The Jpp - JTools package

Maarten de Jong

2023-01-21 16:35:51

Abstract

The Jpp - JTools package is a Java inspired set of C++ interfaces and classes that can be used to make multi-dimensional interpolations of values and to histogram data in any number of dimensions.

1 Introduction

A common problem in computational physics is the interpolation of values in multiple dimensions. Several solutions exist for the one-dimensional case and sometimes two-dimensional [1]. In this note, a CPP software package is described that can be used to interpolate values in any number of dimensions. The available methods include polynomial and cubic spline interpolations. An interpolation can be considered as a functional operation, i.e. $x \rightarrow y$. In this, an abscissa value is mapped onto an ordinate value. In multiple dimensions, the functional operation becomes $(x_0, x_1, \ldots, x_{n-1}) \rightarrow y$. A *set* of abscissa values is now mapped onto an ordinate value. So, the idea is to use multi-dimensional *functional maps*. First, a general framework for collections of elements is presented. Then, a functional interface is defined. Specific implementations of this interface will be presented. It will be shown how this framework can be used to perform multi-dimensional interpolations and to histogram data in any number of dimensions.

2 Collections framework

In general, a collection is an object that represents a group of elements. The collections framework presented here is a unified architecture for representing and manipulating multi-dimensional collections. A set of interfaces is used to define the access mode and search methodology of collections. The various interpolation methods are based on template implementations of these interfaces. Each collection provides for an implementation of a STL map-like interface, namely:

```
template<class JKey_t, class JValue_t>
struct JMappableCollection
{
  typedef JKey_t                    key_type;
  typedef JValue_t                  mapped_type;

  virtual mapped_type& get(typename JClass<key_type>::argument_type key) = 0;

  mapped_type& operator[](typename JClass<key_type>::argument_type key) ;

  void put(typename JClass<key_type>   ::argument_type key,
           typename JClass<mapped_type>::argument_type value);
};
```

In this, the method `get()` defines the interface. The class `JClass` is used to define the proper argument (i.e. copy for primitive data types and otherwise constant reference). The map operator `[]` as well as the method `put()` actually are implemented by this interface. The elements in a collection consist (at least) of an abscissa value and an ordinate value. A basic implementation for an element is the data structure `JElement2D` which has the following policy methods:

```
template<class JAbscissa_t, class JOrdinate_t>
class JElement2D {
public:

  typedef JAbscissa_t                abscissa_type;
  typedef JOrdinate_t                ordinate_type;

  abscissa_type getX() const;

  const ordinate_type& getY() const;
  ordinate_type& getY();

  friend inline JReader& operator>>(JReader& in, JElement2D& element);
  friend inline JWriter& operator<<(JWriter& out, const JElement2D& element);
};
```

In this, `X` and `Y` correspond to the abscissa and ordinate values, respectively. The re-direct operators >> and << are implemented for binary I/O (see below). Any other data structure which provides for these policy methods can also be used in this framework.

2.1 Access to elements

The underlying container of a collection is the `std::vector<>` class[1]. The sequential access to the elements in a collection is based on the `STL` iterators and corresponding member methods of this class, namely:

```
template<typename _Tp, typename _Alloc = std::allocator<_Tp> >
class vector
{
  const_iterator begin() const;
  const_iterator end() const;

  const_reverse_iterator rbegin() const;
  const_reverse_iterator rend() const;

  iterator begin();
  iterator end();

  reverse_iterator rbegin();
  reverse_iterator rend();
};
```

These methods define bi-directional access of the elements in a collection.

[1] The `vector` class provides much faster I/O than `set` or `map`.

2.2 Ordering

The speed of any interpolation method critically depends on the ordering of elements. If elements are ordered, the search for an element can be performed by a binary search method which requires $O(\log(n))$ operations (compared to $O(n)$). Furthermore, for the interpolation one needs to evaluate the distance between elements. To this end, a collection requires one additional template argument `JDistance_t`. The default implementation of this template argument is defined by the template `JDistance` class.

```
template<class JAbscissa_t>
struct JDistance {

  typedef typename JClass<JAbscissa_t>::argument_type    argument_type;

  inline double operator()(argument_type first,
                           argument_type second) const
  {
    return second - first;
  }
};
```

In this, the template argument `JAbscissa_t` refers to the abscissa values of the elements in the collection. As can be seen from this implementation, the default distance between two elements in a collection evaluates to the numerical difference between the corresponding abscissa values. It should be noted that for other abscissa types (e.g date or time), the user can specify a designated distance operator.

The elements in a collection are sorted according the given `JDistance_t` template argument using the internal `JComparator` class. This class acts as a binary "less than" operator `()`. This operator is available as data member `getComparator`.

```
struct JComparator {

  inline bool operator()(const JElement_t& first,
                         const JElement_t& second) const
  {
    return this->getDistance(first.getX(), second.getX()) > 0.0;
  }

  inline bool operator()(const JElement_t& element,
                         typename JClass<abscissa_type>::argument_type x) const
  {
    return this->getDistance(element.getX(), x) > 0.0;
  }

  JDistance_t getDistance;
};
```

As can be seen from the presence of the second binary "less than" operator `()`, the search for an element in a collection is accordingly made.

2.3 Collection

A collection of elements is defined as follows:

```
template<class JElement_t,
         class JDistance_t = JDistance<typename JElement_t::abscissa_type> >
class JCollection
  public JMappableCollection<typename JElement_t::abscissa_type,
                             typename JElement_t::ordinate_type>,
{

  virtual ordinate_type& get(typename JClass<abscissa_type>::argument_type x);

  const_iterator lower_bound(typename JClass<abscissa_type>::argument_type x) const;
  iterator lower_bound(typename JClass<abscissa_type>::argument_type x);

  void configure(..);
  JCollection& add(..);
  JCollection& sub(..);
  JCollection& mul(..);
  JCollection& div(..);

  friend inline JReader& operator>>(JReader& in, JCollection& collection);
  friend inline JWriter& operator<<(JWriter& out, const JCollection& collection);
};
```

The method `get()` provides for an implementation of the `JMappableCollection` interface. The method `lower_bound()` uses the corresponding `STL` binary search method. The re-direct operators >> and << of a collection provide for an implementation of binary I/O by applying the corresponding operators to its elements. The various other methods can be used to fill the collection or modify the ordinate values. As the `JCollection` class implements the `JMappableCollection` interface, a collection can readily be filled using the following syntax.

```
JCollection< JElement2D<double, double> > buffer;

for (double x = -10.0; x <= 10.0; x += 1.0) {
  buffer[x] = ...;
}
```

2.3.1 Equidistant elements

In case the elements in a collection are equidistant, the search for an element can be done at constant time, i.e. $O(1)$ instead of $O(\log(n))$. The `JGridCollection` class can then be used.

```
template<class JElement_t,
         class JDistance_t = JDistance<typename JElement_t::abscissa_type> >
class JGridCollection :
  public JCollection<JElement_t, JDistance_t>
{
  const_iterator lower_bound(typename JClass<abscissa_type>::argument_type x) const;
  iterator lower_bound(typename JClass<abscissa_type>::argument_type x);
};
```

The `JGridCollection` class simply overwrites the `lower_bound()` methods so that the search is done by evaluation of the index based on the lower and upper abscissa values of the collection and the number of elements.

2.4 Bounds

The JAbstractCollection constitutes a simple interface to define a sequence of abscissa values for a collection. The JSet and JGrid classes can be used to define a collection with non-equidistant and equidistant abscissa values, respectively.

2.5 Maps

The JMap class can be used to extend a collection to one more dimension with non-equidistant abscissa values.

```
template<class JKey_t,
         class JValue_t,
         class JDistance_t = JDistance<JKey_t> >
class JMap :
  public JCollection<JElement2D<JKey_t, JValue_t>, JDistance_t>
{};
```

The JGridMap class can be used to extend a collection to one more dimension with equidistant abscissa values.

```
template<class JKey_t,
         class JValue_t,
         class JDistance_t = JDistance<JKey_t> >
class JGridMap :
  public JGridCollection<JElement2D<JKey_t, JValue_t>, JDistance_t>
{};
```

As can be seen from the definition of the JMap and JGridMap classes, maps are equivalent to collections. As a result, the implementation of a specific functionality thus applies to any dimension.

3 Functional collections

The functionality (i.e. the ability of function value evaluation) of a collection is defined by the template interface JFunctional.

```
template<class JArgument_t, class JResult_t>
class JFunctional
{
  typedef JArgument_t                                  argument_type;
  typedef JResult_t                                    result_type;

  virtual void compile() = 0;

  virtual result_type evaluate(const argument_type* pX) const = 0;

  static result_type getValue(const JFunctional& function, const argument_type* pX);

  :
};
```

The method `evaluate()` specifies a recursive method call. An implementation of this method thus applies to the evaluation of a function value in any dimension. The static method `getValue()` handles the proper termination of the recursive method call. The method `compile()` refers to a compilation of elements in the collection that may be required before evaluation of the function value.

The `JFunctional` interface also defines an interface for the handling of exceptions. For this, the interface `JExceptionHandler` is introduced. By default, the member method `action()` throws the exception.

```
template<class JArgument_t, class JResult_t>
class JFunctional
{
  :

  class JExceptionHandler
  {
    virtual result_type action(const JException& exception) const
    {
      throw exception;
    }
  };

  void setExceptionHandler(const JSupervisor& supervisor);
};
```

An exception handler is available which returns a default value in case the function value cannot be evaluated.

```
  class JDefaultResult :
    public JExceptionHandler
  {
  public:
    JDefaultResult(const result_type value);

    virtual result_type action(const JException& exception) const
    {
      return defaultResult;
    }

  private:
    result_type defaultResult;
  };
```

A different exception handler may be installed using the method `setExceptionHandler()`. In this, the argument `JSupervisor` refers to a simple place holder for the `JExceptionHandler` interface. This place holder is used to avoid unnecessary copies of the same exception handler.

3.1 Functions in one dimension

The `JFunction1D` interface refers to the functionality of a collection in one dimension.

```
template<class JArgument_t, class JResult_t>
class JFunction1D :
  public virtual JFunctional<JArgument_t, JResult_t>
{
public:

  enum { NUMBER_OF_DIMENSIONS = 1 };

  result_type operator()(const argument_type x) const
  {
    return this->evaluate(&x);
  }
};
```

The one-dimensional case of the general interface is specified by the function object operator `()`. The static method `getValue()` makes sure that the recursion process immediately terminates.

4 Interpolation in one dimension

In general, the interpolation methods are organised as template classes that derive from the `JCollection` or `JGridCollection` class and that implement the `JFunction1D` interface. The function object operator returns a value which is interpolated between neighbouring elements in the collection. For this, the method `lower_bound()` is used. By design, the `argument_type` of the function corresponds to the `abscissa_type` of the collection. Usually, but not necessarily, the `result_type` of the function corresponds to the `ordinate_type` of the collection. The list of available interpolation methods includes polynomial and cubic spline interpolation. It is interesting to note that *any* ordinate type can be interpolated provided it has arithmetic capabilities. This make it possible to simultaneously interpolate a multitude of values and to determine derivatives or partial integrals of the interpolating function on the fly.

4.1 Elements

The list of elements that can used in conjunction with the interpolation classes includes `JElement2D`, `JSplineElement2D` and `JSplineElement2S`. The `JElement2D` constitutes the basic element of most collections. The `JSplineElement2D` and `JSplineElement2S` extend the `JElement2D` data structure to include the second derivatives and partial integral values for specific spline interpolations. The additional data are evaluated by the method `compile()` of the corresponding class. It should be noted that the additional data are not included in the I/O of these data structures. Hence, the I/O is compatible with that of `JElement2D`.

4.2 Result types

Usually, the result of an interpolation is the same as the ordinate value of the collection. In some cases, more information can be obtained. For this purpose, the data type of the return value of the function object operator can be specified by a designated template argument. The list of possible return types include `JResultDerivative`, `JResultHesse`, `JResultPDF`, and `JResultPolynome`.

The `JResultDerivative` can be used to obtain the function value and its derivative. It is available for both polynomial and spline interpolation.

```
template<class JResult_t>
struct JResultDerivative
{
  JResult_t f;      //!< function value
  JResult_t fp;     //!< first derivative
};
```

The JResultHesse can be used to obtain the function value, its first and its second derivative. It is available for both polynomial and spline interpolation.

```
template<class JResult_t>
struct JResultHesse
{
  JResult_t f;      //!< function value
  JResult_t fp;     //!< first  derivative
  JResult_t fpp;    //!< second derivative
};
```

The JResultPDF can be used to obtain the function value, its derivative and (partial) integrals. It is available for both polynomial and spline interpolation.

```
template<class JResult_t>
struct JResultPDF
{
  JResult_t f;      //!< function value
  JResult_t fp;     //!< first derivative
  JResult_t v;      //!< integral <xmin,x]
  JResult_t V;      //!< integral <xmin,xmax>
};
```

The JResultPolynome can be used to obtain the function value and a specified number of derivatives (up to the degree of the polynomial function used for interpolation). It is available for polynomial interpolation.

```
template<unsigned int N, class JResult_t>
struct JResultPolynome
{
  JResult_t y[NUMBER_OF_POINTS];   //!< values
};
```

These data structure have arithmetic capabilities. Hence, their values can be interpolated in any number of dimensions.

4.3 Polynomial interpolation

The template class, JPolintFunction1D, can be used for polynomial interpolation. The underlying interpolation code is taken from reference [1].

```
template<unsigned int N,
         class JElement_t,
         template<class, class> class JContainer_t,
         class JResult_t   = typename JElement_t::ordinate_type,
         class JDistance_t = JDistance<typename JElement_t::abscissa_type> >
class JPolintFunction1D
{};
```

The first template parameter, `N`, corresponds to the degree of the polynomial function that is used to interpolate the values. The method `compile()` does nothing as there is no need to compile data for the polynomial interpolation.

4.4 Cubic spline interpolation

The template class, `JSplineFunction1D`, can be used for cubic spline interpolation. The underlying interpolation code is taken from reference [1].

```
template<class JElement_t,
         template<class, class> class JContainer_t,
         class JResult_t   = typename JElement_t::ordinate_type,
         class JDistance_t = JDistance<typename JElement_t::abscissa_type> >
class JSplineFunction1D
{};
```

The method `compile()` provides the implementation to determine the second derivatives and the (partial) integrals for the spline interpolation.

4.5 Concrete interpolation classes

A set of concrete interpolation classes is available that can be used for interpolation in one dimension without template specifications. These classes are summarised in table 1. In this, the `JPolint0Function1D_1` and `JGridPolint0Function1D_1` provides for a simple and fast look-up method, respectively.

5 Histogram in one dimension

A histogram can be considered as a function without return value. Instead of evaluating some value, the ordinate value of the element in a collection corresponding to a given abscissa value should simply be incremented by a given weight. The definition of a one-dimensional histogram reads:

```
template<class JElement_t,
         template<class, class> class JContainer_t,
         class JDistance_t = JDistance<typename JElement_t::abscissa_type> >
class JHistogram1D :
  public JContainer_t<JElement_t, JDistance_t>,
  public JHistogram<typename JElement_t::abscissa_type,
                    typename JElement_t::ordinate_type>
{
  virtual void evaluate(const abscissa_type* pX,
                        typename JClass<contents_type>::argument_type w)
  {
    this->fill(*pX, w);
  }

  void fill(typename JClass<abscissa_type>::argument_type x,
            typename JClass<contents_type>::argument_type w);
};
```

interpolator	collection	abscissa	result
JSplineFunction1D_t	JCollection	double	double
JGridSplineFunction1D_t	JGridCollection	double	double
JSplineFunction1H_t	JCollection	double	JResultDerivative<double>
JGridSplineFunction1H_t	JGridCollection	double	JResultDerivative<double>
JSplineFunction1S_t	JCollection	double	JResultPDF<double>
JGridSplineFunction1S_t	JGridCollection	double	JResultPDF<double>
JPolint0Function1D_t	JCollection	double	double
JPolint1Function1D_t	JCollection	double	double
JPolint2Function1D_t	JCollection	double	double
JPolint3Function1D_t	JCollection	double	double
JGridPolint0Function1D_t	JGridCollection	double	double
JGridPolint1Function1D_t	JGridCollection	double	double
JGridPolint2Function1D_t	JGridCollection	double	double
JGridPolint3Function1D_t	JGridCollection	double	double
JPolint0Function1H_t	JCollection	double	JResultDerivative<double>
JPolint1Function1H_t	JCollection	double	JResultDerivative<double>
JPolint2Function1H_t	JCollection	double	JResultDerivative<double>
JPolint3Function1H_t	JCollection	double	JResultDerivative<double>
JGridPolint0Function1H_t	JGridCollection	double	JResultDerivative<double>
JGridPolint1Function1H_t	JGridCollection	double	JResultDerivative<double>
JGridPolint2Function1H_t	JGridCollection	double	JResultDerivative<double>
JGridPolint3Function1H_t	JGridCollection	double	JResultDerivative<double>

Table 1: List of concrete classes for one-dimensional interpolation.

In this, `w` corresponds to the weight. The `contents_type` of a histogram corresponds to the `ordinate_type` of a collection. The method `evaluate()` specifies a recursive method call. An implementation of this method thus applies to the filling of a histogram in any dimension. The method `fill()` provides for an implementation of the histogram functionality.

The `JHistogram` class is a simple data structure for handling general statistics of a histogram.

```
template<class JAbscissa_t, class JContents_t>
class JHistogram
{
  const contents_type& getUnderflow() const;
  const contents_type& getOverflow()  const;
  const contents_type& getIntegral()  const;
};
```

The methods `getUnderflow()` and `getOverflow()` return the summed weights below the lower abscissa value of the histogram and above the upper abscissa value of the histogram, respectively. The method `getIntegral()` returns the sum of all weights.

6 Integration in one dimension

The various interpolating methods can also be used to evaluate the integral of the interpolating function. To this end, the method `integrate()` is provided. This method returns the integral from the lower abscissa

value up to the upper abscissa value of a collection of elements. The results of the partial integrals are also stored. This method is overloaded for the various kinds of interpolating methods.

For the one-dimensional polynomial integration, the implementation reads:

```
template<unsigned int N,
         class JElement_t,
         template<class, class> class JContainer_t,
         class JResult_t,
         class JDistance_t>
inline typename JElement_t::ordinate_type
integrate(const JPolintFunction1D<N,
                                  JElement_t,
                                  JContainer_t,
                                  JResult_t,
                                  JDistance_t>&     input,
          typename JMappable<JElement_t>::map_type& output);
```

The output class should provide for an implementation of the `JMappableCollection` interface. The `JMappable` class provides a simple type definition for this interface. In this specialisation, the Gauss-Legendre integration technique is used to evaluate the partial integrals [1]. The template argument `N` is used to determine a consistent number of points for the integration between two consecutive abscissa values.

For the one-dimensional spline integration, the implementation reads:

```
template<class JElement_t,
         template<class, class> class JContainer_t,
         class JResult_t,
         class JDistance_t>
inline typename JElement_t::ordinate_type
integrate(const JSplineFunction1D<JElement_t,
                                  JContainer_t,
                                  JResult_t,
                                  JDistance_t>&     input,
          typename JMappable<JElement_t>::map_type& output);
```

In this specialisation, the values of the second derivatives for the spline interpolation are used to evaluate the partial integrals.

For a one-dimensional histogram, the implementation reads:

```
template<class JElement_t,
         template<class, class> class JContainer_t,
         class JDistance_t>
inline typename JElement_t::ordinate_type
integrate(const JHistogram1D<JElement_t,
                             JContainer_t,
                             JDistance_t>&          input,
          typename JMappable<JElement_t>::map_type& output);
```

In this implementation, the abscissa values of the partial integrals correspond to the upper edge of each histogram bin.

The method `getIntegral()` can be used to evaluate the integral of a function or histogram without storing the partial integrals. The definition of `getIntegral()` reads.

```
template<class JContainer_t>
inline typename JContainer_t::ordinate_type getIntegral(const JContainer_t& input)
{
  JGarbageCollection<typename JContainer_t::abscissa_type,
                     typename JContainer_t::ordinate_type> garbage;

  return integrate(input, garbage);
}
```

In this, the output is directed to a designated class that provides for a dummy implementation of the `JMappableCollection` interface.

7 Multi-dimensional map

The `JMultiMap` class constitutes a multi-dimensional map (i.e. a map of a map of a map etc.). Here, it is used to describe a point in a multi-dimensional space. For the list of maps, a type list is used.

```
template<class JAbscissa_t,
         class JOrdinate_t,
         class JMaplist_t,
         class JDistance_t = JDistance<JAbscissa_t> >
class JMultiMap;
{
  enum { NUMBER_OF_DIMENSIONS = JMapLength< JMapList<JMap_t, JTail_t> >::value };

  void configure(..);
  void insert(..);

  class super_const_iterator;
  class super_iterator;

  super_const_iterator super_begin() const;
  super_const_iterator super_end() const;

  super_iterator super_begin();
  super_iterator super_end();
};
```

In this, the first template argument refers to the abscissa in each dimension and the second template argument to the overall ordinate. The template arguments `JMaplist_t` refers to a list of maps. The template argument `JDistance` is the distance operator between elements which will be applied to each dimension. The map operator `[]` as well as the methods `configure()` and `insert()` can be used to fill the multi-dimensional map.

```
typedef
  JMapList<JMap,
  JMapList<JMap,
  JMapList<JMap> > >                          JMaplist_t;

typedef JMultiMap<double, double, JMaplist_t>  JMultimap_t;

JMultimap_t buffer;

for (double x = -1.0; x <= +1.0; x += 0.1) {
  for (double y = -1.0; y <= +1.0; y += 0.1) {
    for (double z = -1.0; z <= +1.0; z += 0.1) {
      buffer[x][y][z] = 0.0;
    }
  }
}
```

In addition to the STL iterators that are provided for by each map of a dimension, a "super" iterator is defined that can be used to iterate over all elements in the multi-dimensional map in a single sequence. The smart operator -> represents a multi-dimensional pointer to a single element in the multi-dimensional map. The abscissa and ordinate values can be accessed using data members `first` and `second` recursively. For example, the contents of the above multi-dimensional map can be accessed as follows:

```
for (JMultimap_t::super_const_iterator i  = buffer.super_begin();
                                       i != buffer.super_end();
                                     ++i) {
  cout << i->first                 << ' '
       << i->second->first         << ' '
       << i->second->second->first  << ' '
       << i->second->second->second << endl;
}
```

A multi-dimensional map can thus be viewed as a one-dimensional map with a multi-dimensional key. In this, the keys are constant and cannot be modified by either "super" iterator. The ordinate value is referenced and can be modified. In a similar way, the dereference operator * represents a multi-dimensional pair to a single element in the multi-dimensional map.

7.1 Multi-dimensional key

An implementation of a multi-dimensional key is provided for by the template class `JMultiKey`.

```
template<unsigned int N, class JKey_t>
class JMultiKey :
  public std::pair<JKey_t, JMultiKey<N-1, JKey_t> >
{};
```

7.2 Multi-dimensional pair

An implementation of a multi-dimensional pair is provided for by the template class `JMultiPair`.

```
template<unsigned int N, class JKey_t, class JValue_t>
class JMultiPair
{
  JKey_t                            first;
  JMultiPair<N-1, JKey_t, JValue_t> second;
};
```

8 Functional maps

The functionality (i.e. the ability of function value evaluation) of a map is provided by a corresponding functional collection.

The template class, `JPolintMap`, can be used for polynomial interpolation.

```
template<unsigned int N,
         class JKey_t,
         class JValue_t,
         template<class, class, class> class JMap_t,
         class JResult_t,
         class JDistance_t = JDistance<JKey_t> >
class JPolintMap :
  public JPolintFunction<N,
                         JElement2D<JKey_t, JValue_t>,
                         JMapCollection<JMap_t>::template collection_type,
                         JResult_t,
                         JDistance_t>
{};
```

The template class, `JSplineMap`, can be used for cubic spline interpolation.

```
template<class JKey_t,
         class JValue_t,
         template<class, class, class> class JMap_t,
         class JResult_t,
         class JDistance_t = JDistance<JKey_t> >
class JSplineMap :
  public JMap_t<JKey_t, JValue_t, JDistance_t>,
  public JFunction<JKey_t, JResult_t>
{};
```

In both cases, the interpolation is based on the same functional collection. Hence, there actually is no additional code.

The template class, `JHistogramMap`, can be used to histogram data.

```
template<class JAbscissa_t,
         class JContents_t,
         template<class, class, class> class JMap_t,
         class JDistance_t = JDistance<JAbscissa_t> >
class JHistogramMap :
```

```
  public JMap_t<JAbscissa_t, JContents_t, JDistance_t>,
  public JHistogram<JAbscissa_t, typename JContents_t::contents_type>
{};
```

8.1 Standardized functional maps

A set of standardized functional maps is available that can be used in conjunction with a multi-dimensional map. The corresponding classes require two template arguments, namely the data type of the key and the value. These classes summarised in table 2. In this, the result "same" corresponds to the data type of the return value from the interpolation in the lower dimension. If the result is specified as `JResultDerivative`, the data type of the return value from the interpolation in the lower dimension corresponds to the template argument of this class. As a result, the derivatives in all dimensions can be obtained.

	collection	result
JSplineFunctionalMap	JMap	same
JSplineFunctionalGridMap	JGridMap	same
JPolint1FunctionalMap	JMap	same
JPolint0FunctionalMap	JMap	same
JPolint2FunctionalMap	JMap	same
JPolint3FunctionalMap	JMap	same
JPolint0FunctionalGridMap	JGridMap	same
JPolint1FunctionalGridMap	JGridMap	same
JPolint2FunctionalGridMap	JGridMap	same
JPolint3FunctionalGridMap	JGridMap	same
JPolint1FunctionalMapH	JMap	JResultDerivative
JPolint0FunctionalMapH	JMap	JResultDerivative
JPolint2FunctionalMapH	JMap	JResultDerivative
JPolint3FunctionalMapH	JMap	JResultDerivative
JPolint0FunctionalGridMapH	JGridMap	JResultDerivative
JPolint1FunctionalGridMapH	JGridMap	JResultDerivative
JPolint2FunctionalGridMapH	JGridMap	JResultDerivative
JPolint3FunctionalGridMapH	JGridMap	JResultDerivative

Table 2: List of concrete functional maps for multi-dimensional interpolation.

9 Multi-dimensional interpolation

Multi-dimensional interpolation is based on the template `JMultiFunction` class.

```
template<class JFunction_t,
         class JMaplist_t,
         class JDistance_t = JDistance<typename JFunction_t::argument_type> >
class JMultiFunction :
  public JMultiMap<typename JFunction_t::argument_type,
                   JFunction_t,
                   JMaplist_t,
                   JDistance_t>
{
```

```
  enum { NUMBER_OF_DIMENSIONS = JMapLength<JMaplist_t>::value
                                + JFunction_t::NUMBER_OF_DIMENSIONS };

  result_type operator()(const argument_type x, ...) const;
};
```

where `JFunction_t` refers to the interpolator in the lowest dimension. As can be seen from this definition, the `JMultiFunction` class derives from a `JMultiMap`. Hence, the filling is the same as that of a multi-dimensional map. The function operator corresponds to the recursive interpolation procedure. The number of argument values should match the number of dimensions.

Based on the availability of functional maps, one-dimensional interpolation methods and different collections, various multi-dimensional interpolation methods can be constructed with a different search methodology and interpolation techniques in each dimension. It is interesting to note that the template argument `JFunction_t` could also refer to an interpolation method in more than one dimension. In that case, the number of dimensions in which the interpolation works is expanded to the sum.

10 Multi-dimensional histogram

A multi-dimensional histogram can be made using the template `JMultiHistogram` class.

```
template<class JHistogram_t,
         class JMaplist_t,
         class JDistance_t = JDistance<typename JHistogram_t::abscissa_type> >
class JMultiHistogram :
  public JMultiMap<typename JHistogram_t::abscissa_type,
                   JHistogram_t,
                   JMaplist_t,
                   JDistance_t>
{
  void fill(const abscissa_type x, ...);
};
```

The fill method corresponds to the recursive histogram filling procedure. The number of argument values should match the number of dimensions. Note that one additional value is expected for the histogram weight.

11 Multi-dimensional integration

The method `getIntegral()` can also be used to evaluate the integral of a function or histogram in multiple dimensions. To this end, this method is accordingly overloaded for the various data types.

12 I/O

Binary I/O of data structures is defined by the `JReader` and the `JWriter` interfaces. These interfaces implement the I/O of all primitive data types, most `STL` containers and all other serialisable data structures (i.e. data structures which implement the `JSerialisable` interface). The I/O abilities of serialisable data structures is defined by the `JSerialisable` interface.

```
class JSerialisable {
  virtual JReader& read(JReader& in) = 0;
  virtual JWriter& write(JWriter& out) const = 0;
};
```

12.1 Input

The input of data is defined by the `JReader` interface:

```
class JReader {

  virtual int read(char*, int) = 0;

  JReader& operator>>(JSerialisable& object) { return object.read(*this); }
};
```

The re-direct operator >> defines the input of any data structure that is derived from the `JSerialisable` interface. The interface provides for an implementation of this operator for all primitive data types. The method `read()` defines the interface method for input from any device. The arguments are a pointer to a byte array and the number of bytes to be read, respectively. It should return the actual number of bytes read.

12.2 Output

The output of data is defined by the `JWriter` interface:

```
class JWriter {

  virtual int write(char*, int) = 0;

  JWriter& operator<<(JSerialisable& object) { return object.write(*this); }
};
```

The re-direct operator << defines the output of any data structure that is derived from the `JSerialisable` interface. The interface provides for an implementation of this operator for all primitive data types. The method `write()` defines the interface method for output to any device. The arguments are a pointer to a byte array and the number of bytes to be written, respectively. It should return the actual number of bytes written.

12.3 I/O of multi-dimensional maps

A multi-dimensional map of serialisable objects is also serialisable. This is implemented through recursive application of the I/O operators. As a result, the I/O capabilities of a multi-dimensional map also apply to a function object or histogram in any number of dimensions.

References

[1] Numerical Recipes in C++, W.H. Press, S.A. Teukolsky, W.T. Vetterling and B.P. Flannery, Cambridge University Press.