Structure defining RGB single precision floating point pixel with alpha channel.
 */
typedef struct Gan_RGBAPixel_f
{
   float R, /**< Red channel */
         G, /**< Green channel */
         B, /**< Blue channel */
         A; /**< Alpha channel */
} Gan_RGBAPixel_f;
```

Note that the vector field pixels use Gandalf fixed size vectors to hold the image data.

To set/get a pixel in an image using the higher level routines `gan_image_set_pix()` and `gan_image_get_pix()`, look at the following code fragment.

```
Gan_Image *pImage;
int iRow, iCol;
Gan_Pixel Pixel;

/* create grey-level signed short image */
pImage = gan_image_alloc_gl_s ( 200, 100 );

/* set up pixel format and type */
Pixel.format = GAN_GREY_LEVEL_IMAGE;
Pixel.type = GAN_SHORT;
```

```
/* fill image with ramp data */
for ( iRow = (int)pImage->height-1; iRow >= 0; iRow-- )
   for ( iCol = (int)pImage->width-1; iCol >= 0; iCol-- )
   {
      /* set pixel data */
      Pixel.data.gl.s = iRow+iCol;

      /* fill pixel in image. The format and type of the pixel should
         be the same as that of the image */
      gan_image_set_pix ( pImage, iRow, iCol, &Pixel );
   }

/* print pixel value, should be 27+35 = 62 */
Pixel = gan_image_get_pix ( pImage, 27, 35 );
printf ( "pixel value = %d\n", Pixel.data.gl.s );
```

Here we created an image, filled it with "ramp" data that linearly increases the grey-level value with the row and column coordinates of the image, and extract a single pixel. For an RGB image we could add the following code:

```
/* convert the image to RGB format and unsigned character type */
gan_image_set_rgb_uc ( pImage, 100, 50 );

/* set up pixel format and type */
Pixel.format = GAN_RGB_COLOUR_IMAGE;
Pixel.type = GAN_UCHAR;

/* fill image with ramp data */
for ( iRow = (int)pImage->height-1; iRow >= 0; iRow-- )
   for ( iCol = (int)pImage->width-1; iCol >= 0; iCol-- )
   {
      /* set pixel data */
      Pixel.data.rgb.uc.R = iRow+iCol;
      Pixel.data.rgb.uc.G = iRow;
      Pixel.data.rgb.uc.B = iCol;

      /* fill pixel in image. The format and type of the pixel should
         be the same as that of the image */
      gan_image_set_pix ( pImage, iRow, iCol, &Pixel );
   }

/* print pixel value, should be R=37+11=48, G=37, B=11 */
Pixel = gan_image_get_pix ( pImage, 37, 11 );
printf ( "pixel value R=%d G=%d B=%d\n",
         Pixel.data.rgb.uc.R, Pixel.data.rgb.uc.G, Pixel.data.rgb.uc.B );
```

If you have a `Gan_Pixel` structure in a different format/type to the image, use `gan_image_convert_pixel_[qsi]()` to convert it to the format & type of the image before calling `gan_image_set_pix()`. See Section 4.3.4 for details.

Error detection: The `..._get_pix...()` routines cannot return an error condition. Instead they invoke `gan_assert()` (see Section 2.1.3) to check for errors, which aborts the program if an error is found. The `..._set_pix...()` routines return a boolean value, returning `GAN_TRUE` on success, invoking the Gandalf error handler and returning `GAN_FALSE` on failure. The most likely failure modes are accessing a pixel outside the image (both `..._get_pix...()` and `..._set_pix...()`) and mismatch between image and pixel format/type (`..._set_pix...()` only). These errors are program bugs rather than data-dependent errors, so using `gan_assert()`

to handle errors is fairly safe.

4.3.3 Filling an image with a constant value

To fill a grey-level image with a constant value use this routine:

```
Gan_Image *pImage;

/* create grey-level signed short image */
pImage = gan_image_alloc_gl_s ( 200, 100 );

/* fill with constant */
gan_image_fill_const_gl_s ( pImage, 23 );
```

which fills each pixel with the value 23. For other formats of image you will need to build a structure of the relevant type, for instance

```
Gan_Image *pImage;
Gan_RGBPixel_uc rgbucPixel;

/* create RGB unsigned character image */
pImage = gan_image_alloc_rgb_uc ( 200, 100 );

/* set up pixel */
rgbucPixel.R = 34;
rgbucPixel.G =  2;
rgbucPixel.B = 65;

/* fill with constant RGB value */
gan_image_fill_const_rgb_uc ( pImage, &rgbucPixel );
```

Higher level routines are available using the `Gan_Pixel` structure:

```
Gan_Image *pImage;
Gan_Pixel Pixel;

/* create RGBA single precision floating point image */
pImage = gan_image_alloc_rgba_f ( 200, 100 );

/* set up pixel */
Pixel.format = GAN_RGB_COLOUR_ALPHA_IMAGE;
Pixel.type = GAN_FLOAT;
Pixel.data.rgba.f.R = 0.1F;
Pixel.data.rgba.f.G = 0.2F;
Pixel.data.rgba.f.B = 0.3F;
Pixel.data.rgba.f.A = 0.4F;

/* fill with constant RGBA value. The format & type of the pixel and
   image should match */
gan_image_fill_const ( pImage, &Pixel );
```

If the `Gan_Pixel` structure has a different format/type to the image, use `gan_image_convert_pixel_[qsi]()` to convert it to the format & type of the image before calling `gan_image_fill_const()`. See Section 4.3.4 for details.

There is a special function `gan_image_fill_zero()` to fill an image with zero, whatever format and type it has:

```
Gan_Image *pImage;

/* create RGBA single precision floating point image */
pImage = gan_image_alloc_rgba_f ( 200, 100 );

/* set all image pixels to zero */
gan_image_fill_zero ( pImage );
```

For boolean images (Section 4.4), "zero" is interpreted as false (`GAN_FALSE`), and for pointer images (Section 4.5) "zero" means `NULL`. To fill a single pixel with zero, use

```
/* set a single pixel at position row=10, column=21 to zero */
gan_image_set_pix_zero ( pImage, 10, 21 );
```

There are also routines to fill a rectangular sub-region of an image, either with a constant value or zero:

```
Gan_Image *pImage;
Gan_Pixel Pixel;

/* create grey-level signed short image */
pImage = gan_image_alloc_gl_s ( 200, 100 );

/* set pixels in 30x40 (heightxwidth) pixel region starting at position
   100,30 (row,column) to constant value 125 */
Pixel.format = GAN_GREY_LEVEL_IMAGE;
Pixel.type = GAN_SHORT;
Pixel.data.gl.s = 125;
gan_image_fill_const_window ( pImage, 100, 30, 30, 40, &Pixel );

/* reset image to RGB unsigned character */
gan_image_set_rgb_uc ( pImage, 100, 50 );

/* set pixels in 20x15 (heightxwidth) pixel region starting at position
   10,35 (row,column) to zero */
gan_image_fill_zero_window ( pImage, 10, 35, 20, 15 );
```

Error detection: The image filling routines return a boolean value, so a return value of `GAN_FALSE` indicates failure, with the Gandalf error handling module being invoked.

4.3.4 Converting a pixel to a given format/type

Gandalf routines taking `Gan_Pixel` structure pointers as arguments, such as `gan_image_fill_const()`, require that the format and type of the pixel and image arguments match. This can be done by using the routines in this section. To convert a pixel to a specific format and type use the routine

```
Gan_Pixel Pixel1, Pixel2; /* declare pixels 1 & 2 */

/* let's initialise pixel 1 to a grey-level unsigned character value */
Pixel1.format = GAN_GREY_LEVEL_IMAGE;
Pixel1.type = GAN_UCHAR;
Pixel1.data.gl.uc = 255;

/* now convert pixel to RGB format and floating point type */
```

```
gan_image_convert_pixel_q ( &Pixel1, GAN_RGB_COLOUR_IMAGE, GAN_FLOAT,
                            &Pixel2 );

/* print new pixel value, which should be R=G=B=1 */
printf ( "pixel RGB value %f %f %f\n", Pixel2.data.rgb.f.R,
         Pixel2.data.rgb.f.G, Pixel2.data.rgb.f.B );
```

Another version of this function returns the result as a new pixel:

```
/* convert pixel to RGB format and floating point type */
Pixel2 = gan_image_convert_pixel_s ( &Pixel1, GAN_RGB_COLOUR_IMAGE, GAN_FLOAT );
```

There is also a routine that converts the format and type in-place in the input pixel:

```
/* convert pixel to RGB format and floating point type in-place */
gan_image_convert_pixel_i ( &Pixel1, GAN_RGB_COLOUR_IMAGE, GAN_FLOAT );
```

4.4 Binary images

```
#include <gandalf/image/image_bit.h>
```

Gandalf binary images support compact storage of an array of boolean values. Binary images have format `GAN_GREY_LEVEL_IMAGE` and type `GAN_BOOL`. The complete set of functions described above is available for binary images, as well as other special functions. Here is an illustration of using the standard routines.

```
Gan_Image *pImage;
Gan_Pixel Pixel;

/* allocate 300x200 binary image, and initialise all values to
   zero (false) */
pImage = gan_image_alloc_b ( 300, 200 );
gan_image_fill_zero(pImage);

/* fill rectangular region of image with ones (true) */
Pixel.format = GAN_GREY_LEVEL_IMAGE;
Pixel.type = GAN_BOOL;
Pixel.data.gl.b = GAN_TRUE;
gan_image_fill_const_window ( pImage, 120, 100, 40, 30, &Pixel );

/* reset size of image and fill with zero again */
gan_image_set_b ( pImage, 400, 600 );
gan_image_fill_zero(pImage);

/* set some other pixels to one (true) */
gan_image_set_pix_b ( pImage, 250,   4, GAN_TRUE );
gan_image_set_pix_b ( pImage,  50, 140, GAN_TRUE );
gan_image_set_pix_b ( pImage, 150,  40, GAN_TRUE );

/* free image */
gan_image_free ( pImage );
```

Several other routines are provided for binary images. Firstly there is a routine to return the "active" region of an image, defined as the bounding box around the pixels set to one:

```
Gan_ImageWindow SubWindow;

/* ... set pImage as a binary image with ones and zeros ... */

/* determine image window surrounding "active" pixels, i.e. those
   set to one */
gan_image_get_active_subwindow_b ( pImage, GAN_WORD_ALIGNMENT,
                                   &SubWindow );
```

The `Gan_ImageWindow` result structure was described in Section 4.1. The second argument defines how precisely to determine the horizontal limits of the bounding box. The coarsest method is to find the limits to the nearest word, as in the above example. More precise but slower alignment is possible using either `GAN_BYTE_ALIGNMENT` or `GAN_BIT_ALIGNMENT`.

To compute the number of active bits in a binary image use

```
int iCount;

iCount = gan_image_get_pixel_count_b ( pImage, GAN_TRUE, NULL );
```

The second argument is `GAN_TRUE` to count the ones or `GAN_FALSE` to count the zeroes. The last argument is an optional pointer to a sub-window of the image in which to apply the count.

There are functions to return a boolean value indicating whether a local group of pixels are all set to one. These routines are

```
/* check whether the group of four pixels at positions (100,100),
   (100,101), (101,100), (101,101) are all set to one */
if ( gan_image_pix_get_pix_4group ( pImage, 100, 100 ) )
   printf ( "group of four found\n" );

/* check whether the group of four pixels at positions (99,100),
   (100,99), (100,100), (100,101) and (101,100) are all set to one */
if ( gan_image_pix_get_pix_5group ( pImage, 100, 100 ) )
   printf ( "group of five found\n" );

/* check whether the group of three pixels at positions (100,99),
   (100,100), (100,101) are all set to one */
if ( gan_image_pix_get_pix_3group_horiz ( pImage, 100, 100 ) )
   printf ( "group of three horizontally found\n" );

/* check whether the group of three pixels at positions (99,100),
   (100,100), (101,100) are all set to one */
if ( gan_image_pix_get_pix_3group_vert ( pImage, 100, 100 ) )
   printf ( "group of three vertically found\n" );
```

There is a set of functions to apply a boolean operation to every pixel in one image or a pair of images. Firstly there is are routines to invert a boolean image:

```
Gan_Image *pImage1, *pImage2, *pImage3;

/* ... create and fill image 1 as a boolean image, create image 2 ... */

gan_image_bit_invert_q ( pImage1, pImage2 ); /* invert image 1 into image 2, OR */
pImage3 = gan_image_bit_invert_s ( pImage1 ); /* invert image 1 as a new image, OR */
gan_image_bit_invert_i ( pImage1 ); /* replace image 1 with its inverse */
```

Then there are routines to apply the operations AND, OR, exclusive-OR (EOR) and not-AND (NAND) to a pair of binary images, which must have the same dimensions. Illustrating the AND operation first, we have the options

```
Gan_Image *pImage1, *pImage2, *pImage3, *pImage4;

/* ... create and fill images 1 & 2 as boolean images, create image 3 ... */

gan_image_bit_and_q ( pImage1, pImage2, pImage3 ); /* AND(1,2) into image 3, OR */
pImage4 = gan_image_bit_and_s ( pImage1, pImage2 ); /* AND(1,2) as a new image, OR */
gan_image_bit_and_i ( pImage1, pImage2 ); /* replace image 1 with AND(1,2) */
```

The other operations follow similar lines. Firstly the OR operation:

```
Gan_Image *pImage1, *pImage2, *pImage3, *pImage4;

/* ... create and fill images 1 & 2 as boolean images, create image 3 ... */

gan_image_bit_or_q ( pImage1, pImage2, pImage3 ); /* OR(1,2) into image 3, OR */
pImage4 = gan_image_bit_or_s ( pImage1, pImage2 ); /* OR(1,2) as a new image, OR */
gan_image_bit_or_i ( pImage1, pImage2 ); /* replace image 1 with OR(1,2) */
```

Now the exclusive-OR operation:

```
Gan_Image *pImage1, *pImage2, *pImage3, *pImage4;

/* ... create and fill images 1 & 2 as boolean images, create image 3 ... */

gan_image_bit_eor_q ( pImage1, pImage2, pImage3 ); /* EOR(1,2) into image 3, OR */
pImage4 = gan_image_bit_eor_s ( pImage1, pImage2 ); /* EOR(1,2) as a new image, OR */
gan_image_bit_eor_i ( pImage1, pImage2 ); /* replace image 1 with EOR(1,2) */
```

Finally the not-AND operation:

```
Gan_Image *pImage1, *pImage2, *pImage3, *pImage4;

/* ... create and fill images 1 & 2 as boolean images, create image 3 ... */

gan_image_bit_nand_q ( pImage1, pImage2, pImage3 ); /* NAND(1,2) into image 3, OR */
pImage4 = gan_image_bit_nand_s ( pImage1, pImage2 ); /* NAND(1,2) as a new image, OR */
gan_image_bit_nand_i ( pImage1, pImage2 ); /* replace image 1 with NAND(1,2) */
```

A few more miscellaneous routines are available for binary images. To fill part of a row with either zero or one use the routine

```
/* fill section of row 13 of image with ones (true) starting at
   horizontal position 20 and filling 30 pixels */
gan_image_bit_fill_row ( pImage, 13, 20, 30, GAN_TRUE );
```

To invert part of a row of a binary image use

```
/* invert section of row 13 starting at horizontal position 20 and
   filling 30 pixels */
gan_image_bit_invert_row ( pImage, 13, 20, 30 );
```

Finally if you want to clear a binary image to zero except inside a specified rectangular region of the image, try this:

```
/* clear binary image to zero except in 50(h)x30(w) pixel area starting
   at position 20,60 (y,x) */
gan_image_mask_window_b ( pImage, 20, 60, 50, 30 );
```

Error detection: The standard binary image routines detect errors as described in Section 4.2. The boolean operation routines (`gan_image_bit_invert_q()` etc.) return a pointer to the result image, and return `NULL` on an error. All the other binary image routines, with one exception, return a boolean value; thus `GAN_FALSE` is returned on error. The exception is `gan_image_get_pixel_count_b()`, which returns an integer value, which in case of error is returned as -1. The Gandalf error handler is invoked in all these cases.

4.5 Pointer images

```
#include <gandalf/image/image_pointer.h>
```

Gandalf pointer images allow storage and manipulation of a 2D array of generic pointers, stored as `void *` values. Pointer images have format `GAN_GREY_LEVEL_IMAGE` and type `GAN_POINTER`. All the standard functions given above. Note that when a pointer image is freed, the pointer pixels are left "hanging", so they should if necessary be freed first before freeing the pointer image. This code fragment illustrates the use of pointer image functions.

```
Gan_Image *pImage;
Gan_Vector4 *apv4Vector[5], *pv4Vector;
int iCount, iRow, iCol;

/* allocate 300x200 pointer image, and initialise all pointer "pixels"
   to NULL */
pImage = gan_image_alloc_p ( 300, 200 );
gan_image_fill_zero(pImage);

/* allocate some pointers to 4-vectors */
for ( iCount = 5-1; iCount >= 0; iCount-- )
   apv4Vector[iCount] = gan_malloc_object(Gan_Vector4);

/* set some pointer "pixels" */
gan_image_set_pix_p ( pImage, 271,  39, apv4Vector[0] );
gan_image_set_pix_p ( pImage,  30, 120, apv4Vector[1] );
gan_image_set_pix_p ( pImage,  78,  49, apv4Vector[2] );
gan_image_set_pix_p ( pImage, 147, 120, apv4Vector[3] );
gan_image_set_pix_p ( pImage, 232, 130, apv4Vector[4] );

/* now free allocated vectors by searching for non-NULL values in the
   image */
for ( iRow = (int)pImage->height-1; iRow >= 0; iRow-- )
   for ( iCol = (int)pImage->width-1; iCol >= 0; iCol-- )
      if ( (pv4Vector = gan_image_get_pix_p ( pImage, iRow, iCol )) != NULL )
         free(pv4Vector);

/* free image */
gan_image_free ( pImage );
```

4.6 Copying/converting the whole or part of an image

```
#include <gandalf/image/image_defs.h>
```

To copy a whole image of any format or type, use one of the following routines:

```
Gan_Image *pImage1, *pImage2, *pImage3; /* declare images 1, 2 & 3 */

/* ... create images 1 & 2, fill image 1 ... */
gan_image_copy_q ( pImage1, pImage2 ); /* copy image 1 to image 2, OR */
pImage3 = gan_image_copy_s ( pImage1 ); /* copy image 1 as new image */
```

Image 2 here may have been created with any format, type or dimensions. Gandalf will reset the attributes of image 2 to those of image 1 before copying the image data. These routines make copies of the image data, so image 1 may be destroyed after it is copied.

To copy parts of an image, you will need to include the header file

```
#include <gandalf/image/image_extract.h>
```

The routines to extract sub-parts of an image are `gan_image_extract_q()` and `gan_image_extract_s()`. They have the following extra features over a simple routine to copy image sub-regions:

1. You can convert the image sub-region to any desired format and type, avoiding the need to perform the two steps of extracting and converting sequentially, and thus saving computation and memory.
2. There is an option to make the resulting sub-image point into the source image, rather than copy the pixel data from it. This saves computation time, and the sub-image produced can be manipulated in the same way as other Gandalf images. Obviously use of this feature precludes use of feature 1.

Here are a couple of examples using the sub-region extraction routines. Firstly a code fragment showing showing the simplest form, where the region is copied from the source image and the format/type remain the same.

```
Gan_Image *pImage1, *pImage2; /* declare images 1 & 2 */
Gan_RGBPixel_uc rgbucPixel;

/* create RGB unsigned character image 1 and fill with constant */
pImage1 = gan_image_alloc_rgb_uc ( 200, 100 );
rgbucPixel.R = 128; rgbucPixel.R = 80; rgbucPixel.R = 200;
gan_image_fill_const_rgb_uc ( pImage1, &rgbucPixel );

/* create image 2 in an arbitrary way */
pImage2 = gan_image_alloc_gl_uc(0,0);

/* extract sub-region in image 1 into image 2, with height 60, width 50,
   starting as position 30,40 (y,x), leaving the format/type the same.
   The pixel data is copied */
gan_image_extract_q ( pImage1, 30, 40, 60, 50,
                      pImage1->format, pImage1->type, GAN_TRUE,
                      pImage2 );
```

Now an example continuing from the above, and showing how to make the result image point into the source image.

```
/* extract sub-region in image 1 into image 2, with height 60, width 50,
```

```
      starting as position 30,40 (y,x), leaving the format/type the same.
      Here the pixel data is not copied; instead the result image points
      into the source image */
   gan_image_extract_q ( pImage1, 30, 40, 60, 50,
                         pImage1->format, pImage1->type, GAN_FALSE,
                         pImage2 );
```

Finally an example showing how to convert the sub-region to a different format and type.

```
   {
      Gan_Image *pImage3; /* declare image 3 */

      /* extract sub-region in image 1 into image 2, with height 60, width 50,
         starting as position 30,40 (y,x), converting the format to grey-level
         and the type to unsigned short. Here the format and type are modified
*        as the pixels are extracted from the source image */
      pImage3 = gan_image_extract_s ( pImage1, 30, 40, 60, 50,
                                      GAN_GREY_LEVEL_IMAGE, GAN_USHORT,
                                      GAN_TRUE );
   }
```

There are also routines to convert the whole of an image to a different format or type (or both). These are simpler macro versions of `gan_image_extract_[qs]()`, and can be illustrated as follows:

```
   Gan_Image *pImage1, *pImage2, *pImage3; /* declare images 1, 2 & 3 */

   /* ... create RGB unsigned character image 1 and fill with constant,
          and create image 2 in an arbitrary way ... */

   /* convert image 1 to grey-level format and unsigned short type */
   gan_image_convert_q ( pImage1, GAN_GREY_LEVEL_IMAGE, GAN_USHORT,
                         pImage2 ); /* convert image 1 to image 2, OR */
   pImage3 = gan_image_convert_s ( pImage1, GAN_GREY_LEVEL_IMAGE, GAN_USHORT );
```

Error detection: These routines return the result image pointer, and return `NULL` on error.

4.6.1 Accessing channels of an image

```
   #include <gandalf/image/image_channel.h>
```

Gandalf stores images with the channels combined for each pixel. If you wish to extract a channel of an image as a separate image, Gandalf provides the following function:

```
   Gan_Image *pRGBImage; /* declare RGB image */
   Gan_Image *pRedChannel; /* declare image storing red channel */

   /* ... create and fill RGB image, create red channel image ... */

   /* extract red channel from image */
   gan_image_extract_channel_q ( pRGBImage, GAN_RED_CHANNEL,
                                 0, 0, pRGBImage->height, pRGBImage->width,
                                 pRedChannel );
```

The second argument specifies which channel is to be extracted. The different options are described by the following enumerated type.

```
/**
 * \brief Image channel types for extracting individual channels.
 */
typedef enum
{
   /// for grey-level/alpha images
   GAN_INTENSITY_CHANNEL,

   ///for RGB and RGB/alpha images
   GAN_RED_CHANNEL, GAN_GREEN_CHANNEL, GAN_BLUE_CHANNEL,

   /// for grey-level/alpha and RGB/alpha images
   GAN_ALPHA_CHANNEL,

   /// for 2D and 3D vector field images
   GAN_X_CHANNEL,

   /// likewise
   GAN_Y_CHANNEL,

   /// for 3D vector field images
   GAN_Z_CHANNEL,

   /// all channels
   GAN_ALL_CHANNELS
} Gan_ImageChannelType;
```

The offset (3,4) and dimension (5,6) arguments allow a sub-region to be extracted rather than the whole image, and work in the same way as with `gan_image_extract_q()`. There is also a version which extracts the channel as a new image:

```
pRedChannel = gan_image_extract_channel_s ( pRGBImage, GAN_RED_CHANNEL,
                                            0, 0,
                                            pRGBImage->height, pRGBImage->width );
```

There are also functions for filling a channel of an RGB image with a constant value. For instance

```
Gan_Pixel Pixel;

/* ... fill pRGBImage as an RGB unsigned character image ...*/

/* fill green channel of pRGBImage with constant value */
Pixel.format = GAN_GREY_LEVEL_IMAGE;
Pixel.type = GAN_UCHAR;
Pixel.data.gl.uc = 128;
gan_image_fill_channel_const ( pRGBImage, GAN_GREEN_CHANNEL, &Pixel );
```

sets the all the green pixel components to the value 128. Note that the format of the pixel is set to grey-level, so defining a single channel pixel. To set the channel to zero there is the macro

```
gan_image_fill_channel_zero ( pRGBImage, GAN_GREEN_CHANNEL );
```

instead.

Error detection: The `gan_image_extract_channel_[qs]()` return a pointer to the result image, returning `NULL` and invoking the Gandalf error handler on error. `gan_image_fill_channel_const()` and `gan_image_fill_channel_zero()` return a boolean value, so `GAN_FALSE` is returned on error.

4.7 Displaying images

```
#include <gandalf/image/image_display.h>
```

Gandalf uses OpenGL to display images. It assumes that as well as the standard OpenGL libraries, the GL user toolkit (GLUT) is also installed. Because GLUT is event-driven, program control needs to be passed to the GLUT event handler by calling `glutMainLoop()` after creating the initial windows you want. Remember that creating a window using the `gan_display_new_window()` function (see below) will not make it appear immediately. The window creation event needs to be processed by GLUT. This needs to be borne in mind when reading the description of the functions below. The simplest example using the functions is in `gandalf/image/bitmap_test.c`.

Once an OpenGL window has been set up and a Gandalf image `pImage` created, calls to

```
glRasterPos2i ( 0, 0 );
gan_image_display ( pImage );
```

will display the image using the OpenGL function `glDrawPixels`. This involves a fair amount of OpenGL calls to set the display windows up. To simply create an OpenGL window and display a Gandalf image in it, Gandalf provides functions to make this easy for you. You can use the code

```
Gan_Image *pImage;
int iWindowID;

/* create OpenGL window to display image/graphics with coordinates
   in the range (0-200) vertically and (0-300) horizontally, using a
   zoom factor of 2 so that the size on the screen will be 400x600.
   The window is placed at offset 100,100 from the corner of the screen.
 */
gan_display_new_window ( 200, 300, 2.0, "Graphics", 100, 100,
                         &iWindowID );

/* ... create and fill image pImage ... */

/* display image with top-left pixel at position (0,0) */
gan_image_display ( pImage );
```

The image is drawn so that if its dimensions match those passed as the first two arguments to `gan_display_new_window()`, the displayed image will completely fill the window. If you want the image displayed at a position offset from the top-left corner of the window you will need an appropriate call to `glRasterPos2i()`, such as

```
/* set position in OpenGL window as top-left position in image drawn
 * subsequently by gan_image_display() */
glRasterPos2i ( 30, 40 );
```

The window is available for the standard OpenGL graphics functions. The name of the window ("Graphics" in the above example) is shown in the bar at the top of the graphics window. If the graphics window changes, the window identifier `iWindowID` can be used to switch back to the created window using

```
glutSetWindow ( iWindowID );
```

To create several identically sized graphics windows, use this routine:

```
int iWindowID, *aiWindowID;

/* create an OpenGL window containing 2 rows and 3 columns of
   sub-windows, each containing a 300x200 graphics window, each of
   which contains a (0-900) by (0-600) coordinate frame shrunk to
   300x200 using a zoom factor of 1/3 */
gan_display_new_window_array ( 2, 3, 900, 600, 1.0/3.0, "Graphics",
                               100, 100, &iWindowID, &aiWindowID );
```

The sub-windows are created using the function `glutCreateSubWindow()`. In this case there is both a window identifier for the main display window and an array of window identifiers for the sub-windows, stored in the array in raster-scan order.

Gandalf also provides a routine that creates a window and displays the image all in one, determining the graphics window size from a zoom factor passed in:

```
/* create display window and display image zoomed to double its size */
gan_image_display_new_window ( pImage, 2.0, "Graphics", 100, 100, &iWindowID );
```

This is useful for debugging purposes as the easiest way to display an image.

The `gan_display_new_window()` function stores the windows created in a list, so that the images displayed in the windows can be automatically refreshed. When you have finished with the graphics windows created by `gan_display_new_window()`, remove them using the function

```
gan_image_display_free_windows();
```

Note that this free function only applies to windows created by `gan_display_new_window()`. Graphics windows created using the other functions in this section are refreshed using standard OpenGL routines (`glutDisplayFunc()` etc.).

Error detection: All the routines except `gan_image_display_free_windows()` return a boolean result, which is `GAN_FALSE` if an error occurs, invoking the Gandalf error handler.

4.8 Image pyramids

```
#include <gandalf/image/image_pyramid.h>
```

A quite common construction in image processing is a "pyramid" of images, a multi-resolution representation of an image. We think in fact of an *inverted* pyramid, with the top level of the pyramid representing the image at the original resolution, and lower levels representing the image at lower resolutions (the inversion of the pyramid is to avoid changing the sense of "high" and "low" between describing the "resolution" and the "level"). The pyramid is constructed as an array of structures, one for each resolution level. The structure is defined as

```
/* structure to hold image and mask at a single pyramid level */
typedef struct Gan_ImagePyramid
{
   Gan_Image *img; /* image represented at a single resolution level */
   Gan_Image *mask; /* mask at this level defining which pixels are set */
} Gan_ImagePyramid;
```

Intrinsic to the pyramid is the notion of a "mask" of pixels, a binary image defining which pixels are available in the image. The convention is that available pixels are marked in the mask as ones. Gandalf currently supports pyramids produced in the simplest way, by averaging four adjacent pixels to convert the image from a higher resolution image to a lower resolution image. An example code fragment to create an image pyramid is

```
Gan_Image *pImage, *pMask; /* declare image and mask */
Gan_ImagePyramid *aPyramid;

/* ... create and fill pImage and pMask ... */

/* build image with four resolution levels */
gan_image_pyramid_build ( pImage, pMask, 4, &aPyramid );
```

The image and mask pointer at the original (highest) resolution levels can be accessed as `aPyramid[0].img` and `aPyramid[0].mask` respectively. There are four levels here so the lowest resolution image and mask are `aPyramid[3].img` and `aPyramid[3].mask`. The mask can be passed into `gan_image_pyramid_build()` as `NULL`, indicating that all the pixels in the image are available. In this case the masks at all resolution levels will be set to `NULL`.

To free the pyramid, use

```
gan_image_pyramid_free ( aPyramid, 4, GAN_FALSE );
```

The second argument here is the number of resolution levels of the pyramid. The last argument determines whether the image/mask at the top level of the pyramid are to freed. Here `GAN_FALSE` is passed, so the original image `pImage` and mask `pMask` will not be freed and are available for further processing.

Note that when transferring pixels to a lower resolution, a pixel is computed and a mask bit at the lower resolution only if all four corresponding pixels in the higher resolution image are set.

Error detection: `gan_image_pyramid_build()` returns a boolean value, which is `GAN_FALSE` on error, the Gandalf error handler being invoked.

4.9 Inverting an image

```
#include <gandalf/image_invert.h>
```

Image inversion in Gandalf use `gan_image_invert_[qsi]`. The simplest call is

```
Gan_Image *pImage, *pInvImage; /* declare image and inverted image */

/* ... create and fill pImage ... */

/* invert image the simple way */
pInvImage = gan_image_invert_s ( pImage );
```

or else invert using a pre-allocated result image:

```
/* ... create and fill pImage, allocate pInvImage ... */

/* invert image the simple way */
gan_image_invert_q ( pImage, pInvImage );
```

and finally the in-place version:

```
/* ... create and fill pImage ... */

/* invert image in-place */
gan_image_invert_i ( pImage );
```

4.10 Image sequence I/O

```
#include <gandalf/image/io/movie.h>
```

Gandalf has a module for reading and writing image sequences. These are accessed one image at a time. The `Gan_MovieStruct` structure defines an image sequence. A movie structure is created using the `gan_movie_new()` function. An example call is

```
Gan_MovieStruct *pMovie;

pMovie = gan_movie_new ( "/tmp", "movie.", 3, ".png", 1, 20,
                         GAN_PNG_FORMAT );
```

The arguments to the function define the image sequence attributes, in the following order:

1. The directory in which to find the images;
2. the base name of the image file names;
3. the number of digits in the number part of the image file name;
4. the suffix of each image, usually related to the image file format;
5. the number of the first image in the sequence;
6. the image file format;

The above example defines a sequence of PNG format image files

```
/tmp/movie.001.png
/tmp/movie.002.png
        .
        .
        .
/tmp/movie.020.png
```

Other parameters of a movie structure have defaults which can be set using functions before the movie images are accessed. These functions are

```
gan_movie_set_step ( pMovie, 2 );
```

to set the step in numbers between images. The default is one, and the above call would set the frame numbers to 1, 3, 5 etc.

```
gan_movie_set_crop_window ( pMovie, 5, 10, 8, 12 );
```

sets the values of any crop parameters, i.e. the widths of areas at the edge of each image which should be ignored by image processing operations. The widths are give for the left, right, top and bottom edges respectively.

The movie structure is used both for reading and writing images in a sequence. The number of digits indicates the amount of zero-padding of the file number. A value of zero indicates that no padding is done. To read a single image from the sequence, use the function `gan_movie_image_read()`, for example

```
Gan_Image *pImage;

pImage = gan_movie_image_read ( pMovie, 8, NULL );
```

The second argument indicates which image in the sequence is to be read, from 0 to 19 in this case. The value 8 indicates the file `/tmp/movie.009.png`. This reads the image file into a new image. If `pImage` is already created, you can use

```
gan_movie_image_read ( pMovie, 8, pImage );
```

To write an image in a sequence use

```
gan_movie_image_write ( pMovie, 10, pImage );
```

This will write the file `/tmp/movie.011.png`.

Sometimes it is desirable to build the full name of a movie image file, for instance when generating error messages to say that a given file cannot be read or written. To write an image file name into a string, use the function

```
char acString[300];

gan_movie_image_name ( pMovie, 0, acString, 300 );
```

This writes the name of the first image of the sequence into the provided string, up to the 300 character total size of the `acString` array. For the movie created in the above example this will fill the string `acString` with the value `"/tmp/movie.001.png"`. The second argument is the number of the image in the sequence, so passing 5 would give the string `"/tmp/movie.006.png"`.

Finally, to free a movie structure use

```
gan_movie_free ( pMovie );
```

`Error detection:` `gan_movie_new()` returns a pointer to the allocated movie structure, and `NULL` is returned in case of error. `gan_movie_image_read()` and `gan_movie_image_name()` also return `NULL` on error.
`gan_movie_image_write()` returns a boolean value, so `GAN_FALSE` is returned on error. In all cases the Gandalf error handler is invoked.

Chapter 5

The Vision Package

The **vision** package includes some higher level image manipulation routines. To be able to use any routine or structure in the vision package use the declaration

```
#include <gandalf/vision.h>
```

but including individual module header files instead will speed up program compilation.

5.1 Cameras

The camera modules are separated into single and double precision versions. The double precision camera structure is defined in

```
#include <gandalf/vision/camera.h>
```

and the single precision version in

```
#include <gandalf/vision/cameraf.h>
```

The Gandalf camera defines the transformation from camera 3D coordinates into image coordinates and back again. Ten camera models are defined, all using the assumption that the projected position in the image is independent of the depth. The camera structure (double precision floating point version) is as follows:

```
/**
  * \brief Structure containing camera parameters.
  */
typedef struct Gan_Camera
{
   /// Type of camera
   Gan_CameraType type;

   /// parameters of linear camera

   /// focal distance in x/y pixels
   double fx, fy;

   /// image centre x/y coordinates
```

```
double x0, y0;

/// third homogeneous image coordinate
double zh;

/**
 * \brief Supplementary parameters for non-linear camera models.
 *
 * The thresholds are the square \f$ R^2 \f$ of the undistorted radial
 * camera coordinate \f$ R \f$ where the first reversal of distortion occurs
 * (\a thres_R2), and the similar threshold on the distorted radial
 * distance \f$ d\:R \f$, involving both the distortion coefficient
 * \f$ d \f$  and \f$ F \f$ (thres_dR), at the same reversal point.
 * Both thresholds are set to \c DBL_MAX if there is no reversal.
 */
union
{
   struct
   {
      /// Distortion coefficients
      double K1;

      /// Thresholds on \f$ R^2 \f$ and \f$ d\:R \f$
      double thres_R2, thres_dR;

      /// Outer linear model parameters
      double outer_a, outer_b;
   } radial1;

   struct
   {
      /// Distortion coefficients
      double K1, K2;

      /// Thresholds on \f$ R^2 \f$ and \f$ d\:R \f$
      double thres_R2, thres_dR;

      /// Outer linear model parameters
      double outer_a, outer_b;
   } radial2;

   struct
   {
      /// Distortion coefficients
      double K1, K2, K3;

      /// Thresholds on \f$ R^2 \f$ and \f$ d\:R \f$
      double thres_R2, thres_dR;

      /// Outer linear model parameters
      double outer_a, outer_b;
   } radial3;

   struct { double cxx, cxy, cyx, cyy; } xydist4;
```

```
   } nonlinear;

   /// gamma value of images taken using this camera
   double gamma;

   /// point functions
   struct
   {
      /// point projection function
      Gan_Bool (*project) ( struct Gan_Camera *camera,
                            Gan_Vector3 *X, Gan_Vector3 *p,
                            Gan_Matrix22 *HX, struct Gan_Camera HC[2],
                            int *error_code );

      /// point back-projection function
      Gan_Bool (*backproject) ( struct Gan_Camera *camera,
                                Gan_Vector3 *p, Gan_Vector3 *X,
                                int *error_code );

      /// function to add distortion to a point
      Gan_Bool (*add_distortion) ( struct Gan_Camera *camera,
                                   Gan_Vector3 *pu, Gan_Vector3 *p,
                                   int *error_code );

      /// function to remove distortion from a point
      Gan_Bool (*remove_distortion) ( struct Gan_Camera *camera,
                                      Gan_Vector3 *p, Gan_Vector3 *pu,
                                      int *error_code);
   } point;

   /// line functions
   struct
   {
      /// line projection function
      Gan_Bool (*project) ( struct Gan_Camera *camera,
                            Gan_Vector3 *L, Gan_Vector3 *l );

      /// line back-projection function
      Gan_Bool (*backproject) ( struct Gan_Camera *camera,
                                Gan_Vector3 *l, Gan_Vector3 *L );
   } line;
} Gan_Camera;
```

The single precision version `Gan_Camera_f` is defined similarly. The camera models are defined in `<gandalf/vision/camera_defs.` and are

```
/**
 * \brief Camera models supported by Gandalf.
 */
typedef enum
{
   /// linear camera model
   GAN_LINEAR_CAMERA,

   /// one parameter K1 of radial distortion
```

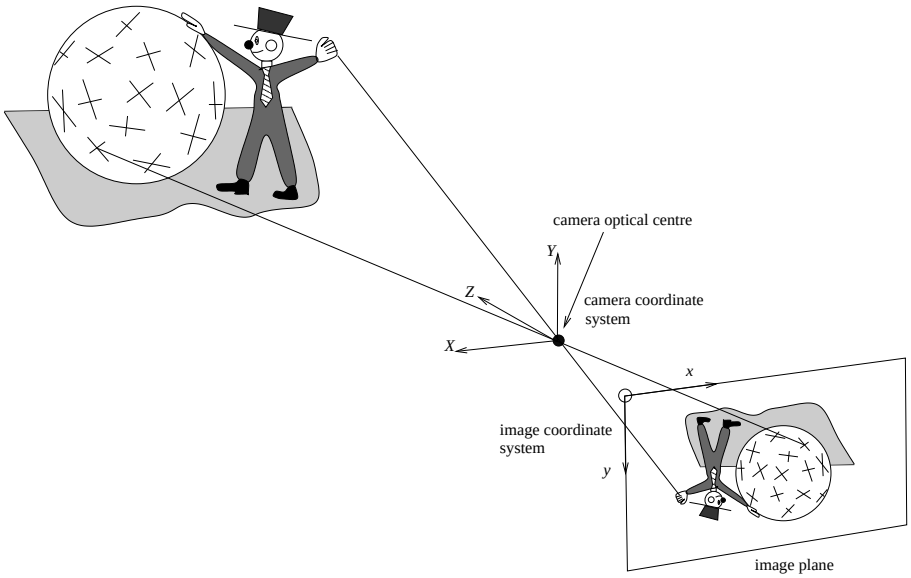

Figure 5.1: Illustration of coordinate frames in projection from camera 3D frame into the image.

```
  GAN_RADIAL_DISTORTION_1,

  /// two parameters K1,K2 of radial distortion
  GAN_RADIAL_DISTORTION_2,

  /// three parameters K1,K2,K3 of radial distortion */
  GAN_RADIAL_DISTORTION_3,

  /// one parameter K1 of inverse radial distortion
  GAN_RADIAL_DISTORTION_1_INV,

  /// stereographic projection
  GAN_STEREOGRAPHIC_CAMERA,

  /// equidistant projection
  GAN_EQUIDISTANT_CAMERA,

  /// sine-law projection
  GAN_SINE_LAW_CAMERA,

  /// equi-solid angle projection
  GAN_EQUI_SOLID_ANGLE_CAMERA,

  /// distortion model as used by 3D Equalizer V4
  GAN_XY_DISTORTION_4,
} Gan_CameraType;
```

The linear and radial distortion models are standard models. The stereographic, equidistant, sine law and equi-solid angle models are wide-angle camera models from [5].

The coordinate frames are illustrated in Figure 5.1.

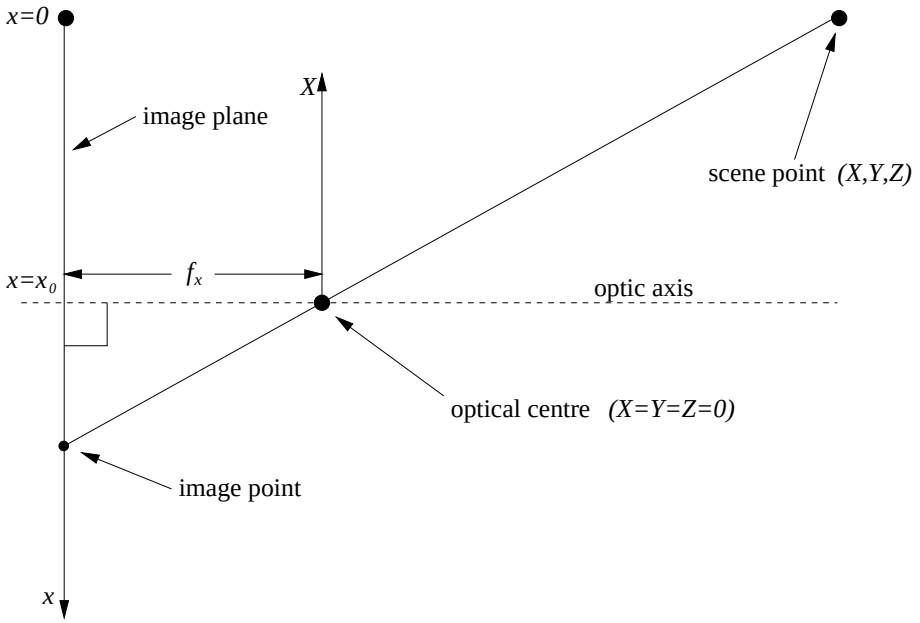

Figure 5.2: Geometrical model of projection from camera 3D coordinates X, Y, Z onto image coordinates x, y for a perfect pinhole camera, here showing only the relationship between X, Z and x. The optic axis is defined as being perpendicular to the image plane and intersecting the optical centre of the camera. Note the reversal between the 3D camera X axis and the image x axis, caused by the projection.

The linear camera model is the simplest standard camera model. It defines the following model relating camera 3D coordinates X, Y, Z to image coordinates x, y:

$$x = x_0 + f_x \frac{X}{Z}, \quad y = y_0 + f_y \frac{Y}{Z} \tag{5.1}$$

This equation derives from the similar triangles apparent in the geometrical model illustrated in Figure 5.2. The image centre coordinates x_0, y_0 and focal distance parameters f_x, f_y correspond to the similarly named `x0, y0` and `fx, fy` fields in the `Gan_Camera` structure. The normal way to write the linear model is in homogeneous coordinates, introducing the third image coordinate z_h, which can be identified with the `zh` field of the `Gan_Camera` structure:

$$\begin{pmatrix} x \\ y \\ z_h \end{pmatrix} = \lambda \begin{pmatrix} f_x & 0 & x_0 \\ 0 & f_y & y_0 \\ 0 & 0 & z_h \end{pmatrix} \begin{pmatrix} X \\ Y \\ Z \end{pmatrix} \quad \text{or} \quad \mathbf{x} = \lambda K \mathbf{X}$$

The λ parameter can be eliminated, to recover Equation 5.1. z_h can be set to one, but a good rule of thumb is to set it to roughly half the range of image x/y coordinates, so that all the image coordinates will be scaled in approximately the same way, which can reduce truncation error in certain situations. K is known as the *camera calibrarion matrix*:

$$K = \begin{pmatrix} f_x & 0 & x_0 \\ 0 & f_y & y_0 \\ 0 & 0 & z_h \end{pmatrix}$$

Note the units the elements of K. f_x and f_y both represent the same distance, the perpendicular distance from the image plane to the optical centre, but they are measured in image x and y pixels respectively. x_0 is the position of the image centre in the image x direction and measured in image x pixels, and similarly for y_0. Note also that f_x and f_y do not measure the focal *length* of the camera. The focal length is purely a property of the lens system. Under normal circumstances the focal distance will be shorter than the focal length, unless the camera is focussing at infinity when the two distances will be the same.

The linear model is an "ideal" model, corresponding to a perfect pinhole camera. It is safe to use this model when the focal length of the lens is large. In practice there will be non-linear distortions, and the simplest model of

distortion is that it is purely `radial`, i.e. directed directly towards or away from the centre of the image[1] A simple model of this distortion is

$$x = x_0 + f_x(1+d)\frac{X}{Z}, \quad y = y_0 + f_y(1+d)\frac{Y}{Z} \tag{5.2}$$

where

$$d = K_1 r^2 + K_2 r^4 + K_3 r^6 \quad \text{and} \quad r^2 = \frac{X^2+Y^2}{Z^2}$$

Here d represents the non-linear distortion. The camera models `GAN_RADIAL_DISTORTION_1`, `GAN_RADIAL_DISTORTION_2` and `GAN_RADIAL_DISTORTION_3` represent the above distortion model with respectively K_1 only, both K_1 & K_2 and all of K_1, K_2 and K_3. Once a camera has been created, it can be used to project from camera to image coordinates, or to back-project from image out into camera coordinates. Note that although the projection is apparently from a 3D space onto a 2D space, you should think instead of a projection between two 2D spaces. All camera 3D points along a straight line through the optical centre project to the same image point, so the projection is between the image plane and the space of *rays* in camera 3D space through the optical centre, a 2D space of rays.

5.1.1 Building cameras

To create a linear camera, you will need the header file

```
#include <gandalf/vision/camera_linear.h>
```

for double precision or

```
#include <gandalf/vision/cameraf_linear.h>
```

Then use the routine

```
Gan_Camera CameraD;

/* build a linear camera in double precision */
gan_camera_build_linear ( &CameraD,
                        /* ZH     FY     FX     Y0     X0 */
                           100.0, 700.0, 500.0, 150.0, 100.0 );
```

to create a double precision linear camera. There is a single-precision camera structure, called a `Gan_Camera_f`. The single precision version of the above function is

```
Gan_Camera_f CameraF;

/* build a linear camera in double precision */
gan_cameraf_build_linear ( &CameraF,
                         /* ZH      FY      FX      Y0      X0 */
                            100.0F, 700.0F, 500.0F, 150.0F, 100.0F );
```

There are similar functions for creating cameras with a radial distortion model, for which you will need one or more of the following header files:

```
#include <gandalf/vision/camera_radial_dist1.h>
#include <gandalf/vision/camera_radial_dist2.h>
#include <gandalf/vision/camera_radial_dist3.h>
#include <gandalf/vision/cameraf_radial_dist1.h>
#include <gandalf/vision/cameraf_radial_dist2.h>
#include <gandalf/vision/cameraf_radial_dist3.h>
```

[1] In practice the centre of distortion will be somewhat different from the image centre [13].

Then the functions are

```
/* build a camera with one radial distortion parameter */
gan_camera_build_radial_distortion_1 ( &CameraD,
                                  /*    ZH     FY     FX     Y0     X0 */
                                        100.0, 700.0, 500.0, 150.0, 100.0,
                                  /*    K1 */
                                        0.001 ); /* OR */
gan_cameraf_build_radial_distortion_1 ( &CameraF,
                                   /*    ZH      FY      FX      Y0      X0 */
                                         100.0F, 700.0F, 500.0F, 150.0F, 100.0F,
                                   /*    K1 */
                                         0.001F );

/* build a camera with two radial distortion parameters */
gan_camera_build_radial_distortion_2 ( &CameraD,
                                  /*    ZH     FY     FX     Y0     X0 */
                                        100.0, 700.0, 500.0, 150.0, 100.0,
                                  /*    K1,    K2 */
                                        0.001, 1.0e-7 ); /* OR */
gan_cameraf_build_radial_distortion_2 ( &CameraF,
                                   /*    ZH      FY      FX      Y0      X0 */
                                         100.0F, 700.0F, 500.0F, 150.0F, 100.0F,
                                   /*    K1,     K2 */
                                         0.001F, 1.0e-7F );
/* build a camera with three radial distortion parameters */
gan_camera_build_radial_distortion_3 ( &CameraD,
                                  /*    ZH     FY     FX     Y0     X0 */
                                        100.0, 700.0, 500.0, 150.0, 100.0,
                                  /*    K1,    K2,     K3 */
                                        0.001, 1.0e-7, -0.0001 ); /* OR */
gan_cameraf_build_radial_distortion_3 ( &CameraF,
                                   /*    ZH      FY      FX      Y0      X0 */
                                         100.0F, 700.0F, 500.0F, 150.0F, 100.0F,
                                   /*    K1,     K2,      K3 */
                                         0.001F, 1.0e-7F, -0.0001F );
```

Note that `Gan_Camera`'s and `Gan_Camera_f`'s are simple structures with no internally allocated data, so there is no `..._free` function for them.

5.1.2 Projecting points and lines

Routines for projecting points and lines into an image are provided. The double precision routines are

```
Gan_Vector3 v3X, v3x; /* declare camera/scene points X, x */
Gan_Vector3 v3L, v3l; /* declare camera/scene lines L, l */

/* fill camera point X & line L with values */
gan_vec3_fill_q ( &v3X, 1.5, -0.8, 1.2 );
gan_vec3_fill_q ( &v3L, 2.7,  3.9, 3.6 );

/* project point from camera 3D coordinates onto the image X --> x */
gan_camera_project_point_q ( &CameraD, &v3X, &v3x );
```

```
/* project line from camera 3D coordinates onto the image L --> l */
gan_camera_project_line_q ( &CameraD, &v3L, &v3l );
```

The point projection function implements the projection equations 5.1 and 5.2. The lines are represented in homogeneous 2D coordinates, so that the line $\mathbf{L}$ in camera coordinates actually describes a *plane* in 3D space intersecting the origin (optical centre). If $\mathbf{X}$ is a point in camera X, Y, Z space, the line in the homogeneous 2D space of camera "rays" is defined by the equation

$$\mathbf{L}.\mathbf{X} = 0 \quad \text{or} \quad L_X X + L_Y Y + L_Z Z = 0$$

The line in image coordinates $\mathbf{x} = (x\ y\ z_h)^\top$ is similarly defined as

$$\mathbf{l}.\mathbf{x} = 0 \quad \text{or} \quad l_x x + l_y y + l_z z_h = 0$$

Projection of lines is only available for linear cameras, since when there is distortion lines in 3D space project to curves on the image. For a linear camera the relationship between $\mathbf{L}$ and $\mathbf{l}$ is

$$\mathbf{l} = K^{-\top}\mathbf{L}$$

There are also versions of the above routines which perform the projection in-place in the input vector. So for instance

```
Gan_Vector3 v3Xx; /* declare point */
Gan_Vector3 v3Ll; /* declare line */

/* fill camera point X & line L with values */
gan_vec3_fill_q ( &v3Xx, 1.5, -0.8, 1.2 );
gan_vec3_fill_q ( &v3Ll, 2.7,  3.9, 3.6 );

/* project point from camera 3D coordinates onto the image in-place */
gan_camera_project_point_i ( &CameraD, &v3Xx );

/* project line from camera 3D coordinates onto the image in-place */
gan_camera_project_line_i ( &CameraD, &v3Ll );
```

Back-projection from image to camera coordinates operates similarly. To back-project a point and a line you can use

```
Gan_Vector3 v3X, v3x; /* declare camera/scene points X, x */
Gan_Vector3 v3L, v3l; /* declare camera/scene lines L, l */

/* fill image point x & line l with values */
gan_vec3_fill_q ( &v3x, 1.5, -0.8, 1.2 );
gan_vec3_fill_q ( &v3l, 2.7,  3.9, 3.6 );

/* back-project point from the image into camera 3D coordinates x --> X */
gan_camera_backproject_point_q ( &CameraD, &v3x, &v3X ); /* OR */
gan_camera_backproject_point_i ( &CameraD, &v3x ); /* in-place */

/* backproject line from the image into camera 3D coordinates l --> L */
gan_camera_backproject_line_q ( &CameraD, &v3l, &v3L ); /* OR */
gan_camera_backproject_line_i ( &CameraD, &v3l ); /* in-place */
```

The single precision versions of these functions operate similarly. The single precision camera to image projection functions are

```
Gan_Vector3_f v3X, v3x; /* declare camera/scene points X, x */
Gan_Vector3_f v3L, v3l; /* declare camera/scene lines L, l */

/* fill camera point X & line L with values */
gan_vec3f_fill_q ( &v3X, 1.5F, -0.8F, 1.2F );
gan_vec3f_fill_q ( &v3L, 2.7F,  3.9F, 3.6F );

/* project point from camera 3D coordinates onto the image X --> x */
gan_cameraf_project_point_q ( &CameraF, &v3X, &v3x ); /* OR */
gan_cameraf_project_point_i ( &CameraF, &v3X ); /* in-place */

/* project line from camera 3D coordinates onto the image L --> l */
gan_cameraf_project_line_q ( &CameraF, &v3L, &v3l ); /* OR */
gan_cameraf_project_line_i ( &CameraF, &v3L ); /* in-place */
```

The single precision image to camera back-projection functions are

```
Gan_Vector3_f v3X, v3x; /* declare camera/scene points X, x */
Gan_Vector3_f v3L, v3l; /* declare camera/scene lines L, l */

/* fill image point x & line l with values */
gan_vec3f_fill_q ( &v3x, 1.5F, -0.8F, 1.2F );
gan_vec3f_fill_q ( &v3l, 2.7F,  3.9F, 3.6F );

/* project point from camera 3D coordinates onto the image X --> x */
gan_cameraf_backproject_point_q ( &CameraF, &v3x, &v3X ); /* OR */
gan_cameraf_backproject_point_i ( &CameraF, &v3x ); /* in-place */

/* project line from camera 3D coordinates onto the image L --> l */
gan_cameraf_backproject_line_q ( &CameraF, &v3l, &v3L ); /* OR */
gan_cameraf_backproject_line_i ( &CameraF, &v3l ); /* in-place */
```

5.1.3 Adding/removing camera distortion

Gandalf also supplies some functions for adding and removing the image plane distortion from an image point. So for instance

```
Gan_Camera CameraD;
Gan_Vector3 v3x, v3xu;

/* build camera with one parameter of radial distortion */
gan_camera_build_radial_distortion_1 ( &CameraD,
                                /*    ZH     FY     FX     Y0     X0 */
                                      100.0, 700.0, 500.0, 150.0, 100.0,
                                /*    K1 */
                                      0.001 );

/* build image point x assumed to have distortion */
gan_vec3_fill_q ( &v3x, 50.0, -80.0, 100.0 );

/* remove distortion from image point x --> xu */
gan_camera_remove_distortion_q ( &CameraD, &v3x, &v3xu );
```

removes the distortion from the image point x, producing an undistorted point xu. Given the camera 3D point **X** that projects onto x, xu is defined as the point on the image onto which the equivalent linear camera (i.e. the linear camera with the same f_x, f_y, x_0, y_0 and z_h) would project when applied to **X**. The in-place version of this function is

```
/* remove distortion from image point x --> xu in-place */
gan_camera_remove_distortion_i ( &CameraD, &v3x );
```

The reverse is to add distortion to an image point. Given a non-linear camera, this means converting a point projected with the equivalent linear camera to a point projected with the non-linear camera:

```
/* build image point xu assumed to have NO distortion */
gan_vec3_fill_q ( &v3xu, 50.0, -80.0, 100.0 );

/* add distortion to image point xu --> x */
gan_camera_add_distortion_q ( &CameraD, &v3xu, &v3x ); /* OR */
gan_camera_add_distortion_i ( &CameraD, &v3xu ); /* in-place */
```

The single precision versions of these routines are

```
Gan_Camera_f CameraF;
Gan_Vector3_f v3x, v3xu;

/* build camera with one parameter of radial distortion */
gan_cameraf_build_radial_distortion_1 ( &CameraF,
                                  /*    ZH      FY      FX      Y0      X0 */
                                        100.0F, 700.0F, 500.0F, 150.0F, 100.0F,
                                  /*    K1 */
                                        0.001F );

/* build image point x assumed to have distortion */
gan_vec3f_fill_q ( &v3x, 50.0F, -80.0F, 100.0F );

/* remove distortion from image point x --> xu */
gan_cameraf_remove_distortion_q ( &CameraF, &v3x, &v3xu ); /* OR */
gan_cameraf_remove_distortion_i ( &CameraF, &v3x ); /* in-place */

/* build image point xu assumed to have NO distortion */
gan_vec3f_fill_q ( &v3xu, 50.0F, -80.0F, 100.0F );

/* add distortion to image point xu --> x */
gan_cameraf_add_distortion_q ( &CameraF, &v3xu, &v3x ); /* OR */
gan_cameraf_add_distortion_i ( &CameraF, &v3xu ); /* in-place */
```

5.1.4 Building the camera calibration matrix

The camera calibration matrix K is triangular. Gandalf provides a routine to build K from the calibration structure. The double precision version is

```
Gan_Camera CameraD; /* declare camera structure */
Gan_SquMatrix33 sm33K; /* declare camera calibration matrix K */

/* ... build camera using e.g. gan_camera_build_linear() ... */
```

```
/* build camera calibration matrix K */
sm33K = gan_camera_fill_matrix_s ( &CameraD );
```

and the single precision version is

```
Gan_Camera_f CameraF; /* declare camera structure */
Gan_SquMatrix33_f sm33K; /* declare camera calibration matrix K */

/* ... build camera using e.g. gan_cameraf_build_linear() ... */

/* build camera calibration matrix K */
sm33K = gan_cameraf_fill_matrix_s ( &CameraF );
```

Note that although K is an upper triangular matrix, the routines above produce a *lower* triangular matrix, since that is the only form of fixed size triangular matrix supported by Gandalf. As explained in Section 3.1.2, Any operations involving upper triangular matrices can be implemented using implicit transpose of a lower triangular matrix.

5.1.5 Converting cameras between precisions

It is sometimes necessary to convert from a double precision `Gan_Camera` to a single precision `Gan_Camera_f` or vice versa. Gandalf provides two versions of these routines:

```
Gan_Camera   CameraD; /* double precision camera */
Gan_Camera_f CameraF; /* single precision camera */

/* ... build CameraD using e.g. gan_cameraf_build_linear() ... */

/* convert camera from double precision to single precision */
gan_cameraf_from_camera_q ( &CameraD, &CameraF ); /* OR */
CameraF = gan_cameraf_from_camera_s ( &CameraD );

/* convert camera back from single precision to double precision */
gan_camera_from_cameraf_q ( &CameraF, &CameraD ); /* OR */
CameraD = gan_camera_from_cameraf_s ( &CameraF );
```

5.2 Computing the fundamental/essential matrix

```
#include <gandalf/vision/fundamental.h>
#include <gandalf/vision/essential.h>
```

The fundamental matrix [8] encodes all the geometrical constraints available given two images of a rigid scene. Given two images with point locations $\mathbf{x}_1 = (x_1\ y_1\ z_h)^\top$ and $\mathbf{x}_2 = (x_2\ y_2\ z_h)^\top$ in homogeneous coordinates, the relationship between image points projected from the same scene point is

$$\mathbf{x}_2^\top F \mathbf{x}_1 = 0$$

F is the 3×3 fundamental matrix. To compute F you can use multiple point matches to solve the above homogeneous linear equations. The standard technique is to use pre-conditioning followed by symmetric matrix eigendecomposition to solve for the nine elements of F up to an undetermined scale factor.

```
Gan_SymMatEigenStruct SymEigen;
Gan_Vector3 *av3Point1, *av3Point2; /* arrays of image points */
Gan_Matrix33 m33F;

/* allocate arrays of image coordinates, one array for each image */
av3Point1 = gan_malloc_array ( Gan_Vector3, 100 );
av3Point2 = gan_malloc_array ( Gan_Vector3, 100 );

/* ... fill arrays av3Point1 and av3Point2 with point correspondence
       data for 100 points ... */

/* create structure for computing eigenvalues and eigenvectors,
   initialising accumulated matrix S (here 9x9) to zero */
gan_symeigen_form ( &SymEigen, 9 );

/* solve for fundamental matrix */
gan_fundamental_matrix_fit ( av3Point1, av3Point2, 100, &SymEigen, &m33F );

/* free stuff */
gan_symeigen_free ( &SymEigen );
gan_free_va ( av3Point2, av3Point1, NULL );
```

The essential matrix E is the equivalent of the fundamental matrix in the case of known camera calibration parameters. In this case the rotation between the cameras can be computed, and also the translation vector between them up to an unknown scale factor. The mathematical model is that the images are related by the equation

$$\mathbf{x}_2'^{\top} E \mathbf{x}_1' = 0$$

involving the essential matrix E, where $\mathbf{x}_1'$, $\mathbf{x}_2'$ are *ideal* image coordinates for images 1 & 2, transformed from the original image coordinates $\mathbf{x}_1$, $\mathbf{x}_2$ so that $\mathbf{x}_1'$ and $\mathbf{x}_2'$ are the projected coordinates for an *ideal camera*, which is to say a linear camera with focal distances $f_x = f_y = 1$, image centre $x_0 = y_0 = 0$, and homogeneous z-coordinate $z_h = 1$. The camera calibration matrix for an ideal camera is $K = I_{3\times 3}$. This means that the 3D camera coordinate frame can be identified with the ideal image frame (up to scale as usual). The essential matrix can be written as

$$E = R[\mathbf{T}]_\times$$

where R is the rotation matrix, $\mathbf{T}$ is the translation vector between the camera positions and $[\mathbf{T}]_\times$ is the "cross product matrix" of $\mathbf{T}$, defined as

$$\mathbf{T} = \begin{pmatrix} T_X \\ T_Y \\ T_Z \end{pmatrix}, \quad [\mathbf{T}]_\times = \begin{pmatrix} 0 & -T_Z & T_Y \\ T_Z & 0 & -T_X \\ -T_Y & T_X & 0 \end{pmatrix}.$$

It is termed the cross product matrix because given any 3-vectors $\mathbf{x}$ and $\mathbf{y}$, $[\mathbf{x}]_\times \mathbf{y} = \mathbf{x} \times \mathbf{y}$. For more details of the essential matrix see [4].

The main difference in Gandalf between computing the fundamental and essential matrices is in the computation of the ideal image coordinates. These can be computed by back-projecting the original image coordinates out into 3D camera (ideal image) coordinates, using the Gandalf back-projection function described in Section 5.1. Here is a code fragment to compute the essential matrix, represented by the rotation R and translation $\mathbf{T}$.

```
Gan_SymMatEigenStruct SymEigen;
Gan_Vector3 *av3Point1, *av3Point2; /* arrays of image points */
Gan_Camera Camera;
Gan_Euclid3D Pose;
```

```
/* allocate arrays of image coordinates, one array for each image */
av3Point1 = gan_malloc_array ( Gan_Vector3, 100 );
av3Point2 = gan_malloc_array ( Gan_Vector3, 100 );

/* ... fill arrays av3Point1 and av3Point2 with point correspondence
       data for 100 points ... */

/* build a camera with two radial distortion parameters */
gan_camera_build_radial_distortion_2 ( &Camera,
                                       100.0, 700.0, 500.0, 150.0, 100.0,
                                       0.001, 1.0e-7 );

/* create structure for computing eigenvalues and eigenvectors,
   initialising accumulated matrix S (here 9x9) to zero */
gan_symeigen_form ( &SymEigen, 9 );

/* compute essential matrix */
gan_essential_matrix_fit ( av3Point1, av3Point2, 100, &Camera, &Camera,
                           &SymEigen, &Pose );

/* free stuff */
gan_symeigen_free ( &SymEigen );
gan_free_va ( av3Point2, av3Point1, NULL );
```

The Gandalf 3D Euclidean transformation structure `Gan_Euclid3D` is used to store the result rotation and translation.

Error detection: Both fundamental and essential matrix routines return a boolean value, which is `GAN_FALSE` on error, invoking the Gandalf error handler.

5.3 Computing a homography between 2D scene and image

```
#include <gandalf/vision/homog33_fit.h>
```

If a part of the viewed scene is planar, or the camera is undergoing a pure rotation (or both), the (part of the) scene can be reconstructed using 2D methods. Here we assume a point-cloud representation, so the scene is represented by n points $\mathbf{X}_i$ in homogeneous coordinates, $i = 1, \ldots, n$. The relationship between the $\mathbf{X}_i$ and points $\mathbf{x}_i$ in an image of the same (part of the) scene is a simple linear projective transformation or homography:

$$\mathbf{x}_i = \lambda_i P \mathbf{X}_i \tag{5.3}$$

P is a 3×3 matrix representing the homography and λ is a scale factor. This equation also assumes that the camera employed in projecting the points onto the image is linear, but if the camera is non-linear AND the camera parameters are known, the distortion can be removed first by applying the function `gan_camera_remove_distortion_[qi]()` to the image points $\mathbf{x}_i$ as described in Section 5.1. Given four or more point correspondences in the image (in general position), the homography matrix P can be computed. This can be done by first eliminating λ to obtain homogeneous linear equations for the elements of P. Given that $\mathbf{X} = (X\ Y\ Z)^\top$ and $\mathbf{x} = (x\ y\ z_h)^\top$, we can obtain the equations

$$x\mathbf{P}_3.\mathbf{X} - z_h\mathbf{P}_1.\mathbf{X} = 0, \quad y\mathbf{P}_3.\mathbf{X} - z_h\mathbf{P}_2.\mathbf{X} = 0 \tag{5.4}$$

where P is separated into rows as

$$P = \begin{pmatrix} \mathbf{P}_1^\top \\ \mathbf{P}_2^\top \\ \mathbf{P}_3^\top \end{pmatrix}$$

From four points we get eight such equations, which allows P to be computed up to a scale factor using the same symmetric eigensystem routines as are used to solve for the fundamental and essential matrices above.

Note that this formulation differs from the normal formulation which considers the homographies *between* images. That is a special case of our formulation, because we can take an image as the projective "scene" representation $\mathbf{X}_i$. The scene/image formulation also allows us to represent the motion over a sequence of k images in a compact way as the set of homographies $P_{(j)}$ for images $j = 1, \ldots, k$ mapping the scene $\mathbf{X}_i$ to each set of image points $\mathbf{x}_{i(j)}$, rather than as an arbitrary collection of pairwise homographies.

To start the calculation, define an accumulated symmetric matrix eigensystem structure and initialise it using the following routine:

```
Gan_SymMatEigenStruct SymEigen;

/* initialise eigensystem matrix */
gan_homog33_init ( &SymEigen );
```

Then for each point correspondence, build the equations 5.4 and increment the accumulated symmetric eigensystem matrix by calling the following function:

```
int iEqCount=0, iCount;
Gan_Vector3 v3X, v3x; /* declare scene and image points X & x */

for ( iCount = 0; iCount < 100; iCount++ )
{
   /* ... build scene and image point coordinates into X and x ... */

   /* increment matrix using point correspondence */
   gan_homog33_increment_p ( &SymEigen, &v3X, &v3x, 1.0, &iEqCount );
}
```

The fourth argument `1.0` is a weighting factor for the equations as described in Section 3.2.2.15. The last argument `iEqCount` is a running count of the total number of equations processed thus far, to be passed below to the function to solve for P.

Once the point correspondences have been processed in this way, you can solve the equations using

```
Gan_Matrix33 m33P; /* homography matrix P */

gan_homog33_solve ( &SymEigen, iEqCount, &m33P );
```

to compute the homography P. If you want to repeat the calculation of a homography with new data, you can start again by calling

```
gan_homog33_reset ( &SymEigen );
```

At the end of the homography calculation(s) you can free the eigensystem structure using the function

```
gan_homog33_free ( &SymEigen );
```

Given correspondences between lines, it is also possible to generate homogeneous linear equations for P and either combine with points or compute P purely from lines. To see how to derive the equations for lines, take the line equations

$$\mathbf{L}.\mathbf{X} = 0, \quad \mathbf{l}.\mathbf{x} = 0$$

define the homogeneous line parameters $\mathbf{L}$ in the scene and $\mathbf{l}$ in the image. We can derive the relationship between $\mathbf{L}$, $\mathbf{l}$ and P using the point projection equation 5.3, yielding

$$\mathbf{L} = \mu P^\top \mathbf{l}$$

for a scale factor μ. Separating P into columns as

$$P = \begin{pmatrix} \mathbf{P}'_1 & \mathbf{P}'_2 & \mathbf{P}'_3 \end{pmatrix},$$

eliminating μ from the above equation, and writing $\mathbf{L} = (L_X \; L_Y \; L_Z)^\top$, we obtain the two homogeneous linear equations

$$L_X \mathbf{P}'_3.\mathbf{l} = L_Z \mathbf{P}'_1.\mathbf{l}, \quad L_Y \mathbf{P}'_3.\mathbf{l} = L_Z \mathbf{P}'_2.\mathbf{l}.$$

Given correspondence between a known scene line $\mathbf{L}$ and a known image line $\mathbf{l}$, the following routine generates these equations and accumulates them in the calculation of P:

```
Gan_Vector3 v3L, v3l; /* declare scene line L and image line l */

/* ... fill L and l with values for corresponding lines ... */

/* increment matrix using line correspondence */
gan_homog33_increment_l ( &SymEigen, &v3L, &v3l, 1.0, &iEqCount );
```

This is assuming that the endpoints of the scene line are unknown. In practice the scene line will normally be created from previous matching of image lines, which are line *segments*, so that the endpoints $\mathbf{X}_1$ and $\mathbf{X}_2$ of the line in scene coordinates will be approximately known. Note that we don't depend on locating the actual endpoints of the line accurately, which is a notoriously difficult problem. You should think of the two points $\mathbf{X}_1$ and $\mathbf{X}_2$ instead as *representative* points on the line. In this case there is an alternative way of incorporating the line information which seems to give better numerical performance. We note that the scene line endpoints $\mathbf{X}_1$ and $\mathbf{X}_2$ should project onto the image line $\mathbf{l}$, so we obtain

$$\mathbf{l}.(P\mathbf{X}_1) = 0, \quad \mathbf{l}.(P\mathbf{X}_2) = 0$$

These are homogeneous linear equations in the elements of P which can be directly fed into the accumulated matrix calculation for P, using the routine

```
Gan_Vector3 v3X1, v3X2; /* declare scene line endpoints X1 & X2 */
Gan_Vector3 v3l; /* image line homogeneous coordinates l */

/* ... set X1, X2 and l for corresponding scene line and image line ... */

/* add equations for two endpoints */
gan_homog33_increment_le ( &SymEigen, &v3X1, &v3l, 1.0, &iEqCount );
gan_homog33_increment_le ( &SymEigen, &v3X2, &v3l, 1.0, &iEqCount );
```

Error detection: `gan_homog33_init()` returns a pointer to the initialised structure, and returns `NULL` on error. All the other routines except the `void` routine `gan_homog33_free()` return a boolean value, which is `GAN_FALSE` on error. The Gandalf error handler is invoked when an error occurs.

5.3.1 Computing a 2D homography from an array of feature matches

The above routines are designed for incremental computation of the homography P as more point/line feature matches become available. An alternative is to store all the feature matches in an array of match structures; indeed the array can in practice be the result of feature matching. The match structure defined here has the same match options as the above routines, encapsulated into the following enumerated type.

```
/* type of matching feature when computing 2D homography */
typedef enum { GAN_HOMOG33_POINT, /* Match scene point to image point */
               GAN_HOMOG33_LINE, /* Match scene line to image line */
               GAN_HOMOG33_LINE_ENDPOINTS, /* Match scene line endpoints to
                                              image line */
               GAN_HOMOG33_IGNORE } /* rejected match */
 Gan_Homog33MatchType;
```

where GAN_HOMOG33_IGNORE denotes a match that has been rejected. The match structure contains the details of the match:

```
/* structure to hold details of scene and image data to be used in
 * computing 2D homographies
 */
typedef struct
{
   Gan_Homog33MatchType type;
   union
   {
      struct { Gan_Vector3 X, x; } p; /* point --> point match */
      struct { Gan_Vector3 L, l; } l; /* line --> line match */
      struct { Gan_Vector3 X1, X2, l; } le; /* line endpoints --> line match */
   } d;
} Gan_Homog33Match;
```

Given an array of the Gan_Homog33Match structures, you can compute the homography from scene to image by calling

```
Gan_Homog33Match *aMatch;
unsigned uiNoMatches;
Gan_Matrix33 m33P;

/* ... create and fill array of matches, set uiNoMatches to the number
       of structures in the array ... */

/* fit projective 2D homography */
gan_homog33_fit ( aMatch, uiNoMatches, &m33P );
```

Error detection: gan_homog33_fit() returns a boolean value; hence GAN_FALSE is returned on error and the Gandalf error handler is invoked.

5.3.2 Computing a 2D affine homography

```
#include <gandalf/vision/affine33_fit.h>
```

If the region of the scene in which a homography is to be computed is small, or a long focal length lens is being used, an affine 2D model of motion is usually adequate, and indeed computing a full projective model can become unstable. The function defined in this module is a version of gan_homog33_fit() for computing an affine 2D homography, which can be formed from a full projective homography by imposing the constraints $P_{31} = P_{32} = 0$, $P_{33} = 1$. To fit an affine 2D homography replace the call to gan_homog33_fit() in the above code fragment with

```
/* fit affine 2D homography */
gan_affine33_fit ( aMatch, uiNoMatches, &m33P );
```

`Error detection:` `gan_affine33_fit()` returns a boolean value; hence `GAN_FALSE` is returned on error and the Gandalf error handler is invoked.

5.4 Computing a homography between 3D scene and image

```
#include <gandalf/vision/affine34_fit.h>
```

Pose estimation is the procedure to compute the position of a camera relative to a known scene. In projective terms it means estimating the 3×4 homography matrix P representing the projection from the 3D scene into the 2D image. Here we assume a point-cloud representation, so the scene is represented by n 3D points $\mathbf{X}_i$ in homogeneous coordinates, $i = 1, \ldots, n$. The relationship between the $\mathbf{X}_i$ and points $\mathbf{x}_i$ in an image of the same (part of the) scene is a simple linear projective transformation or homography:

$$\mathbf{x}_i = \lambda_i P \mathbf{X}_i \tag{5.5}$$

P is a 3×4 homography matrix and λ is a scale factor. This equation also assumes that the camera employed in projecting the points onto the image is linear, but if the camera is non-linear AND the camera parameters are known, the distortion can be removed first by applying the function `gan_camera_remove_distortion_[qi]()` to the image points $\mathbf{x}_i$ as described in Section 5.1. Given six or more point correspondences (in general 3D position) in two images, the homography matrix P can be computed. This can be done by first eliminating λ to obtain homogeneous linear equations for the elements of P. Given that $\mathbf{X} = (X\ Y\ Z\ W)^\top$ and $\mathbf{x} = (x\ y\ z_h)^\top$, we can obtain the equations

$$x\mathbf{P}_3.\mathbf{X} - z_h\mathbf{P}_1.\mathbf{X} = 0, \quad y\mathbf{P}_3.\mathbf{X} - z_h\mathbf{P}_2.\mathbf{X} = 0 \tag{5.6}$$

where P is separated into rows as

$$P = \begin{pmatrix} \mathbf{P}_1^\top \\ \mathbf{P}_2^\top \\ \mathbf{P}_3^\top \end{pmatrix}$$

From six points we get twelve such equations, which allows P to be computed up to a scale factor[2] using the same symmetric eigensystem routines as are used to solve for the fundamental and essential matrices in Section 5.2.

To start the calculation, define an accumulated symmetric matrix eigensystem structure and initialise it using the following routine:

```
Gan_SymMatEigenStruct SymEigen;

/* initialise eigensystem matrix */
gan_homog34_init ( &SymEigen );
```

Then for each point correspondence, build the equations 5.6 and increment the accumulated symmetric eigensystem matrix by calling the following function:

```
int iEqCount=0, iCount;
Gan_Vector4 v4X; /* declare scene point X */
Gan_Vector3 v3x; /* declare image point x */

for ( iCount = 0; iCount < 100; iCount++ )
{
   /* ... build scene and image point coordinates into X and x ... */

   /* increment matrix using point correspondence */
   gan_homog34_increment_p ( &SymEigen, &v4X, &v3x, 1.0, &iEqCount );
}
```

[2] In fact only eleven equations are required.

The fourth argument `1.0` is a weighting factor for the equations as described in Section 3.2.2.15. The last argument `iEqCount` is a running count of the total number of equations processed thus far, to be passed below to the function to solve for P.

Once the point correspondences have been processed in this way, you can solve the equations using

```
Gan_Matrix34 m34P; /* homography matrix P */

gan_homog34_solve ( &SymEigen, iEqCount, &m34P );
```

to compute the homography P. If you want to repeat the calculation of a homography with new data, you can start again by calling

```
gan_homog34_reset ( &SymEigen );
```

At the end of the homography calculation(s) you can free the eigensystem structure using the function

```
gan_homog34_free ( &SymEigen );
```

If line matches are available, and the endpoints of the 3D line are approximately known, the line information can also be incorporated into the calculation. Since the scene line will normally be created from previous matching of image lines, which are line *segments*, the endpoints $\mathbf{X}_1$ and $\mathbf{X}_2$ of the line in scene coordinates should indeed be known. Note that we don't depend on locating the actual endpoints of the line accurately, which is a notoriously difficult problem. You should think of the two points $\mathbf{X}_1$ and $\mathbf{X}_2$ instead as *representative* points on the line. We note that $\mathbf{X}_1$ and $\mathbf{X}_2$ should project onto the image line $\mathbf{l}$, and so we obtain the equations

$$\mathbf{l}.(P\mathbf{X}_1) = 0, \quad \mathbf{l}.(P\mathbf{X}_2) = 0$$

These are homogeneous linear equations in the elements of P which can be directly fed into the accumulated matrix calculation for P, using the routine

```
Gan_Vector4 v4X1, v4X2; /* declare scene line endpoints X1 & X2 */
Gan_Vector3 v3l; /* image line homogeneous coordinates l */

/* ... set X1, X2 and l for corresponding scene line and image line ... */

/* add equations for two endpoints */
gan_homog34_increment_le ( &SymEigen, &v4X1, &v3l, 1.0, &iEqCount );
gan_homog34_increment_le ( &SymEigen, &v4X2, &v3l, 1.0, &iEqCount );
```

Error detection: `gan_homog34_init()` returns a pointer to the initialised structure, and returns `NULL` on error. All the other routines except the `void` routine `gan_homog34_free()` return a boolean value, which is `GAN_FALSE` on error. The Gandalf error handler is invoked when an error occurs.

5.5 Smoothing an image using a 1D convolution mask

```
#include <gandalf/vision/mask1D.h>
#include <gandalf/vision/convolve1D.h>
```

This module deals with creating 1D convolution masks, used in Gandalf for convolving an image with a separable filter, which is a filter whose functional form can be factored into independent one-dimensional filters in the x and y directions. 2D Gaussian convolution, for instance, can be implemented using two 1D convolutions in sequence, one in the x direction and one in the y direction. In this case the 1D convolution mask would be symmetrical

around zero. Convolution by a derivative of Gaussian filter is also separable, but in this case the derivative filter is antisymmetric. Knowledge of the specific shape of the filter can help improve the efficiency of the convolution, by reducing the number of required multiplications. Gandalf defines an enumerated type defining the shape of a convolution mask:

```
/* format of convolution mask */
typedef enum { GAN_MASK1D_SYMMETRIC, GAN_MASK1D_ANTISYMMETRIC,
               GAN_MASK1D_GENERIC }
   Gan_Mask1DFormat;
```

`GAN_MASK1D_GENERIC` should be used when the filter does not fit one of the special types. The following code creates a symmetrical convolution mask.

```
Gan_Mask1D *pMask;

/* create symmetric 1D convolution mask */
pMask = *gan_mask1D_alloc ( GAN_MASK1D_SYMMETRIC, GAN_FLOAT, 9 );
```

This mask can be filled with data by directly accessing the `data.f` field of the mask structure, in this case an array of five `float`s containing the positive x half of the convolution mask. For Gaussian convolutions there is a function to create the mask and fill it with values:

```
Gan_Mask1D *pMask;

/* create symmetric 1D convolution mask */
pMask = gan_gauss_mask_new ( GAN_FLOAT,
                             1.0, /* standard deviation of Gaussian */
                             9, /* size of mask */
                             1.0, /* scaling of values */,
                             NULL );
```

The convolution mask can then be applied to an image, using the following routines:

```
Gan_Image *pOriginalImage; /* declare original image */
Gan_Image *pXSmoothedImage; /* declare image smoothed in x-direction */
Gan_Image *pXYSmoothedImage; /* declare image smoothed in x & y directions */
Gan_Mask1D *pMask;

/* ... create and fill original image, create smoothed images, and
       build Gaussian convolution mask ... */

/* apply smoothing in the x direction */
gan_image_convolve1Dx_q ( pOriginalImage, GAN_INTENSITY_CHANNEL,
                          pMask, pXSmoothedImage );

/* apply smoothing in the y direction */
gan_image_convolve1Dy_q ( pXSmoothedImage, GAN_INTENSITY_CHANNEL,
                          pMask, pXYSmoothedImage );
```

The second `Gan_ImageChannelType` argument allows you to selectively convolve a single channel of a multi-channel image, such as an RGB colout image. The result of this pair of 1D convolutions is a 2D Gaussian image convolution (they could be applied in the reverse order to achieve the same result). The convolution is computed only where all the pixels within the mask are available, so, for instance, convolution in the x-direction with a Gaussian mask of size nine reduces the width of the result image by eight pixels.

There are also functions to compute the convolved images without first creating them:

```
/* apply smoothing in the x direction */
pXSmoothedImage = gan_image_convolve1Dx_s ( pOriginalImage, GAN_INTENSITY_CHANNEL,
                                            pMask );

/* apply smoothing in the y direction */
pXYSmoothedImage = gan_image_convolve1Dy_s ( pXSmoothedImage, GAN_INTENSITY_CHANNEL,
                                             pMask );
```

To free a convolution mask use the function

```
gan_mask1D_free ( pMask );
```

5.6 Smoothing an image using a 2D convolution mask

```
#include <gandalf/vision/mask2D.h>
#include <gandalf/vision/convolve2D.h>
```

This module deals with creating 2D convolution masks, used in Gandalf for convolving an image with a bidimensional filter (understood as a matrix). The dimensions of these masks (number of rows and columns) must be necessarily odd, otherwise an error will occur.

Similarly to 1D convolutions, three types of masks are considered, although the meaning of their names is a bit different from that of `Gan_Mask1D`.

```
/* format of 2D convolution mask */
typedef enum { GAN_MASK2D_SYMMETRIC, GAN_MASK2D_ANTISYMMETRIC,
               GAN_MASK2D_GENERIC }
   Gan_Mask2DFormat;
```

On the one hand, `GAN_MASK2D_GENERIC` represents a $m \times n$ generic mask with no regularity in the values that contains, where m is the number of rows and n is the number of columns (both are odd).

If we divide the 2D mask in four sections or quadrants by taking the vertical and horizontal axes through the center of the mask, we shall consider the upper left quadrant as representative of the values of the whole mask for both `GAN_MASK2D_SYMMETRIC` and `GAN_MASK2D_ANTISYMMETRIC` types. For the symmetric case, the four quadrants of the mask contain exactly the same values, and therefore they are symmetric with respect to the before mentioned axes. There are only $((m-1)/2+1) \times ((n-1)/2+1)$ independent elements of the mask.

On the other hand, for the antisymmetric case, the upper left quadrant and the lower right one have the same values, while the upper right quadrant and the lower left one have the opposite values. The elements located exactly at the vertical and horizontal axes through the center are equal zero. In this case there are only $((m-1)/2) \times ((n-1)/2)$ independent elements of the mask.

To create a 2D generic convolution mask of floats, for example, the following code can be used:

```
Gan_Mask2D *pMask_gen;
unsigned int rows = 9, cols = 7;

/* create symmetric 2D convolution mask */
pMask_gen = *gan_mask2D_alloc ( GAN_MASK2D_GENERIC,
                                GAN_FLOAT, rows, cols );
```

For a 2D symmetric convolution mask of the same size, we would have written the following lines:

```
Gan_Mask2D *pMask_sym;
unsigned int rows = 9, cols = 7;

/* create symmetric 2D convolution mask */
pMask_sym = *gan_mask2D_alloc ( GAN_MASK2D_SYMMETRIC,
                                GAN_FLOAT, rows, cols );
```

Remember that in this case only 5×4 elements have to be specified (see below how). Notice, however, that the numbers of rows and columns in the mask creation refer to the total size, so we still request for a 9×7 mask. Internally the `gan_mask2D_alloc` function knows how many elements have to be allocated according to the mask format.

Similarly, for a 2D antisymmetric convolution mask of the same size, only 4×3 elements need to be specified and we could have:

```
Gan_Mask2D *pMask_antisym;
unsigned int rows = 9, cols = 7;

/* create antisymmetric 2D convolution mask */
pMask_antisym = *gan_mask2D_alloc ( GAN_MASK2D_ANTISYMMETRIC,
                                    GAN_FLOAT, rows, cols );
```

In all the three previous cases, there is memory allocation that is transparent and "intelligent" for the end-user, in the sense that only the adequate number of elements is allocated according to the mask format.

Another way to initialize a mask is by means of the following function:

```
Gan_Mask2D *gan_mask2D_alloc_data ( Gan_Mask2DFormat format,
                                    Gan_Matrix *data,
                                    unsigned int rows,
                                    unsigned int cols );
```

In this case, a 2D convolution mask is generated, the memory for the corresponding number of elements is allocated (as in the previous fucntion), and, as well, these elements are given a value by means of a `Gan_Matrix` parameter. Bear in mind that this matrix must necessarily have the adequate size.

In the following example, a 9×7 symmetric mask is initialized with a 5×4 matrix of data.

```
// Generation of a 9x7 symmetric convolution mask.
Gan_Matrix *mat = gan_mat_alloc(5,4);
mat = gan_mat_fill_zero_q(mat,5,4);
gan_mat_set_el(mat,0,0,1.);
gan_mat_set_el(mat,1,2,2.);
gan_mat_set_el(mat,2,1,3.);
gan_mat_set_el(mat,3,3,1.);
gan_mat_set_el(mat,4,0,2.);
gan_mat_set_el(mat,0,2,3.);
gan_mat_set_el(mat,1,1,1.);
gan_mat_set_el(mat,3,1,2.);
```

```
gan_mat_set_el(mat,4,2,3.);
gan_mat_set_el(mat,2,2,1.);

Gan_Mask2D *mask_sym;
mask_sym = gan_mask2D_alloc_data (GAN_MASK2D_SYMMETRIC,mat,9,7);
```

The convolution mask can then be applied to an image, by means of the following routines (with the usual convention _q for the "quick" version and _s for the "slow" one):

```
Gan_Image *gan_image_convolve2D_q ( Gan_Image *image,
                                    Gan_ImageChannelType channel,
                                    Gan_Mask2D *mask,
                                    Gan_Image *dest );

Gan_Image *gan_image_convolve2D_s ( Gan_Image *image,
                                    Gan_ImageChannelType channel,
                                    Gan_Mask2D *mask );
```

In these functions, the parameters are: the image to convolve, the channel to convolve and the 2D convolution mask to use. For the "quick" version, there is a fourth parameter, which is the image that stores the result, with no memory allocation for it.

Let's see an example of use:

```
Gan_Image *pOriginalImage; /* declare original image */
Gan_Image *pSmoothedImage; /* declare smoothed image */
Gan_Mask2D *pMask;

/*
  Here we have to do the following operations:
    1. Fill the original image with values (for example, with
       grey levels).
    2. Fill the 2D mask with values as shown before.
    3. Allocate memory for the result image.
*/

/* Apply smoothing */
gan_image_convolve2D_q ( pOriginalImage,
                         GAN_INTENSITY_CHANNEL,
                         pMask,
                         pSmoothedImage );
```

According to the image format, the following channels can be used:

- For grey level images, GAN_INTENSITY_CHANNEL and GAN_ALL_CHANNELS apply the mask to the whole image (both have the same effect). If GAN_X_CHANNEL, GAN_Y_CHANNEL or GAN_Z_CHANNEL are used, then the grey-level image is convolved and the result is a 2D image with 3D vectors, where the convolution is stored in the X, Y or Z component of the vectors, respectively.
- For RGB colour images (with or without alpha channel), a monochromatic convolution can be applied (GAN_RED_CHANNEL, GAN_GREEN_CHANNEL, GAN_BLUE_CHANNEL), so that the result is a grey-level image. A global convolution can also be performed (GAN_ALL_CHANNELS).

- For GAN_VECTOR_FIELD_3D images, a global convolution can be performed (GAN_ALL_CHANNELS), or only for one of the components (GAN_X_CHANNEL, GAN_Y_CHANNEL, GAN_Z_CHANNEL).

Notice that the convolution is applied only where all the pixels within the mask are available. Therefore, if the size of the original image is $DIMY \times DIMX$ and the size of the mask is $m \times n$, then the size of the convolved image is $(DIMY - m + 1) \times (DIMX - n + 1)$.

To free a convolution mask use the following function:

```
gan_mask2D_free ( pMask );
```

5.7 Feature detection

Gandalf currently has versions of the Canny edge detector, the Plessey corner detector and a line segment finder. The different feature detectors follow the same layered scheme, having a feature module which supports other feature detectors of the same type, below the specialised algorithm which implements a specific feature detector. Another important part of the Gandalf feature detectors is the *local feature map*. This encapsulates a local blocked representation of s feature map, allowing very fast search for features in regions of the image, and supporting the development of feature tracking/matching.

There is an important issue concerning coordinate frames for representing the feature positions, which is covered in Section 5.7.1 We then describe the different feature detection modules. For each feature type, we explain the use of the lower level feature module, which can be used to build new feature detectors, followed by the built-in feature detector.

5.7.1 Image feature coordinate frames

Feature detectors are often applied to a rectangular sub-region of an image, and may be applied to a down-sampled version of the image for greater speed. The most natural coordinate frame to represent the coordinates of features is then the local coordinate frame of the feature map. On the other hand, when using the features for higher level computations such as computing homographies or structure from motion, it is most effecient to use the coordinate frame of the original image to represent the features, so that features detected in different regions can be easily combined in the same coordinate frame. In Gandalf the convention used is that the integer pixel positions are provided in the local coordinate frame of the feature map, while floating point positions are in a user-defined "global" coordinate frame, specified as an affine transformation of the local coordinate frame. The situation is illustrated in Figure 5.3.

Let the position of a feature in the local coordinate frame in homogeneous coordinates be $\mathbf{x}_l = (x_l \ y_l \ 1)^\top$. Then the global coordinates $\mathbf{x}_g = (x_g \ y_g \ 1)^\top$ in global coordinates are related to $\mathbf{x}_l$ as

$$\mathbf{x}_l = A\mathbf{x}_g \quad \text{or} \quad \begin{pmatrix} x_l \\ y_l \\ 1 \end{pmatrix} = \begin{pmatrix} A_{xx} & A_{xy} & A_{xz} \\ A_{yx} & A_{yy} & A_{yz} \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_g \\ y_g \\ 1 \end{pmatrix}$$

where A is an 2D affine homography matrix. Normally A will represent a simple offset, with perhaps a scaling of coordinates, but this representation allows for more general coordinate transformations. The matrix is passed in by the user program to the feature detection algorithms, as is explained below.

5.7.2 Edge detection

```
#include <gandalf/vision/edge_feature.h>
```

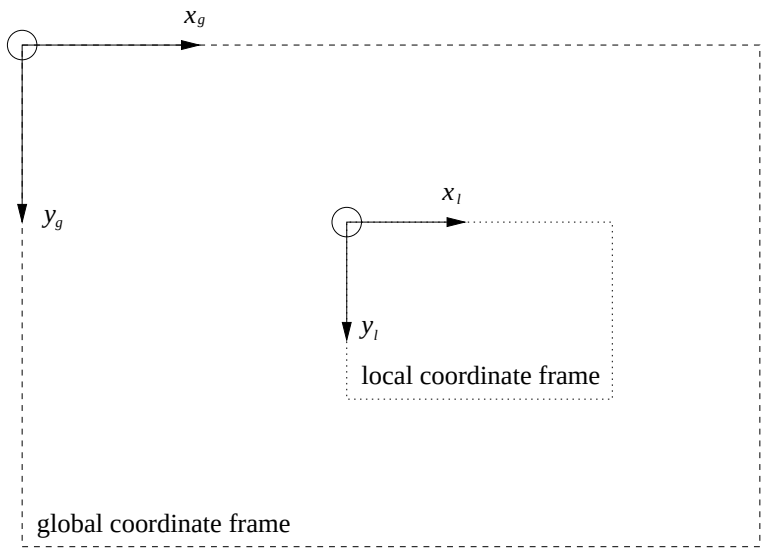

Figure 5.3: Illustration of the local and global coordinate frames for feature detection. The features are detected in the smaller rectangular region described by the local coordinate frame, while for many purposes it is more convenient to also represent the feature positions in a user-defined "global" coordinate frame, which is normally that of the original image.

An *edge map* is a collection of edge points (or edgels) stored in an edge map structure. The edge structure is

```
/* Definition of basic 2D edge feature structure */
typedef struct Gan_EdgeFeature
{
   unsigned short r, c; /* row/column coordinates in coordinate frame of 2D
                           feature array */
   Gan_Vector2_f p; /* potentially sub-pixel coordinates of edge feature in
                       coordinate frame defined by edge map */
   Gan_Vector2_f pu; /* coordinates of feature with any non-linear image
                        distortion removed */
   float strength; /* edge feature strength/contrast value */
   float angle; /* orientation of edge, where applicable. The angle is
                   measured clockwise from the positive x axis, and should
                   be in the range [-pi,pi]. The angle should point in the
                   direction of higher image intensity, or a suitably
                   analagous direction. */
   float cov; /* covariance of feature edge in direction given by the
                 orientation field (angle) */

   /* fields for user program to define */
   short status;
   short index;

   /* next and previous features in list for when edges are stored in a
      list */
   struct Gan_EdgeFeature *next, *prev;
} Gan_EdgeFeature;
```

The `r, c` fields are the integer local coordinates of the edge feature. `p` and `pu` are coordinates in the user-defined coordinate frame. Note that here and elsewhere in the feature detection structures we employ single precision

floating point in order to save memory and computation. The edge structures are designed to be placed into doubly linked *strings* of edges. The edge string is defined as

```
/* Structure defining a connected string of edge features
 */
typedef struct Gan_EdgeString
{
   Gan_EdgeFeature *first, *last;
   unsigned length;
} Gan_EdgeString;
```

The sense of the direction of the edge string is such that as you traverse the string from the `first` edge to the `last` edge, the brighter region is on the left. So if you are walking along the string from `first` to `last` and stick your left arm out sideways, it will point approximately in the edge direction (the `angle` field of a `Gan_EdgeFeature`). New edge detection algorithms should be written to conform to this convention, since the string direction is relevant to other procedures, such as finding line segments given an edge map as input.

The edgels and strings are built into an edge map structure as follows:

```
/* Definition of 2D edge feature map structure */
typedef struct Gan_EdgeFeatureMap
{
   unsigned nedges;       /* number of edge features stored */
   Gan_EdgeFeature *edge; /* array of edge features */
   unsigned max_nedges;   /* allocated limit on number of edge features */

   unsigned nstrings;      /* number of connected edge strings stored */
   Gan_EdgeString *string; /* array of connected strings of edges */
   unsigned max_nstrings;  /* allocated limit on number of strings */

   /* dimensions of image region in which edge features have been computed */
   unsigned height, width;

   /* whether the following A, Ai fields are set */
   Gan_Bool A_set;

   /* transformation between region coordinates (0..width) and (0..height)
      and edge coordinates, and its inverse */
   Gan_Matrix23_f A, Ai;

   /* calibration structure defining camera used for non-linear distortion
      correction */
   Gan_Camera_f camera;

   /* local blocked feature index map */
   Gan_LocalFeatureMap local_fmap;

   /* whether this structure was dynamically allocated */
   Gan_Bool alloc;
} Gan_EdgeFeatureMap;
```

The fields are fairly self-explanatory. The A transformation matrix between local and user-defined global coordinates is defined by the `A` field. If it is not set (`A_set` having value `GAN_FALSE`) then the two coordinate systems are identical.

The most important function in this module sets up the edge feature map structure, a necessary precursor to filling it with edges. Here is in an example call to the routine.

```
Gan_EdgeFeatureMap EdgeMap;

/* initialise edge map */
gan_edge_feature_map_form ( &EdgeMap,
                            10000,   /* initial limit on number of edges */
                              500 ); /* initial limit on number of edge strings */
```

If the initially allocated number of edges or strings is exceeded, `gan_realloc_array()` is used to reallocate the array(s), so if you have no idea what reasonable initial limits should be, you can pass zero for either or both.

The edge detection algorithm will then add edges and strings to the edge map, using the functions `gan_edge_feature_add()` and `gan_edge_feature_string_add()` defined in the `edge_feature.[ch]` module. To free the edge map afterwards, call

```
/* free edge map */
gan_edge_feature_map_free ( &EdgeMap );
```

The other low-level edge routines defined in the `edge_feature.[ch]` module are relevant only if you are developing your own edge detector; examples of their use can be found in the Canny edge detector code.

5.7.3 Displaying an edge map

```
#include <gandalf/vision/edge_disp.h>
```

There are functions to display both whole edge maps and individual edges, the latter being used, for instance, to highlight edges in a different colour. The functions invoke OpenGL routines, and the OpenGL display window must be set up beforehand. Also you will need to set up some colours as single precision floating point RGB pixel structures. For instance the following will create primary colours for you:

```
Gan_RGBPixel_f blue = {0.0F, 0.0F, 1.0F}, green = {0.0F, 1.0F, 0.0F},
               yellow = {1.0F, 1.0F, 0.0F}, red = {1.0F, 0.0F, 0.0F};
```

Once this is done, an example call to display an edge map is

```
/* display a whole edge map using OpenGL */
gan_edge_feature_map_display ( &EdgeMap, 0.0F /* displayed size of each edge */,
                               NULL, /* affine transformation of coordinates */
                               &red,     /* colour of below-threshold edges */
                               &green,   /* colour of edge strings */
                               &blue,    /* colour of first edge in string */
                               &yellow,  /* colour of last edge in string */
                               &green ); /* colour of bounding box */
```

The second argument is the size of the square box used to display each edge point. If it is passed as zero, as is the case here, a single point is drawn on the image. The third argument is an affine transformation of coordinates that allows additional freedom in positioning and scaling the edge map on the display window.`vision_test.c` has some example code using this routine.

To highlight a single edgel, use instead

```
Gan_EdgeFeature *pEdge;

/* ... set pEdge to point to a Gan_EdgeFeature structure ... */

/* display a single edgel using OpenGL */
gan_edge_feature_display ( pEdge, 0.0F /* displayed size of each edge */,
                           NULL, /* affine transformation of coordinates */
                           &yellow ); /* colour to highlight edgel */
```

Note that the `NULL` passed for the affine coordinate transformation indicates an identity transformation.

5.7.4 The Canny edge detector

```
#include <gandalf/vision/canny_edge.h>
```

The Canny edge detector is described in [3]. It involves three stages:

1. **Directional gradients** are computed by smoothing the image and numerically differentiating the image to compute the x and y gradients.
2. **Non-maximum suppression** finds peaks in the image gradient.
3. **Hysteresis thresholding** locates edge strings.

Here is a code fragment illustrating the use of the Canny edge detector. More example code can be found in the `vision_test.c` test program.

```
Gan_Image *pImage; /* declare image from which edges will be computed */
Gan_Mask1D *pFilter; /* convolution mask */
Gan_EdgeFeatureMap EdgeMap; /* declare edge map */

/* ... fill image ... */

/* initialise edge map */
gan_edge_feature_map_form ( &EdgeMap,
                            10000,   /* initial limit on number of edges */
                              500 ); /* initial limit on number of edge strings */

/* create convolution mask */
pFilter = gan_gauss_mask_new ( GAN_FLOAT, 1.0, 9, 1.0, NULL );

/* apply Canny edge detector */
gan_canny_edge_q ( pImage, /* input image */
                   NULL, /* or binary mask of pixels to be processed */
                   pFilter, pFilter, /* image smoothing filters */
                   0.008, /* lower edge strength threshold */
                   0.024, /* upper edge strength threshold */
                   10, /* threshold on string length */
                   NULL, /* or affine coordinate transformation */
                   NULL, /* or pointer to camera structure defining
                            distortion model */
                   NULL, /* or parameters of local feature map */
                   &EdgeMap ); /* result edge map */
```

```
/* free convolution mask and edge map */
gan_mask1D_free ( pFilter );
gan_edge_feature_map_free ( &EdgeMap );
```

5.7.5 Corner detection

```
#include <gandalf/vision/corner_feature.h>
```

An *corner map* is a collection of "corner" points stored in a corner map structure. The corner structure is

```
/* Definition of basic 2D corner feature structure */
typedef struct Gan_CornerFeature
{
   unsigned short r, c; /* row/column coordinates in coordinate frame of 2D
                           feature array */
   Gan_Vector2_f p; /* potentially sub-pixel coordinates of corner feature in
                       coordinate frame defined by corner map */
   Gan_Vector2_f pu; /* coordinates of feature with any non-linear image
                        distortion removed */
   float strength; /* corner feature strength/contrast value */
   Gan_SquMatrix22_f N, Ni; /* covariance and inverse covariance for feature
                               point position */

   /* fields for user program to define */
   short status;
   short index;
} Gan_CornerFeature;
```

The `r, c` fields are the integer local coordinates of the corner feature. `p` and `pu` are coordinates in the user-defined coordinate frame.

The corners are stored in the corner map structure as follows:

```
/* Definition of 2D corner feature map structure */
typedef struct Gan_CornerFeatureMap
{
   unsigned ncorners;         /* number of corner features stored */
   Gan_CornerFeature *corner; /* array of corner features */
   unsigned max_ncorners;     /* allocated limit on number of corner features*/

   /* dimensions of image region in which corner features have been computed */
   unsigned height, width;

   /* whether the following A, Ai fields are set */
   Gan_Bool A_set;

   /* transformation between region coordinates (0..width) and (0..height)
      and corner coordinates, and its inverse */
   Gan_Matrix23_f A, Ai;

   /* calibration structure defining camera used for non-linear distortion
      correction */
   Gan_Camera_f camera;
```

```
    /* local blocked feature index map */
    Gan_LocalFeatureMap local_fmap;

    /* whether this structure was dynamically allocated */
    Gan_Bool alloc;
} Gan_CornerFeatureMap;
```

To create a corner map with an initially allocated number of corners, use the following routine:

```
Gan_CornerFeatureMap CornerMap;

/* initialise corner map */
gan_corner_feature_map_form ( &CornerMap,
                              10000 ); /* initial limit on number of corners */
```

If the initially allocated number of corners is exceeded, `gan_realloc_array()` is used to reallocate the array, so if you have no idea what reasonable initial limit should be, you can pass zero.

The corner detection algorithm will then add corners to the corner map, using the functions `gan_corner_feature_add()` defined in the `corner_feature.[ch]` module. To free the corner map afterwards, call

```
/* free corner map */
gan_corner_feature_map_free ( &CornerMap );
```

The other low-level corner routines defined in the `corner_feature.[ch]` module are relevant only if you are developing your own corner detector; examples of their use can be found in the Harris corner detector code.

5.7.6 Displaying a corner map

```
#include <gandalf/vision/corner_disp.h>
```

There are functions to display both whole corner maps and individual corners, the latter being used, for instance, to highlight corners in a different colour. The functions invoke OpenGL routines, and the OpenGL display window must be set up beforehand. Also you will need to set up some colours as single precision floating point RGB pixel structures. We can use the colour structures set up for the edge detector in Section 5.7.3. Then an example call to display an corner map is

```
/* display a whole corner map using OpenGL */
gan_corner_feature_map_display ( &CornerMap, 0.0F /* displayed size of a corner */,
                                 NULL, /* affine transformation of coordinates */
                                 &red, /* corner colour */
                                 &green ); /* colour of bounding box */
```

The second argument is the size of the square box used to display each corner point. If it is passed as zero, as is the case here, a single point is drawn on the image. The third argument is an affine transformation of coordinates that allows additional freedom in positioning and scaling the corner map on the display window.`vision_test.c` has some example code using this routine.

To highlight a single corner, use instead

```
Gan_CornerFeature *pCorner;

/* ... set pCorner to point to a Gan_CornerFeature structure ... */
```

```
/* display a single corner using OpenGL */
gan_corner_feature_display ( pCorner, 0.0F /* displayed size of each corner */,
                             NULL, /* affine transformation of coordinates */
                             &yellow ); /* colour to highlight corner */
```

Note that the `NULL` passed for the affine coordinate transformation indicates an identity transformation.

5.7.7 The Harris corner detector

```
#include <gandalf/vision/harris_corner.h>
```

The Harris corner detector [6] computes the locally averaged moment matrix computed from the image gradients, and then combines the eigenvalues of the moment matrix to compute a corner "strength", of which maximum values indicate the corner positions. Here is an example code fragment using the Harris corner detector.

```
Gan_Image *pImage; /* declare image from which corners will be computed */
Gan_Mask1D *pFilter; /* convolution mask */
Gan_CornerFeatureMap CornerMap; /* declare corner map */

/* ... fill image ... */

/* initialise corner map */
gan_corner_feature_map_form ( &CornerMap,
                              1000 ); /* initial limit on number of corners */

/* create convolution mask */
pFilter = gan_gauss_mask_new ( GAN_FLOAT, 1.0, 9, 1.0, NULL );

/* apply Harris corner detector */
gan_harris_corner_q ( pImage, /* input image */
                      NULL, /* or binary mask of pixels to be processed */
                      NULL, NULL, /* or image pre-smoothing masks */
                      pFilter, pFilter, /* gradient smoothing */
                      0.04, /* kappa used in computing corner strength */
                      0.04, /* corner strength threshold */
                      NULL, /* or affine coordinate transformation */
                      0, /* status value to assign to each corner */
                      NULL, /* or pointer to camera structure defining
                               distortion model */
                      NULL, /* or parameters of local feature map */
                      &CornerMap ); /* result corner map */

/* free convolution mask and corner map */
gan_mask1D_free ( pFilter );
gan_corner_feature_map_free ( &CornerMap );
```

5.7.8 Line segment detection

```
#include <gandalf/vision/line_feature.h>
```

A *line map* is a collection of line segments stored in a line map structure. The line structure is

```
/* Definition of basic 2D line feature structure */
typedef struct Gan_LineFeature
{
   unsigned r1, c1, r2, c2; /* row/column coordinates in coordinate frame of 2D
                               feature array */
   Gan_Vector2_f p1, p2; /* endpoints of line */
   double strength; /* line feature strength/contrast value */
   Gan_Vector3_f l; /* line parameters a*x + b*y + c = 0 scaled so that
                       a^2 + b^2 = 1 */
   Gan_SquMatrix22_f N, Ni; /* covariance and inverse covariance for canonical
                               line parameters a/b in y=ax+b, with x/y system
                               centred on midpoint of line (p1+p2)/2 with
                               positive x-axis along the line towards p2
                               endpoint, and positive y-axis 90 degrees
                               anticlockwise from x-axis */

   /* fields for user program to define */
   int status;
   int index;

   /* array of points attached to this line */
   Gan_Vector2_f *point;
   unsigned npoints;
} Gan_LineFeature;
```

The `r1, c1, r2, c2` fields are the integer local coordinates of the line segment endpoints. `p1` and `p2` are coordinates in the user-defined coordinate frame.

The lines are stored in the line map structure as follows:

```
/* Definition of 2D line feature map structure */
typedef struct Gan_LineFeatureMap
{
   unsigned nlines;       /* number of line features stored */
   Gan_LineFeature *line; /* array of line features */
   unsigned max_nlines;   /* allocated limit on number of line features */

   /* dimensions of image region in which line features have been computed */
   unsigned height, width;

   /* whether the following A, Ai fields are set */
   Gan_Bool A_set;

   /* transformation between region coordinates (0..width) and (0..height)
      and line coordinates, and its inverse */
   Gan_Matrix23_f A, Ai;

   /* local blocked feature index map */
   Gan_LocalFeatureMap local_fmap;

   /* points making up line (optional) */
   Gan_Vector2_f *point;  /* array of points used to fit the lines to:
                             may be NULL */
   unsigned npoints;     /* current number of points */
   unsigned max_npoints; /* maximum (allocated) number of points */
```

```
   /* whether this structure was dynamically allocated */
   Gan_Bool alloc;
} Gan_LineFeatureMap;
```

To create a line map with an initially allocated number of lines, use the following routine:

```
Gan_LineFeatureMap LineMap;

/* initialise line map */
gan_line_feature_map_form ( &LineMap,
                            10000 ); /* initial limit on number of lines */
```

If the initially allocated number of lines is exceeded, `gan_realloc_array()` is used to reallocate the array, so if you have no idea what reasonable initial limit should be, you can pass zero.

The line detection algorithm will then add lines to the line map, using the functions `gan_line_feature_add()` defined in the `line_feature.[ch]` module. To free the line map afterwards, call

```
/* free line map */
gan_line_feature_map_free ( &LineMap );
```

The other low-level line routines defined in the `line_feature.[ch]` module are relevant only if you are developing your own line detector; examples of their use can be found in the Harris line detector code.

5.7.9 Displaying a line map

```
#include <gandalf/vision/line_disp.h>
```

There are functions to display both whole line maps and individual lines, the latter being used, for instance, to highlight lines in a different colour. The functions invoke OpenGL routines, and the OpenGL display window must be set up beforehand. Also you will need to set up some colours as single precision floating point RGB pixel structures. We can use the colour structures set up for the edge detector in Section 5.7.3. Then an example call to display an line map is

```
/* display a whole line map using OpenGL */
gan_line_feature_map_display ( &LineMap,
                               NULL, /* affine transformation of coordinates */
                               &green, /* line colour */
                               &blue, /* colour of start of line */
                               &yellow, /* colour of end of line */
                               &blue, /* colour to display points */
                               &green ); /* colour of bounding box */
```

The second argument is an affine transformation of coordinates that allows additional freedom in positioning and scaling the line map on the display window.`vision_test.c` has some example code using this routine.

To highlight a single line, use instead

```
Gan_LineFeature *pLine;

/* ... set pLine to point to a Gan_LineFeature structure ... */

/* display a single line using OpenGL */
```

```
gan_line_feature_display ( pLine,
                           NULL, /* affine transformation of coordinates */
                           &yellow, /* colour to highlight line */
                           &yellow, /* colour of start of line */
                           &yellow, /* colour of end of line */
                           &blue ); /* colour to display points */
```

Note that the `NULL` passed for the affine coordinate transformation indicates an identity transformation.

5.7.10 The Gandalf line detector

```
#include <gandalf/vision/orthog_line.h>
```

The Gandalf line detector takes an edge map as input, and automatically segments each edge string into line segments. Here is an example code fragment using the Harris corner detector.

```
Gan_EdgeFeatureMap EdgeMap; /* declare edge map */
Gan_LineFeatureMap LineMap; /* declare line map */

/* ... build edge map ... */

/* initialise line map */
gan_line_feature_map_form ( &LineMap,
                            1000 ); /* initial limit on number of lines */

/* compute lines from the edge map */
gan_orthog_line_q ( &EdgeMap,
                    10, /* minimum length of line segment */
                    2, /* cut size, i.e. amount to shrink line segment */
                    1.0, /* RMS error threshold on fitted line */
                    NULL, /* or pointer to local feature map */
                    GAN_TRUE, /* whether to copy points */
                    &LineMap );

/* free line map */
gan_line_feature_map_free ( &LineMap );
```

5.8 Representing 3D rotations

```
#include <gandalf/vision/rotate3D.h>
```

Gandalf has an extensive set of routines for handling 3D rotations. Four different representations are available, defined by the following enumerated type:

```
/* types of rotation handled by Gandalf */
typedef enum { GAN_ROT3D_QUATERNION, GAN_ROT3D_EXPONENTIAL,
               GAN_ROT3D_ANGLE_AXIS, GAN_ROT3D_MATRIX }
   Gan_Rot3D_Type;
```

These representations are now described in turn.

The **quaternion** representation uses the following structure:

```
/* quaternion structure */
typedef struct Gan_Quaternion
{
   double q0, q1, q2, q3;
} Gan_Quaternion;
```

The relationship between a quaternion $\mathbf{q} = (q_0\ q_1\ q_2\ q_3)^\top$ and the equivalent rotation matrix R is

$$R = \begin{pmatrix} q_0q_0 + q_1q_1 - q_2q_2 - q_3q_3 & 2(q_1q_2 - q_0q_3) & 2(q_1q_3 + q_0q_2) \\ 2(q_2q_1 + q_0q_3) & q_0q_0 - q_1q_1 + q_2q_2 - q_3q_3 & 2(q_2q_3 - q_0q_1) \\ 2(q_3q_1 - q_0q_2) & 2(q_3q_2 + q_0q_1) & q_0q_0 - q_1q_1 - q_2q_2 + q_3q_3 \end{pmatrix}.$$

Here the quaternion is assumed to have been scaled to unit length, i.e. $|\mathbf{q}|^2 = 1$.

The **exponential** representation uses a 3-vector $\mathbf{r}$ related to the equivalent rotation matrix R according to

$$\begin{aligned} R &= e^{[\mathbf{r}]_\times} \\ &= I + \frac{\sin\theta}{\theta}[\mathbf{r}]_\times + \frac{(1-\cos\theta)}{\theta^2}(\mathbf{r}\mathbf{r}^\top - \|\mathbf{r}\|^2 I) \end{aligned}$$

where $\theta = \|\mathbf{r}\|$ is the rotation angle, and the cross-product matrix $[\mathbf{r}]_\times$ is explained in Section 5.2.

The **angle/axis** representation $a/\mathbf{a}$ is strongly related to a quaternion, according to the formula

$$\mathbf{q} = \begin{pmatrix} \cos(a/2) \\ a_x \sin(a/2) \\ a_y \sin(a/2) \\ a_z \sin(a/2) \end{pmatrix}$$

where the rotation axis $\mathbf{a} = (a_x\ a_y\ a_z)^\top$ is assumed to be scaled to $|\mathbf{a}|^2 = 1$. The rotation angle a is measured in radians.

The **matrix** representation uses a 3×3 matrix R as above to represent a rotation

This variety of representations is necessary because of the corresponding variety of operations that can be applied. For instance, quaternions are perhaps the most natural representation, and are a good representation when combining rotations, because quaternion product has a simply linear formula. Given two quaternions $\mathbf{q}_1 = (q_{10}\ q_{11}\ q_{12}\ q_{13})$ and $\mathbf{q}_2 = (q_{20}\ q_{21}\ q_{22}\ q_{23})$, with corresponding rotation matrices R_1, R_2, the product

$$\mathbf{q}_3 = \begin{pmatrix} q_{10}q_{20} - q_{11}q_{21} - q_{12}q_{22} - q_{13}q_{23} \\ q_{10}q_{21} + q_{11}q_{20} + q_{12}q_{23} - q_{13}q_{22} \\ q_{10}q_{22} + q_{12}q_{20} + q_{13}q_{21} - q_{11}q_{23} \\ q_{10}q_{23} + q_{13}q_{20} + q_{11}q_{22} - q_{12}q_{21} \end{pmatrix}$$

is equivalent to the rotation matrix product

$$R_3 = R_1R_2$$

You should always use quaternions for combining rotations in this way, because if you use matrices the rounding error will accumulate over repeated matrix multiplications, and cause the matrix to become non-orthogonal. With quaternions the scale will drift slightly, but it is much easier to fix the scale of a quaternion than to correct the matrix representation. The angle/axis form is mainly useful as an intuitive way of defining rotations. It has the problem of having no unique representation of zero rotations, since in this case the axis is arbitrary. The exponential representation is unique in being a `minimal` representation with its three parameters. It is mainly useful however only for `small` rotations, since it has severe singularity problems for large rotations. For optimisation purposes, you can use the exponential form of rotation to represent a small change in the estimated rotation, with a quaternion used to represent the latest rotation estimate. Similar *local coordinate* forms of optimisation have been employed in 3D reconstruction by Taylor & Kriegman [14]. A local rotation representation has also been developed independently by Pennec & Thirion [11]

First we describe some routines specific to the quaternion representation. Then we provide details of how the Gandalf structure `Gan_Rot3D` is used to create and manipulate rotations of any of the currently supported representations.

5.8.1 Quaternion routines

There is a special set of functions and macros to handle quaternions. A `Gan_Quaternion` is basically the same object as a `Gan_Vector4`, and a lot of the routines are macro calls to the equivalent 4-vector routines. Here is a code fragment illustrating the use of quaternions in Gandalf.

```
Gan_Quaternion q1, q2, q3; /* declare three quaternions */

/* fill q1 the "quick" way and scale it to unit length */
gan_quat_fill_q ( &q1, 0.5, -0.2, 0.3, 0.3 );
gan_quat_unit_i ( &q1 );

/* fill q2 the "slow" way and scale it to unit length */
q2 = gan_quat_fill_s ( -0.4, 0.7, 0.1, 0.9 );
q2 = gan_quat_unit_s ( &q2 );

/* add two quaternions */
gan_quat_add_q ( &q1, &q2, &q3 );

/* subtract two quaternions */
gan_quat_sub_q ( &q1, &q2, &q3 );

/* multiply a quaternion by a scalar */
gan_quat_scale_q ( &q1, 2.0, &q3 ); /* q3 = 2*q1, OR */
q3 = gan_quat_scale_s ( &q1, 2.0 ); /* q3 = 2*q1, OR */
gan_quat_scale_i ( &q1, 2.0 ); /* replace q1 = 2*q1 in-place */

/* divide a quaternion by a scalar */
gan_quat_divide_q ( &q1, 2.0, &q3 ); /* q3 = q1/2, OR */
q3 = gan_quat_divide_s ( &q1, 2.0 ); /* q3 = q1/2, OR */
gan_quat_divide_i ( &q1, 2.0 ); /* replace q1 = q1/2 in-place */

/* print squared length of quaternion */
printf ( "quaternion squared length |q|^2=%f\n",
         gan_quat_sqrlen_q(&q3) ); /* macro version, OR */
printf ( "quaternion squared length |q|^2=%f\n",
         gan_quat_sqrlen_s(&q3) ); /* function version */
```

Error detection: The routines `gan_quat_divide_[qi]()` and `gan_quat_unit_[qi]()` return `NULL` upon division by zero error, invoking the Gandalf error handler, whereas the equivalent `..._s()` routines abort the program on error.

5.8.2 General rotation routines

The different rotation representations are combined into the following structure:

```
/* structure representing 3D rotation */
typedef struct Gan_Rot3D
{
   Gan_Rot3D_Type type; /* representation used */
   union
   {
      Gan_Quaternion                              q; /* quaternion form */
```

```
      Gan_Vector3                                  r; /* exponential form */
      struct { Gan_Vector3 axis; double angle; } aa; /* angle/axis form */
      Gan_Matrix33                                 R; /* matrix form */
   } d;
} Gan_Rot3D;
```

This structure can then be used to implement unary and binary operations involving rotations. Firstly there is a set of functions to build a rotation structure. For quaternions the function is

```
Gan_Rot3D Rot; /* declare rotation structure */
double q0, q1, q2, q3; /* declare quaternion elements */

/* ... set q0, q1, q2 and q3 to quaternion coordinates, and scale to
       unit length if desired ... */

/* build rotation structure from quaternion */
gan_rot3D_build_quaternion ( &Rot, q0, q1, q2, q3 );
```

For the exponential representation use

```
Gan_Rot3D Rot; /* declare rotation structure */
double rx, ry, rz; /* declare exponential rotation vector elements */

/* ... set rx, ry & rz ... */

/* build rotation structure from exponential rotation vector */
gan_rot3D_build_exponential ( &Rot, rx, ry, rz );
```

For the angle/axis representation use

```
Gan_Rot3D Rot; /* declare rotation structure */
double angle, ax, ay, az; /* declare angle and axis elements */

/* ... set angle, ax, ay & az ... */

/* build rotation structure from rotation angle and axis */
gan_rot3D_build_angle_axis ( &Rot, angle, ax, ay, az );
```

Finally for the matrix representation we have

```
Gan_Rot3D Rot; /* declare rotation structure */
double Rxx, Rxy, Rxz, Ryx, Ryy, Ryz, Rzx, Rzy, Rzz; /* declare matrix elements */

/* ... set matrix elements Rxx, Rxy etc. ... */

/* build rotation structure from rotation angle and axis */
gan_rot3D_build_matrix ( &Rot, Rxx, Rxy, Rxz,
                               Ryx, Ryy, Ryz,
                               Rzx, Rzy, Rzz );
```

Another way to build a rotation structure is from a fixed-size Gandalf vector or matrix. These routines give the added flexibility of allowing conversion to another rotation representation. So for instance

```
Gan_Rot3D Rot; /* declare rotation structure */
```

```
Gan_Quaternion q; /* declare quaternion */

/* ... fill quaternion q using e.g. gan_quat_fill_q() ... */

/* fill rotation structure with rotation matrix equivalent to
   quaternion q */
gan_rot3D_from_quaternion_q ( &Rot, &q, GAN_ROT3D_MATRIX ); /* OR */
Rot = gan_rot3D_from_quaternion_s ( &q, GAN_ROT3D_MATRIX );
```

builds the rotation structure `Rot` containing the rotation matrix equivalent to the quaternion `q`. This routine and others in the same family do rescale and adjust as necessary to an exact rotation. The other routines are:

```
Gan_Rot3D Rot; /* declare rotation structure */
Gan_Vector3 r; /* declare exponential rotation vector */
double angle; Gan_Vector3 axis; /* declare angle an axis */
Gan_Matrix33 R; /* declare rotation matrix */

/* ... fill vector r using e.g. gan_vec3_fill_q() ... */

/* fill rotation structure with quaternion equivalent to
   exponential rotation vector r */
gan_rot3D_from_exponential_q ( &Rot, &r, GAN_ROT3D_QUATERNION ); /* OR */
Rot = gan_rot3D_from_exponential_s ( &r, GAN_ROT3D_QUATERNION );

/* ... fill angle and axis ... */

/* fill rotation structure with rotation matrix equivalent to
   angle and axis */
gan_rot3D_from_angle_axis_q ( &Rot, angle, &axis, GAN_ROT3D_QUATERNION ); /* OR */
Rot = gan_rot3D_from_angle_axis_s ( angle, &axis, GAN_ROT3D_QUATERNION );

/* ... fill rotation matrix R ... */

/* fill rotation structure with angle and axis equivalent to
   rotation matrix */
gan_rot3D_from_matrix_q ( &Rot, &R, GAN_ROT3D_QUATERNION ); /* OR */
Rot = gan_rot3D_from_matrix_s ( &R, GAN_ROT3D_QUATERNION );
```

Next are a pair of routines to set a rotation to zero:

```
Gan_Rot3D Rot; /* declare rotation structure */

/* set a rotation structure to be a quaternion representation and
   set it to a zero rotation */
gan_rot3D_ident_q ( &Rot, GAN_ROT3D_QUATERNION ); /* OR */
Rot = gan_rot3D_ident_s ( GAN_ROT3D_QUATERNION );
```

To apply a rotation to a 3D point use one of the routines

```
Gan_Vector3 v3X, V3Xp; /* declare 3D points X & Xp */

/* ... fill 3D point X with values ... */

/* apply rotation such as Xp = R*X */
```

```
gan_rot3D_apply_v3_q ( &Rot, &v3X, &v3Xp ); /* OR */
v3Xp = gan_rot3D_apply_v3_s ( &Rot, &v3X );
```

To combine two rotations use

```
Gan_Rot3D Rot1, Rot2, Rot3; /* declare rotations R1, R2 & R3 */

/* ... fill R1 and R2 with rotation parameters of the same type ... */

/* combine two rotations into a third: for matrices R3 = R1*R2 */
gan_rot3D_mult_q ( &Rot1, &Rot2, &Rot3 ); /* OR */
Rot3 = gan_rot3D_mult_s ( &Rot1, &Rot2 );
```

The second rotation structure may also be implicitly inverted, yielding

```
/* combine two rotations into a third: for matrices R3 = R1*R2^-1 */
gan_rot3D_multI_q ( &Rot1, &Rot2, &Rot3 ); /* OR */
Rot3 = gan_rot3D_multI_s ( &Rot1, &Rot2 );
```

There is also a set of arithmetical routines. For binary arithmetical operations, both structures must have the same representation, and the operation is a pure parameter addition/subtraction etc., without rescaling or otherwise adjusting the rotation parameters to conform to an actual rotation. This is often required when implementing optimisation involving rotation parameters, for instance computing derivatives numerically. Firstly there are routines for multiplying or dividing rotation parameters by a scalar:

```
Gan_Rot3D Rot1, Rot2; /* declare rotations R1 & R2 */

/* ... fill R1 with rotation parameters ... */

/* multiply the rotation parameters R1 by 3, writing them into R2 */
gan_rot3D_scale_q ( &Rot1, 3.0, &Rot2 ); /* R2 = 3*R1, OR */
R2 = gan_rot3D_scale_s ( &Rot1, 3.0 ); /* R2 = 3*R1, OR */
gan_rot3D_scale_i ( &Rot1, 3.0 ); /* replace R1 = 3*R1 */

/* divide the rotation parameters R1 by 3, writing them into R2 */
gan_rot3D_divide_q ( &Rot1, 3.0, &Rot2 ); /* R2 = R1/3, OR */
R2 = gan_rot3D_divide_s ( &Rot1, 3.0 ); /* R2 = R1/3, OR */
gan_rot3D_divide_i ( &Rot1, 3.0 ); /* replace R1 = R1/3 */
```

Next a set of routines each for adding and subtracting rotation parameters:

```
Gan_Rot3D Rot1, Rot2, Rot3; /* declare rotations R1, R2 & R3 */

/* ... fill R1 and R2 with rotation parameters of the same type ... */

/* add the rotation parameters R1 and R2 */
gan_rot3D_add_q ( &Rot1, &Rot2, &Rot3 ); /* R3 = R1 + R2, OR */
Rot3 = gan_rot3D_add_s ( &Rot1, &Rot2 ); /* R3 = R1 + R2, OR */
gan_rot3D_increment ( &Rot1, &Rot2 ); /* replace R1 = R1 + R2 in-place, OR */
gan_rot3D_add_i1 ( &Rot1, &Rot2 ); /* replace R1 = R1 + R2 in-place, OR */
gan_rot3D_add_i2 ( &Rot1, &Rot2 ); /* replace R2 = R1 + R2 in-place */

/* subtract the rotation parameters R1 and R2 */
gan_rot3D_sub_q ( &Rot1, &Rot2, &Rot3 ); /* R3 = R1 - R2, OR */
```

```
Rot3 = gan_rot3D_sub_s ( &Rot1, &Rot2 ); /* R3 = R1 - R2, OR */
gan_rot3D_decrement ( &Rot1, &Rot2 ); /* replace R1 = R1 - R2 in-place, OR */
gan_rot3D_sub_i1 ( &Rot1, &Rot2 ); /* replace R1 = R1 - R2 in-place, OR */
gan_rot3D_sub_i2 ( &Rot1, &Rot2 ); /* replace R2 = R1 - R2 in-place */
```

There are a couple of routines to convert a rotation structure from one representation to another:

```
Gan_Rot3D Rot1, Rot2; /* declare rotations R1 & R2 */

/* ... fill R1 with rotation parameters ... */

/* convert rotation R1 to matrix representation in R2 */
gan_rot3D_convert_q ( &Rot1, GAN_ROT3D_MATRIX, &Rot2 ); /* OR */
Rot2 = gan_rot3D_convert_s ( &Rot1, GAN_ROT3D_MATRIX );
```

Finally a utility routine to correct a 3×3 matrix to the "nearest" orthogonal matrix, using SVD:

```
Gan_Matrix33 m33R; /* declare matrix R */

/* ... set up R as "nearly" a rotation matrix */

/* adjust matrix R to be exactly a rotation matrix */
gan_rot3D_matrix_adjust ( &m33R );
```

For statistical optimisation purposes there is a structure designed to hold covariance information for rotation parameters, currently supporting only quaternion and exponential representations, they being the most likely representations to use for optimising rotation parameters:

```
/* structure representing covariance of 3D rotation */
typedef struct Gan_Rot3D_Cov
{
   Gan_Rot3D_Type type;
   union
   {
      Gan_SquMatrix44 q; /* covariance of quaternion */
      Gan_SquMatrix33 r; /* covariance of exponential rotation vector */
   } data;
} Gan_Rot3D_Cov;
```

Error detection: The `gan_rot3D_build_...()` and all the `gan_rot3D_..._[qi]()`, `gan_rot3D_..._i1()`, `gan_rot3D_..._i2()`, `gan_rot3D_..._increment()` and `gan_rot3D_..._decrement()` routines return a boolean value, and return `GAN_FALSE` on error, invoking the Gandalf error handler. The main error modes are difference of the representations between two rotation structures for the arithmetic and combination routines, and illegal parameter values.

5.9 Representing 3D Euclidean transformations

```
#include <gandalf/vision/euclid3D.h>
```

This module allows you to manipulate 3D Euclidean transformations, to represent for instance camera pose relative to a 3D scene. The basic structure contains a rotation and a translation:

```
/* 3D pose */
typedef struct
{
   Gan_Rot3D   rot;   /* rotation parameters */
   Gan_Vector3 trans; /* translation parameters */
} Gan_Euclid3D;
```

To build a Euclidean transformation structure, you need to decide on a representation for the rotation, and then call one of the following routines:

```
Gan_Euclid3D Euc; /* Euclidean transformation structure */
double TX, TY, TZ; /* translation vector */
double q0, q1, q2, q3; /* quaternion parameters */
double rx, ry, rz; /* exponential rotation vector parameters */
double angle, ax, ay, az; /* angle/axis parameters */
double Rxx, Rxy, Rxz, Ryx, Ryy, Ryz, Rzx, Rzy, Rzz; /* matrix rotation parameters */

/* ... set up translation and rotation parameters ... */

/* build Euclidean transformation structure using quaternion rotation */
gan_euclid3D_build_quaternion ( &Euc, TX, TY, TZ, q0, q1, q2, q3 );

/* build Euclidean transformation structure using exponential rotation */
gan_euclid3D_build_exponential ( &Euc, TX, TY, TZ, rx, ry, rz );

/* build Euclidean transformation structure using angle/axis rotation */
gan_euclid3D_build_angle_axis ( &Euc, TX, TY, TZ, angle, ax, ay, az );

/* build Euclidean transformation structure using matrix rotation */
gan_euclid3D_build_matrix ( &Euc, TX, TY, TZ,
                            Rxx, Rxy, Rxz, Ryx, Ryy, Ryz, Rzx, Rzy, Rzz );
```

There is a pair of routines to set up a null Euclidean transformation (zero translation and rotation):

```
Gan_Euclid3D Euc; /* declare Euclidean transformation structure */

/* set a null Euclidean transformation structure using a quaternion
   representation of rotation */
gan_euclid3D_ident_q ( &Euc, GAN_ROT3D_QUATERNION ); /* OR */
Euc = gan_euclid3D_ident_s ( GAN_ROT3D_QUATERNION );
```

There is also a set of arithmetical routines. For binary arithmetical operations, both structures must have the same rotation representation, and the operation is a pure parameter addition/subtraction etc., without rescaling or otherwise adjusting the translation & rotation parameters to conform to an actual rotation. This is often required when implementing optimisation, for instance computing derivatives numerically. Firstly there are routines for multiplying or dividing transformation parameters by a scalar:

```
Gan_Euclid3D Euc1, Euc2; /* declare Euclidean pose parameters T1,R1 and T2,R2 */

/* ... fill T1,R1 with translation & rotation parameters ... */

/* multiply the T1,R1 parameters by 3, writing them into T2,R2 */
gan_euclid3D_scale_q ( &Euc1, 3.0, &Euc2 ); /* T2 = 3*T1, R2 = 3*R1, OR */
R2 = gan_euclid3D_scale_s ( &Euc1, 3.0 ); /* T2 = 3*T1, R2 = 3*R1, OR */
```

```
gan_euclid3D_scale_i ( &Euc1, 3.0 ); /* replace T1 = 3*T1, R1 = 3*R1 */

/* divide the rotation parameters R1 by 3, writing them into R2 */
gan_euclid3D_divide_q ( &Euc1, 3.0, &Euc2 ); /* T2 = T1/3, R2 = R1/3, OR */
R2 = gan_euclid3D_divide_s ( &Euc1, 3.0 ); /* T2 = T1/3, R2 = R1/3, OR */
gan_euclid3D_divide_i ( &Euc1, 3.0 ); /* replace T1 = T1/3, R1 = R1/3 */
```

Next a set of routines each for adding and subtracting Euclidean transformation parameters:

```
Gan_Euc3D Euc1, Euc2, Euc3; /* declare rotations T1,R1, T2,R2 & T3,R3 */

/* ... fill T1,R1 and T2,R2 with translation & rotation parameters ... */

/* add the translation/rotation parameters T1,R1 and T2,R2 */
gan_euclid3D_add_q ( &Euc1, &Euc2, &Euc3 ); /* T3 = T1 + T2, R3 = R1 + R2 */

/* subtract the rotation parameters R1 and R2 */
gan_euclid3D_sub_q ( &Euc1, &Euc2, &Euc3 ); /* T3 = T1 - T2, R3 = R1 - R2 */
```

For statistical optimisation purposes there is a structure designed to hold covariance information for 3D pose parameters. Writing the rotation parameters as a vector $\mathbf{R}$ (which could be a 4-parameter quaternion vector or a 3-parameter exponential vector, for instance), we can write the covariance as

$$\mathrm{Cov}\begin{pmatrix}\mathbf{R}\\ \mathbf{T}\end{pmatrix} = \begin{pmatrix}\mathrm{Cov}_{\mathbf{RR}} & \mathrm{Cov}^{\top}_{\mathbf{TR}}\\ \mathrm{Cov}_{\mathbf{TR}} & \mathrm{Cov}^{\top}_{\mathbf{TT}}\end{pmatrix}$$

```
/* covariance of 3D pose */
typedef struct
{
   Gan_Rot3D_Cov      Crr; /* covariance of rotation parameters */
   Gan_Euclid3D_TRCov Ctr; /* cross-covariance between translation and rotation */
   Gan_SquMatrix33    Ctt; /* covariance of translation parameters */
} Gan_Euclid3r_Cov;
```

The cross-covariance structure between $\mathbf{T}$ and $\mathbf{R}$ is

```
/* cross-covariance between rotation and translation */
typedef struct Gan_Euclid3D_TRCov
{
   Gan_Rot3D_Type type;
   union
   {
      Gan_Matrix34 q;  /* quaternion representation (4 parameters) */
      Gan_Matrix33 le; /* exponential representation (3 parameters) */
   } d;
} Gan_Euclid3D_TRCov;
```

Error detection: The `gan_euclid3D_build_...()` and all the `gan_euclid3D_..._[qi]()` routines return a boolean value, and return `GAN_FALSE` on error, invoking the Gandalf error handler. The main error modes are difference of the representations between two rotation parts of the structures for the arithmetic and combination routines, and illegal parameter values.

5.10 Levenberg-Marquardt minimisation

```
#include <gandalf/vision/lev_marq.h>
```

The Levenberg-Marquardt algorithm [9, 2] is a general non-linear downhill minimisation algorithm for the case when derivatives of the objective function are known. It dynamically mixes Gauss-Newton and gradient-descent iterations. We shall develop the L-M algorithm for a simple case in our notation, which is derived from Kalman filtering theory [1]. The Gandalf implementation of Levenberg-Marquardt will then be presented. Let the unknown parameters be represented by the vector $\mathbf{x}$, and let noisy measurements of $\mathbf{x}$ be made:

$$\mathbf{z}_{(j)} = \mathbf{h}(j;\mathbf{x}) + \mathbf{w}_{(j)}, \quad j = 1, \ldots, k \tag{5.7}$$

where $\mathbf{h}_{(j)}$ is a measurement function and $\mathbf{w}_{(j)}$ is zero-mean noise with covariance $N_{(j)}$. Since we are describing an iterative minimization algorithm, we shall assume that we have already obtained an estimate $\hat{\mathbf{x}}^-$ of $\mathbf{x}$. Then the maximum likelihood solution for a new estimate $\hat{\mathbf{x}}$ minimizes

$$J(\hat{\mathbf{x}}) = \sum_{j=1}^{k} (\mathbf{z}_{(j)} - \mathbf{h}(j;\hat{\mathbf{x}}))^\top {N_{(j)}}^{-1} (\mathbf{z}_{(j)} - \mathbf{h}(j;\hat{\mathbf{x}})). \tag{5.8}$$

We form a quadratic approximation to $J(.)$ around $\hat{\mathbf{x}}^-$, and minimize this approximation to $J(.)$ to obtain a new estimate $\hat{\mathbf{x}}^+$. In general we can write such a quadratic approximation as

$$J(\mathbf{x}) \approx a - 2\mathbf{a}^\top(\mathbf{x} - \hat{\mathbf{x}}^-) + (\mathbf{x} - \hat{\mathbf{x}}^-)^\top A(\mathbf{x} - \hat{\mathbf{x}}^-)$$

for scalar a, vectors $\mathbf{a}$, $\mathbf{b}$ and matrices A, B. Note that here and in equation (5.8) the signs and factors of two are chosen WLOG to simplify the resulting expressions for the solution. Differentiating, we obtain

$$\frac{\partial J}{\partial \mathbf{x}} = -2\mathbf{a} + 2A(\mathbf{x} - \hat{\mathbf{x}}^-),$$

$$\frac{\partial^2 J}{\partial \mathbf{x}^2} = 2A,$$

At the minimum point $\hat{\mathbf{x}}$ we have $\partial J/\partial \mathbf{x} = \mathbf{0}$ which means that

$$A(\hat{\mathbf{x}}^+ - \hat{\mathbf{x}}^-) = \mathbf{a}. \tag{5.9}$$

Thus we need to obtain $\mathbf{a}$ and A to compute the update. We now consider the form of $J(.)$ in (5.8). Writing the Jacobian of $\mathbf{h}(j, \mathbf{x})$ as

$$H_{(j)} = \frac{\partial \mathbf{h}_{(j)}}{\partial \mathbf{x}},$$

we have

$$\frac{\partial J}{\partial \mathbf{x}} = -2\sum_{j=1}^{k} {H_{(j)}}^\top {N_{(j)}}^{-1} (\mathbf{z}_{(j)} - \mathbf{h}(j;\mathbf{x})), \tag{5.10}$$

$$\begin{aligned} \frac{\partial^2 J}{\partial \mathbf{x}^2} &= 2\sum_{j=1}^{k} {H_{(j)}}^\top {N_{(j)}}^{-1} H_{(j)} - 2\sum_{j=1}^{k} \left(\frac{\partial H_{(j)}}{\partial \mathbf{x}}\right)^\top {N_{(j)}}^{-1} (\mathbf{z}_{(j)} - \mathbf{h}(j;\mathbf{x})) \\ &\approx 2\sum_{j=1}^{k} {H_{(j)}}^\top {N_{(j)}}^{-1} H_{(j)}, \end{aligned} \tag{5.11}$$

In the last formula for $\partial^2 J/\partial \mathbf{x}^2$, the terms involving the second derivatives of $\mathbf{h}_{(j)}(.)$ have been omitted. This is done because these terms are generally much smaller and can in practice be omitted, as well as because the second derivatives are more difficult and complex to compute than the first derivatives.

Now we solve the above equations for $\mathbf{a}$ and A given the values of function $\mathbf{h}_{(j)}$ and the Jacobian $H_{(j)}$ evaluated at the previous estimate $\hat{\mathbf{x}}^-$. We have immediately

$$A = \sum_{j=1}^{k} {H_{(j)}}^\top {N_{(j)}}^{-1} H_{(j)}.$$

We now write the *innovation* vectors $\boldsymbol{\nu}_{(j)}$ as

$$\boldsymbol{\nu}_{(j)} = \mathbf{z}_{(j)} - \mathbf{h}(j; \hat{\mathbf{x}}^-)$$

Then we have

$$\mathbf{a} = \sum_{j=1}^{k} H_{(j)}{}^\top N_{(j)}{}^{-1} \boldsymbol{\nu}_{(j)} \tag{5.12}$$

Combining equations (5.9) and (5.12) we obtain the linear system

$$A(\hat{\mathbf{x}}^+ - \hat{\mathbf{x}}^-) = \mathbf{a} = \sum_{j=1}^{k} H_{(j)}{}^\top N_{(j)}{}^{-1} \boldsymbol{\nu}_{(j)} \tag{5.13}$$

to be solved for the adjustment $\hat{\mathbf{x}}^+ - \hat{\mathbf{x}}^-$. The covariance of the state is

$$P = A^{-1}.$$

The update (5.13) may be repeated, substituting the new $\hat{\mathbf{x}}^+$ as $\hat{\mathbf{x}}^-$, and improving the estimate until convergence is achieved according to some criterion. Levenberg-Marquardt modifies this updating procedure by adding a value λ to the diagonal elements of the linear system matrix before inverting it to obtain the update. λ is reduced if the last iteration gave an improved estimate, i.e. if J was reduced, and increased if J increased, in which case the estimate of $\mathbf{x}$ is reset to the estimate before the last iteration. It is this that allows the algorithm to smoothly switch between Gauss-Newton (small λ) and gradient descent (large λ).

This version is a generalization of Levenberg-Marquardt as it is normally presented (e.g. [12]) in that we incorporate vector measurements $\mathbf{z}_{(j)}$ with covariances $N_{(j)}$, rather than scalar measurements with variances. The full algorithm is as follows:

1. Start with a prior estimate $\hat{\mathbf{x}}^-$ of $\mathbf{x}$. Initialize λ to some starting value, e.g. 0.001.
2. Compute the updated estimate $\hat{\mathbf{x}}^+$ by solving the linear system (5.13) for the adjustment, having first added λ to each diagonal element of A. Note that the Lagrange multiplier diagonal block should remain at zero.
3. Compute the least-squares residuals $J(\hat{\mathbf{x}}^-)$ and $J(\hat{\mathbf{x}}^+)$ from (5.8).
   (a) If $J(\hat{\mathbf{x}}^+) < J(\hat{\mathbf{x}}^-)$, reduce λ by a specified factor (say 10), set $\hat{\mathbf{x}}^-$ to $\hat{\mathbf{x}}^+$, and return to step 2.
   (b) Otherwise, the update failed to reduce the residual, so increase λ by a factor (say 10), forget the updated $\hat{\mathbf{x}}^+$ and return to step 2.

The algorithm continues until some pre-specified termination criterion has been met, such as a small change to the state vector, or a limit on the number of iterations.

If the measurements $\mathbf{z}_{(j)}$ are unbiased and normally distributed, the residual $J(\hat{\mathbf{x}}^+)$ is a χ^2 random variable, and testing the value of J against a χ^2 distribution is a good way of checking that the measurement noise model is reasonable. The number of degrees of freedom (DOF) of the χ^2 distribution can be determined as the total size of the measurement vectors, minus the size of the state. If the $SIZE(.)$ function returns the dimension of its vector argument, then the degrees of freedom may be computed as

$$DOF = \sum_{j=1}^{k} SIZE(\mathbf{z}_{(j)}) - SIZE(\mathbf{x}). \tag{5.14}$$

5.10.1 Robust observations

An important drawback of standard least-squares algorithm such as Levenberg-Marquardt is that they assume that all observations are correct. Various types of estimators have been successfully used to deal with the presence

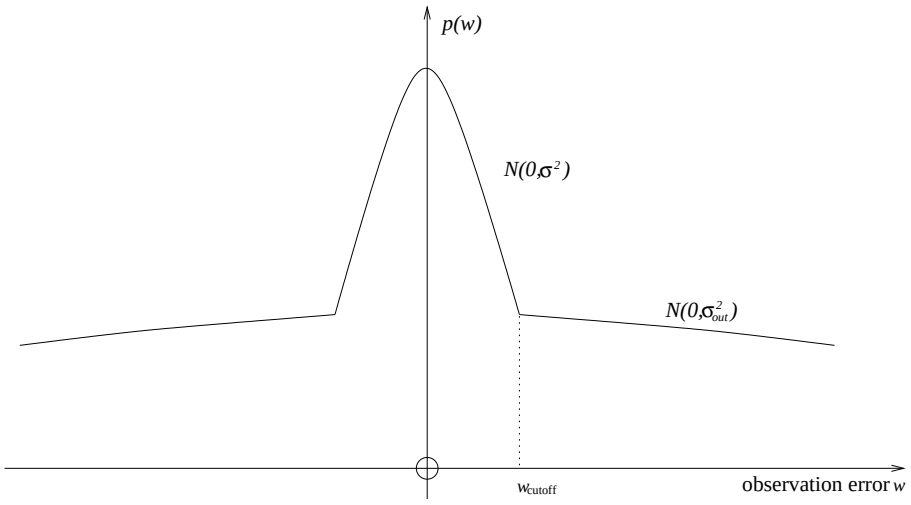

Figure 5.4: The error model used to model outliers in the observations incorporated in robust Levenberg-Marquardt, a combination of a narrow inlier Gaussian with variance σ^2, and wide Gaussian for outliers with variance σ^2_{out}. Both distributions on the observation error w have zero mean.

of outliers in the data. Examples are least median-of-squares, RANSAC and Hough transform estimators. These estimators involve a radical redesign of the measurement error model. We employ what is probably the simplest method of "robustifying" the standard Gaussian error model. The robust error model used here assumes that the errors follow a distribution combining a narrow "inlier" Gaussian with a wide "outlier" Gaussian, as shown for a one-dimensional distribution in Figure 5.4. The distribution is a function of the observation error[3] $\mathbf{w} = \mathbf{z} - \mathbf{h}(\mathbf{x})$. The relative vertical scaling of the two Gaussians is chosen so that the two distribution functions are equal at a chosen point x_{offset}.

For a general multi-dimensional observation, we have a inverse covariance matrix N^{-1} for the inlier distribution. We restrict the outlier distribution N^{-1}_{out} to be a rescaled version of the inlier distribution, so that

$$N^{-1}_{\text{out}} = \frac{1}{V} N^{-1}$$

for some value $V > 1$. We then set choose a cutoff hypersphere in the state space $\mathbf{x}$ for switching between the two distributions as a particular value of the χ^2. So the probability distribution function is

$$p(\boldsymbol{\nu}) = \begin{cases} e^{-\boldsymbol{\nu}^\top N^{-1} \boldsymbol{\nu}} & \text{if } \boldsymbol{\nu}^\top N^{-1} \boldsymbol{\nu} < \chi^2_{\text{cutoff}} \text{ (inlier)} \\ \\ e^{K^{-1}-1} e^{-\boldsymbol{\nu}^\top N^{-1}_{\text{out}} \boldsymbol{\nu}} & \text{otherwise (outlier)} \end{cases}$$

The scaling of the outlier distribution is chosen so that the two distributions are correctly aligned at the chosen cutoff point χ^2_{cutoff}. This leads directly to the correct "compensation" value for the likelihood function $1 - K^{-1}\chi^2_{\text{cutoff}}$, to be added to the least-squares residual when the outlier distribution is selected during application of a minimisation iteration. The simple scheme used to decide switching between the two distributions is detailed below. Note that each Levenberg-Marquardt observation can be chosen as robust or standard (non-robust), and potentially with a different choice for K and χ^2_{cutoff}.

5.10.2 Generalised observations

The observation function $\mathbf{h}(.)$ in Equation 5.7 does not encapsulate the most general form of observation, since it assumes that the observation vector $\mathbf{z}$ can be separated as a function of the state $\mathbf{x}$. It is sometimes therefore

[3]The innovation $\boldsymbol{\nu}$ is the observation error relative to that computed from the *estimated* state: $\boldsymbol{\nu} = \mathbf{z} - \mathbf{h}(\hat{\mathbf{x}})$.

necessary to introduce a generalised observation of the form

$$\mathbf{F}(\mathbf{x}, \mathbf{z} - \mathbf{w}) = \mathbf{0}$$

where $\mathbf{w}$ again represents a random noise vector having covariance N. However with some manipulation and extra computation we can effectively convert the linearised version of the $\mathbf{F}$-type function into an $\mathbf{h}$-type function, allowing it to be incorporated in the same way. We linearise $\mathbf{F}(.)$ with respect to $\mathbf{x}$ and $\mathbf{z}$ around the estimated state $\hat{\mathbf{x}}$ and observation $\mathbf{z}$, assuming that the noise $\mathbf{w}$ is small:

$$\mathbf{F}(\mathbf{x}, \mathbf{z}_t) = \mathbf{F}(\hat{\mathbf{x}}, \mathbf{z}) + \frac{\partial \mathbf{F}}{\partial \mathbf{x}}(\mathbf{x} - \hat{\mathbf{x}}) - \frac{\partial \mathbf{F}}{\partial \mathbf{z}}\mathbf{w} = \mathbf{0}$$

where $\mathbf{x}$ here represents the true value of the state vector, and $\mathbf{z}_t$ is the true observation vector (as opposed to the actually measured vector $\mathbf{z}$), so that $\mathbf{w} = \mathbf{z} - \mathbf{z}_t$. We identify the following quantities with their equivalents for an $\mathbf{h}$-type observation:

The **innovation** vector is $\boldsymbol{\nu} = -\mathbf{F}(\hat{\mathbf{x}}, \mathbf{z})$.

The **Jacobian** matrix is $H = \frac{\partial \mathbf{F}}{\partial \mathbf{x}}$.

The **noise vector** is $\mathbf{w}' = \frac{\partial \mathbf{F}}{\partial \mathbf{z}}\mathbf{w}$.

The **noise covariance** matrix is $N' = \frac{\partial \mathbf{F}}{\partial \mathbf{z}} N \left(\frac{\partial \mathbf{F}}{\partial \mathbf{z}}\right)^\top$.

Extra computation is therefore needed to convert the observation covariance from N to N'. The innovation vector $\boldsymbol{\nu}$, Jacobian matrix H and observation covariance N' are substituted into the Levenberg-Marquardt algorithm in place of their equivalents for the $\mathbf{h}$-type observation. There is no reason why there should not be a robust version of the $\mathbf{F}$-type observation, but currently it is not implemented.

5.10.3 Levenberg-Marquardt software

The following code extracts are taken from the `vision/vision_test.c` test program. The example application is fitting a quadratic function through points x, y on a plane. The function to be fitted is

$$y = ax^2 + bx + c$$

The state vector of unknown parameters to be estimated is thus

$$\mathbf{x} = \begin{pmatrix} a \\ b \\ c \end{pmatrix}$$

and we wish to compute the least-squares solution that minimises

$$J(\mathbf{x}) = \sum_{j=1}^{k} (y_j - ax_j^2 - bx_j - c)^2 \sigma_j^{-2}$$

given k points x_j, y_j for $j = 1, \ldots, k$ and independent noise levels σ_j for each point j. This can be solved directly by linear methods, and this feature makes it useful as a test algorithm because test program can compare the results with the Levenberg-Marquardt solution. The problem can be put into the form of the Levenberg-Marquardt algorithm described above by identifying

$$\mathbf{z}_{(j)} = (y_j), \quad N^{-1}{}_{(j)} = (\sigma^{-2}), \quad \mathbf{h}(\mathbf{x}) = ax_j^2 + bx_j + c$$

To initialise a Levenberg-Marquardt algorithm instance use the call

```
Gan_LevMarqStruct *lm;

/* initialise Levenberg-Marquardt algorithm */
lm = gan_lev_marq_alloc();
```

We build the points from ground-truth values for the quadratic coefficients a, b, c, and add random Gaussian noise:

```
/* number of points */
#define NPOINTS 100

/* ground-truth quadratic coefficients a,b,c */
#define A_TRUE 2.0
#define B_TRUE 3.0
#define C_TRUE 4.0

/* noise standard deviation */
#define SIGMA 1.0

/* arrays of x & y coordinates */
double xcoord[NPOINTS], ycoord[NPOINTS];

/* build arrays of x & y coordinates */
for ( i = NPOINTS-1; i >= 0; i-- )
{
   /* x-coordinates evenly spaced */
   xcoord[i] = (double) i;

   /* construct y = a*x^2 + b*y + c + w with added Gaussian noise w */
   ycoord[i] = A_TRUE*xcoord[i]*xcoord[i] + B_TRUE*xcoord[i]
               + C_TRUE + gan_normal_sample(0.0, SIGMA);
}
```

Here we defined a noise level `SIGMA` as the estimated standard deviation of the random observation errors, the same for each point. Now that we have constructed the input data, the next thing is to create observations for each point. Gandalf's version of Levenberg-Marquardt uses callback functions to evaluate observation $\mathbf{h}(.)$ and observation Jacobians H. A non-robust h-type observation is then defined by:

- The observation vector $\mathbf{z}$;
- The observation inverse covariance N^{-1}, and;
- The observation callback function $\mathbf{h}(.)$.

We construct the observations for the quadratic fitting problem using the following code:

```
Gan_Vector *z; /* define observation vector */
Gan_SquMatrix *Ni; /* define observation inverse covariance */

/* allocate observation vector z and inverse covariance Ni */
z = gan_vec_alloc(1);
Ni = gan_symmat_fill_va ( NULL, 1, 1.0/(SIGMA*SIGMA) );

for ( i = NPOINTS-1; i >= 0; i-- )
{
   /* construct point observation */
```

```
   z = gan_vec_fill_va ( z, 1, ycoord[i] );
   cu_assert ( z != NULL );
   cu_assert ( gan_lev_marq_obs_h ( lm, z, &xcoord[i], Ni, quadratic_h )
               != NULL );
}
```

We create the observation vector with size one; this can be adjusted dynamically if necessary; see Section 3.2.1.2.

The observation callback function `quadratic_h()` is defined as follows:

```
/* observation callback function for single point */
static Gan_Bool
 quadratic_h ( Gan_Vector *x,  /* state vector */
               Gan_Vector *z,  /* observation vector */
               void *zdata,    /* user pointer attached to z */
               Gan_Vector *h,  /* vector h(x) to be evaluated */
               Gan_Matrix *H ) /* matrix dh/dx to be evaulated or NULL */
{
   double a, b, c;

   /* read x-coordinate from user-defined data pointer */
   double xj = *((double *) zdata);

   /* read quadratic parameters from state vector x=(a b c)^T*/
   if ( !gan_vec_read_va ( x, 3, &a, &b, &c ) )
   {
      gan_err_register ( "quadratic_h", GAN_ERROR_FAILURE, NULL );
      return GAN_FALSE;
   }

   /* evaluate h(x) = h(a,b,c) = y = a*x*x + b*x + c */
   if ( gan_vec_fill_va ( h, 1, a*xj*xj + b*xj + c ) == NULL )
   {
      gan_err_register ( "quadratic_h", GAN_ERROR_FAILURE, NULL );
      return GAN_FALSE;
   }

   /* if Jacobian matrix is passed as non-NULL, fill it with the Jacobian
      matrix (dh/da dh/db dh/dc) = (x*x x 1) */
   if ( H != NULL &&
        gan_mat_fill_va ( H, 1, 3, xj*xj, xj, 1.0 ) == NULL )
   {
      gan_err_register ( "quadratic_h", GAN_ERROR_FAILURE, NULL );
      return GAN_FALSE;
   }

   /* success */
   return GAN_TRUE;
}
```

Note the the x-coordinate passed in as the third "user pointer" argument to `gan_lev_marq_obs_h()` is read into the variable `xj` in `quadratic_h()`. Using pointers in this way is the standard method to pass extra information into the callback routines.

So far we have merely registered the observations and their callback routines. No processing has started. To get

started with some actual optimisation we need to initialise the state vector with some values for a, b and c. This involves invoking the routine

```
double residual;

/* initialise Levenberg-Marquardt algorithm */
gan_lev_marq_init ( lm, quadratic_init, NULL, &residual );
```

`quadratic_init()` is another callback routine that computes values for the state vector $\mathbf{x}$ given the observations $\mathbf{z}_{(j)}$. The observations are presented to `quadratic_init()` in a linked list. The third argument to `gan_lev_marq_init()` is another user pointer, not used in this example. The last `residual` argument is returned as the initial value of the least-squares residual $J(\mathbf{x})$. This is the full code for the `quadratic_init()` function, whose operation is self-explanatory:

```
/* initialisation function for state vector */
static Gan_Bool
 quadratic_init ( Gan_Vector *x0,     /* state vector to be initialised */
                  Gan_List *obs_list, /* list of observations */
                  void *data )        /* user data pointer */
{
   int list_size = gan_list_get_size(obs_list);
   Gan_LevMarqObs *obs;
   Gan_Matrix33 A;
   Gan_Vector3 b;
   double xj, y;

   /* we need at least three points to fit a quadratic */
   if ( list_size < 3 ) return GAN_FALSE;

   /* initialise quadratic by interpolating three points: the first, middle and
      last point in the list of point observations. We construct equations

         (y1)   (x1*x1 x1 1) (a)
         (y2) = (x2*x2 x2 1) (b) = A * b for 3x3 matrix A and 3-vector b
         (y3)   (x3*x3 x3 1) (c)

      and solve the equations by direct matrix inversion (not pretty...) to
      obtain our first estimate of a, b, c given points (x1,y1), (x2,y2) and
      (x3,y3).
   */

   /* first point */
   gan_list_goto_pos ( obs_list, 0 );
   obs = gan_list_get_current ( obs_list, Gan_LevMarqObs );
   xj = *((double *) obs->details.h.zdata); /* read x-coordinate */
   A.xx = xj*xj; A.xy = xj; A.xz = 1.0; /* fill first row of equations in A */
   gan_vec_read_va ( &obs->details.h.z, 1, &y );
   b.x = y; /* fill first entry in b vector */

   /* middle point */
   gan_list_goto_pos ( obs_list, list_size/2 );
   obs = gan_list_get_current ( obs_list, Gan_LevMarqObs );
   xj = *((double *) obs->details.h.zdata); /* read x-coordinate */
   A.yx = xj*xj; A.yy = xj; A.yz = 1.0; /* fill first row of equations in A */
```

```
   gan_vec_read_va ( &obs->details.h.z, 1, &y );
   b.y = y; /* fill second entry in b vector */

   /* last point */
   gan_list_goto_pos ( obs_list, list_size-1 );
   obs = gan_list_get_current ( obs_list, Gan_LevMarqObs );
   xj = *((double *) obs->details.h.zdata); /* read x-coordinate */
   A.zx = xj*xj; A.zy = xj; A.zz = 1.0; /* fill first row of equations in A */
   gan_vec_read_va ( &obs->details.h.z, 1, &y );
   b.z = y; /* fill second entry in b vector */

   /* invert matrix and solve (don't do this at home) */
   A = gan_mat33_invert_s(&A);
   b = gan_mat33_multv3_s ( &A, &b );

   /* fill state vector x0 with our initial values for a,b,c */
   gan_vec_fill_va ( x0, 3, b.x, b.y, b.z );
   return GAN_TRUE;
}
```

We are now ready to apply optimisation iterations using the routine `gan_lev_marq_iteration()`. The following code applies ten iterations, adjusting the damping factor in the way suggested in [12]. This simple scheme decreases the damping when the residual decreases, and vice versa.

```
double lambda = 0.1; /* damping factor */
double new_residual;

/* apply iterations */
for ( i = 0; i < 10; i++ )
{
   gan_lev_marq_iteration ( lm, lambda, &new_residual );
   if ( new_residual < residual )
   {
      /* iteration succeeded in reducing the residual */
      lambda /= 10.0;
      residual = new_residual;
   }
   else
      /* iteration failed to reduce the residual */
      lambda *= 10.0;
}
```

To extract the optimised solution, use the code

```
Gan_Vector *x;

/* get optimised solution */
x = gan_lev_marq_get_x ( lm );
```

Note that the `x` pointer passed back here points to a vector internal to the Levenberg-Marquardt software, and should *not* be freed. To free the Levenberg-Marquardt structure and the matrices & vectors created above, use the code

```
gan_squmat_free ( Ni );
```

```
gan_vec_free ( z );
gan_lev_marq_free ( lm );
```

5.11 Fast Hough Transform

```
#include <gandalf/vision/fast_hough_transform.h>
```

A Hough transform is a mapping from an *observation space* into a *parameter space*. In computer vision, observation space could be a digital image, an edge map etc. Now assume that a certain structure is thought to be present in image space. For an edge map, this could be a straight line or a circle. The parameters of the structure define parameter space (gradient and intercept for a line, radius and centre coordinates for a circle). In a Hough transform, each point in image space "votes" for that part of parameter space which describes structures which include the point. For instance, to find circles in an edge map, edges vote for the region in parameter space (in fact a conical surface) which describes circles that pass through them. A part of parameter space receiving a large number of votes corresponds to a possible fit.

In the normal Hough transform approach, parameter space is bounded by setting lower and upper limits on the parameter values, and then divided into blocks in each direction, and an accumulator assigned to each block. The Hough transform proceeds with each point in image space being transformed to an region in parameter space as described in the previous paragraph. When the region intersects one of the blocks, the corresponding accumulator is incremented. The block whose accumulator has the most votes can then be taken as the best fit of the structure to the image points, the values of the parameters usually being calculated at the centre of the block.

5.11.1 The Fast Hough Transform (FHT)

The above method has two main drawbacks: large memory requirement and slowness. In order to find the plane parameters accurately, parameter space must be divided finely in all three directions, and an accumulator assigned to each block. Also it takes a long time to fill the accumulators when there are so many. The Fast Hough Transform described in [7] gives considerable speed up and reduces memory requirement. Instead of dividing parameter space uniformly into blocks, the FHT "homes in" on the solution, ignoring areas in parameter space relatively devoid of votes. The relative speed advantage of the FHT increases for higher dimensional parameter spaces.

The FHT applies to those Hough transform problems in which a feature F_j votes for a *hyperplane* in parameter space (a hyperplane is a $k-1$ dimensional generalisation of a plane, where k is the dimension of parameter space). This means that the relationship between feature space and parameter space must be linear in the parameters. In addition, it must be known in advance how many votes the solution will receive in the Hough transform. The FHT is also restricted in that it only supplies one "best fit" solution, whereas for the more conventional Hough transform method above it is plausible to consider local maxima as alternatives to the global maximum, i.e. the block in parameter space receiving the most votes.

A major advantage of the FHT is that it only uses addition and multiplication by two, which in integer arithmetic can be done efficiently using bitwise shifts. This is very convenient for computers on which integer arithmetic is much faster than floating point arithmetic.

5.11.1.1 Notation

The following notation is taken from [7]. Hyperplanes are represented by the equations

$$a_{0j} + \sum_{i=1}^{k} a_{ij} X_i = 0 \quad \text{for } j \;=\; 1, 2, \ldots, n \tag{5.15}$$

where $(X_1, X_2, \ldots, X_k)$ is parameter space, rescaled so that the initial ranges of each X_i are the same and centred around zero. The initial ranges thus form a *hypercube* (generalisation of a cube) in parameter space. Each a_{ij} is a function of F_j normalised such that $\sum_{i=1}^{k} a_{ij}^2 = 1$.

5.11.1.2 The FHT Algorithm

A coarse Hough Transform is applied to the initial "root" hypercube in parameter space by dividing it into 2^k "child" hypercubes formed by halving the root along each of the k dimensions and assigning an accumulator to each child. Each hyperplane passing through a child hypercube increments its accumulator. Those children receiving greater than a threshold T votes are recursively subdivided, becoming parents themselves, and so on.

A limit is set on the level of subdivision. Because the range of the parameters in a hypercube is halved for each level of subdivision, This is equivalent to having a precision threshold:

$$\text{maximum subdivision level} = \log_2 \frac{\text{initial range}}{\text{required accuracy}}.$$

An extra speed up is made possible by keeping track of which features vote for (i.e. which hyperplanes intersect) each hypercube. Only those features need be tested for intersection between hyperplane and child hypercubes, since children lie inside their "parents".

Li et al. consider two methods of calculating whether a hyperplane passes through a hypercube. The "brute force" method is to test whether all the vertices of the hypercube are on the same side of the hyperplane by evaluating the left hand side of equation 5.15 for each vertex. For increased speed, Li et al. recommend determining whether the hyperplane intersects the hypercube's circumscribing hypersphere. This is an approximation, since some planes will intersect the hypersphere but not the hypercube. One effect of this is that the total number of subdivisions will be increased, because a hypersphere may get more than T votes when the hypercube on its own (being smaller) would not have. However the increased speed more than compensates for this, especially for high-dimensional parameter spaces. The method is fast because the perpendicular distance from the centre of a child hypercube to a hyperplane can be calculated simply in terms of the corresponding distance from the parent's centre to the hyperplane, as is shown on page 147. This distance is then compared with the radius of the hypersphere.

5.11.1.3 Example: Line Fitting

Figure 5.5 shows how the FHT works for $k = 2$, when parameter space is a plane, hyperplanes are straight lines and hypercubes are squares whose associated hyperspheres are circles passing through the vertices of the squares (figure 5.5). This is applicable to the problem of finding a straight line through points on a plane. If the plane has coordinates (u, v) the line can be written

$$v = \alpha u + \beta$$

where α and β are constant. Each point (u_j, v_j) votes for a line in parameter space:

$$\beta = v_j - \alpha u_j.$$

Let the initial ranges of α and β, defining the root hypercube, be L_α and L_β centred around α_0 and β_0 respectively. Then the above equation can put in the form of equation 5.15 using the transformation

$$X_1 = \frac{\alpha}{L_\alpha},\ X_2 = \frac{\beta}{L_\beta},\ a_{0j} = \frac{-v_j}{q},\ a_{1j} = \frac{L_\alpha u_j}{q},\ a_{2j} = \frac{L_\beta}{q}$$

where $q = \sqrt{L_\alpha^2 u_j^2 + L_\beta^2}$.

5.11.2 Example: Plane Fitting

The method described here was used in [10]. A pair of images is rectified so that their epipolar lines are horizontal and parallel. Edges are detected in each image. A region of the left image is then matched to the right image using a planar fit in (x, y, d) space where (x, y) are the coordinates of edges in the left image and d is the disparity, such that if (x_r, y_r) are the coordinates of the corresponding edge in the right image,

$$x_r = x + d$$

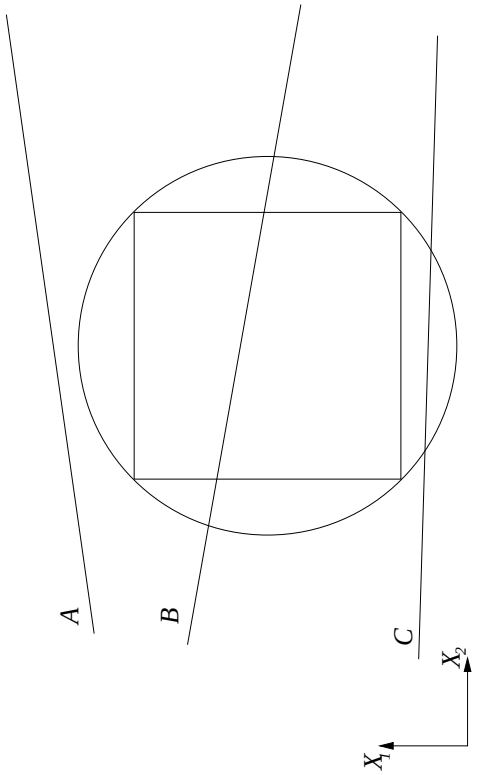

Figure 5.5: This illustrates the approximate method of calculating intersections between hyperplane and hypercube in the Fast Hough Transform in the case $k = 2$. A line is tested for intersection with the square's circumscribing circle rather than the square itself. This method give the same result for lines like A and B, but not for line C

and because of the rectification procedure, $y_r = y$.

A planar fit is attempted to the disparity points from candidate edge matches between the image, in the knowledge that planes in disparity space (x, y, d) correspond to planes in the world.

A large number of the disparity points in the box will be incorrect matches, so direct fitting, for example for least squares, would not work. However the Hough transform is well suited to such a problem. It is used to select a large subset of the disparity points that lie near a plane. The Fast Hough transform gives a large increase in speed and decrease in storage requirement over the standard Hough transform method. It is applicable because, with an appropriate parametrisation, plane fitting is a linear Hough transform problem, to which the FHT is restricted.

5.11.2.1 Calculating the Intersection of a Plane and a Sphere

The perpendicular, and therefore nearest, distance from the plane to the centre of the cube is calculated. If it is smaller than the radius of the circumscribing sphere, the plane intersects the sphere, otherwise it misses. The distance is normalised by dividing it by the side length of the cube. The normalised distance of a plane from the centre of a child cube can be calculated simply from the normalised distance of the plane from it's parent's centre, as shown below. The plane is defined by the following equation:

$$a_0 + a_1 X_1 + a_2 X_2 + a_3 X_3 = 0$$

where $a_1^2 + a_2^2 + a_3^2 = 1$. Let the child cube have indices $[b_1, b_2, b_3]$, centre (C_1, C_2, C_3) and side length L_{child}. The radius of the circumscribing sphere is $\frac{\sqrt{3}L_{\text{child}}}{2}$. The perpendicular distance of the plane to the centre of the child cube is

$$\begin{aligned} \text{perpendicular distance} &= \frac{a_0 + a_1 C_1 + a_2 C_2 + a_3 C_3}{\sqrt{a_1^2 + a_2^2 + a_3^2}} \\ &= a_0 + a_1 C_1 + a_2 C_2 + a_3 C_3. \end{aligned}$$

When normalised this becomes

$$R_{\text{child}} = \frac{a_0 + a_1 C_1 + a_2 C_2 + a_3 C_3}{L_{\text{child}}}. \tag{5.16}$$

The normalised distance of the centre of the parent from the plane, where the parent has centre (p_1, p_2, p_3) and side length L_{parent}, is

$$R_{\text{parent}} = \frac{a_0 + a_1 p_1 + a_2 p_2 + a_3 p_3}{L_{\text{parent}}}.$$

The child is half the size of the parent, so $L_{\text{parent}} = 2L_{\text{child}}$. Also the centres of the cubes are related by the following equation:

$$C_i = p_i + \frac{b_i L_{\text{child}}}{2}. \tag{5.17}$$

and substituting the RHS into equation 5.16 yields

$$\begin{aligned} R_{\text{child}} &= \frac{a_0 + a_1 p_1 + a_2 p_2 + a_3 p_3 + \frac{L_{\text{child}}}{2}(a_1 b_1 + a_2 b_2 + a_3 b_3)}{L_{\text{child}}} \\ &= 2R_{\text{parent}} + \frac{1}{2}(a_1 b_1 + a_2 b_2 + a_3 b_3). \end{aligned} \tag{5.18}$$

This formula is used to calculate the normalised distance for a child in terms of that of it's parent, and is fast because the 2^k values of the term $\frac{1}{2}(a_1 b_1 + a_2 b_2 + a_3 b_3)$ can be stored as a look-up table for each set of coefficients a_{ij} (i.e. each disparity point).

The normalised distance of the plane from the root cube, which has side length one and centre $(0, 0, \frac{d_{\text{peak}}}{w})$, is

$$R_{\text{root}} = a_0 + a_3 \frac{d_{\text{peak}}}{w}. \tag{5.19}$$

The normalised distances are calculated initially from equation 5.19 and from then on using the formula 5.18. The normalised distances are compared with $\frac{\sqrt{3}}{2}$, the radius of the circumscribing sphere of a cube with side length one. This is equivalent to comparing the un-normalised distance with the circumscribing sphere of the original cube.

5.11.2.2 Calculating the Plane Parameters of a Child Cube

When a new "best so far" fit cube is found, the cube needs to find out where it is in parameter space. The FHT does not explicitly propagate information from parent to child concerning the position of hypercubes in parameter space. Of course it would be trivial to do this, by, for instance, calculating the centre of the child relative to the centre of the parent, using equation 5.17, and passing the centre coordinates on to the child. However, since centre coordinates are needed *only* when an improved planar fit is found, this would involve quite a lot of wasted multiplication and division.

The method used in Needles provides each cube with the part of its "family tree" comprising the direct line of descent from the root to itself. It therefore knows how it is related to its parent, how its parent was related to its grandparent, and so on. A cube knows only about its own "branch", which is passed down to it by its parent.

The relationship between parent and child cubes is contained in the child indices $[b_1, b_2, b_3]$. Each cube therefore receives a list of such index triplets, detailing the relationships between ancestors of different generations. The list is linked *backwards*, i.e. from child to root. The formulae to calculate a cube's centre coordinates are simpler when the list is traversed in this direction. They are computational in nature, involving iteration when traversing the list, so we have not included them here, but they are given in the next section in procedure *cube_centre*.

5.11.2.3 Formal Statement of the FHT Plane Fitting Algorithm

We have:

1. A set of disparity points (x_j, y_j, d_j), $j = 1 \ldots n$.
2. An array of weights W_j, $j = 1 \ldots n$, one for each disparity point (x_j, y_j, d_j).
3. Parameter space (a, b, c), restricted to the box with centre $(0, 0, d_{\text{centre}})$ and side lengths L_a, L_b and L_c. d_{centre} is the centre of the disparity search range.
4. A vote threshold $T = \theta W$, W being the total weight of the edges in the square patch.
5. A minimum level of subdivision $l_{\min}$ (e.g. 3), and a maximum level $l_{\max}$ (e.g. 10).

The algorithm is as follows:

begin

Declare new variables:

An array $\mathbf{R}$ of distances R_j, $j = 1 \ldots n$.
An array $\mathbf{P}$ of boolean flags $P_1, P_2, \ldots, P_n$ where each P_j either takes the value $true$ or $false$.
Integer variables max_level and max_value, initialised to zero.
Variables a_{best}, b_{best} and c_{best} making up a point in (a, b, c) space.
Another array of boolean flags $\mathbf{P}_{\text{best}}$.

The variables max_level, max_value, a_{best}, b_{best}, c_{best} and array $\mathbf{P}_{\text{best}}$ are used to keep track of the best fit as the algorithm proceeds.

Set all the P_j to $false$.

For each disparity point (x_j, y_j, d_j):

begin

Calculate the plane parameters for (X_1, X_2, X_3) space:

$$a_{0j} = \frac{-d_j}{Q},\ a_{1j} = \frac{L_a x_j}{Q},\ a_{2j} = \frac{L_b y_j}{Q},\ a_{3j} = \frac{L_c}{Q}$$

where $Q = \sqrt{L_a^2 x_j^2 + L_b^2 y_j^2 + L_c^2}$.

Calculate the normalised perpendicular distance R_j from the plane to the centre of the root cube:

$$R_j = a_{0j} + \sum_{i=1}^{3} a_{ij} C_i = a_{0j} + \frac{a_{3j} d_{\text{centre}}}{Q L_c}.$$

If $R_j < \frac{\sqrt{3}}{2}$ the plane passes through the root cube's circumscribing sphere: set P_j to *true*.

end

Call the procedure *divide_cube* with arguments as follows:

1. Array of boolean flags **P**.
2. Array **R** of normalised distances.
3. Initial level of subdivision 0.
4. Initial accumulator value 0.
5. Empty line of descent list.

end

Procedure *divide_cube* (array of flags **P**, array of normalised distances **R**, subdivision level l, accumulator value v, line of descent list):

begin

Declare new variables:

Eight boolean flags arrays $\mathbf{P_b}$, one for each **b**, where **b** is a child cube index triplet (b_1, b_2, b_3) and each b_i is either -1 or 1. All the elements of the flag arrays are initialised to *false*.

Eight arrays $\mathbf{R_b}$ of normalised distances.

Accumulators $A_\mathbf{b}$ for each **b**, initialised to zero.

Boolean flag s, initialised to *false*.

Sum the votes for the child cubes: for each j such that $P_j = true$:

begin

For each child cube with index **b**:

begin

Calculate the normalised distance $R_{j\mathbf{b}}$ from plane j to the centre of the child cube:

$$R_{j\mathbf{b}} = 2R_j + \frac{1}{2}(a_{1j}b_1 + a_{2j}b_2 + a_{3j}b_3).$$

(The $a_{1j}b_1 + a_{2j}b_2 + a_{3j}b_3$ values can be efficiently calculated by storing them in eight arrays, one for each **b**.)

If $R_{j\mathbf{b}} < \frac{\sqrt{3}}{2}$ then

begin

Add weight W_j to $A_\mathbf{b}$

Set flag $P_{j\mathbf{b}}$ to *true*.

end

end

end

Subdivide child cubes receiving greater than or equal to T votes: If $l < l_{\max}$:

begin

For each child cube with index **b**:

begin

If $A_{\mathbf{b}} \geq T$ recursively subdivide child cube:

begin

Add **b** to original line of descent.
Call *divide_cube* with arguments $\mathbf{P_b}$, $\mathbf{R_b}$, $l+1$, $A_{\mathbf{b}}$ and extended line of descent.
Set flag s to *true*.

end

end

If s is still $false$ (i.e. no further subdivision of this cube), test for best fit so far:

If $l > max\_level$ or if $l = max\_level$ and $v > max\_value$:

begin

Set max_level to l.
Set max_value to v.
Calculate centre of cube in (a, b, c) space by calling procedure *cube_centre* with the line of descent as argument. The results are copied into a_{best}, b_{best} and c_{best}.
Copy array of flags $\mathbf{P}$ into array $\mathbf{P}_{\text{best}}$.

end

end (procedure *divide_cube*)

Procedure *cube_centre* (line of descent list): **begin**

Declare new variables:

Three arrays $\mathbf{r}_a$, $\mathbf{r}_b$ and $\mathbf{r}_c$, each of three elements with indices -1, 0 and 1, so that $\mathbf{r}_a = [r_a(-1), r_a(0), r_a(1)]$ and similarly for $\mathbf{r}_b$ and $\mathbf{r}_c$. The zero (middle) elements of each array are not used.

Set arrays $\mathbf{r}_a$, $\mathbf{r}_b$ and $\mathbf{r}_c$ as follows:

$$\begin{array}{ccc} r_a(-1) = \frac{-L_a}{4}, & r_b(-1) = \frac{-L_b}{4}, & r_c(-1) = \frac{-L_c}{4}, \\ r_a(1) = \frac{L_a}{4}, & r_b(1) = \frac{L_b}{4}, & r_c(1) = \frac{L_c}{4}. \end{array}$$

Set a_{best}, b_{best} and c_{best} to zero.

While line of descent not traversed (i.e. root cube not yet reached):

begin

Take next triplet of indices **b** on line of descent.

Replace old values of a_{best}, b_{best} and c_{best}:

$$\begin{aligned} a_{\text{best}} &\leftarrow \tfrac{a_{\text{best}}}{2} + r_a(b_1) \\ b_{\text{best}} &\leftarrow \tfrac{b_{\text{best}}}{2} + r_b(b_2) \\ c_{\text{best}} &\leftarrow \tfrac{c_{\text{best}}}{2} + r_c(b_3) \end{aligned}$$

end

Add centre c-coordinate of root cube to c_{best} (a and b coordinates of root are zero):

$$c_{\text{best}} \leftarrow c_{\text{best}} + d_{\text{centre}}$$

end (procedure *cube_centre*)

At the end of the algorithm *max_level* is the highest level of subdivision reached. This must be greater than or equal to $l_{\min}$ for the fit to be used. *max_value* is the highest accumulator value of a cube at subdivision level *max_level*, the cube has centre $(a_{\text{best}}, b_{\text{best}}, c_{\text{best}})$ and these are the best fit plane parameters. The flags $\mathbf{P}_{j\text{best}}$ are *true* if the j'th disparity point voted for the winning cube, and those disparity points constitute the subset of all the points which best fit a plane, given that the total weight of the points must exceed T. They and the values of *max_level* and *max_value* can stored for use. More precise estimates of the plane parameters can be calculated using orthogonal regression.

5.11.3 Speed Improvement to FHT Line Finder

```
#include <gandalf/vision/modified_fht.h>
```

The Fast Hough Transform can be used to fit a line to points on a plane. However the modification to the FHT described here has proved to be slightly faster for the line fitting problem, and it requires less memory. It can readily be generalised and is applicable to the same class of Hough transform problems as standard FHT. However for higher dimensional parameter spaces (e.g. plane fitting) standard FHT takes over as the faster method. To simplify the notation the modified FHT (referred to henceforth as the MFHT) is described for line fitting only.

The FHT line fitter was described in section 5.11.1.3 and the notation used there will be followed. Points (u_j, v_j), $j = 1 \ldots n$ are scattered on the (u, v) plane. A straight line in the plane is defined as

$$v = \alpha u + \beta$$

where α and β are constant. Each point (u_j, v_j) "votes" for a line in parameter space (α, β). Ranges for α and β are specified, so $\alpha_1 \leq \alpha \leq \alpha_2$ and $\beta_1 \leq \beta \leq \beta_2$. Like the standard FHT, the MFHT proceeds by dividing this root rectangle (hypercube) into four (2^k) child rectangles and counting the lines (hyperplanes) passing through each child rectangle. If the number of such intersections for any child is greater than a threshold T, the child is subdivided. The MFHT differs in that the circumscribing hypersphere (in this case ellipse) approximation is not used. Intersections between line and rectangle are calculated exactly. Hence there is no need to normalise parameter space in order to transform the root rectangle into a square, as is required for standard FHT.

The line in (α, β) space defined by a point (u_j, v_j) is

$$\beta = -u\alpha + v.$$

Intersection between line and rectangle is calculated by comparing the β coordinate of the vertices of the rectangle with the points of intersection between the line and the constant α sides of the rectangle. This is illustrated in figure 5.6. The ranges $[\alpha_{\text{low}}, \alpha_{\text{high}}]$ and $[\beta_{\text{low}}, \beta_{\text{high}}]$ define a rectangle in (α, β) space. Three lines A, B and C are shown. For each line, the β values of the intersection points with the lines $\alpha = \alpha_{\text{low}}$ and $\alpha = \alpha_{\text{high}}$ (points A_1 and A_2, B_1 and B_2, C_1 and C_2 in figure 5.6) are compared with β_{low} and β_{high}. The β values can be thought of as intercept values of the line with the α = constant lines. It is clear that line and rectangle miss each other if and only if if all four vertices are either below or above (i.e. their β values are all $<$ or $>$) both intercepts. Obviously this is true for line B but false for lines A and C.

In practice it saves repeated computational effort if intersections between a parent's four child rectangles and the lines are calculated at the same time. Let us assume that enough lines intersect the rectangle in figure 5.6 for it to be subdivided. It is divided into four child rectangles R_{00}, R_{01}, R_{10} and R_{11} as shown in figure 5.7. For each line such as line A shown, there are three intercept β values, at α_{low}, α_{av} and α_{high}. Obviously $\alpha_{\text{av}} = (\alpha_{\text{low}} + \alpha_{\text{high}})/2$. The intercepts are each to be compared with the three β values β_{low}, β_{av} and β_{high}. A boolean flag is defined for each of the nine vertices (V_{ll}, V_{la} etc. in figure 5.7) of the rectangular subdivision. Each flag is set to *false* if the vertex lies below the line (i.e. if the β value of the vertex is less than the corresponding intercept β value of the line) and to *true* if it lies above it. Label the flags v_{ll}, v_{la} etc. Then it follows that:-

1. The line misses rectangle R_{00} if and only if v_{ll}, v_{la}, v_{al}, v_{aa} are all either *true* or *false*.

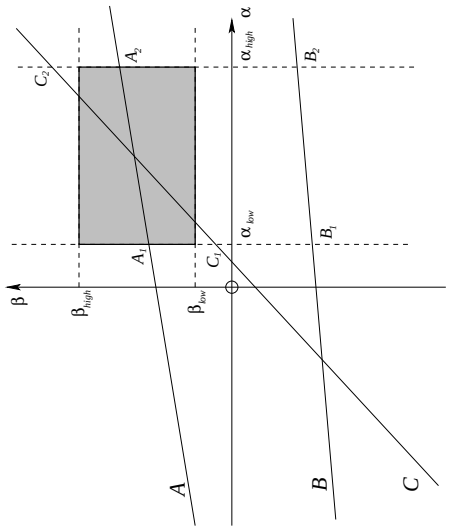

Figure 5.6: Deciding whether a line intersects a rectangle (see text).

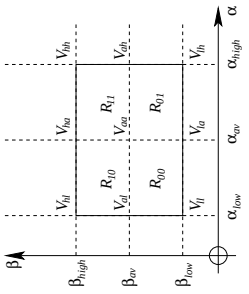

Figure 5.7: Subdivision of rectangle into four "children" (see text).

2. The line misses rectangle R_{01} if and only if v_{la}, v_{lh}, v_{aa}, v_{ah} are all either $true$ or $false$.
3. The line misses rectangle R_{10} if and only if v_{al}, v_{aa}, v_{hl}, v_{ha} are all either $true$ or $false$.
4. The line misses rectangle R_{11} if and only if v_{aa}, v_{ah}, v_{ha}, v_{hh} are all either $true$ or $false$.

In the standard version of the FHT, the normalised distances from hyperplane to centre of child hypercubes are calculated from the normalised distance of the parent from the hyperplane. The child "inherits" the new normalised distances from the parent, and passes them on to its own offspring. The MFHT also passes information from parent to child, in this case the intercept β values.

5.11.3.1 Formal Statement of Algorithm

We have:

1. A set of points on a plane (u_j, v_j), $j = 1 \ldots n$.
2. An array of weights W_j, $j = 1 \ldots n$, one for each point (u_j, v_j).
3. Parameter space (α, β), restricted to the rectangle $\alpha_1 \leq \alpha \leq \alpha_2$, $\beta_1 \leq \beta \leq \beta_2$.
4. A vote threshold T.
5. A maximum level of subdivision $l_{\max}$.

The algorithm is as follows:

begin

Declare new variables:

Arrays $\mathbf{I}^{\text{low}}$ and $\mathbf{I}^{\text{high}}$ of intercepts I_j^{low}, I_j^{high}, $j = 1 \ldots n$.
An array $\mathbf{P}$ of boolean flags $P_1, P_2, \ldots, P_n$.
Integer variables *max_level* and *max_value*.
Variables α_{best} and β_{best} comprising a point in (α, β) space.
Another array of flags $\mathbf{P}_{\text{best}}$.
Boolean flags s_{00}, s_{01}, s_{10} and s_{11}.

The variables *max_level*, *max_value*, α_{best}, β_{best} and array $\mathbf{P}_{\text{best}}$ are used to keep track of the best fit as the algorithm proceeds.

Set all the P_j to $false$.

For each point (u_j, v_j):

begin

Set flags corresponding to intercepts lying below/above vertices of the root rectangle:
if $v_j - \alpha_1 u_j - \beta_1 > 0$ set s_{00} to $false$, otherwise set it to $true$.
if $v_j - \alpha_2 u_j - \beta_1 > 0$ set s_{01} to $false$, otherwise set it to $true$.
if $v_j - \alpha_1 u_j - \beta_2 > 0$ set s_{10} to $false$, otherwise set it to $true$.
if $v_j - \alpha_2 u_j - \beta_2 > 0$ set s_{11} to $false$, otherwise set it to $true$.
If the s's are either all true or all false, the line $\beta = -u_j \alpha + v_j$ misses the root rectangle, otherwise they intersect. If they intersect, set I_j^{low} to $v_j - \alpha_1 u_j$, I_j^{high} to $v_j - \alpha_2 u_j$ and P_j to $true$.

end

Call the procedure *divide_rectangle* with arguments as follows:

1. Array of flags $\mathbf{P}$.
2. Arrays $\mathbf{I}^{\text{low}}$ and $\mathbf{I}^{\text{high}}$.
3. Variables β_1 and β_2.
4. Initial level of subdivision 0.
5. Initial accumulator value 0.
6. Empty line of descent list.

end

Procedure *divide_rectangle* (array of flags $\mathbf{P}$, arrays of intercepts $\mathbf{I}^{\text{low}}$ and $\mathbf{I}^{\text{high}}$, variables β_{low} and β_{high}, subdivision level l, accumulator value v, line of descent list):

begin

Declare new variables:

Boolean flag arrays $\mathbf{P}^{00}$, $\mathbf{P}^{01}$, $\mathbf{P}^{10}$ and $\mathbf{P}^{11}$ each with n elements. All the elements of the flag arrays are initialised to $false$.
Array $\mathbf{I}^{\text{av}}$ of β intercepts with n elements.
Accumulators A_{00}, A_{01}, A_{10} and A_{11}, initialised to zero.
Variable β_{av}.
Nine boolean flags v_{ll}, v_{la}, v_{lh}, v_{al}, v_{aa}, v_{ah}, v_{hl}, v_{ha}, v_{hh}.
Boolean flag s, initialised to $false$.

Set β_{av} to $(\beta_{\text{low}} + \beta_{\text{high}})/2$.

Sum the votes for the child rectangles: For each j such that $P_j = true$:

begin

Set flags v_{ll}, v_{la} etc. to $false$.
Set $I_j^{\rm av}$ to $(I_j^{\rm low} + I_j^{\rm high})/2$.
If $\beta_{\rm low} < I_j^{\rm low}$ then:
begin

Set v_{ll} to $true$.
If $\beta_{\rm av} < I_j^{\rm low}$ then:
begin

Set v_{la} to $true$.
If $\beta_{\rm high} < I_j^{\rm low}$ then set v_{lh} to $true$.

end

end

Repeat for intercept $I_j^{\rm av}$ and flags v_{al}, v_{aa} and v_{ah}.
Repeat for intercept $I_j^{\rm high}$ and flags v_{hl}, v_{ha} and v_{hh}.

end

If v_{ll}, v_{la}, v_{al} and v_{aa} are not all either $true$ or $false$:
begin

Add weight W_j to A_{00}
Set flag P_j^{00} to $true$.

end

Repeat for flags v_{la}, v_{lh}, v_{aa} and v_{ah}, accumulator A_{01} and flag P_j^{01}.
Repeat for flags v_{al}, v_{aa}, v_{hl} and v_{ha}, accumulator A_{10} and flag P_j^{10}.
Repeat for flags v_{aa}, v_{ah}, v_{ha} and v_{hh}, accumulator A_{11} and flag P_j^{11}.

end

Subdivide child rectangles receiving greater than or equal to T votes: If $l < l_{\rm best}$:

begin

If $A_{00} \geq T$ recursively subdivide child rectangle:
begin

Add [-1,-1] to original line of descent.
Call *divide_rectangle* with arguments $\mathbf{P}^{00}$, $\mathbf{I}^{\rm low}$, $\mathbf{I}^{\rm av}$, $\beta_{\rm low}$, $\beta_{\rm av}$, A_{00}, $l+1$ and extended line of descent.
Set flag s to $true$.

end

Repeat with accumulator A_{01}, index pair [-1,1], flag list $\mathbf{P}^{01}$, intercept arrays $\mathbf{I}^{\rm low}$ and $\mathbf{I}^{\rm av}$, and β values $\beta_{\rm av}$ and $\beta_{\rm high}$.
Repeat with accumulator A_{10}, index pair [1,-1], flag list $\mathbf{P}^{10}$, intercept arrays $\mathbf{I}^{\rm av}$ and $\mathbf{I}^{\rm high}$, and β values $\beta_{\rm low}$ and $\beta_{\rm av}$.
Repeat with accumulator A_{11}, index pair [1,1], flag list $\mathbf{P}^{11}$, intercept arrays $\mathbf{I}^{\rm av}$ and $\mathbf{I}^{\rm high}$, and β values $\beta_{\rm av}$ and $\beta_{\rm high}$.
If s is still $false$ (i.e. no further subdivision of this rectangle), test for best fit so far:

If $l > max\_level$ or if $l = max\_level$ and $v > max\_value$:
begin

Set *max_level* to l.
Set *max_value* to v.
Calculate centre of rectangle in α direction by calling procedure *alpha_centre* with the line of descent as argument. The result is copied into α_{best}.
Set β_{best} to β_{av}.
Copy array of flags $\mathbf{P}$ into array $\mathbf{P}_{\text{best}}$.

end

end (procedure *divide_rectangle*)

Procedure *alpha_centre* (line of descent list): **begin**

Declare new variables:

Array $\mathbf{r}_\alpha$ of three elements with indices -1, 0 and 1. The zero (middle) element of $\mathbf{r}_\alpha$ is not used.

Set array $\mathbf{r}_\alpha$ as follows:

$$r_\alpha(-1) = \frac{\alpha_1 - \alpha_2}{4}, \quad r_\alpha(1) = \frac{\alpha_2 - \alpha_1}{4}.$$

Set α_{best} to zero.

While line of descent not traversed (i.e. root rectangle not yet reached):

begin

Take next pair of indices $[b_1, b_2]$ on line of descent.
Replace old value of α_{best}:

$$\alpha_{\text{best}} \leftarrow \frac{\alpha_{\text{best}}}{2} + r_\alpha(b_1).$$

end

Add centre α-coordinate of root cube to α_{best}:

$$\alpha_{\text{best}} \leftarrow \alpha_{\text{best}} + \frac{\alpha_1 + \alpha_2}{2}.$$

end (procedure *alpha_centre*)

At the end of the algorithm *max_level* is the highest level of subdivision reached, *max_value* the highest accumulator value at level *max_level*, while α_{best} and β_{best} hold the best fit line parameters.

Note that the points (u_j, v_j) are never used by the procedure *divide_rectangle*. All the information about the points is contained in the intercept arrays. The β index part of the line of descent is actually superflous since the β positions of rectangles are passed from parent to child via β_{low} and β_{high}.

5.11.3.2 Comparision of Speed and Memory Requirement with Standard FHT

Direct speed tests on a Sun have shown that the MFHT is about 8% faster than an FHT line fitter adapted from the plane fitter described in section 5.11.2.3. This improvement is almost entirely due to the hypersphere (circle) approximation to a hypercube (square) used in the FHT, which increases the total number of subdivisions. The speeds of individual subdivisions in the two versions are almost identical.

When memory is limited, the MFHT is preferable. Ignoring the arrays of flags and individual variables, the MFHT allocates space for one array, $\mathbf{I}^{\text{av}}$, at each call to *divide_rectangle*. On the contrary, the FHT requires four: $\mathbf{R_b}$ for each value [-1,-1], [-1,1], [1,-1] and [1,1] of $\mathbf{b}$ (refer to procedure *divide_cube* in section 5.11.2.3). The FHT also uses global arrays, whereas the MFHT does not. The subdivision procedure in the FHT can be reordered so as to require only one array (which is used four times), but at the cost of introducing repeated tests into the code, slowing the algorithm by about 50%.

	FHT	MFHT
multiplications/divisions	2^k	$3^{k-1} - 2^{k-1}$
additions	2^k	$3^{k-1} - 2^{k-1}$
comparisons	2^k	$3^k/2$
boolean expressions	0	2^k

Table 5.1: Complexity comparison of FHT and MFHT. The figures are the number of calculations at each hypercube subdivision, for each hyperplane.

5.11.3.3 Complexity Comparison of FHT and MFHT

As the dimensionality k of parameter space increases, the MFHT becomes slower than the FHT. This is because the computational work at each hypercube subdivision increases faster for the MFHT as k increases than for the FHT.

Most of the work in the subdivision procedures (*divide_cube* for the FHT and *divide_rectangle* for the MFHT) goes into two stages. Firstly, arrays are calculated that are to be passed on to children. For the FHT plane finder these are the eight (2^k in general) $\mathbf{R_b}$ normalised distance arrays, whereas for the MFHT line finder, $\mathbf{I}^{\mathrm{av}}$ is the only such array (there are $3^{k-1} - 2^{k-1}$ arrays in general). In both cases the calculation of an array element involves two calculations: an addition and multiplication by two in the FHT, an addition and a division by two in the MFHT. The second time consuming part is the voting, i.e. the test for intersection of hyperplane with hypersphere/hypercube. For the FHT plane finder this is just a single comparison for each of the eight child cubes (2^k comparisons generally) whereas for the MFHT line finder, 9/2 comparisons (on average) and four boolean expressions are evaluated (for general k these figures become $3^k/2$ and 2^k respectively). All these figures are summarised in table 5.1.

Since 3^k increases faster than 2^k, the FHT will overtake the MFHT for speed as k increases. In fact tests have shown the MFHT plane fitter to be about 25% slower than the FHT. The memory space advantage of the MFHT is also reduced, since $3^2 - 2^2 = 5$ new arrays are required at each subdivision as against eight for the FHT (which again could be altered so as to require only one array, at the cost of some loss of speed).

Chapter 6

The Test Framework

There is extensive testing code in the `gandalf/TestFramework` directory. A simple way to test your installation of Gandalf is to execute the following commands:-

```
cd gandalf/TestFramework
make
./cUnit -all
```

This compiles the main Gandalf test program `cUnit` and runs through all the tests. There is at least one test program for each Gandalf package.

You can also build test programs individually by `cd`'ing into a Gandalf package and typing the command `make all`. For instance, the commands

```
cd gandalf/common
make all
./list_test
```

will compile all the test programs in the Common package, and run the linked list test program. The main test program `TestFramework/cUnit.c` compiles and links with all the individual test programs in the Gandalf packages.

Input files are all in the `gandalf/TestInput` directory. Output files are written into the `gandalf/TestOutput` directory. Test programs are designed to purge any old `TestOutput` files before running the test program.

6.1 Adding new tests

New code written for Gandalf should come with a test harness that tests the functionality of the module. Currently there is only one module-specific test harness in Gandalf, that for linked lists in `common/linked_list.[ch]`, the rest of the test programs combining tests over several modules in a package. In the future we wish to push Gandalf in the direction of Extreme Programming, with test harnesses for each module and for testing interactions between modules.

Let us say we have written a new module `histogram.[ch]` in a (currently fictitious) `statistics` package. We now wish to add a test harness `statistics/histogram_test.[ch]` (actually we should write the test harness *first* according to Extreme Programming principles). The first thing is to copy another test program, say `common/list_test.[ch]`. Remove the test code and change the names of all the definitions and strings to correspond to the new test program, leaving the following template files. Firstly `histogram_test.h` should like like this:

```
/**
```

```
 * File:          $RCSfile: testing.tex,v $
 * Module:        Histogram test program
 * Part of:       Gandalf Library
 *
 * Revision:      $Revision: 1.3 $
 * Last edited:   $Date: 2003/02/24 10:06:15 $
 * Author:        $Author: pm $
 * Copyright:     (c) 2002 YOUR INSTITUTION
 *
 * Notes:         Tests the histogram functions
 */

/* This library is free software; you can redistribute it and/or
   modify it under the terms of the GNU Lesser General Public
   License as published by the Free Software Foundation; either
   version 2.1 of the License, or (at your option) any later version.

   This library is distributed in the hope that it will be useful,
   but WITHOUT ANY WARRANTY; without even the implied warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
   Lesser General Public License for more details.

   You should have received a copy of the GNU Lesser General Public
   License along with this library; if not, write to the Free Software
   Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
*/

#include <gandalf/TestFramework/cUnit.h>

#ifndef CUNIT_HISTOGRAM_TEST_H
#define CUNIT_HISTOGRAM_TEST_H

cUnit_test_suite * histogram_test_build_suite(void);

#endif
```

Make sure you keep the header and license sections. The `histogram_test.c` file should be:

```
/**
 * File:          $RCSfile: testing.tex,v $
 * Module:        Histogram test program
 * Part of:       Gandalf Library
 *
 * Revision:      $Revision: 1.3 $
 * Last edited:   $Date: 2003/02/24 10:06:15 $
 * Author:        $Author: pm $
 * Copyright:     (c) 2002 YOUR INSTITUTION
 *
 * Notes:         Tests the histogram functions
 */

/* This library is free software; you can redistribute it and/or
   modify it under the terms of the GNU Lesser General Public
   License as published by the Free Software Foundation; either
   version 2.1 of the License, or (at your option) any later version.
```

```
   This library is distributed in the hope that it will be useful,
   but WITHOUT ANY WARRANTY; without even the implied warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
   Lesser General Public License for more details.

   You should have received a copy of the GNU Lesser General Public
   License along with this library; if not, write to the Free Software
   Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
*/

#include <stdlib.h>
#include <stdio.h>

#include <gandalf/TestFramework/cUnit.h>
#include <gandalf/statistics/histogram_test.h>
#include <gandalf/statistics/histogram.h>

static Gan_Bool setup_test(void)
{
   printf("\nSetup for histogram_test completed.\n\n");
   return GAN_TRUE;
}

static Gan_Bool teardown_test(void)
{
   printf("\nTeardown for histogram_test completed.\n\n");
   return GAN_TRUE;
}

/* Tests all the histogram functions */
static Gan_Bool run_test(void)
{
   return GAN_TRUE;
}

#ifdef HISTOGRAM_TEST_MAIN

int main ( int argc, char *argv[] )
{
   /* set default Gandalf error handler without trace handling */
   gan_err_set_reporter ( gan_err_default_reporter );
   gan_err_set_trace ( GAN_ERR_TRACE_OFF );

   setup_test();
   if ( run_test() )
      printf ( "Tests ran successfully!\n" );
   else
      printf ( "At least one test failed\n" );

   teardown_test();
   gan_heap_report(NULL);
   return 0;
}
```

```
#else

/* This function registers the unit tests to a cUnit_test_suite. */
cUnit_test_suite *histogram_test_build_suite(void)
{
   cUnit_test_suite *sp;

   /* Get a new test session */
   sp = cUnit_new_test_suite("histogram_test suite");

   cUnit_add_test(sp, "histogram_test", run_test);

   /* register a setup and teardown on the test suite 'histogram_test' */
   if (cUnit_register_setup(sp, setup_test) != GAN_TRUE)
      printf("Error while setting up test suite histogram_test");

   if (cUnit_register_teardown(sp, teardown_test) != GAN_TRUE)
      printf("Error while tearing down test suite histogram_test");

   return( sp );
}

#endif /* #ifdef HISTOGRAM_TEST_MAIN */
```

There are now three functions, `setup_test()`, `teardown_test()` and `run_test()` for you to fill with your test code. `setup_test()` should create any data structures to be used multiple times by the tests. Then `run_test()` performs the tests, and `teardown_test()` frees the memory allocated by `setup_test()`. You can leave `setup_test()` and `teardown_test()` blank if you like, and allocate & free the memory in `run_test()`. It is up to you.

The next stage is to add a rule in the package `Makefile.in` program to make your test program. Add `histogram-test` as a target to the `all:` line in `statistics/Makefile.in`. Then add the following lines to `statistics/Makefile.in`:

```
histogram-test:
        $(LIBTOOL) $(CC) -I$(TOPLEVEL)/.. $(CFLAGS) -DHISTOGRAM_TEST_MAIN histogram_test.c $(PATH_BU
```

Remember that there must be a tab character at the start of the `$(LIBTOOL)` line. Note the predefined symbol `HISTOGRAM_TEST_MAIN`. This is to make sure that the section of `histogram_test.c` with the `main()` function is compiled in. The other section of the code is for when the test functions are linked against the Gandalf test harness. For now, rerun `./configure` from the `gandalf` directory to recreate `statistics/Makefile` with the new rules, and make the test program with the commands:

```
cd statistics
make histogram-test
./histogram_test
```

(or `make all`). The tests should be designed so that if the data is successfully allocated and all the tests pass, `setup_test()`, `run_test()` and `teardown_test()` should all return `GAN_TRUE`. There is a special macro `cu_assert()`, which operates like `assert()` in the sense that it tests an expression and fails if zero is returned. In the `cu_assert()` either `GAN_TRUE` (one, success) is returned if the expression is non-zero, and `GAN_FALSE` (zero, failure) is returned if the expression is zero. In the latter case an error message is also printed, providing the line at which failure occurs. This provides a convenient shorthand for testing the results of the tests.

The next stage is to incorporate the test into the main Gandalf test harness. To do this, first edit `gandalf/TestFramework/Makef`
and add the following:

1. Add `histogram_test.o` to the `OBJS` list.
2. Add the line

```
HISTOGRAM_TEST_C  = $(TOPLEVEL)/statistics/hisogram_test.c
```

   to the list below the `OBJS` list.
3. Add the rule

```
histogram_test.o: $(HISTOGRAM_TEST_C)
        $(LIBTOOL) $(CC) -I$(TOPLEVEL)/.. $(CFLAGS) -c $(HISTOGRAM_TEST_C)
```

   (remember the tab character again before `$(LIBTOOL)`).

Now you will need to edit `TestFramework/cUnit.c`. Add the header file declaration

```
#include <gandalf/statistics/histogram_test.h>
```

among the other `#include` declarations. Find the line which has `#define maxAutoTests` in it and add one to the number you see there. You will also need to add a line

```
auto_tests[iIndex++] = "histogram_test";
```

in the list of similar lines below, and finally the lines

```
pSuite = cUnit_add_test_suite(auto_tests[iIndex++],
                              histogram_test_build_suite);
gan_list_insert_last(glstAutoSuiteList, pSuite);
```

at the corresponding place in the next set of similar lines. You will need to run `configure` again to recreate the `TestFramework/Makefile` file, and then typing

```
cd TestFramework
make
./cUnit -menu
```

should give you the extended menu of test programs with your new test as one of the options.

Bibliography

[1] Y. Bar-Shalom and T. E. Fortmann. *Tracking and Data Association.* Academic Press, 1988.

[2] Åke Björck. *Numerical Methods for Least Squares Problems.* SIAM Press, Philadelphia, PA, 1996.

[3] J. Canny. A computational approach to edge detection. *IEEE Trans. Pattern Analysis and Machine Intelligence*, 8(6):679–698, 1986.

[4] O.D. Faugeras. *Three-Dimensional Computer Vision.* MIT Press, 1993.

[5] M. Fleck. Perspective projection: The wrong imaging model. Technical Report 95-01, Computer Science, University of Iowa, 1994.

[6] C. J. Harris and M. Stephens. A combined corner and edge detector. In *Proc. 4th Alvey Vision Conf., Manchester*, pages 147–151, 1988.

[7] H. Li, M.A. Lavin, and R.J. Le Master. Fast hough transform: A hierarchical approach. *Computer Vision, Graphics and Image Processing*, 36:139–161, 1986.

[8] Q.-T. Luong and O.D. Faugeras. The fundamental matrix: theory, algorithms, and stability analysis. *International Journal of Computer Vision*, 17(1):43–76, 1996.

[9] D. W. Marquardt. An algorithm for least-squares estimation of nonlinear parameters. *Journal of the Society for Industrial and Applied Mathematics*, 11:431–441, 1963.

[10] P.F. McLauchlan, J.E.W. Mayhew, and J.P. Frisby. Stereoscopic recovery and description of smooth textured surfaces. *Image and Vision Computing*, 9(1):20–26, February 1991.

[11] X. Pennec and J.P. Thirion. A framework for uncertainty and validation of 3-d registration methods based on points and frames. *International Journal of Computer Vision*, 25(3):203–229, December 1997.

[12] W.H. Press, B.P. Flannery, S.A. Teukolsky, and W.T. Vetterling. *Numerical Recipes in C.* Cambridge University Press, 1988.

[13] C.C. Slama, C. Theurer, and S.W. Henriksen, editors. *Manual of Photogrammetry.* American Society of Photogrammetry, 1980.

[14] C.J. Taylor and D.J. Kriegman. Structure and motion from line segments in multiple images. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 17(11):1021–1032, November 1995.