Interpolation Routines

Alan Louis Scheinine, Senior Researcher, CRS4

CRS4
Centro di Ricerca, Sviluppo e Studi Superiori in Sardegna
Sesta Strada, Ovest
Zona Industriale Macchiareddu
09010 Uta (Cagliari) Italy

E-mail: scheinin@crs4.it

Contents

List of Figures

List of Tables

1 Basic Classes

This section describes simple classes that deal with vectors and matrices, or deal with polynomials used for interpolation, or utility classes.

1.1 StencilParams

The class StencilParams **declared in file** *Basic/stencil_params.hh* **associates a dimension and size for a given stencil tag. The tag can be an integer or the class** StencilSitesTag**. As an integer, the tag can have the follow values: 3 or 5 for one dimension, 9, 13 or 25 for two dimensions and 27, 33 or 57 for three dimensions.**

The class StencilSitesTag **is used to identify the tag that describes a stencil rather than the tag that describes the polynomical terms used for the interpolation function. Some algorithms use both kinds of tags so to avoid confusion the type descriptor is a class.**

1.2 StencilSites

The class StencilSites **declared in file** *Basic/stencil_sites.hh* **and implemented in the file** *Basic/stencil_sites.C* **creates the arrays**

```
ReadOnlyNumArray<signed char> stencil_sites_x;
ReadOnlyNumArray<signed char> stencil_sites_y;
ReadOnlyNumArray<signed char> stencil_sites_z;
```

which give lattice indices of a stencil in 1, 2, or 3 dimensions. This class inherits from StencilParams**. The constructor of the class has an argument that defines the specific stencil.**

```
StencilSites(int tag_in)
StencilSites(const StencilSitesTag& tag_in)
```

The integer tag that describes which stencil type happens to correspond to the size of the stencil: 3 or 5 for one dimension, 9, 13, 21 or 25 for two dimensions and 27, 33 or 57 for three dimensions.

In the same files *Basic/stencil_params.hh* and *Basic/stencil_params.C* are the declaration and implementation of ArbitrarySites in which the stencil is defined by arbitary positions in space (specified by floating point numbers). This class has not been tested.

1.3 StencilVector

The typedef StencilVector declared in file *Basic/stencil_vector.hh* is defined as follows

```
typedef  TNTVect<double>  StencilVector;
```

Though the array is called StencilVector, it can be used to hold any vector. Note, unlike an STL vector, resizing destroys the contents.

The array type is derived from the C++ templates TNT (Template Numerical Toolkit: Linear Algebra Module) *http://math.nist.gov/tnt* with some changes to increase the speed of simple operations that were implemented by Gassan Abdoulaev. The difference between the TNT class for vectors and the standard template library is that TNT class is optimized for numbers. The third edition of the book *The C++ Programming Language* describes a valarray for numbers but this is not implemented (or I could not find the declarations) in the 2.xx version of g++. The user can forget about the type

```
TNTVect<double>
```

by simply using

```
StencilVector_pointer newStencilVector();
StencilVector_pointer newStencilVector(int n);
```

which are implemented in the file *Basic/stencil_vector.hh*.

In some cases, this class is used as coefficients for the polynomial terms of the interpolation function. While the size of StencilVector is arbitrary, the actual use as stencil coefficients assumes that the size is fixed, in analogy with the size of the position template and polynomial terms being fixed.

1.4 Bspline

The class BsplineEquations and the class Bspline declared and implemented in file *Basic/bspline.hh* deal with a specific type of function that has a shape similar to a Gaussian function. These functions are useful for interpolation.

The class BsplineEquations implements the functions

```
static inline double bspline2(double x)
static inline double bspline2_derivative(double x)
static inline double bspline2_integral(double x, double y)
static inline double bspline2_integral(double x)
static inline double bspline3(double x)
static inline double bspline3_derivative(double x)
static inline double bspline3_integral(double x, double y)
static inline double bspline3_integral(double x)
static inline double bspline4(double x)
static inline double bspline4_derivative(double x)
static inline double bspline4_integral(double x, double y)
static inline double bspline4_integral(double x)
```

as well as functions that represent the sum of two neighboring bsplines

```
static inline double bsplinepair2(double x)
static inline double bsplinepair2_derivative(double x)
static inline double bsplinepair2_integral(double x, double y)
static inline double bsplinepair2_integral(double x)
static inline double bsplinepair3(double x)
static inline double bsplinepair3_derivative(double x)
static inline double bsplinepair3_integral(double x, double y)
static inline double bsplinepair3_integral(double x)
static inline double bsplinepair4(double x)
static inline double bsplinepair4_derivative(double x)
static inline double bsplinepair4_integral(double x, double y)
static inline double bsplinepair4_integral(double x)
```

The function bspline2() has polynomial terms up to second order, has four pieces and is non-zero for

```
0  <= |x| < 1/2
```

```
1/2 <= |x| < 3/2
```

whereas the function bspline3() has polynomial terms up to third order, has four pieces and is non-zero for

```
 0  <= |x| <  1
 1  <= |x| <  2
```

and finally the function bspline4() has polynomial terms up to fourth order, has six pieces and is non-zero for

```
 0  <= |x| < 1/2
1/2 <= |x| < 3/2
3/2 <= |x| < 5/2
```

The adjacent intervals in the above equations refer to the pieces.

The class Bspline implements the basis functions for a user-defined scale, user defined offset and one, two, or three dimensions. The two and three-dimensional functions are the products of one-dimensional bspline functions.

When the user calls the function

```
double bspline(const double* location) const
```

the argument to the standard bspline function is

```
BsplineEquations::bspline2((invr_scale_[i]*(a - center_[i]))
                           - (0.5*extended_[i]);
```

(or bspline3 or bspline4) for each dimension indexed by i where `invr_scale_[i]` is the inverse of the scale factors. The term *scale factors* are synonymous with the term *voxel size*. The order of approximation of the bspline can be different for each dimension. The voxel size and order of approximation is set by the arguments of the constructor whereas the position of the center is set by `setCenter()`.

The array `extended_[i]` is only used when the bspline is actually a bsplinepair. The value of `extended_[i]` is -1 if a stencil site is a lower edge and is $+1$ if a stencil site is an upper edge for dimension `i`.

The derivative can be calculated using the function

```
void bspline_derivative(const double* location,
                        double* d) const
```

where the array `d[]` **is the derivative in each direction.**

The integrated value of the bspline is calculated by

```
double bspline_integral(const double* box) const
```

where `box[0] = lower_x`, `box[1] = upper_x`, `box[2] = lower_y`, **etc. The average value within a box is calculated by**

```
double bspline_avg(const double* box) const
```

The function implementations are all in a header file so that the compiler can inline them.

1.5 StencilTerms

The class StencilTerms **declared in file** *Basic/stencil_terms.hh* **and implemented in file** *Basic/stencil_terms.C* **implements the polynomial functionals that take a position to a value. The implementation of a Taylor series can be found in** *Basic/stencil_terms.C* **whereas the implementation using bsplines is taken from class** Bspline.

The choice of functions for the basis is described by the class TermsTag **whose constructor is given by**

```
TermsTag(int tag_in, int basis_in)
```

For a Taylor expansion, the tag value of 3, 5, 9, 13, 27 or 33 describes the dimension and the order of approximation. For example, the number 27 refers to a 3x3x3 cube whereas 33 refers to a cube for which the template has extra points along the axes and therefore a higher order approximation is needed. The basis parameter should be 1 for Taylor expansion and 2 for overlapping gaussian bsplines.

For a Taylor expansion the constructor is either one of the following:

```
StencilTerms(int tag_in)
StencilTerms(const TermsTag& tag_in)
```

For bsplines, the constructors are

```
StencilTerms(int tag_in, double x)
StencilTerms(const TermsTag& tag_in, double x)
StencilTerms(int tag_in, double x, double y)
StencilTerms(const TermsTag& tag_in, double x, double y)
StencilTerms(int tag_in, double x, double y, double z)
StencilTerms(const TermsTag& tag_in, double x, double y, double z)
```

where the double precision numbers are the pixel or voxel sizes.

Public vectors that hold the results of function calls are

```
StencilVector terms;
StencilVector xterms;
StencilVector yterms;
StencilVector zterms;
```

The same result vector $terms$ is used for two different types of calculations implemented the functions

```
void make_point_terms(const double *location)
void make_ntgrl_terms(const double *box)
```

It is assumed that only one of the two functions will be called for a given dataset. The polynomial terms calculated here correspond to intensity values when multiplied by the appropriate coefficients calculated elsewhere. The function `make_point_terms(const double *location)` **gives an intensity value at the point $location$[] when an inner product is taken between the $terms$ and the coefficients. The function** `make_ntgrl_terms(const double *box)` **gives an average intensity value within the voxel box[] when an inner product is taken between the $terms$ and the coefficients.**

For interpolation, if we take the view that a given voxel of a dataset represents the average of underlying function (rather than the value of the function at the center of the voxel) then the call `make_ntgrl_terms()` **should be used.**

The arrays $xterms$**,** $yterms$ **and** $zterms$ **are assigned values when the function**

```
void make_gradient_terms(const double *location)
```

is called.

The choice of approximation method depends on a tradeoff between speed and accuracy. The choice of approximation method, made during construction of the class StencilTerms**, is then transparent to the user for the function calls**

```
void make_point_terms(const double *location)
void make_ntgrl_terms(const double *box)
void make_gradient_terms(const double *location)
```

1.6 ImageBase

The class ImageBase **declared in file** *Basic/image_base.hh* **does not contain a block of data that could represent a field or image. It only contains the basic information, in particular, the dimension (1, 2, or 3), the length (bounds) and the pixel/voxel size (aspect). These values are set using the functions**

```
virtual void set_Dimension(unsigned char dimension)
virtual void set_Bounds(const int *lattice_bounds)
virtual void set_Aspect(const double *aspect_ratio)
```

The class ImageBase **also contains static constants such as**

```
static const int ImageBase::IMAGE_DATA_TYPE_UNSIGNED_SHORT =  5;
```

which can be used to determine the numerical type of data in a derived class when a base class pointer is used that does not specify the numerical type of the image data.

In the namespace BasicDataType the function

```
int BasicDataType::toDataType(const char* type_in)
```

returns the data type integer tag when the argument is a character string that describes C type such as "unsigned short". In addition, the data type integer tag can be obtained using a templated function such as

```
template<> int BasicDataType::toDataType<double>()
```

These templates, implemented in the file *Basic/image_base.C* simplify the construction of a typed field, that is, a typed field that is defined by a templated class can return the correct data type tag by returning

```
int BasicDataType::toDataType<T>()
```

where T is the template parameter.

Two important functions defined in the class ImageBase are

```
bool find_Indices_Nearest(const double* const coord,
                          int* const lattice_site,
                          double* const location)
bool find_Indices_Nearest(const double* const coord,
                          int* const lattice_site,
                          double* const location,
                          const signed char* field_pos)
```

Since the length in each direction and the pixel or voxel size are data of this class, given a point coord[] it is possible to say which box of the lattice contains the point. The output variable lattice_site[] specifies the box and the output variable location[] gives the displacement from the middle of the box. The second function has an input variable field_pos[] which can have a value of -1, 0 or 1 for each dimension. The most common cases correspond to either -1 or 0. The value -1 corresponds to the coordinate system beginning at an edge and having only positive values, whereas, the value 0 corresponds to the coordinate system being centered on the lattice.

1.7 StencilMatrix

The class StencilMatrix **and the class** ArbitraryMatrix **are declared in file** *Basic/stencil_matrix.hh* **and implemented in file** *Basic/stencil_matrix.C*. **The class** ArbitraryMatrix **is for a template of arbitrary positions but has not been tested.**

The class StencilMatrix **inherits from** StencilSites, **though one could argue that it would be better to use a** HAS-A **rather than** IS-A **relationship between the two classes. For a lattice with uniform spacing, there is a unique matrix that multiplies intensity values at a stencil of sites to obtain weights for the polynomials (basis functions) used to interpolate the intensity values at any arbitrary position near the center of the stencil. The class** StencilMatrix **contains the data**

```
vector<double> _col_by_col_matrix;
vector<double> _row_by_row_matrix;
```

which is the interpolation matrix. Initially the class had the data

```
TNT::Matrix<double> _tnt_matrix;
```

but the matrix is read much more often than written so the linear vectors (unrolled matrix) are more efficient.

The matrix is constructed by taking the inverse of a matrix constructed from the basis functions. Public functions of this class include the following

```
int fill_matrix(const double *position,
                const ImageBase *field,
                int value_mode)
int take_inverse()
void mat_vec_mult(StencilVector_const_ref stencil_coefs_in,
                  StencilVector_ref stencil_coefs_out) const
void vec_mat_mult(StencilVector_const_ref stencil_coefs_in,
                  StencilVector_ref stencil_coefs_out) const
virtual int generateLatticeInverse(const ImageBase* lattice,
                                   int value_mode)
int numSites() const
```

but in practice only generateLatticeInverse() **is used outside this class. It is not necessary to have an actual lattice, just the simple base class** ImageBase **that**

contains the voxel size (since the aspect ratio has an effect on the interpolation). The degree of the polynomial approximation and the use of Taylor expansion or Gaussian weights (bsplines) is specified in the constructor

```
StencilMatrix(int tag_in, int basis_in) :
  StencilSites(tag_in), _terms_tag(tag_in, basis_in)
```

the polynomial degree implied by *tag_in* and the Taylor expansion or bsplines specified by *basis_in*. In the function call

```
C<generateLatticeInverse(const ImageBase* lattice, int value_mode)>
```

the *value_mode* can have either of two values: POINT_VALUE_MODE or BOX_VALUE_MODE. The first means that the intensities used as input for interpolation refer to specific points whereas the second means that the intensities refer to the average value over a box centered at the point. One way to describe the meaning of these two modes is to say that the interpolation polynomials (combined with their coefficients) should reproduce the actual values of stencil points at specific points in the former case, while in the latter case, the interpolation polynomials (combined with their coefficients) should reproduce the average of the area/volume of a box. Stated another way, the intensity values of the lattice are associated with either discrete points or are averages over a pixel/voxel.

1.8 Constants

The file *Basic/interpol.hh* defines some constants that describe the mode of interpolation. The style of the file may be interesting for other applications because it shows how to define the same constants for Fortran, C and C++.

1.9 StencilHandle

The class StencilHandle declared and implemented in file *Basic/stencil_handle.hh* is a reference counting wrapper for a pointer. It takes control of a pointer and deletes the pointer when the reference count goes to zero. (It is widely used in CORBA.) Of course, reference counting does not work for circular references, nonetheless, this class is often useful. One use of *handle* types is the automatic deletion of objects created with *new*. This is especially useful when the a function can exit

from various points. The StencilHandle should be declared in the function, rather than created with *new*. Then it will be placed on the stack and deleted when the function exits. The StencilHandle should be assigned a pointer of something that one wants to delete automatically when exiting a function. When StencilHandle is deleted, it calls *delete* on the pointer that it holds, unless this StencilHandle has assigned its value to another StencilHandle. In the latter case, the other StencilHandle continues to hold the pointer. For a given pointer being handled by an StencilHandle, its associated reference count is a heap variable so that all StencilHandle instantations that handle the same pointer use the same reference count.

2 Linear Algebra Classes

As well as defining interpolation routines, this package defines a field (an image). In order to do arithmetic on fields without making reference to the primitive type of the field variable, abstract types have been defined for scalars and vectors for linear algebra. These class were originally developed for a parallel program, so this abstraction is at a sufficiently high level that the implementation class could actually be a distributed vector. However, the distributed vector classes have not been put into this package.

Most of the classes of the Interpol package concern the representation of a scalar-valued field. The structure of classes is rather complex but the end result should be simple for the user. Two aspects for which the classes are simple for the user (though complex in the Interpol implementation) are the following.

The field that represents an image can have a variety of numerical types, generalized by using a templated class. However, nearly all of the functions of a field are virtual and are defined in a base class that does not have a type. My personal preference is to avoid templated classes, perhaps simply due to the bad performance of compilers of the past for which templates sometimes caused internal compiler errors. Templated classes lead to more templated class because of the type checking of C++. That is, if a variable is an instantation of a templated class, then a user's class must also be templated in order to have functions generalized for use with all the potential types of the templated variable. In contrast, the Interpol package gives the developer a non-templated class for use in his or her classes. The details of the implementation of a field classes inside the Interpol package are complicated because the virtual functions are repeated in several places. A further example of the complexity is that there is a class just for assignment of fields, ImageFieldAssign. This class does type conversion and implements the operators += , -= , *- , and

/= , as well as assignment. Moreover, these assignment operators convert between field types using the class LimitRange to avoid compiler complaints and run-time errors. The program developer does not need to be concerned about details such as the ImageFieldAssign class.

A second aspect of the field classes which is complicated for someone studying the source code but should simplify the use for a developer is that the fields inherit from LinAlgVector. A LinAlgVector was initially developed for the Nast++ project, to be combined with a LinAlgMatrix. In the case of Nast++, the LinAlgVector hides both the type of the field variable and also the distribution among processors for a parallel application. For the Interpol package, LinAlgVector just hides the base type of the field variable. The key point of LinAlgVector is that it implements arithmetic on vectors, including cases in which dummy variables are needed by the compiler. The problem is a technical detail of C++ for which a following brief description may not be clear to all readers. The operation A += B does not require a copy function. On the other hand, A = B + C needs to generate a temporary for (B + C) that does not modify B nor C. The consequence is that a copy function is needed. But if A, B and C are bases classes then they to not contain the field array, since the base class has no type. Typically, A, B, and C would be references to the derived classes that contain the field of a specific type. But an actual copy function that returns a temporary dummy can only return the base type, thereby loosing the field array. A variable whose type is the class LinAlgVector can be either a reference (or pointer) to a derived type or if the variable is actually a LinAlgVector, then it contains a pointer to a derived type. The latter case allows returning a copy of the base class without loosing the array that contains the field. The added complexity is that derived classes need to define all the arithmetic operations defined as virtual functions in LinAlgVector. The advantage for the user is that an algorithm of vector arithmetic can be described in terms of LinAlgVector (and LinAlgScalar, which encapsulates a scalar without specifying the type) and the algorithm can then be applied to any field that is derived from LinAlgVector.

The subdirectory *LinAlg* contains some classes not related to linear algebra. This subdirectory is the first compiled so it has everything simple used by other classes.

2.1 LinAlgVector

The classes LinAlgScaler and LinAlgVector declared in file *LinAlg/lin_alg_vector.hh* implement scalars and vectors that can be used in linear algebra. A linear algebra scalar of a specific type is declared as

```
template <class T>
class LinAlgScalarTyped : public LinAlgScalar
```

The historical origin of LinAlgVector resides in the parallelization by Alan Scheinine of a linear algebra solver and fluid dynamic solver written by Gassan Abdoulaev, a C++ program called Nast++.

The original Nast++ used templates for classes that contained various interative solvers. Which class could be used could be found by compiling and seeing if all operations required by a function could be found in the class given as a template parameter. The Nast++ program was gradually modified to use virtual functions of a base class, instead of templates, because the base class can serve to rigorously declare which functions need to be implemented for the iterative solvers. The actual implementation of LinAlgVector has solved one interesting problem that often presents itself in arithmetic expressions. For example `X = A + (B + C)` requires a temporary for `(B + C)` because none of the variables can hold the temporary value. (In contrast, for `X = A + B` the variable X can hold the result of `A + B` .) This temporary requires a copy of a return value but copies from a base class can only give a base class. The solution will not be described in the document. The overall result is that LinAlgVector as a base class has few restrictions on the type of arithmetic statement in which it can appear. For Nast++ the actual implementation can include communication between parallel processes on different computers, but with regard to this interpolation package it is used for describing arithmetic on fields (images) without regard to the underlying primitive numerical type.

2.2 Number

The classes Number and NumberTyped declared in file *LinAlg/number.hh* and implemented in the file *LinAlg/number.C* are similar to the classical example used in many introductory C++ textbooks. A Number is a wrapper for any primitive numerical type for which arithmetic can be defined. The class NumberTyped is defined as follows

```
template <class T>
class NumberTyped : public Number
```

Normally the programmer would use a pointer or reference to Number which is actually a NumberTyped. The class Number inherits from the class LinAlgScalar and is primarily motivated by the use of LinAlgVector.

The class Number is not of fundamental importance for implementing a generalized linear algebra but is convenient when a typeless scalar is needed. A Number differs from a LinAlgScalar in that a Number does conversion to and from numerical types.

2.3 LimitRange

The class templated class LimitRange declared in file *LinAlg/limit_range.hh* and implemented in the file *LinAlg/limit_range.C* is used for type conversions, in particular, for converting all values of a field. Though it would be unwise to converted a signed field to an unsigned field, by using this class the results will at least be defined. A second purpose is to avoid compiler warnings. In particular, the *Field* classes provide conversion between all types of fields (of numerical primitive types). Before the class LimitRange was introduced, the compilation created a large number of warnings.

2.4 Vector1, Vector2, Vector3

The classes Vector1, Vector2, Vector3, LinAlgVectorSpace and VecSpecificDim are declared in file *LinAlg/vector123.hh* and implemented in *LinAlg/vector123.C*. The former three are simple one, two, and three-component vectors. In the series of books *Graphics Gems* there are defined simple structures (with similar names) for one, two, and three-dimensional points; these classes serve the same purpose. The latter two classes, declared as follows

```
class LinAlgVectorSpace : public virtual LinAlgVector
template <class T>
class VecSpecificDim : public virtual LinAlgVectorSpace
```

put the simple vectors into the framework of LinAlgVector.

2.5 TNTVect

The class TNTVect declared and implemented in file *LinAlg/tnt_vect.hh* has the following class declaration

```
template <class T>
class TNTVect : public Vector<T>
```

where Vector **is a class of TNT (Template Numerical Toolkit) and** TNTVect **includes some changes, implemented by Gassan Abdoulaev, to increase the speed of simple operations.**

One useful note, the `newsize()` **function of TNT destroys the original data, unlike the Standard Template Library (STL)** `resize()` **function which saves what can be saved.**

2.6 ReadOnlyNumArray

The class ReadOnlyNumArray **declared and implemented in file** *LinAlg/read_only_num_arra* **is a templated class that has a TNTVect vector. To protect the vector from being changed the function** `setRO()` **can be called.**

2.7 Timer

The class Timer **declared and implemented in file** *LinAlg/timer.hh* **implements an accumulator for the processor time of a program, implemented by calling the C function** `clock()`**. Each instantation of** Timer **is an independent stopwatch.**

2.8 CastToSelfType

The global function `cast_to_self_type` **defined in namespace** CastToSelfType **and implemented in file** *LinAlg/cast_to_self_type.hh* **is part of a trick needed when a copy must be made of a base class. It is used, for example, in** LinAlgVector **and in** ImageFieldTyped**. For example, the assignment operator in the templated class**

```
ImageFieldTyped<T> { // ...


// virtual method of ImageField
ImageField& operator=(const ImageField& object_in) {
  return operator=(cast_to_self_type(object_in));
}
```

The assignment operator is a virtual function of ImageField and therefore both the input argument and the return value do not specify the type specified by the template. By casting to the templated type (which is the derived type) the correct `operator=` of the derived type is chosen.

2.9 Matrix Inversion

The function

```
int invert_matrix(double *matrix,
                  int size,
                  int ifdebug)
```

in the file *LinAlg/invert_matrix.c* inverts the square matrix given as the first argument.

In the file *LinAlg/invert.hh* are declarations of C type for functions compiled with Fortran for the matrix inversion.

3 Image Field

The description of the classes defined in the directory *Field* will not be presented in hierachical order. The first class described, ImageFieldTyped, is the first member of the hierarchy that brings together a large number of virtual functions and actual implementions. By describing this class first, the reader can understand the goal of the hierarchy of classes. The next group of classes described will be the those classes from which ImageFieldTyped is derived. After which, the classes that implement interpolation will be described.

3.1 ImageFieldTyped

The templated class ImageFieldTyped defined in file *image_field_typed.hh* defines a field (image) of a given numerical type. The class declaration is

```
template <class T>
```

```
class ImageFieldTyped :
  public ImageFieldAssign<T>,
  virtual public ImageField
```

The inheritance combines both a series of classes that declare virtual functions and a series of classes that implementation functions or contain data. This class does not contain data. Most of the functions of this class concern the implementation of arithmetic operations. Other functions of this class are implemented using static functions defined in the class ImageFieldAlgorithms. There are too many functions in ImageFieldTyped to list them all, just some examples will be given.

This class inherits from ImageFieldAssign, which defines various assignment operators. Because this class does not contain data, it can use the ImageFieldAssign assignment operator to convert between various templated versions of this class. The implementation is shown below.

```
template <class U>
ImageFieldTyped<T>& operator=(const ImageFieldBase<U>& object_in) {
  ImageFieldAssign<T>::operator=<U>(object_in);
  return *this;
}
```

This class implements the virtual functions of LinAlgVector. The direct base class ImageField inherits from LinAlgVector. These functions are mostly arithmetic operations. An example of an implementation is the following implementation of the operator +=

```
template<class U>
inline ImageFieldTyped<T>& operator+=(const ImageFieldTyped<U>& A) {
  ImageFieldAssign<T>::operator+=<U>(A);
  return *this;
}
```

which takes advantage of the actual implementation in ImageFieldAssign. The implementation of the analogous virtual function of LinAlgVector then takes advantage the += of this class.

```
inline LinAlgVector& operator+=(const LinAlgVector& lav) {
  return operator+=(cast_to_self_type(lav));
}
```

The class ImageField, as well as inheriting virtual functions from LinAlgVector, declares a large number of abstract virtual functions. For example, to get individual pixels as well as converting the type (implemented using LimitRange) there are declared in ImageField functions such as

```
virtual short getImage_short(int ix) const = 0;
virtual short getImage_short(int ix, int iy) const = 0;
virtual short getImage_short(int ix, int iy, int iz) const = 0;
```

and similar for putting a pixel/voxel value.

```
virtual void setImage(int ix, short v) = 0;
virtual void setImage(int ix, int iy, short v) = 0;
virtual void setImage(int ix, int iy, int iz, short v) = 0;
```

The number of getImage and setImage functions is about 70.

Other virtual functions of ImageField, such as

```
virtual ImageField* new_proj_x_avg(int lower, int upper) = 0;
virtual ImageField* new_proj_z_max(int lower, int upper) = 0;
virtual ImageField* new_cropped(const int *lattice_box) = 0;
virtual ImageField* new_imbedded(const int *lattice_box,
                                 double background) = 0;
virtual ImageField* new_diffused(double diff_coef,
                                 int num_iters) = 0;
```

are implemented in ImageFieldTyped by calling static functions defined in ImageFieldAlgorithms. The static functions in ImageFieldAlgorithms all use ImageField, that is, no type needs to be specified for the field for used in the algorithms of ImageFieldAlgorithms.

In most cases, the pointer to a field class can be of type ImageField with calls to virtual functions being implemented by functions of ImageFieldTyped. The use of LinAlgVector as the base type for pointers is more abstract than is necessary for this interpolation package. The original motivation for having LinAlgVector as a base class comes from a parallel program in which there were several very different implementations for vectors (distributed vectors, local vectors and local vectors composed of blocks of vectors). The use in the package is motivated by the

fact that LinAlgVector has a reasonably complete set of declarations of arithmetic operators for vectors, so it serves as a useful prototype.

There are a group of functions for which there are pairs of functions with similar names, such as,

```
getDimension() and get_Dimension()
find_indices_nearest() and find_Indices_Nearest()
```

with identical implementations, in fact the former calls the latter. The reason is that for class ImageFieldTyped there is a convergence of two lines of class hierarchies. One hierarchy contains predominantly implementations of functions, while the other hierarchy contains predominantly declarations of abstract classes. However, the hierarchy of implementations also contains some abstract classes because during the development of the classes the structure of the more highly derived classes was not certain. For the hierarchy in which the implementations predominate, it can be useful to write programs that take advantage of virtual functions without arriving at the highly derived level of ImageFieldTyped. To give a specific example, the class ImageBase is so primitive that it does not contain a field, only the size of the lattice and the size of each pixel/voxel. Yet that is enough for implementing the function `find_Indices_Nearest()` in the class ImageBase, a function which is declared virtual. The class ImageBase is part of the hierarchy that contains implementations, for example, there is the following inheritance

```
template<class T>
class ImageFieldBase : public ImageBase
```

where the class ImageFieldBase contains the actual field data in the data member

```
TNTVect<T> image
```

The other hierarchy of abstract classes contains a similar declaration in the class ImageField

```
virtual bool find_indices_nearest(...) = 0;
```

Because ImageFieldTyped inherits both, the names are different. At one point in the develop of the package, the abstract virtual function in ImageField

and the virtual function in ImageBase had the same names, both being inherited by ImageFieldTyped. The GNU g++ did not complain and the test programs worked correctly. Nonetheless, the situation seems ambiguous and likely to invoke an internal compiler error (even if technically legal) so the names were changed to avoid ambiguity.

In addition, there are virtual functions that are actually implemented in FieldInterpolAlgorithms with function calls in this class simply calling the same function in FieldInterpolAlgorithms. The functions in the class FieldInterpolAlgorithms never use ImageFieldTyped explicitly, the algorithms rely on the virtual functions of ImageField. For this reason, the algorithms are treated as a separate class, in accord with the concept of an algorithm as logically abstract.

3.2 ImageField

The classes ImageField and NewImageField defined in files *Field/image_field.hh* and *Field/image_field.C* declare abstract functions. A partial list will be given to show the breadth of the class.

```
ImageField()
ImageField(const LinAlgVector& lav)
[ functions implemented in ImageBase, such as ]
virtual void setBounds(const int *lattice_bounds) = 0;
virtual int getBounds(int i) const = 0;
virtual void setAspect(const double *aspect_ratio) = 0;
virtual double getAspect(int i) const = 0;
virtual void setDimension(unsigned char dimension) = 0;
virtual unsigned char getDimension() const = 0;
virtual const int* const getBounds_array() const = 0;
virtual const double* const getAspect_array() const = 0;
virtual bool find_indices_nearest(const double* const coord,
                                  int* const lattice_site,
                                  double* const location) = 0;
virtual bool find_indices_nearest(const double* const coord,
                                  int* const lattice_site,
                                  double* const location,
                                  const signed char* field_pos) = 0
virtual const ImageBase* getImageBase() const = 0;
[ other methods include ]
virtual int getDataType() const = 0;
```

```
virtual int toDataType(const char* type_in) = 0;
virtual ImageField* newImageField() const = 0;
virtual ImageField* newImageField(...) const = 0;
virtual ImageField* cloneImageField() const = 0;
virtual Number getImage_Number(int ix) const = 0;
virtual Number getImage_Number(int ix, int iy) const = 0;
virtual Number getImage_Number(int ix, int iy, int iz) const = 0;
virtual char getImage_char(int ix) const = 0;
virtual char getImage_char(int ix, int iy) const = 0;
virtual char getImage_char(int ix, int iy, int iz) const = 0;
[ etc. other types ]
virtual int getImage_int(int ix) const = 0;
virtual int getImage_int(int ix, int iy) const = 0;
virtual int getImage_int(int ix, int iy, int iz) const = 0;
virtual float getImage_float(int ix) const = 0;
virtual float getImage_float(int ix, int iy) const = 0;
virtual float getImage_float(int ix, int iy, int iz) const = 0;
[ etc. other types ]
virtual void setImage(int ix, Number v) = 0;
virtual void setImage(int ix, int iy, Number v) = 0;
virtual void setImage(int ix, int iy, int iz, Number v) = 0;
virtual void setImage(int ix, char v) = 0;
virtual void setImage(int ix, int iy, char v) = 0;
virtual void setImage(int ix, int iy, int iz, char v) = 0;
[ etc. other types ]
virtual void setImage(int ix, unsigned short v) = 0;
virtual void setImage(int ix, int iy, unsigned short v) = 0;
virtual void setImage(int ix, int iy, int iz, unsigned short v) = 0;
virtual void setImage(int ix, float v) = 0;
virtual void setImage(int ix, int iy, float v) = 0;
virtual void setImage(int ix, int iy, int iz, float v) = 0;
[ etc. other types ]
virtual ImageField* new_proj_x(int lower, int upper, int mode) = 0;
virtual ImageField* new_proj_y(int lower, int upper, int mode) = 0;
virtual ImageField* new_proj_z(int lower, int upper, int mode) = 0;
virtual ImageField* new_proj_x_avg(int lower, int upper) = 0;
virtual ImageField* new_proj_y_avg(int lower, int upper) = 0;
virtual ImageField* new_proj_z_avg(int lower, int upper) = 0;
virtual ImageField* new_proj_x_max(int lower, int upper) = 0;
virtual ImageField* new_proj_y_max(int lower, int upper) = 0;
virtual ImageField* new_proj_z_max(int lower, int upper) = 0;
virtual ImageField* new_cropped(...) = 0;
virtual ImageField* new_imbedded(...) = 0;
```

```
virtual ImageField* new_diffused(...) = 0;
```

Since the functions are abstract, one may wonder why the file *image_field.C* is needed. This file contains the implementation of

```
ImageField* NewImageField::newImageField(int image_data_type ...
ImageField* NewImageField::newImageField(int image_data_type ...
```

which generates pointers whose actual type is ImageFieldTyped.

This class is not templated, thereby giving the advantage of having a unique base class for the virtual functions. For every type of primitive number for which the function `limit_range()` is defined, there is a corresponding indexed get and set for field values.

Using an instantation of a derived, expressed as a pointer to this base class, a new instantation of the derived class can be constructed using this base class as a prototype. The methods are the following.

```
virtual ImageField* newImageField() const = 0;
virtual ImageField* newImageField(int dimension,
                                  const int *lattice_bounds,
                                  const double *aspect_ratio) const = 0;
```

The same idea, but with field values set equal to the prototype can be obtained using the following.

```
virtual ImageField* clone() const = 0;
```

For interpolation, the following virtual functions are defined:

```
virtual FieldInterpol* new_extend_by_two() const = 0;
virtual int make_stencil_rhs(StencilVector_ref rhs,
                             StencilSites& sites,
                             const int* lattice_site) const = 0;
virtual int check_template_with_field(int dimension_must_be,
                                      const int* lattice_site,
                                      int extent) const = 0;
```

```
virtual FieldInterpol* new_by_interpol(const ImageBase& grid,
                                       const MapDef& mapping,
                                       int precision_level,
                                       const PrecisionChoice& pc) const = 0;
```

3.3 ImageFieldAlgorithms

The class ImageFieldAlgorithms defined in files *Field/image_field_algorithms.hh Field/image_field_algorithms.C* are algorithms implemented by static functions. Functions with the same name are declared as virtual in the class ImageField and the implementation in the class ImageFieldTyped is to call the function defined in ImageFieldAlgorithms. The idea is that ImageField and ImageFieldTyped have primarily an organizational role. The class ImageField declares abstract virtual functions and the class ImageFieldTyped is a templated class that uses only a header file so it should not be too complex to avoid long recompilations. Nowhere in the class ImageFieldAlgorithms is the class ImageFieldTyped used, the algorithms are valid for any type. These algorithms are rather simple, namely

```
static ImageField* new_proj_x(int image_data_type,
static ImageField* new_proj_y(int image_data_type,
static ImageField* new_proj_z(int image_data_type,
static ImageField* new_proj_x_avg(int image_data_type,
static ImageField* new_proj_y_avg(int image_data_type,
static ImageField* new_proj_z_avg(int image_data_type,
static ImageField* new_proj_x_max(int image_data_type,
static ImageField* new_proj_y_max(int image_data_type,
static ImageField* new_proj_z_max(int image_data_type,
static ImageField* new_cropped(int image_data_type,
static ImageField* new_imbedded(int image_data_type,
static ImageField* new_diffused(int image_data_type,
```

The first functions create slices, the last three either crop to a smaller image, or imbeds a field into a larger field of with background of a given value, or creates a diffuse image. One purpose of an imbedded image is to allow a simple interpolation algorithm to scan over all sites of the original (inner) image using a stencil without special cases for the near-border regions. Diffusion has been put into this base class because it is considered to be generally useful for segmentation. The word *new* is used in the names to emphasize that space is allocated, a field is actually contructed. The use of an algorithm class is a scheme of organization that is more relevant for the class FieldInterpolAlgorithms in which the algorithms are complicated.

3.4 ImageFieldAssign

To continue the descriptions by working backwards from ImageFieldTyped **the class** ImageFieldAssign **defined in file** *image_field_assign.hh* **is a direct base class of** ImageFieldTyped**. The purpose is the implement type conversion of fields using the function**

```
LimitRange<T>::limit_range()
```

Both the assignment operator and the copy constructor can also convert between types.

The declaration of the class is

```
template <class T>
class ImageFieldAssign : public ImageFieldBase<T>
```

which shows us that the next link in the chain of inheritance is the class ImageFieldBase**.**

3.5 ImageFieldBase

The class ImageFieldBase **defined in file** *image_field_base.hh* **is part of the hierarchy of inheritance that contains data and implementations. This class contains the actual array of field data. This class is a templated class that inherits from** ImageBase

```
template<class T>
class ImageFieldBase : public ImageBase
```

For faster arithmetic, direct access is given to the class that hold the field data.

```
inline const TNTVect<T>& getImageArray() const
inline TNTVect<T>& getImageArray()
```

Two virtual functions of ImageBase **are redefined,**

```
void setDimension(unsigned char dimension)
void setBounds(const int *lattice_bounds)
```

so as to give an error message if called. The consequence is that the number of elements and shape of the array cannot be changed. On the other hand, the physical size of each pixel or voxel can be changed.

3.6 FieldInterpolAlgorithms

The classes FieldInterpolAlgorithms **and** FieldInterpolHelper **defined in files** *Field/field_interpol_algorithms.hh Field/field_interpol_algorithms.C* **implement algorithms for interpolation of field defined on a regular grid. These functions are**

```
static ImageField* new_extend_by_two(int image_data_type,
                                     const ImageField& fldntrpl);
static int check_template_with_field(int image_data_type,
                                     const ImageField& fldntrpl,
                                     int dimension_must_be,
                                     const int* lattice_site,
                                     int extent);
static int make_stencil_rhs(int image_data_type,
                            const ImageField& fldntrpl,
                            StencilVector_ref rhs,
                            StencilSites& sites,
                            const int* lattice_site);
static ImageField* new_by_interpol(int image_data_type,
                                   const ImageField& fldntrpl,
                                   const ImageBase& grid,
                                   const MapDef& mapping,
                                   int precision_level,
                                   const PrecisionChoice& pc);
static int point_to_voxel(int image_data_type,
                          const ImageField* field_pnt,
                          ImageField* field_out,
                          const PrecisionChoice& pc);
```

The function `new_extend_by_two()` **extends the field in every direction by two sites, using extrapolation. By doing this, a template centered anywhere within the original field will have reasonable field values for all pixels/voxels of the template.**

That is, for a template centered near the edge of the original field, at least on leg of the template will extend beyond the original field, so the template is used on the field generated by the function `new_extend_by_two()` **rather than used on the original field. This approach allows the use of the template over the entire field (the original width of the field) without having to use a special procedure near the edge.**

The interpolated field is generated by *new_by_interpol()*. **It is up to the user to delete the result of the interpolation when it is no longer needed, hence the word "new" in the function name. The calling argument** `int image_data_type` **describes the type of the generated field, since the return value of** `ImageField*` **is actually a pointer to the templated class** `ImageFieldTyped`. **The calling argument** `const ImageField& fldntrpl` **is the field to be interpolated. The calling argument** `const ImageBase& grid` **describes the pixel or voxel size and the bounds, that is, the number of pixels or voxels in each direction. The calling argument** `const MapDef& mapping` **is the mapping from the new field to the old field. This direction of the mapping may seem backwards but the given positions are on the new grid and those positions must be mapped to the input field. The calling argument** `int precision_level` **describes the technique of interpolation. Useful values are either PRECISION_LEVEL3 or PRECISION_LEVEL4. The latter involves over-sampling, which takes more time. The calling argument** `PrecisionChoice& pc` **is a way of specifying the stencil and type of basis function without specifying the exact stencil size. The public member PrecisionChoice.stencil_type can be 1 or 2 for any basis type and can be 3 for 2 or 3 dimensions with basis_type 2. The larger value indicates a larger stencil. The public member PrecisionChoice.basis_type can be 1 or 2, which corresponds to Taylor expansion or Gaussian weights, respectively.**

3.7 LinTrans

The class LinTrans **defined in files** *Field/lin_trans.hh Field/lin_trans.C* **contains one 3 by 3 matrix and one 3 component vector as public members.**

```
double _mat[9];
double _vec[3];
```

Methods of the class can be used to construct the matrix and to add to the existing transform.

```
void transform_identity();
```

```
void transform_translate(double x, double y, double z);
void pre_transform_translate(double x, double y, double z);
void post_transform_translate(double x, double y, double z);
void transform_scale(double x, double y, double z);
void pre_transform_scale(double x, double y, double z);
void post_transform_scale(double x, double y, double z);
void transform_rotate(double degrees,
                      double x_axis, double y_axis, double z_axis);
void pre_transform_rotate(double degrees,
                          double x_axis, double y_axis, double z_axis);
void post_transform_rotate(double degrees,
                           double x_axis, double y_axis, double z_axis);
```

The "post_transform" methods are applied to the vector _vec[] as well as the matrix _mat[], which is consistent with how the vector part is used in the class MapDef.

3.8 MapDef

The class MapDef defined in files *Field/map_def.hh Field/map_def.C* contains one method, which implements a linear mapping of a point.

```
int map_a_point(const ImageBase& field_in,
                const ImageBase& field_out,
                const double *location_in,
                double *location_out) const;
```

The class contains the linear mapping as a public data member

```
LinTrans linear;
```

The matrix of the linear transform is applied first, then the vector translation is applied.

The class contains two arrays that describe the coordinate system as being based on a corner or the middle of the grid.

```
signed char beg_field_pos[3];
signed char end_field_pos[3];
```

The case

```
beg_field_pos[3] = { -1 , -1 , -1 }
```

means that a corner is (0.0, 0.0, 0.0). Whereas the case

```
beg_field_pos[3] = { 0 , 0 , 0 }
```

means that the center is (0.0, 0.0, 0.0). For former is more typical of an image and the latter is useful for rotations. For interpolation, typically the mapping should be from the destination to the source because given destination points need to sample the source wherever they are put by the mapping.

3.9 ObjVar

The class ObjVar **declared and implemented in file** *Field/obj_var.hh* **is a reference counting wrapper for a pointer. It has the same implementation as** StencilHandle**. In the future it might inherit from a base class that is specific to fields.**

3.10 DataExplorer

The class DataExplorer **defined in files** *Field/data_explorer.hh Field/data_explorer.C* **reads and writes simple grids in IBM Open Data Explorer format. This class is not now used in order to avoid having the compilation of the interpolation routines depend upon the installation of Data Explorer.**

4 Utilities

Some higher-level algorithms are in the directory Utils.

4.1 RegridBrick

The class RegridBrick **defined in files** *Utils/regrid_brick.hh* **and** *Utils/regrid_brick.C* **has two public methods.**

```
static ImageField* new_brick_from_aspect(int image_data_type,
                                         const ImageField& fldin,
                                         const double* aspect_ratio,
                                         const PrecisionChoice& pc,
                                         int debug);
static ImageField* new_brick_from_bounds(int image_data_type,
                                         const ImageField& fldin,
                                         const int* lattice_bounds,
                                         const PrecisionChoice& pc,
                                         int debug);
```

The type of the output field (the return value) is specified by the calling parameter `int image_data_type`**. The word "new" in the function name is intended to remind the user that new space is allocated by the functions and that it is up to the user to delete the space when no longer needed. The input field is** `const ImageField& fldin`**. The overall size of the field is constant so if the user specifies** `const double* aspect_ratio` **then the number of pixels or voxels of the new field is changed accordingly. If the user specifies the** `const int* lattice_bounds` **then the size of each pixel or voxel is changed accordingly. The calling argument** `const PrecisionChoice& pc` **specifies the stencil and basis to be used for interpolation. For normal use, the value of** `int debug` **should be zero.**

4.2 GridSlice

The class GridSlice **defined in files** *Utils/grid_slice.hh* **and** *Utils/grid_slice.C* **generates two-dimensional slices from a three-dimensional field. The field that is sliced needs to be specified in the constructor**

```
GridSlice(const ImageField& field_in);
```

or by calling

```
void setSource(const ImageField& field_in);
```

When the field is specified, a wider field is created by extrapolation, using `FieldInterpolAlgorithms:: new_extend_by_two()`**.**

A slice is created by calling

```
ImageField* newImageField(const double* position,
                          const double* xdir,
                          const double* ydir,
                          const ImageBase& grid,
                          double thickness,
                          int precision_level,
                          const PrecisionChoice& pc,
                          int debug) const;
```

The center of the slice is given by `const double* position`**. The directions of the axes of the slice are given by** `const double* xdir` **and** `const double* ydir`**. The pixel size and number of pixels (aspect ratio and lattice bounds) is given by** `const ImageBase& grid`**. The value of each pixel can be an average over a given thickness, as specified by the calling argument** `double thickness`**. As described for** `FieldInterpolAlgorithms:: new_by_interpol()`**, the** `int precision_level` **describes the interpolation technique and <const PrecisionChoice& pc> describes the stencil and basis. For normal use, the value of** `int debug` **should be zero.**

Acknowledgments

This work has been partially supported by the Sardinian Regional Authorities.