Contents

Chapter 1

Introduction

This note is the documentation of version 2.0 of the NUTILITY package, which contains basic tools for statistical computations. The work is supported by The Research Council of Norway through the research program no. STP 28402 "Toolkits in Industrial Mathematics", where the Norwegian Computing Center (NR) is one of three participants, the other two being SINTEF Oslo and the University of Oslo (UiO).

The routines are implemented in C++, and the module is organized in class hierarchies to simplify addition of new models and methods. Version 2.0 contains implementation of matrix and vector operations, least squares algorithms, and random number generators for several probability distributions. The class documentations do not include descriptions of private members, protected members, and member functions considered as implementation details, unless these descriptions are required for clarification of the functionality.

The note is organized in four chapters following this introduction. Chapter two contains the documentation of the matrix and vector classes. Some general remarks about the matrix- and vector-hierarchy are given in section 2.1. Chapter three contains the documentation of the classes for least-squares computations, and chapter four the documentation of the random number generators. The random number generators are organized in a class hierarchy with base classes **RandomGen**, **RandomCont** and **RandomDisc**, documented in section 4.2.

Routines for computing the factorial and binomial functions, and a function counting the number of entries on a file, are documented in chapter five. This chapter also includes documentation of a base classes for sets of data.

Some of the classes and functions make use of basic tools implemented in the module Diffpack, developed at SINTEF Applied Mathematics and University of Oslo, Department of Mathematics and Department of Informatics. Diffpack is released as public access software, that is public domain software for non-commercial use. Parameterized classes for simple vectors are used to implement vectors of matrices, and the classes in NUTILITY make extensive use of a set of error handling functions, as well as an implementation of boolean variables. Documentations of these classes and functions from Diffpack are included in appendix A.

Differences between versions 1.0 and 2.0.

In version 2.0, errors are corrected, and adjustments due to experience with using the library are performed. The matrix multiplication and decomposition of rectangular matrices are made more efficient. The random generators for the Poisson and gamma distributions are reimplemented. Some general base classes included in version 1.0, considered to be closely related to specific methods, are not included in the new version.

Chapter 2

Matrix and vector classes

2.1 General remarks

2.1.1 Representation

The matrix and vector classes in the NUTILITY package are especially designed for use in solution of statistical problems. They are derived from two base classes **BaseArray** and **BaseIntArray**, which implements representations of double and integer arrays, respectively.

The representation includes a copy counting mechanism. If an object in the matrix/vector hierarchy is copied, no new memory is allocated, so that several matrices or vectors are sharing the same representation in memory. When an object is changed, it is detached from the shared representation. The copy count mechanism keeps account of the number of objects sharing the same memory location, so that the memory is not freed until the last object goes out of scope.

Separate classes are implemented for symmetric and triangular matrices, taking advantage of the fact that only the elements on and below the diagonal need to be stored. These classes are derived from the class for rectangular double matrices, **nMatrix**. As a result of the differences in representation, some member functions of the base class **nMatrix** should not be invoked for objects of the derived classes **TriangMatrix** and **SymMatrix**. Such functions are made virtual, and the implementation of those functions in the derived classes consists of an error message stating that the function call is illegal.

The arithmetic operations (multiplication, addition and subtraction of matrices) are implemented for all combinations of the different matrix classes.

2.1.2 Communication with C and FORTRAN subroutines

In NUTILITY, both vectors and matrices are dynamically allocated and stored as one-dimensional arrays. The matrices are stored row by row. The classes **BaseArray** and **BaseIntArray** are equipped with member functions that returns a pointer to the first (or second) element of a floating point array or integer array. These functions can be used when a matrix or vector object is to be transferred to a C-function or FORTRAN subroutine. FORTRAN relies on arrays beeing allocated in one bunch of storage. By representing matrices as well as vectors as a one-dimensional array, this requirement is met. Since multidimensional arrays in FORTRAN are stored columnwise, the representation of the transpose of the matrices should be used when calling FORTRAN subroutines.

2.2 Base classes

2.2.1 BaseArray

NAME

BaseArray - a base class for double arrays

INCLUDE

```
include "matrix.h"
```

SYNTAX

```
class BaseArray{
 protected:
   struct vrep{     // actual data, ref.count update instead of copy
     BASE *v;       // pointer to data
     int n;         // reference count
     int len;       // length of array
     void init(int l){
       if (l<=0) errorFP("vrep::init (in BaseArray)",
                         "Illegal length %d.!\n",l);
       n=1; v=new BASE[len=l];
       if (v==NULL)
         errorFP("vrep::init (in BaseArray)",
                 "Problems with allocation, using new. Length: %d.\n",l);
       v--;
     }
     vrep(int l) { init(l);}
     vrep(){n=1; len=0; v=NULL; }
     vrep(const vrep& q)
       {
         init(q.len);
         for(int i=1; i<=len; i++)
           v[i] = q.v[i];
       }
     ~vrep(){ if (v!=NULL) { v++; delete v;}}
   };
   vrep* p;

   virtual inline BASE& iter(int& k)  // returns elements in canonical order
     {return p->v[k++];}              // no index range check
   virtual inline BASE iter(int& k) const
     {return p->v[k++];}              // no index range check

   void detachIfMultiples();
   //  copies vrep into new vrep if multiple references exist

   BaseArray(vrep*);

 public:
   BaseArray(int);
   BaseArray() { p = new vrep();}
   BaseArray(int,BASE);
   BaseArray(int,BASE*);
   BaseArray(const BaseArray&);
   virtual ~BaseArray();

   BaseArray& operator=(const BaseArray&);
```

```
    BooLean operator==(const BaseArray&) const;

    inline BaseArray& operator+(){return *this;};
    BaseArray& operator-();

    virtual BaseArray& operator+=(const BaseArray&);
    virtual BaseArray& operator-=(const BaseArray&);

    friend BaseArray& operator+=(BaseArray& ,BASE);
    friend BaseArray& operator*=(BaseArray& ,BASE);
    friend BaseArray& operator/=(BaseArray& ,BASE);

    virtual BASE* ptrBase0 ();
    virtual BASE* ptrBase1 ();

    // *** minimum, maximum and summary measures:

    double maxval() const;            // maximum of elements
    double minval() const;            // minimum of elements
    virtual double sum() const;       // sum of elements
    virtual double mean() const;      // mean of elements
};
```

KEYWORDS

array, double array

DESCRIPTION

The class is a base class for implementations of different kinds of arrays of elements of type BASE. In the current revision, `BaseArray` is used as base class for double vectors and matrices, and the macro BASE is set equal to double.

The class is primarily meant to be a base class for vectors and matrices, and not to be used directly.

The array is represented by a C-array of elements, but the array is shifted so that the first index is one. To prevent unnecessary copying of objects, a reference counter is implemented. When a new object of a class in the hierarchy is created from an already excisting instance, the objects will share the same memory location until one of the objects is changed. Then, the object operated on will be detached from the shared representation.

CONSTRUCTORS AND INITIALIZATION

The class has five public constructors, including a default constructor not allocating any memory, and a copy constructor. The three other constructors all have an integer argument indicating the length of the array. In addition, a BASE (in this revision double) value or a pointer to a BASE C-array might be specified, giving initial values for the array elements. It is assumed that the C-array has first index zero. By default, all elements of the array are initialized to zero.

MEMBER FUNCTIONS

Most member functions are self explanatory, and their intended use and function should be clear from the class declaration.

`operator+` - unary + -operator

`operator-` - unary - -operator, overwrites the elements of the object with their negative values.

`ptrBase0` - returns the representation of the array as a pointer to a double C-array with first index zero. If other objects share the same representation, the object is detached from the shared representation. The function is useful for sending an object of a class in the `BaseArray` hierarchy to a C-function requiring a pointer. Since the value returned is the address of the first element of the actual array, this function can be used for sending the contents of an object to a FORTRAN subroutine.

`ptrBase1` - similar to `ptrBase0`, except that the first index of the array pointed to is one. The function is useful for sending an object of a class in the `BaseArray` hierarchy to a C-function requiring a base 1 array.

FILES

base_array.C

EXAMPLE

See class nMatrix and class nVector.

SEEALSO

class nMatrix, class nVector

AUTHOR

Jon Helgeland, NR

2.2.2 BaseIntArray

NAME

BaseIntArray - a base class for integer arrays

INCLUDE

```
include "matrix.h"
```

SYNTAX

```
class BaseIntArray{
 protected:
   struct vrep{     // actual data, ref.count update instead of copy
     int *v;        // pointer to data
     int n;         // reference count
     int len;       // length of array
     void init(int l){
       if (l<=0) errorFP("vrep::init (in BaseIntArray)",
                         "Illegal length %d.!\n",l);
       n=1; v=new int[len=l];
       if (v==NULL)
         errorFP("vrep::init (in BaseIntArray)",
                 "Problems with allocation, using new. Length: %d.\n",l);
       v--;
     }
     vrep(int l) { init(l); }
     vrep(){ n=1; len=0; v=NULL; }
     vrep(const vrep& q)
       {
         init(q.len);
         for(int i=1; i<=len ; i++)
           v[i] = q.v[i];
       }
     ~vrep(){ if (v!=NULL) {v++; delete v;} }
   };
   vrep *p;
   virtual inline int& iter(int& k)    // returns elements in canonical order
     { return p->v[k++];}              // no index range check
   virtual inline int iter(int& k) const
     { return p->v[k++];}              // no index range check

   void detachIfMultiples();
   //  copies vrep into new vrep if multiple references exist

   BaseIntArray(vrep*);

 public:
   BaseIntArray() { p = new vrep();}
   BaseIntArray(int);
   BaseIntArray(int, int);
   BaseIntArray(int, int*);
   BaseIntArray(const BaseIntArray&);
   virtual ~BaseIntArray();

   BaseIntArray& operator=(const BaseIntArray&);
   BooLean operator==(const BaseIntArray&) const;

   virtual BaseIntArray& operator+=(const BaseIntArray&);
   virtual BaseIntArray& operator-=(const BaseIntArray&);
   virtual BaseIntArray& operator*=(int);
```

```
    virtual int* ptrBase0 ();
    virtual int* ptrBase1 ();

    int maxval() const;   // maximum of elements
    int minval() const;   // minimum of elements
    int sum() const;      // sum of elements
};
```

KEYWORDS

array, integer array

DESCRIPTION

The class is a base class for implementions of different kinds of integer arrays. It is primarily not meant to be used directly. The array is represented by a C-array of elements, but the array is shifted so that the first index is one. To prevent unnecessary copying of objects, a reference counter is implemented. When a new object of a class in the hierarchy is created from an already excisting instance, the objects will share the same memory location until one of the objects is changed. Then, the object operated on will be detached from the shared representation.

CONSTRUCTORS AND INITIALIZATION

The class has five public constructors, including a default constructor, not allocating any memory, and a copy constructor. The three other constructors all have an integer argument indicating the length of the array. I addition, an integer value or a pointer to an integer array might be specified, giving initial values for the array elements. It is assumed that the C-array has first index zero. By default, all elements of the array are initialized to zero.

MEMBER FUNCTIONS

Most member functions are self explanatory, and their intended use and function should be clear from the class declaration.

`ptrBase0` - returns the representation of the array as a pointer to an integer C-array with first index zero. If other objects share the same representation, the object is detached from the shared representation. The function is useful for sending an object of a class in the `BaseIntArray` hierarchy to a C-function requiring a pointer. Since the value returned is the address of the first element of the actual array, this function can be used for sending the contents of an object to a FORTRAN subroutine.

`ptrBase1` - similar to `ptrBase0`, except that the first index of the array pointed to is one. The function is useful for sending an object of a class in the `BaseIntArray` hierarchy to a C-function requiring a base 1 array.

FILES

base_array.C

EXAMPLE

See class nIntVector.

SEEALSO

class nIntVector

AUTHOR

Jon Helgeland, NR

2.3 Double and integer vectors

2.3.1 nVector

NAME

nVector - a class for double vectors

INCLUDE

```
include "matrix.h"
```

SYNTAX

```
class nVector:public BaseArray
{
   int n;                                   // length of vector

 public:
   nVector(int len)
     : BaseArray(len),n(len){}
   nVector(int len, BASE value)
     : BaseArray(len,value),n(len){}
   nVector(int len, BASE* value)
     : BaseArray(len,value),n(len){}
   nVector(const nVector& v)
     : BaseArray(v),n(v.n) {}
   nVector()
     : BaseArray(),n(0){}

   // *** operators:

   nVector& operator=(const nVector&);
   nVector& setEqual(const nVector&);       // operator= with index check

   // subscript operators:
   inline double& operator()(int);
   inline double operator()(int) const;
   inline double& el (int i) { detachIfMultiples(); return p->v[i];}
   inline double el (int i) const { return p->v[i];}

   double normE() const;                    // Euclidean norm

   BooLean redim(int);

   nVector& operator+=(const nVector&);     // overwrites the current object
   nVector& operator-=(const nVector&);     // overwrites the current object

   int getLen() const { return n;}          // returns length of vector

   operator nMatrix() const;                // converts to nMatrix object
   nMatrix asColVector() const;             // returns n x 1 matrix
   nMatrix asRowVector() const;             // returns 1 x n matrix

   nVector preMult(const nMatrix& A) const; // A*b
   nVector postMult(const nMatrix& A) const; // b'*A

   nVector elMult(const nVector&) const;    // elementwise multiplication
   nVector elDiv(const nVector&) const;     // elementwise division
```

```
   void elop (double (*pf)(double));
             // operates on each element with given function pointer,
             // overwrites the current object

   SymMatrix diag() const;                  // generates diagonal matrix

   // *** concatenation and extraction:

   nVector concat(const nVector&) const;
   nVector concat(double) const;

   nVector subvector(const nIntVector&) const;
   nVector subvector(int from, int to) const;
   nVector excl(const nIntVector& ind) const;
             // extracts vector excluding elements with indices  ind

   void setSubvector(int from, int to, const nVector&) const;

   // *** minimum, maximum and summary measures:

   double maxval() const { return BaseArray::maxval(); }
   double minval() const { return BaseArray::minval(); }
   double sum() const { return BaseArray::sum(); }
   double mean() const { return BaseArray::mean(); }
   double ss() const;                       // sum of squares
   double var() const;                      // variance

   // *** inner and outer products:

   double innerProduct(const nVector& v) const;
   nMatrix outerProduct(const nVector& v) const;

   friend long double innerProduct(int m, int n,
                const nVector&, const nVector&, long double c=0.0);

   // *** input and output:

   virtual void scan(Is is);
   virtual void print(Os os) const;

   friend Is& operator>>(Is& istr,nVector& a);
   friend Os& operator<<(Os& ostr,const nVector& a);

   friend double relDiff(const nVector&, const nVector&);
   friend double mahaDist(const nVector&, const nVector&, const SymMatrix&);

   // *** friend classes:
   friend class nMatrix;
   friend class TriangMatrix;
   friend class SymMatrix;
};

// *** arithmetic operations:

nVector operator+(const nVector& a,const nVector& b);
nVector operator-(const nVector& a,const nVector& b);
nVector operator*(BASE a,const nVector& b);
nVector operator*(const nVector& a,BASE b);
nVector operator/(const nVector& a,BASE b);
```

KEYWORDS

double vector, vector

DESCRIPTION

The class implements a double vector, represented by an array with first index one. The class has several member functions implementing common operations on vectors, such as subscripting, arithmetic operations and inner products, as well as functions for editing and input and output of the contents of a vector.

The class is derived from class `BaseArray`, a common base class for vectors and matrices.

CONSTRUCTORS AND INITIALIZATION

The class has five constructors, including a copy constructor and a default constructor not allocating any memory. The three other constructors all take the length of the vector as argument. By default, all elements of the vector are initialized to zero. Other values might be specified by an argument of type `BASE` (all values equal) or `BASE*` (a pointer to an ordinary base 0 C-array of values), where the macro `BASE` equals double in this revision.

MEMBER FUNCTIONS

Many functions are self explanatory, and the intended use and function of member functions not further documented, should be clear from the class declaration.

`Subscripting:`

The class has two pairs of subscript functions, each pair consisting of functions for const and non-const objects. The overloaded operator () performs index checking, while the member functions named `el` do not.

`Input and output:`

The functions for input and output of the contents of the vector, `scan`, `print`, and the overloaded operators >> and <<, all use the classes `Is` and `Os` instead of the standard `istream` and `ostream` classes. These classes, implemented in Diffpack, offer more flexibility, in that they provide a general interface to several sources of input and output. The following calls to `scan` (and similar calls to `print`) are all legal:

```
scan(cin);

ifstream infile("input.file",ios::in);    scan(infile);

scan("FILE=input.file");
```

This is obtained by automatic conversion from the standard input/output sources to `Is/Os`.

`operator nMatrix()` - user-defined conversion operator converting a `nVector` of length `n` to a `n` x `1` `nMatrix`. Note that in functions using reference arguments, `nMatrix&`, automatic type conversion is not allowed because a temporary argument (`nMatrix(nVector)`) is created, and changes done on this object within the function will not be visible outside.

`asColVector` - returns a `nMatrix` object with dimensions `n` by 1, representing the length `n` vector as a column vector.

`asRowVector` - returns a `nMatrix` object with dimensions 1 by `n`, representing the length `n` vector as a row vector.

`concat` - overloaded function returning a vector equal the concatenation of the vector with another vector or a single double value. The function leaves the object unchanged.

`elop` - performs on each element the operation indicated by the function pointer. The function overwrites the object for which it is called. The argument is a pointer to a function of double returning double.

`excl` - returns the subvector corresponding to the vector for which the member function is invoked, excluding elements with indices indicated by the `nIntVector` reference argument.

`normE` - returns the Euclidean norm of the vector.

`postMult` - performs post-multiplication of the `nVector` object (`b`) with the matrix indicated by the `nMatrix` reference argument (`A`), and returns the vector `y=transp(b)*A`.

`preMult` - performs pre-multiplication of the `nVector` object (`b`) with the matrix indicated by the `nMatrix` reference argument (`A`), and returns the vector `y=Ab`.

`redim` - If the integer argument equals the old length, the function leaves the vector unchanged, and returns dpFALSE. If not, the vector object is redimensioned, the element values set to zero, and dpTRUE is returned.

`setSubvector` - replaces the elements of the subvector with indices in the interval `[from,to]`, by the elements of the `nVector` object entered as argument to the function.

`subvector` - overloded function returning the subvector corresponding to the indices indicated by the `nIntVector` reference argument, or in the range `[from,to]` (including the endpoints). The function leaves the object unchanged.

FRIEND FUNCTIONS

`innerProduct` - computes the inner product of two vectors `a` and `b` in the index range `m` to `n`, with the offset `c`.

`mahaDist` - returns the Mahalanobis distance between two vectors `a` and `b` with the covariance matrix `S` indicated by the `SymMatrix` reference argument. The Mahalanobis distance is given by `transp(a-b)*inv(S)*(a-b)`.

`relDiff` - returns the relative difference between two vectors `a` and `b`, given by the sum over all elements `i` of `abs(a(i)-b(i))/(1.0+0.5*abs(a(i)+b(i)))`.

FILES

vector.C

EXAMPLE

```
#include <matrix.h>
#include <stdlib.h>

main(int argc, char* argv[])
{
  int n=atoi(argv[1]);   // length of vector

  nVector v1(n);
  v1.scan("FILE=input.file");

  // computes (x_i-mean(x))^2/var(x), i=1 to n:

  double mval = v1.mean();
  nVector vscaled = (v1-nVector(n,mval))/sqrt(v1.var());
```

```
  double maxv = vscaled.maxval();
  double minv = vscaled.minval();

  ofstream out("out.file",ios::out);
  vscaled.print(out);
  out << "\nMaximum and minimum values: " << maxv << " " << minv << "\n";

  // weighted sum of elements:

  nVector weights(n);
  ifstream winput("weights.dat");
  weights.scan(winput);

  double wsum = vscaled.innerProduct(weights);
  out << "\nWeighted sum: " << wsum << "\n";


  // extract subvector with indexvector (1,3,5,...)

  int m = n/2;
  nIntVector indvec(m);
  for (int i=1;i<=m;i++)
    indvec(i) = 2*i;

  nVector vexcl = vscaled.excl(indvec);

  vexcl.print("FILE=half.vector");
}
```

SEEALSO

class BaseArray, class nMatrix

AUTHOR

Jon Helgeland and Turid Follestad, NR

2.3.2 nIntVector

NAME

nIntVector - a class for integer vectors

INCLUDE

```
include "matrix.h"
```

SYNTAX

```
class nIntVector: public BaseIntArray
{
   int n;                                   // length of vector

 public:
   nIntVector() : BaseIntArray(),n(0){}
   nIntVector(int len)
        : BaseIntArray(len),n(len){}
   nIntVector(int len, int val)
        : BaseIntArray(len,val),n(len){}
   nIntVector(int len, int* val)
        : BaseIntArray(len,val),n(len){}
   nIntVector(const nIntVector& v)
        : BaseIntArray(v),n(v.n){}

   // *** operators:

   nIntVector& operator=(const nIntVector& b);
   nIntVector& setEqual(const nIntVector&);  // operator= with index check

   operator nVector();                      // converts to nVector object
   int getLen() const { return n;}          // returns length of vector

   // subscript operators:
   inline int& operator()(int);
   inline int operator()(int) const;
   inline int& el (int i)  { detachIfMultiples(); return p->v[i];}
   inline int el (int i) const { return p->v[i];}

   void indexRangeCheck (int min, int max, char* errtext) const;
   BooLean duplicateCheck ();
   BooLean redim(int);

   // *** concatenation and extraction:

   nIntVector concat (const nIntVector&) const;
   nIntVector concat (int) const;

   nIntVector subvector(const nIntVector&) const;
   nIntVector subvector(int from, int to) const;

   nIntVector excl(const nIntVector& ind) const;
               // extracts vector excluding elements with indices  ind

   void setSubvector(int from, int to, const nIntVector&) const;

   // *** minimum, maximum and summary measures:

   int maxval() const { return BaseIntArray::maxval(); }
   int minval() const { return BaseIntArray::minval(); }
```

```
    int sum() const { return BaseIntArray::sum(); }

    // *** input and output:

    void scan(Is is);
    void print(Os os) const;

    friend Is& operator>>(Is& istr,nIntVector& a);
    friend Os& operator<<(Os& ostr,const nIntVector& a);
};
```

KEYWORDS

index vector, integer vector, vector

DESCRIPTION

The class implements an integer vector, represented by an array with first index one. Index vectors are typical objects of this class.

Included are routines for input and output, subscripting, and editing of vectors.

The class is derived from class `BaseIntArray`.

CONSTRUCTORS AND INITIALIZATION

The class has five constructors, including a copy constructor and a default constructor not allocating any memory. The three other constructors all take the length of the vector as argument.

By default, all elements of the vector are initialized to zero. Other values might be specified by an additional integer argument (all values equal) or a pointer to an ordinary base 0 C-array of integer values.

MEMBER FUNCTIONS

Many functions are self explanatory, and the intended use and function of member functions not further documented, should be clear from the class declaration.

`Subscripting:`

The class has two pairs of subscript functions, each pair consisting of functions for const and non-const objects. The overloaded operator () performs index checking, while the member functions named `el` do not.

`Input and output:`

The functions for input and output of the contents of the vector, `scan`, `print`, and the overloaded operators >> and <<, all use the classes `Is` and `Os` instead of the standard `istream` and `ostream` classes. These classes, implemented in Diffpack, offer more flexibility, in that they provide a general interface to several sources of input and output. The following calls to `scan` (and similar calls to `print`) are all legal:

```
scan(cin);

ifstream infile("input.file",ios::in);    scan(infile);

scan("FILE=input.file");
```

This is obtained by automatic conversion from the standard input/output sources to `Is/Os`.

`operator nVector()` - user defined conversion operator converting an integer vector to a double vector. Note that in functions using reference arguments, `nVector&`, automatic type conversion is not allowed because a temporary argument (`nVector(nIntVector)`) is created, and changes done on this object within the function will not be visible outside.

No arithmetic operations are implemented for the integer vector. If such operations are needed, the integer vector should be treated as a double vector.

`concat` - overloaded function returning a vector equal to the concatenation of the vector with another vector or a single integer value. The function leaves the object unchanged.

`duplicateCheck` - removes duplicate values, overwrites the object with the integer vector without duplicates, keeping the first instance of duplicate value elements, and returns dpTRUE if one or more duplicates are found.

`excl` - returns the subvector corresponding to the vector for which the member function is invoked, excluding elements with indices indicated by the `nIntVector` reference argument.

`indexRangeCheck` - returns dpTRUE if the integer arguments to the function are within the index range of the `nIntVector` object. The `char*` argument is a string specifying the name of the function from which the member function is invoked.

`redim` - If the integer argument equals the old length, the function leaves the vector unchanged, and returns dpFALSE. If not, the vector object is redimensioned, the element values set to zero, and dpTRUE is returned.

`setSubvector` - replaces the elements of the subvector with indices in the interval `[from,to]`, by the elements of the `nIntVector` object entered as argument to the function.

`subvector` - overloded function returning the subvector corresponding to the indices indicated by the `nIntVector` reference argument or in the range [`from`,`to`] (including the endpoints). The function leaves the object unchanged.

FILES

vector.C

EXAMPLE

See class nVector.

SEEALSO

class BaseIntArray, class nVector

AUTHOR

Jon Helgeland and Turid Follestad, NR

2.4 Double matrices

2.4.1 nMatrix

NAME

nMatrix - a class for rectangular double matrices

INCLUDE

```
include "matrix.h"
```

SYNTAX

```
class nMatrix: public BaseArray
{
 protected:
   int m;                                  // no. of rows
   int n;                                  // no. of columns

 public:
   nMatrix() : BaseArray(),m(0),n(0) {}
   nMatrix(int rdim, int cdim);
   nMatrix(int rdim, int cdim, BASE value);
   nMatrix(const nMatrix& a);

   nMatrix(char* filename, int rdim, int cdim, BooLean numbcheck=dpTRUE);

   virtual String typeId() const { return "nMatrix"; }

   // *** operators:

   nMatrix& operator=(const nMatrix&);
   nMatrix& operator=(const SymMatrix&);
   nMatrix& operator=(const TriangMatrix&);
   nMatrix& setEqual(const nMatrix&);        // operator= with index check

   virtual BooLean operator==(const nMatrix&) const;

   // subscripting:
   virtual double& operator()(int,int);
   virtual double operator()(int,int) const;

   virtual double& el (int i, int j)
         { detachIfMultiples(); return p->v[(i-1)*n+j];}
   virtual double el (int i, int j) const { return p->v[(i-1)*n+j];}

   // arithmetic operations, overwrites the current object:
   virtual nMatrix& operator+=(const nMatrix&);
   virtual nMatrix& operator-=(const nMatrix&);
   virtual nMatrix& operator*=(const nMatrix&);

   int getRdim() const {return m;}           // returns number of rows
   int getCdim() const {return n;}           // returns number of columns

   virtual BooLean redim(int i, int j);

   double normE() const;                     // Euclidean norm of elements
                                             // (square root of sum of squares)
   nVector diag() const;                     // returns diagonal
```

```
// *** conversion:

virtual nMatrix rectangular() const { return *this;}
operator nVector() const;        // converts column vectors to type nVector

BooLean isSymmetric() const;
BooLean isLowTriangular() const;

nVector asVector() const;        // returns m x 1 or 1 x n matrix as vector
SymMatrix asSymMatrix() const;   // returns symmetric matrix
TriangMatrix asTriangMatrix() const;      // returns lower triangular matrix

// *** special matrix multiplications, A=(*this):

virtual SymMatrix multAtA() const;        // A'A, returns symmetric matrix
SymMatrix multAAt() const;                // AA', returns symmetric matrix

nMatrix mAAt() const;                     // AA'
nMatrix mAtA() const;                     // A'A
nMatrix multAB(const nMatrix& B) const;   // AB
nMatrix multAtB(const nMatrix& B) const;  // A'B
nMatrix multABt(const nMatrix& B) const;  // AB'
nMatrix multAtBA(const nMatrix& B) const; // A'BA
nMatrix multABAt(const nMatrix& B) const; // ABA'

// *** get two-dimensional C-array representation:

double** ptr2dBase1 ();   // base 1, returns pointer to array of pointers
double** ptr2dBase0 ();   // base 0, returns pointer to array of pointers

// *** extract and insert rows, columns and submatrices:

virtual nVector row(int i) const;         // returns row number i
virtual nVector col(int j) const;         // returns column number j

// set rows of current object:
virtual void setRow(int i, const nVector& r);  // sets row i to r
virtual void setCol(int j, const nVector& c);  // sets column j to c

void setSubmatrix(const nIntVector&, const nIntVector&, const nMatrix&);
void setSubmatrix(int r1,int c1,const nMatrix&);

nMatrix rowConc(const nMatrix&) const;         // stacks matrices
nMatrix colConc(const nMatrix&) const;         // matrices side by side

// extract submatrix:
nMatrix submatrix(const nIntVector&, const nIntVector&) const;
nMatrix submatrix(int r1, int r2, int c1, int c2) const;
nMatrix submatrix(const nIntVector&, int c1, int c2) const;
nMatrix submatrix(int r1, int r2, const nIntVector&) const;

nMatrix rowExcl(const nIntVector&) const;
nMatrix colExcl(const nIntVector&) const;

// *** columnwise and rowwise operations:

nVector colSum()              const;
nVector rowSum()              const;

nVector colMean()             const;
nVector rowMean()             const;

nVector colSs()               const;     // column sum of squares
nVector rowSs()               const;     // row sum of squares

nMatrix rowAdd(const nVector& v)  const;  // add v to each row
```

```
nMatrix colAdd(const nVector& v)  const;  // add v to each column

nMatrix rowSub(const nVector& v)  const;  // subtract v from each row
nMatrix colSub(const nVector& v)  const;  // subtract v from each column

nMatrix rowMult(const nVector& v) const;  // elementwise mult., each row
nMatrix colMult(const nVector& v) const;  // elementwise mult., each column

nMatrix rowDiv(const nVector& v)  const;  // elementwise div., each row
nMatrix colDiv(const nVector& v)  const;  // elementwise div., each column

// *** elementwise operations:

nMatrix elMult(const nMatrix&) const;     // elementwise multiplication
void elop (double (*pf)(double));
          // operates on each element of the current object with given
          // function pointer

// *** minimum, maximum and summary measures:

double maxval() const { return BaseArray::maxval(); }
double minval() const { return BaseArray::minval(); }
double sum() const { return BaseArray::sum(); }
double mean() const { return BaseArray::mean(); }
virtual double ss() const;  // sum of squares
virtual double var() const; // variance

// *** transpose:

virtual void mtotransp();                 // overwrites the current object
virtual nMatrix transp() const;

virtual int rank() const;                 // rank of matrix

// *** determinant and trace, matrix square:

BooLean isQuadratic() const; // returns dpTRUE if matrix is square
virtual double det() const;  // determinant, if matrix is square
double trace()     const;    // sum of digonal elements, if matrix is square

// *** decomposition and inverse of matrix, matrix square:

virtual BooLean mtodecomp(double& d1, int& d2, nIntVector& piv,
                          BooLean eflag=dpTRUE);
                  // LU-decomposition, returns determinant and overwrites
virtual void mtoinv();                    // overwrites with inverse

virtual nMatrix decomp(double& d1, int& d2, nIntVector& piv) const;
                  // LU-decomposition, returns determinant
virtual nMatrix inv() const;              // inverse
virtual nMatrix geninv() const;           // generalized inverse


// *** solution of AX = B:

virtual void solve(nMatrix& B);
                // LU-decomposition of A and solution of A=XB
void solveLU(nMatrix& B, const nIntVector& piv) const;
                // assumes (*this) is the LU-decomposition of A

// *** singular value decomposition:

nVector singVal()   const; // singular values
Svd svd()           const; // sing.val.decomp: A=U*diag(l)*transp(V)
Svd svdStrip()      const; // sing.val.decomp, excluding negative singular
                           // values
```

```
   // *** input and output:

   virtual void scan(Is is);
   virtual void print(Os os) const;

   friend Is& operator>>(Is& istr,nMatrix& a);
   friend Os& operator<<(Os& ostr,const nMatrix& a);

   // *** matrix multiplication:

   friend nMatrix operator*(const nMatrix& a,const nMatrix& b);

   // *** friend classes:
   friend class nVector;
   friend class TriangMatrix;
   friend class SymMatrix;
};

// *** arithmetic operations on matrices:

nMatrix operator+(const nMatrix& a,const nMatrix& b);
nMatrix operator-(const nMatrix& a,const nMatrix& b);
nMatrix operator*(BASE a,const nMatrix& b);
nMatrix operator*(const nMatrix& a,BASE b);
nMatrix operator/(const nMatrix& a,BASE b);

// *** deletion of two-dimension arrays representing matrices:

void delPtr2dBase1 (double** arr, int m);
void delPtr2dBase0 (double** arr, int m);
```

KEYWORDS

double matrix, matrix, rectangular matrix, square matrix

DESCRIPTION

The class implements a rectangular double matrix, with first indices one. The class has several member functions implemeting common operations on matrices, such as subscripting, arithmetic operations, inversion and decomposition, as well as functions for editing data matrices and input and output of the contents of a matrix.

The class is derived from class `BaseArray`, a common base class for vectors and matrices, and the matrix elements are stored in a one-dimensional array, row by row.

If a class `nMatrix` object is to be used as input to a FORTRAN subroutine, the member function `ptrBase0` inherited from `BaseArray` could be used to get the C-array representation. Note that since the matrix is stored row by row, in contradiction to what is expected by a FORTRAN subroutine, the function `ptrBase0` should be applied to the transpose of the matrix.

CONSTRUCTORS AND INITIALIZATION

The class has five public constructors, including a default constructor not allocating any memory, and a copy constructor.

The constructors take the row and column dimensions of the matrix as argument. By default, all elements of the matrix are initialized to zero. Other values might be specified by an argument of type `BASE` (all values equal), where `BASE` equals double in this revision.

One constructor reads the contents of the matrix from a file named `filename`. If the last argument, `numbcheck`, equals dpTRUE, the number of elements on the file is counted, and an error message is issued if this number not equals `rdim*cdim`.

Using the default constructor impiles that the row and column dimensions are set to zero.

MEMBER FUNCTIONS

Many functions are self explanatory, and the intended use and function of member functions not further documented, should be clear from the class declaration.

`General remarks:`

For some matrix operations, two member functions are implemented. Functions with names beginning with `mto` (move to) overwrites the object for which the member function is invoked, while the corresponding function with no prefix creates a new `nMatrix` object, leaving the current object unchanged.

`Subscripting:`

The class has two pairs of subscript functions, each pair consisting of functions for const and non-const objects. The overloaded operator () performs index checking, while the member functions named `el` do not.

`Input and output:`

The functions for input and output of the matrix, `scan`, `print`, and the overloaded operators >> and <<, all use the classes `Is` and `Os` instead of the standard `istream` and `ostream` classes. These classes, implemented in Diffpack, offer more flexibility, in that they provide a general interface to several sources of input and output. The following calls to `scan` (and similar calls to `print`) are all legal:

```
scan(cin);

ifstream infile("input.file",ios::in);    scan(infile);

scan("FILE=input.file");
```

This is obtained by automatic conversion from the standard sources to `Is/Os`.

`asSymMatrix` - returns a symmetric matrix as a `SymMatrix` object. If the matrix is not symmetric, the `SymMatrix` object is created from the lower triangular part of the matrix, and a warning message is issued.

`asTriangMatrix` - returns a lower triangular matrix as a `TriangMatrix` object. If the matrix is not lower triangular, the lower triangular part is extracted, and a warning message is issued.

`asVector` - returns a `m` by 1 or 1 by `n` matrix as a `nVector` object with length `m` or `n` respectively.

`colExcl` - extracts the columns indicated by the `nIntVector` reference argument, returning the matrix made up by the remaining columns.

`decomp` - performs the same operation as `mtodecomp`, but a new matrix holding the decomposition is created, and the function leaves the object for which the member function is called, unchanged.

`elop` - performs on each element the operation indicated by the function pointer. The function overwrites the object for which it is called. The argument is a pointer to a function of double returning double.

inv - virtual function computing the inverse of a square matrix, by calling the member function **mtoinv** for a copy of the object. The inverse is returned as a class **nMatrix** object even if **inv** is called for an object of a derived class, implying that care should be taken when calling this member function for a derived class object.

mtodecomp - virtual function that overwrites the object with a decomposition of the matrix. The type of decomposition depends on the subclass for which the function is called. The default, implemented for class **nMatrix**, is the LU-decomposition **A=LU** for a square matrix **A**. **L** is stored in the lower triangular part of the object, and **U**, a unit diagonal, upper triangular matrix, in the upper triangular part of the object, ommitting the diagonal.

The pivot vector **piv** keeps the interchanges made to the rows of **A**, such that the i-th row and the piv(i)-th row were interchanged at the i-th step.

The member function is intended for use together with **solveLU**, when an equation system **AX=B** is to be solved for the same matrix **A**, but different right hand sides **B**.

The procedure will fail if the matrix is singular or almost singular when the eflag given is dpTRUE. If an eflag parameter of dpFALSE is given, the procedure will return dpTRUE if it succeded, and dpFALSE if it failed.

The determinant of A can be computed from the double (**d1**) and integer (**d2**) reference arguments as **det = d1*2^d2**

The function is adapted from the procedure *unsymdet* in J.H.Wilkinson and C.Reinsch: "Linear Algebra", Springer Verlag, Berlin Heidelberg New York, 1971, pp. 99-100.

operator nVector() - user-defined conversion operator converting a **n x 1** matrix to a double vector of length **n**. Note that in functions using reference arguments, **nVector&**, automatic type conversion is not allowed because a temporary argument (**nVector(nMatrix)**) is created, and changes done on this object within the function will not be visible outside. A warning will be issued in such cases.

ptr2dBase0 - returns the representation of the array as a pointer to a double C-array of pointers, with first index zero. If other objects share the same representation, the object is detached from the shared representation. The function is useful for sending an object of a class in the **BaseArray** hierarchy to a C-function requiring a two-dimensional array.

ptr2dBase1 - similar to **ptrBase0**, except that the first indices of the arrays pointed to are one.

rectangular - virtual function that transforms the object to a rectangular matrix, of type class **nMatrix**. This function is needed in classes derived from **nMatrix**, since the data representations of the derived classes **TriangMatrix** and **SymMatrix** are not the same as for **nMatrix**.

redim - If the integer arguments equal the old dimensions, the function leaves the matrix unchanged, and returns dpFALSE. If not, the matrix object is redimensioned, the element values set to zero, and dpTRUE is returned.

rowExcl - function extracting the rows indicated by the **nIntVector** reference argument, returning the matrix made up by the remaining rows.

solve - virtual function that finds the solution of the equation system **AX=B**. The member function **mtodecomp** is called, computing the LU-decomposition of the square matrix **A**. The input to the function is **p** right hand sides, stored in the **n x p** matrix **B**. The **nMatrix** reference argument is overwritten by the **n x p** solution matrix **X**. If the equation system is to be solved for several different matrices **B**, the member funtions **mtodecomp** and **solveLU** should be used, decomposing the square matrix **A** only once, and calling **solveLU** for each **B**. The current object is overwritten by the LU-decomposition.

solveLU - function that finds the solution of the equation system **AX=B** for a square matrix **A**. It is assumed that **mtodecomp** is called on beforehand, so that the object holds

the LU-decomposition of the square matrix `A`. The input to the function is `p` right hand sides, stored in the `n x p` matrix `B`, and the pivot vector returned from the decomposition as computed by `mtodecomp`. The `nMatrix` reference argument is overwritten by the `n x p` solution matrix `X`.

`setSubmatrix` - overloaded function that sets the element values of a submatrix of the `nMatrix` object. The submatrix is specified either by two `nIntVector` reference arguments, that is expected to contain the `p` row- and `q` column indices of the elements to be replaced, or by the starting points `r1` and `c1` of two index intervals.

The elements are replaced by the elements of the `nMatrix` object entered as argument to the function.

`singVal` - returns a vector holding the singular values of the matrix.

`submatrix` - overloaded function extracting a submatrix of the `nMatrix` object. The function parameters indicates the row an column indices of the elements that are to be extracted, beginning with the row indices. The indices might be specified by an index vector, represented by the `nIntVector` reference argument, or alternatively by two integer values specifying the endpoints of an index interval.

`svd` - performs a singular value decomposition of the matrix, and returns the result in an object of type class `Svd`.

`svdStrip` - performs a singular value decomposition of the matrix, excludes singluar values less than zero, and returns the result in an object of type class `Svd`.

`transp` - virtual function computing the transpose of a matrix, by calling the member function `mtotransp` for a copy of the object. The transpose is returned as a class `nMatrix` object even if `transp` is called for an object of a derived class, implying that care should be taken when calling this member function for a derived class object.

FILES

matrix.C, matrix_svd.C

EXAMPLE

```
#include <matrix.h>
#include <stdlib.h>

main(int argc, char* argv[])
{
  //--------- Solution of equation system Y = XB:

  int m=atoi(argv[1]);     // number of rows
  int n=atoi(argv[2]);     // number of columns

  nMatrix X(m,n);
  X.scan("FILE=input.file1");

  nMatrix Y("Y.dat",m,1);  // checking no. of values on file

  double d1;
  int d2;
  nIntVector piv;

  // LU-decomposition:

  X.mtodecomp(d1,d2,piv);
```

```
  cout << "Determinant: " << d1*pow(2,d2) << "\n";

  // solution of X*B = Y:

  X.solveLU(Y,piv);

  cout << "Solution:\n" << Y;

  // --------- arithmetic operations

  nMatrix M1(X);           // copy constructor, M1=X
  nMatrix M2(m,n);
  M2.scan("FILE=input.file2");

  nMatrix M3 = M1+M2;      // matrix addition

  M3.mtotransp();          // overwrite with transpose

  nMatrix M4 = M3*M1;      // matrix multiplication

  // ---------- elementwise operations:

  M2.elop(&sqrt);          // overwrites each element by its square root

  // ---------- extracting submatrix/concatenation:

  int p=n/2;
  nIntVector indvec(p);

  for (int i=1;i<=p;i++)
    indvec(i) = 2*i;

  // extract every second column of M2 (column 2,4,6,...,n (or n-1)):

  nMatrix M5 = M2.submatrix(1,m,indvec);

  cout << "submatrix:\n";
  cout << M5;

  // concatenate M2 and 2*M2:

  M5 = M2.colConc(2*M2);   // column concatenation (M2 | (2*M2))
                           // operator=: no dimension check

  cout << "\nConcatenation: " << M5;
}
```

SEEALSO

class BaseArray, class Svd, class nVector

AUTHOR

Jon Helgeland, Magne Aldrin and Turid Follestad, NR

2.4.2 SymMatrix

NAME

SymMatrix - a class for symmetric double matrices

INCLUDE

```
include "matrix.h"
```

SYNTAX

```
class SymMatrix: public nMatrix
{
 public:
   SymMatrix()
     : nMatrix()
       { ;}
   SymMatrix(int dim);
   SymMatrix(const SymMatrix& S)
     : nMatrix(S.m*(S.m+1)/2,S)
       {m = n = S.m;}

   SymMatrix(char* filename, int dim, BooLean numbcheck=dpTRUE);

   SymMatrix& operator=(const SymMatrix&);
   SymMatrix& setEqual(const SymMatrix&);   // operator= with index check

   BooLean operator==(const SymMatrix& m) const
     { return nMatrix::operator==(m);}

   String typeId() const { return "SymMatrix"; }

   // *** subscript operators:

   double& operator()(int,int);
   double operator()(int,int) const;

   double& el (int i, int j)
     {  detachIfMultiples();
        return (i > j) ? p->v[(i-1)*i/2+j] : p->v[(j-1)*j/2+i]; }
   double el (int i, int j) const
     { return (i > j) ? p->v[(i-1)*i/2+j] : p->v[(j-1)*j/2+i]; }

   nMatrix rectangular() const;   // converts symmetric matrix to square matrix

   SymMatrix& operator+=(const SymMatrix&);
   SymMatrix& operator-=(const SymMatrix&);

   // *** S +=(-=)(*=) A (class nMatrix) illegal

   nMatrix& operator*=(const nMatrix&)
     { errorFP("SymMatrix::operator*=(const nMatrix&)",
               "Illegal operation! Returns *this.\n"); return *this;}
   nMatrix& operator+=(const nMatrix&)
     { errorFP("SymMatrix::operator+=(const nMatrix&)",
               "Illegal operation! Returns *this.\n"); return *this;}
   nMatrix& operator-=(const nMatrix&)
     { errorFP("SymMatrix::operator-=(const nMatrix&)",
               "Illegal operation! Returns *this.\n"); return *this;}

   double sum() const;                     // sum of elements
```

```
   double ss() const;                      // sum of squares

   // *** extract row and column:

   nVector row(int i) const;               // returns row i
   nVector col(int j) const;               // returns column j

   SymMatrix mPow(double p) const;         // matrix power
   int rank() const;

   void mtotransp() const {}               // overwrites with transpose
   void mtoinv();                          // overwrites with inverse

   BooLean mtodecomp(BooLean eflag=dpTRUE);
                           // overwrites with Cholesky decomposition
   void mtosemidecomp(nIntVector& piv);
                           // overwrites with modified Cholesky decomposition
   BooLean mtoreversedecomp(BooLean eflag=dpTRUE);
                           //  overwrites with reverse Cholesky decomposition
   SymMatrix semidecomp(nIntVector& piv) const;
                           //  modified Cholesky decomposition

   // *** solutions of SX = B:

   void solve(nMatrix& B);     // solves SX=B
   void solveSemi(nMatrix& B); // solves SX=B, S almost singular
   void solveLL(nMatrix& B) const;
                           // assumes that mtodecomp has been called
   void solveLL(nMatrix& B, const nIntVector& piv) const;
                           // assumes that mtosemidecomp has been called

   SymMatrix symtransp() const {return *this;}
   SymMatrix syminv() const;               // returns inverse
   SymMatrix gensyminv() const;            // returns generalized inverse

   // *** eigenvectors and eigenvalues

   nVector eigVal()    const;              // eigenvalues
   Eigen eigen()       const;              // eigenvalues and eigenvectors
   Eigen eigenStrip()  const;              // eigenvalues and eigenvectors
                                           // stripped for eigenvalues appr.=0
   Eigen eigenPos()  const;                // eigenvalues and eigenvectors
                                           // stripped for eigenvalues <= 0
   // *** friend class:

   friend class nMatrix;
};

// *** aritmetic operations:

SymMatrix operator+(const SymMatrix& a,const SymMatrix& b);
SymMatrix operator-(const SymMatrix& a,const SymMatrix& b);
SymMatrix operator*(BASE a,const SymMatrix& b);
SymMatrix operator*(const SymMatrix& a,BASE b);

nMatrix operator*(const SymMatrix& a,const nMatrix& b);
nMatrix operator+(const SymMatrix& a,const nMatrix& b);
nMatrix operator-(const SymMatrix& a,const nMatrix& b);

SymMatrix identity(int n);                 // returns the identity matrix I_n

// *** Cholesky factor:

TriangMatrix cholFactor(const SymMatrix&); // Cholesky decomposition
TriangMatrix cholFactorSemi(const SymMatrix&, nIntVector& piv);
                                          // Cholesky decomposition
                                          // for semidefinite matrices
```

KEYWORDS

double matrix, matrix, symmetric matrix

DESCRIPTION

The class implements a symmetric double matrix, with first indices one. The class differs from its base class **nMatrix** in that only the diagonal and below diagonal elements of a **SymMatrix** object are stored. A **SymMatrix** object might be transformed to a **nMatrix** object by the member function **rectangular**.

CONSTRUCTORS AND INITIALIZATION

The class has four constructors: a copy constructor, a default constructor not allocating any memory, a constructor taking an integer argument specifying the number of rows (and columns) of the symmetric matrix, and a constructor reading the contents of the matrix from a file named **filename**. If the last argument of this constructor, **numbcheck**, equals dpTRUE, the number of elements on the file is counted, and an error message is issued if this number not equals **rdim*cdim**.

Using the default constructor impiles that the row and column dimensions are set to zero.

MEMBER FUNCTIONS

Many functions are self explanatory, and the intended use and function of member functions not further documented, should be clear from the class declaration.

General remarks:

Note that several of the member functions of the base class **nMatrix** returns class **nMatrix** objects. This applies to some virtual functions as well, a fact that might be undesirable in some cases. For member functions where a **SymMatrix** return value is wanted, a function with a different name than the base class virtual function is implemented. This applies to the functions **syminv** and **symtransp**. So when a **SymMatrix** return type is desired, the virtual mechanism should be avoided, and the alternative name functions used.

Subscripting:

The class has two pairs of subscript functions, each pair consisting of functions for const and non-const objects. The overloaded operator () performs index checking, while the member functions named **el** do not.

decomp - performs the same operation as **mtodecomp**, but a new matrix holding the Cholesky decomposition is created, and the object for which the member function is called, is left unchanged.

eigen - returns the eigenvalues and eigenvectors of the matrix as an object of type **Eigen**. The computations of eigenvalues and eigenvectors are based on routines adapted from Press, Flannery, Teukolsky and Vetterling: "Numerical Recipes in C", Cambridge University Press, 1988.

eigenPos - computes eigenvalues and eigenvectors, removing negative eigenvalues and the corresponding eigenvectors from the **Eigen** object that is returned.

eigenStrip - computes eigenvalues and eigenvectors, removing eigenvalues approximately equal to zero and the corresponding eigenvectors from the **Eigen** object that is returned.

`eigVal` - computes the `m` eigenvalues of the matrix, returning the values as a vector of length `m`.

`mtodecomp` - overwrites the object with the Cholesky decomposition, `S = L*Lt`, where `Lt` is the transpose of `L`, so that the lower triangular matrix `L` is stored in the lower triangular part of the matrix.

It is assumed that the matrix is positive definite. The procedure will fail if the matrix is not positive definite or almost singular when the eflag given is dpTRUE. If an eflag parameter of dpFALSE is given, the procedure will return dpTRUE if it succeded, and dpFALSE if it failed.

`mtoreversedecomp` - overwrites the object with the reverse Cholesky factor `L`, computed from `S=Lt*L`. The lower triangular matrix `L` is stored in the lower triangular part of the object. It is assumed that the matrix is positive definite.

`mtosemidecomp` - overwrites the object with a modified Cholesky decomposition, `Pt*S*P + E = L*Lt`, of a `n` by `n` symmetric matrix `S`. Here, `P` is a permutation matrix made up from the pivot vector `piv` from the decomposition, by

```
P(i,piv(i)) = 1.0 for all i in [1,n],
```

and

```
P(i,j) = 0.0 elsewhere.
```

This decomposition is intended for use when the symmetric matrix is expected to be almost singular, or approximately semidefinite. At each step, an amount (`E(i,i)`) is added to the diagonal if an ordinary Cholesky update would result in a negative value of the diagonal element. The lower triangular Cholesky factor `L` is stored in the lower triangular part of the matrix, and the pivots are returned by the `nIntVector` reference argument.

The member function is intended for use together with `solveLL`, when an equation system `SX=B` is to be solved for the same matrix `S`, but different right hand sides `B`.

The function is based on the public domain FORTRAN routine `modchl` implemented by Elizabeth Eskow and Robert B. Schnabel, obtained from NetLib.

`rectangular` - function that transforms a `SymMatrix` object to a rectangular matrix, of type class `nMatrix`. This function is needed because the representation of symmetric and rectangular matrices are not the same.

`semidecomp` - performs the same operation as `mtosemidecomp`, but creates a new matrix holding the modified Cholesky decomposition and leaves the object for which the member function is called, unchanged. The pivot vector from the decomposition is returned by the `nIntVector` reference argument.

`solve` - function that finds the solution of the equation system `SX=B` for a positive definite symmetric matrix `S`. The member function `mtodecomp` is called, computing the Cholesky-decomposition `S=LLt` of `S`. The input to the function is `p` right hand sides, stored in the `n x p` matrix `B`. The `nMatrix` reference argument is overwritten by the `n x p` solution matrix `X`. If the equation system is to be solved for several different matrices `B`, the member funtions `mtodecomp` and `solveLL` should be used, decomposing the symmetric matrix `S` only once, and calling `solveLL` for each `B`.

It is assumed that the matrix is positive definite. The current object is overwritten by the Cholesky-decomposition.

`solveSemi` - function that finds the solution of the equation system `SX=B` for an almost singular, or positive semidefinite symmetric matrix `S`. The member function `mtosemidecomp` is called, computing the modified Cholesky-decomposition `PtSP + E =LLt` of `S`. The input

to the function is `p` right hand sides, stored in the `n x p` matrix `B`. The `nMatrix` reference argument is overwritten by the `n x p` solution matrix `X`. If the equation system is to be solved for several different matrices `B`, the member funtions `mtosemidecomp` and `solveLL` should be used, decomposing the symmetric matrix `S` only once, and calling `solveLL` for each `B`.

`solveLL` - overloaded function that finds the solution of the equation system `SX=B`. It is assumed that `mtodecomp` or `mtosemidecomp` is called on beforehand, so that the object for which this member function is called, holds the Cholesky decomposition `S=LLt`, or `PSPt = LLt` in the semidefinite case, of the `n x n` matrix `S`. The input to the function is `p` right hand sides, stored in the `n x p` matrix `B`, and, if `mtosemidecomp` is called, the pivot vector returned from that function. The `nMatrix` reference argument is overwritten by the `n x p` solution matrix `X`.

`syminv` - returns the inverse of a positive definite symmetric matrix, leaving the object unchanged. The inverse is computed using the Cholesky decomposition of the symmetric matrix. The function is introduced because calling the member function `inv` inherited from class `nMatrix`, results in returning a class `nMatrix` and not `SymMatrix` object.

The member function `inv` should be used if the matrix is not positive definite.

`symtransp` - returns the transpose of the symmetric matrix, leaving the object unchanged. The function is introduced because calling the member function `transp` inherited from class `nMatrix`, results in returning a class `nMatrix` and not `SymMatrix` object.

`gensyminv` - returns the generalized inverse of the symmetric matrix, leaving the object unchanged. The function is introduced because calling the member function `geninv` inherited from class `nMatrix`, results in returning a class `nMatrix` and not `SymMatrix` object.

GLOBAL FUNCTIONS

`cholFactor` - computes the Cholesky factor `L` of the symmetric matrix `S` entered as argument to the function, so that `S=LLt`. It is assumed that the symmetric matrix is positive definite. If the matrix is almost singular, the function `cholFactorSemi` could be used.

`cholFactorSemi` - computes the Cholesky factor `L` for a semidefinite or almost singular symmetric matrix, `S`. The function has two arguments, a `SymMatrix` holding the matrix to be decomposed, and an `nIntVector` reference argument, returning the pivot vector `piv` from the decomposition. If `P` is the permutation matrix with ones at the elements `(i,piv(i))` and zeros elsewhere, the decomposistion can be written `PtSP+E=LLt`.

The decomposition is done by a call to the member function `mtosemidecomp` of class `SymMatrix`.

FILES

spec_matr.C, matrix_eigen.C

EXAMPLE

```
// Decomposition of an almost singular matrix
#include <matrix.h>
#include <stdlib.h>

main(int argc, char* argv[])
{
  int n = atoi(argv[1]);
```

```
  SymMatrix S1(n);
  S1.scan("FILE=input.file");

  SymMatrix S2(S1);      // copy constructor
  nIntVector piv;
  S2.mtosemidecomp(piv); // modified Cholesky decomposition

  nMatrix Y(n,1);        // response variables

  Y.scan("FILE=response.dat");
  S2.solveLL(Y,piv);     // solves S2*X = Y; Y overwritten by X

  cout << "Solution\n:";
  Y.print(cout);
}
```

SEEALSO

class Eigen, class nMatrix, class TriangMatrix, class nVector

AUTHOR

Jon Helgeland, Magne Aldrin and Turid Follestad, NR

2.4.3 TriangMatrix

NAME

TriangMatrix - a class for lower triangular double matrices

INCLUDE

```
include "matrix.h"
```

SYNTAX

```
class TriangMatrix: public nMatrix
{
 public:
   TriangMatrix()
     : nMatrix()
       { ;}
   TriangMatrix(int dim);
   TriangMatrix(const TriangMatrix& T)
     : nMatrix(T.m*(T.m+1)/2,T)
       {m = n = T.m;}
   TriangMatrix(const SymMatrix& S)             // extracts lower triangle of S
     : nMatrix(S.getRdim()*(S.getRdim()+1)/2,S)
       {m = n = S.getRdim();}
   TriangMatrix(char* filename, int dim, BooLean numbcheck=dpTRUE);

   TriangMatrix& operator=(const TriangMatrix&);
   TriangMatrix& setEqual(const TriangMatrix&); // operator= with index check

   BooLean operator==(const TriangMatrix& m) const
     { return nMatrix::operator==(m);}

   String typeId() const { return "TriangMatrix"; }

   nMatrix rectangular() const; // converts triangular matrix to square matrix

   // special inner products:
   long double innerColCol(int,int,int,
                           const nMatrix&,int,long double = 0.0) const;
   long double innerRowCol(int,int,int,int,long double = 0.0) const;
             // inner product of row and column
   long double innerColCol(int,int,int,int,long double = 0.0) const;
             // inner product of column and column

   // *** subscript operators:

   double& operator()(int,int);
   double operator()(int,int) const;
   double& el (int i, int j)
     { detachIfMultiples();
       return (i > j) ? p->v[(i-1)*i/2+j] : zero;}
   double el (int i, int j) const
     { return (i > j) ? p->v[(i-1)*i/2+j] : zero;}

   // *** arithmetic operations:

   TriangMatrix& operator+=(const TriangMatrix&); // overwrites current object
   TriangMatrix& operator-=(const TriangMatrix&); // overwrites current object

   // *** Illegal operations, functions inherited from class nMatrix:
```

```
   nMatrix& operator*=(const nMatrix&)
     { errorFP("TriangMatrix::operator*=(const nMatrix&)",
               "Illegal operation! Returns *this.\n"); return *this;}
   nMatrix& operator+=(const nMatrix&)
     { errorFP("TriangMatrix::operator+=(const nMatrix&)",
               "Illegal operation! Returns *this.\n"); return *this;}
   nMatrix& operator-=(const nMatrix&)
     { errorFP("TriangMatrix::operator-=(const nMatrix&)",
               "Illegal operation! Returns *this.\n"); return *this;}

   // *** extract or reset row and column:

   nVector row(int i) const;                    // returns row i
   nVector col(int j) const;                    // returns column j

   // set rows of current object:
   void setRow(int i, const nVector& r);        // sets rows i to r
   void setCol(int j, const nVector& c);        // sets column j to c

   double det() const;                          // returns determinant

   void mtoinv();                               // overwrites with inverse
   TriangMatrix trianginv() const;              // returns inverse

   void solve(nMatrix& B) const;                // solves LX=B
   void transpSolve(nMatrix& B) const;          // solves L'X=B

   SymMatrix multAtA() const;                   // returns A'A

   // *** friend class:

   friend class nMatrix;
};

// *** arithmetic operations:

TriangMatrix operator+(const TriangMatrix& a,const TriangMatrix& b);
TriangMatrix operator-(const TriangMatrix& a,const TriangMatrix& b);
TriangMatrix operator*(BASE a,const TriangMatrix& b);  // BASE = double
TriangMatrix operator*(const TriangMatrix& a,BASE b);  // BASE = double

nMatrix operator*(const TriangMatrix& a,const nMatrix& b);
nMatrix operator+(const TriangMatrix& a,const nMatrix& b);
nMatrix operator-(const TriangMatrix& a,const nMatrix& b);
```

KEYWORDS

double matrix, lower triangular matrix, matrix, triangular matrix, square matrix

DESCRIPTION

The class implements a lower triangular double matrix, with first indices one. The class differs from its base class `nMatrix` in that only the diagonal and below diagonal elements of the matrix are stored. A `TriangMatrix` object might be transformed to a `nMatrix` object by the member function `rectangular`.

CONSTRUCTORS AND INITIALIZATION

The class has five constructors, including a copy constructor and a default constructor. One constructor takes an integer argument specifying the number of rows (and columns) of the triangular matrix, one of the constructors extracts the lower triangular part of a symmetric matrix, and the last one reads the contents of the matrix from a file named `filename`. If the last argument of this constructor, `numbcheck`, equals dpTRUE, the number of elements on the file is counted, and an error message is issued if this number not equals `rdim*cdim`.

By default, the row and column dimensions are set to zero, and no memory is allocated.

MEMBER FUNCTIONS

Many functions are self explanatory, and the intended use and function of member functions not further documented, should be clear from the class declaration.

`General remarks:`

Note that several of the member functions of the base class `nMatrix` returns class `nMatrix` objects. This applies to some virtual functions as well, a fact that might be undesirable in some cases. For member functions where a `TriangMatrix` return value is wanted, a function with a different name than the base class virtual function is implemented, as is the case with the function `trianginv`. So when a `TriangMatrix` return type is desired, the virtual mechanism should be avoided, and the alternative name function used.

`Subscripting:`

The class has two pairs of subscript functions, each pair consisting of functions for const and non-const objects. The overloaded operator () performs index checking, while the member functions named `el` do not.

`rectangular` - function that transforms a `TriangMatrix` object to a rectangular matrix, of type class `nMatrix`. This function is needed because the representation of triangular and rectangular matrices are not the same.

`mtoinv` - overwrites the object with the inverse of the triangular matrix.

`trianginv` - returns the inverse of the triangular matrix. The function is introduced because calling the member function `inv` inherited from class `nMatrix`, results in returning a class `nMatrix` and not `TriangMatrix` object.

`solve` - function that finds the solution of the equation system `LX=B` by foreward substitution. The input to the function is `p` right hand sides, stored in the `n x p` matrix `B`. The `nMatrix` reference argument is overwritten by the `n x p` solution matrix `X`.

`transpSolve` - function that finds the solution of the equation system `LtX=B` by backward substitution. The input to the function is `p` right hand sides, stored in the `n x p` matrix `B`. The `nMatrix` reference argument is overwritten by the `n x p` solution matrix `X`.

FILES

spec_matr.C

EXAMPLE

```
/*************************************************************************/
/* Computation of a=y'inv(S)y, where S is a symmetric matrix, avoiding the */
/* direct computation of the inverse of S.                                 */
```

```
/*                                                                      */
/* Using the Cholesky decomposition S = LL', a can be written as:       */
/*   a = y'inv(L')inv(L)y = x'x, where x = inv(L)y                      */
/*                                                                      */
/* a is computed by solving Lx=y, and computing the inner product x'x   */
/************************************************************************/

#include <stdlib.h>
#include <matrix.h>


main(int argc, char* argv[])
{
  int n=atoi(argv[1]);

  // read rom files, including counting the number of entries:
  SymMatrix S("input.file",n);    // symmetric matrix S
  nMatrix y("Y.dat",n,1);         // right hand side

  TriangMatrix L = cholFactor(S); // Cholesky factor of S

  L.solve(y);     // solves Lx=y, using nMatrix and not nVector because
                  // of the nMatrix reference argument in function  solve.
                  // y is overwritten by x

  double a = 0.0;
  for (int i=1;i<=n;i++)
    a+=y(i,1)*y(i,1);

  cout << "Result:\t" << a << "\n";
}
```

SEEALSO

class nMatrix, class SymMatrix, class nVector

AUTHOR

Jon Helgeland and Turid Follestad, NR

2.5 Eigenvalues and singular values

2.5.1 Eigen

NAME

Eigen - a class containing eigenvalues and eigenvectors

INCLUDE

```
include "matrix.h"
```

SYNTAX

```
class Eigen
{
private:
  nVector lambda;                   // eigenvalues
  nMatrix Gamma;                    // eigenvectors

public:
  Eigen(int m) : lambda(m), Gamma(m,m) {};
  Eigen(const Eigen& d) : lambda(d.lambda), Gamma(d.Gamma) {};
  Eigen() : lambda(), Gamma() {};

  BooLean redim(int m);

  nVector getLambda() const {return lambda;}
  nMatrix getGamma() const {return Gamma;}

  friend class SymMatrix;
};
```

KEYWORDS

eigenvalues, linear algebra, matrix, spectral theorem

DESCRIPTION

The class contains the eigenvalues and eigenvectors of a symmetric matrix, calculated by the functions `eigen`, `eigenPos` or `eigenStrip` of class `SymMatrix`.

The relation between the symmetric matrix `A` and the eigenvalues `lambda` and the matrix of eigenvectors `Gamma` is given by

```
A=Gamma*diag(lambda)*Gamma.transp().
```

CONSTRUCTORS AND INITIALIZATION

The class has one constructor taking the number of eigenvalues, `m`, as input, a copy-constructor, and a default constructor not allocating any memory.

MEMBER FUNCTIONS

The `get`-functions returns the data members.

`redim` - redimensions the members according to the new number of eigenvalues, `m`, if this number is different from the old one, returning dpTRUE if redimensioning is done.

FILES

matrix_eigen.C

EXAMPLE

```
#include <matrix.h>

main()
{
  int n=3;
  SymMatrix A(n);

  A(1,1)=1;
  A(2,1)=2;
  A(3,1)=3;

  // computes eigenvalues and eigenvectors of A:
  Eigen eig=A.eigen();

  // prints eigenvalues and eigenvectors to standard output:
  eig.getLambda().print(cout);
  cout << endl;
  cout << endl;
  eig.getGamma().print(cout);
}
```

SEEALSO

class SymMatrix, class Svd

AUTHOR

Magne Aldrin, NR

2.5.2 Svd

NAME

Svd - a class containing the result of a singular value decomposition

INCLUDE

```
include "matrix.h"
```

SYNTAX

```
class Svd
{
private:
  nMatrix U;
  nMatrix l;
  nMatrix V;

public:
  Svd(int m, int r, int n) : U(m,r), l(r,1), V(n,r) {};
  Svd(const Svd& d) : U(d.U), l(d.l), V(d.V) {};
  Svd() : U(), l(), V() {};

  BooLean redim(int m, int r, int n);

  nMatrix getU() const {return U;}
  nMatrix getL() const {return l;}
  nMatrix getV() const {return V;}

  friend class nMatrix;
};
```

KEYWORDS

linear algebra, matrix, singular values, singular value theorem

DESCRIPTION

The class contains the result of a singular value decomposition of a matrix, calculated by the functions `svd` or `svdStrip` of class `nMatrix`.

The singular decomposition of the matrix `A` is given by

```
A=U*diag(l)*V.transp().
```

CONSTRUCTORS AND INITIALIZATION

The class has one constructor taking the number of singular values, `r`, and the dimensions of the matrix `A`, `m` and `n`, as input, a copy-constructor, and a default constructor not allocating any memory.

MEMBER FUNCTIONS

The `get`-functions returns the data members.

`redim` - redimensions the members according to the new number of singular values, `r`, and the new matrix dimensions `m` and `n`, if these values are different from the old ones, returning dpTRUE if redimensioning is done.

FILES

matrix_svd.C

EXAMPLE

```
#include <matrix.h>

main()
{
  int m=4, n=3;
  nMatrix A(m,n);

  A(1,1)=1;
  A(2,1)=2;
  A(3,1)=3;
  A(4,1)=4;

  // computes the singular value decomposition of A:
  Svd svdA=A.svd();

  // prints the singular value decomposition to standard output:
  svdA.getL().print(cout);
  cout << endl;
  svdA.getU().print(cout);
  cout << endl;
  svdA.getV().print(cout);
}
```

SEEALSO

class nMatrix, class Eigen

AUTHOR

Magne Aldrin, NR

2.6 Simple arrays of matrices

2.6.1 VecSimplest(nMatrix)

NAME

VecSimplest(nMatrix) - a very simple vector of matrices

INCLUDE

```
include "VecSimplest_nMatrix.h"
```

SYNTAX

```
#define Type nMatrix
#include <VecSimplest.h>
#undef Type
```

KEYWORDS

matrix, simple vector, vector

DESCRIPTION

`VecSimplest(nMatrix)` is a class implementing a very simple vector of `nMatrix` objects, and is a specification of the parametric class `VecSimplest(Type)` in Diffpack, with parameter `Type` equal `nMatrix`.

The only operation available is subscripting, so that this class is suitable for storing vectors of matrices not intended for use in numerical computations. The index base is 1. Assignment operators and input and output routines are provided by the derived class `VecSimple(nMatrix)`.

See documentation of the parametric class `VecSimplest(Type)` implemented in Diffpack for a description of the interface of the class.

SEEALSO

class VecSimplest(Type), class VecSimple(nMatrix)

2.6.2 VecSimple(nMatrix)

NAME

VecSimple(nMatrix) - a simple vector of matrices

INCLUDE

```
include "VecSimple_nMatrix.h"
```

SYNTAX

```
#define Type nMatrix
#include <VecSimple.h>
#undef Type
```

KEYWORDS

matrix, simple vector, vector

DESCRIPTION

`VecSimple(nMatrix)` is a class implementing a simple vector of `nMatrix` objects, and is a specification of the parametric class `VecSimple(Type)` in Diffpack, with parameter `Type` equal `nMatrix`.

The class is derived from class `VecSimplest(nMatrix)`, and in addition to the inherited member functions, functions performing assignment and input and output of the contents of the array, are provided. The index base is 1.

See documentation of the parametric class `VecSimple(Type)` in Diffpack for a description of the interface of the class.

SEEALSO

class VecSimple(Type), class VecSimplest(nMatrix)

2.6.3 VecSimplest(nIntVector)

NAME

VecSimplest(nIntVector) - a very simple vector of integer vectors

INCLUDE

```
include "VecSimplest_nIntVector.h"
```

SYNTAX

```
#define Type nIntVector
#include <VecSimplest.h>
#undef Type
```

KEYWORDS

vector, simple vector

DESCRIPTION

`VecSimplest(nIntVector)` is a class implementing a very simple vector of `nIntVector` objects, and is a specification of the parametric class `VecSimplest(Type)` in Diffpack, with parameter `Type` equal `nIntVector`.

The only operation available is subscripting. The index base is 1. Assignment operators and input and output routines are provided by the derived class `VecSimple(nIntVector)`.

See documentation of the parametric class `VecSimplest(Type)` implemented in Diffpack for a description of the interface of the class.

SEEALSO

class VecSimplest(Type), class VecSimple(nIntVector)

2.6.4 VecSimple(nIntVector)

NAME

VecSimple(nIntVector) - a simple vector of integer vectors

INCLUDE

```
include "VecSimple_nIntVector.h"
```

SYNTAX

```
#define Type nIntVector
#include <VecSimple.h>
#undef Type
```

KEYWORDS

vector, simple vector

DESCRIPTION

`VecSimple(nIntVector)` is a class implementing a simple vector of `nIntVector` objects, and is a specification of the parametric class `VecSimple(Type)` in Diffpack, with parameter `Type` equal `nIntVector`.

The class is derived from class `VecSimplest(nIntVector)`, and in addition to the inherited member functions, functions performing assignment and input and output of the contents of the array, are provided. The index base is 1.

See documentation of the parametric class `VecSimple(Type)` in Diffpack for a description of the interface of the class.

SEEALSO

class VecSimple(Type), class VecSimplest(nIntVector)

2.6.5 ArrayGenSimplest(nMatrix)

NAME

ArrayGenSimplest(nMatrix) - a general array of matrices, with variable number of indices

INCLUDE

```
include "ArrayGenSimplest_nMatrix.h"
```

SYNTAX

```
#define Type nMatrix
#include <ArrayGenSimplest.h>
#undef Type
```

KEYWORDS

matrix, array

DESCRIPTION

`ArrayGenSimplest(nMatrix)` is a class implementing a general array of `nMatrix` objects, and is a specification of the parametric class `ArrayGenSimplest(Type)` in Diffpack, with parameter `Type` equal `nMatrix`.

The base and the number of dimensions of the array are both variable. The base is set manually. The only operations available are subscripting and an iterator for traversal of the array, so this class is suitable for storing arrays of matrices not intended for use in numerical computations.

Assignment operators and input and output routines are provided by the derived class `ArrayGenSimple(nMatrix)`.

See documentation of the parametric class `ArrayGenSimplest(Type)` implemented in Diffpack for a description of the interface of the class.

SEEALSO

class ArrayGenSimplest(Type), class ArrayGenSimple(nMatrix), class VecSimplest(nMatrix)

2.6.6 ArrayGenSimple(nMatrix)

NAME

ArrayGenSimple(nMatrix) - a general array of matrices, with variable number of indices

INCLUDE

```
include "ArrayGenSimple_nMatrix.h"
```

SYNTAX

```
#define Type nMatrix
#include <ArrayGenSimple.h>
#undef Type
```

KEYWORDS

matrix, array

DESCRIPTION

`ArrayGenSimple(nMatrix)` is a class implementing a general array of `nMatrix` objects, and is a specification of the parametric class `ArrayGenSimple(Type)` in Diffpack, with parameter `Type` equal `nMatrix`.

The base and the number of dimensions of the array are both variable. The base is set manually. The class is derived from class `ArrayGenSimplest(nMatrix)`, and in addition to the inherited member functions, functions performing assignment and input and output of the contents of the array, are provided.

See documentation of the parametric class `ArrayGenSimple(Type)` implemented in Diffpack for a description of the interface of the class.

SEEALSO

class ArrayGenSimple(Type), class ArrayGenSimplest(nMatrix), class VecSimple(nMatrix)

Chapter 3

Least squares computations

3.1 Ordinary least squares

3.1.1 LeastSquaresQR

NAME

LeastSquaresQR - a class for ordinary least squares computations

INCLUDE

```
include "least_squares.h"
```

SYNTAX

```
class LeastSquaresQR
{
protected:
  nMatrix qr;
  nVector alpha;
  nIntVector pivot;

  void decompose();

public:
  LeastSquaresQR(const nMatrix& X);
  virtual ~LeastSquaresQR () { }

  virtual nMatrix solve(const nMatrix& Y) const; // returns coefficient matrix
  virtual nMatrix fit(nMatrix& Y) const;         // overwrites Y with fit and
                                                 // returns coefficient matrix
  virtual SymMatrix XtXinv () const;             // computes inv(X'X)

  void Qtrans(nMatrix& X) const;                 // overwrites X with QX
  void QbackTrans(nMatrix& X) const;             // overwrites X with Q'X
  nMatrix backFit(const nMatrix & z) const;      // bactransforms fitted part of Z

  nMatrix solveCoeff(const nMatrix& z) const;    // solves for coefficient matrix

  TriangMatrix getRt() const;
  nIntVector getPivot() const { return pivot;}   // returns pivot vector

  // computation of the least squares solution B of Y = XB:
  friend nMatrix leastSquaresSolution(const nMatrix& X, const nMatrix& Y);
};
```

KEYWORDS

least squares, QR decomposition

DESCRIPTION

The class implements least squares solutions through QR decomposition. We determine `B` minimizing the Euclidean norm of `Y-XB`, by Householder transformation of `X`: `QX = R`. Here, `R` is upper triangular and `Q` unitarian. Pivoting is used. Multiple columns of `Y` and `B` are allowed. `X` must have full rank.

The QR-representation itself should be used for computing quantities of interest such as the variance estimate, residual vector, hat matrix etc., rather than brute force computations using `X`, `B` and `Y` directly. Auxiliary functions are provided for these purposes.

The method is numerically stable, but may be slower than methods based on solving the normal equations by Cholesky decomposition.

The QR-transformation follows the algorithm from Businger and Golub (1971), and the regression problem is solved by using the methods described in Goodall (1993).

REFERENCES:

Businger, P. and Golub, G.H.: "Linear Least Squares Solutions by Householder Transformations", in Wilkinson, J.H. and Rheinsch, C.: Linear Algebra, Springer-Verlag, 1971.

Goodall, C.R.: "Computations Using the QR Decomposition", in Rao, C.R. (ed.): Handbook of Statistics 9. Computational Statistics, 1993, pp. 492-496.

CONSTRUCTORS AND INITIALIZATION

The class has one constructor, taking the `X` matrix as argument. Transformation to QR form is performed by the constructor, and may be reused.

MEMBER FUNCTIONS

See also the SYNTAX section.

The basic operations are solving for `B`, computing the fitted `Y`, and transformations by Q and its inverse (equal to its transpose). The member functions vary in the way arguments are passed and in combinations of basic operations.

`backFit` - backtransforms fitted part of the transformed `Y`, `Z=QY`. The argument to the function is the `m x p` matrix `Zhat`, and the fitted value of `Y`, `Yhat`, is returned. The function performs a similar operation as `QbackTrans`, but takes advantage of the fact that the last `m-n` rows of `Zhat` are zero. Here `n` is the rank of the matrix `X` (`X` must have full rank) in the regression problem `Y=XB`.

`fit` - solves the system `Y=XB`, overwrites the `nMatrix` reference argument `Y` with the fitted value, `yhat`, and returns the estimated coefficient matrix.

`solve` - solves the system `Y=XB`. The only argument is the matrix `Y`, and the estimated coefficient matrix is returned.

`solveCoeff` - solves the system `Z=QXB`, where the `nMatrix` argument `Z` is the Q-transformed of `Y`, and returns the coefficient matrix `B`.

`XtXinv` - computes the matrix `inv(X'X)`, to be used in computation of the covariance matrix of the estimated coefficients.

FILES

least_squares.C

EXAMPLE

```
/*************************************************************************/
/* Computation of the hat-matrix H in yhat = Hy, and the leverage, diag(H),  */
```

```
/* for the regression problem y=Xb.                                          */
/*                                                                           */
/* Let X be a n by p matrix, and y n by q. The fitted y is given by          */
/*                                                                           */
/*     yhat = Q'zhat,                                                        */
/*                                                                           */
/* where zhat is the solution of the Q-transformed problem:                  */
/*     Qy = z = QXb = Rb.                                                    */
/*                                                                           */
/* Since the last n-p rows of zhat are zero, yhat can be computed by         */
/*                                                                           */
/*     yhat = Q1'z1hat = Q1'z1 = Q1'R1b = Q1'Q1Xb = Q1'Q1y                   */
/*                                                                           */
/* and                                                                       */
/*     H = Q1'Q1                                                             */
/*                                                                           */
/* Q1 is the n by p left submatrix of Q, and z1 the upper n by q submatrix   */
/* of z = Qy.                                                                */
/*****************************************************************************/

#include <iostream.h>
#include <fstream.h>
#include <least_squares.h>

main(int argc, char* argv[])
{
  int n=atoi(argv[1]);
  int p=atoi(argv[2]);
  nMatrix X(n,p);          // n by p matrix of independent variables

  ifstream in("input.file",ios::in);
  X.scan(in);

  LeastSquaresQR lsq(X);   // QR-decomposition of X

  nMatrix H1(identity(n)); // n by n identity matrix

  lsq.Qtrans(H1);          // computes QH1 = QI = Q (n by n)


  // backFit takes into account the zeros below row p, so that
  // Q1'Q1, and not Q'Q as would be the case with Qbacktrans, is returned.

  nMatrix H = lsq.backFit(H1);

  ofstream out("out.file",ios::out);

  out << "Hat matrix:\n" << H;
  out << "leverage:\n" << H.diag();
}
```

SEEALSO

class GenLeastSquaresQR, class nMatrix

AUTHOR

Jon Helgeland, NR

3.2 Generalized least squares

3.2.1 GenLeastSquaresQR

NAME

GenLeastSquaresQR - a class for generalized least squares computations

INCLUDE

```
include "least_squares.h"
```

SYNTAX

```
class GenLeastSquaresQR: public LeastSquaresQR
{
 protected:
  TriangMatrix Bt;

  void QSymmQt(SymMatrix&);
  TriangMatrix setBt(SymMatrix&);

 public:
  GenLeastSquaresQR(const nMatrix& X, SymMatrix& Sigma);
  // Sigma is overwritten

  nMatrix solve(const nMatrix& Y) const; // returns coefficient matrix
  nMatrix fit(nMatrix& Y) const;         // overwrites Y with fit and returns
                                         // coefficient matrix
  nMatrix solveResid(const nMatrix& )const;
                              // From z1, finds fitted residuals (eta1, nu2)
  SymMatrix XtXinv () const;             // matrix factor of covariance matrix
  TriangMatrix getBt() const {return Bt;}
};
```

KEYWORDS

generalized least squares, least squares, QR decomposition

DESCRIPTION

The class implements generalized least squares solutions. The solution vector `b` minimizes

```
(y-Xb)'inv(Sigma)(y-Xb),
```

where `Sigma` is symmetric and positive definite, and `X` has full rank.

Householder transformation of `X` is used: `QX=R`, as well as reverse Cholesky decomposition of the transformed Sigma matrix: `BBt = Q Sigma Qt`, with `B` upper triangular.

This algorithm is numerically more stable than methods based on Cholesky decomposition of `Sigma` followed by transformation to an ordinary least squares problem.

The generalized least squares computations are based on the algorithms described in Goodall (1993).

REFERENCE:

Goodall, C.R.: "Computations Using the QR Decomposition", in Rao, C.R. (ed.) Handbook of Statistics 9. Computational Statistics, 1993, pp. 492-496.

CONSTRUCTORS AND INITIALIZATION

The constructor takes one `nMatrix` argument `X` and one `SymMatrix` argument `Sigma`, holding the covariance matrix for the residuals. A QR-decomposition of the matrix `X` is performed by the inherited `LeastSquaresQR` constructor. To conserve memory in large problems, `Sigma` is overwritten and the storage used for the `TriangMatrix` holding the reverse Cholesky decomposition of the matrix `Q Sigma Qt`.

Changes to the `SymMatrix` entered as actual argument to the function will have no impact on the `TriangMatrix` member because of the copy count mechanism for the matrix classes, but should be avoided to prevent creation of a copy of the matrix representation.

MEMBER FUNCTIONS

The member functions `solve` and `fit` are virtual functions inherited from the base class `LeastSquaresQR`, and re-implemented for generalized least squares.

`getBt` - returns the `TriangMatrix Bt`, where `BBt` is the reverse Cholesky decomposition of `Q Sigma Qt`.

`setBt` - protected member function used in the constructor, returning the reverse Cholesky decomposition `Q Sigma Qt` as a lower triangular matrix `Bt`.

`solveResid` - this function is called by `solve` and `fit`, and computes the residuals of the transformed least squares problem:

```
min  nu'nu; z = QX beta + eta; eta = B nu.
```

The function argument is the transformed y-matrix `z=Qy`, and it is assumed that the matrix is a `m` x 1 column matrix.

The return value is the minimizing value of the vector

```
(eta1', nu2')',
```

where `eta1` is the upper `n`-dimensional subvector of `eta`, and `nu2` the lower `m-n`-dimensional subvector of `nu`.

See the reference Goodall (1993) for a further description of the method.

`XtXinv` - computes the matrix `inv(Xt*inv(Sigma)*X)`, to be used in computation of the covariance matrix of the estimated coefficients.

FILES

least_squares.C

EXAMPLE

```
// Computation of the coefficient matrix for a generalized least squares
// problem

#include <iostream.h>
#include <fstream.h>
#include <least_squares.h>

main(int argc, char* argv[])
{
  int n=atoi(argv[1]);
  int p=atoi(argv[2]);
  nMatrix D(n,p+1);      // n by p+1 matrix of independent variables and
                         // one response variable

  ifstream in1("data.file",ios::in);
  D.scan(in1);

  nMatrix X = D.submatrix(1,n,1,p);
  nMatrix y = D.submatrix(1,n,p+1,p+1);

  SymMatrix Cov(n);

  ifstream in2("covar.file",ios::in);
  Cov.scan(in2);

  GenLeastSquaresQR glsq(X,Cov);   // QR-decomposition of X and reversed
                                   // Cholesky decomp. of QCovQ'

  nMatrix BCoef = glsq.solve(y);

  ofstream out("out.file",ios::out);

  out << "Coefficient matrix:\n" << BCoef;
}
```

SEEALSO

class LeastSquaresQR, class nMatrix

AUTHOR

Jon Helgeland, NR

Chapter 4

Random number generators

4.1 A stream of random numbers

4.1.1 RandomStream

NAME

RandomStream - a class for generating a stream of random numbers

INCLUDE

```
include "rand.h"
```

SYNTAX

```
class RandomStream
{
 public:
  RandomStream(int i1,int i2=0) {xint1=i1; xint2=i2;}
  RandomStream();
};
```

KEYWORDS

random numbers, random stream

DESCRIPTION

The class keeps in order the stream of uniformly distributed random numbers that is generated by the random number generators in the `RandomGen` hierarchy.

The random numbers are generated by the rule (Knuth, p.105)

$$X(n+1) = (271828183 * X(n) - 314159269 * X(n-1)) \bmod (2^{31} - 1).$$

The period is

$$4.61 * 10^{19}.$$

REFERENCE:

Knuth, D.E.: "Seminumerical Algorithms", The art of computer programming, Vol. 2, 2nd ed., Addison-Wesley, 1981.

CONSTRUCTORS AND INITIALIZATION

One constructor has two integer arguments, giving two seeds for the generator. The first integer has to be specified, the other has default value zero. The class also provides a default constructor. By calling this constructor, the second seed is set to zero, and the first to the point of time.

A global pointer to `RandomStream` is always created. This pointer will be initialized by the default construction of objects in the `RandomGen` hierarchy.

MEMBER FUNCTIONS

None.

FILES

rand.C

EXAMPLE

See class RandUnif

SEEALSO

class RandomCont, class RandomDisc, class RandomGen

AUTHOR

Jon Helgeland, NR

4.2 Abstract base classes

4.2.1 RandomGen

NAME

RandomGen - a base class for generating random numbers

INCLUDE

```
include "rand.h"
```

SYNTAX

```
class RandomGen
{
 protected:
  RandomStream *rstr;
 public:
  RandomGen(RandomStream* r)
    : rstr(r) { }
  RandomGen(int i1, int i2=0);
  RandomGen();

  virtual ~RandomGen() { }
};
```

KEYWORDS

random numbers, random stream

DESCRIPTION

The class is a base class for generation of random numbers.

The numbers are generated on a global random stream, if no other stream is explicitly specified by the `RandomStream` pointer argument of the first constructor. See also CONSTRUCTORS AND INITIALIZATION.

CONSTRUCTORS AND INITIALIZATION

The class has three constructors. The first, taking a pointer to a `RandomStream` as argument, constructs a random number generator operating on this local random stream. The two other constructors initialize a global random stream. Using the default constructor implies that the seeds for the stream are set to the point of time and zero respectively. Alternatively, one or both seeds can be specified by using the second constructor.

Note that it is assumed that the integer argument constructor is called for only one object of the classes in the `RandomGen` hierarchy in a program, so that all random number generators constructed, operate on the same random stream. If one tries to use this constructor a second time, a warning is issued, and the new seeds are ignored.

If one prefers to operate on different random streams for different generators, a `RandomStream` object should be constructed for each generator, and its address entered as actual argument to the first constructor.

MEMBER FUNCTIONS

None.

FILES

rand.C

EXAMPLE

See class RandomCont, class RandUnif

SEEALSO

class RandomCont, class RandomDisc, class RandomStream

AUTHOR

Jon Helgeland, NR

4.2.2 RandomCont

NAME

RandomCont - a base class for generating continuously distributed random numbers

INCLUDE

```
include "rand.h"
```

SYNTAX

```
class RandomCont : public RandomGen
{
 public:
  RandomCont(RandomStream* r)
    : RandomGen(r) { }
  RandomCont(int i1, int i2=0)
    : RandomGen(i1,i2) { };
  RandomCont()
    : RandomGen() { };

  virtual double operator()() = 0;
};
```

KEYWORDS

continuous distribution, random numbers, random stream

DESCRIPTION

The class is a base class for classes generating random numbers from continuous distributions. It is inherited from the general random generator class `RandomGen`.

The numbers will be generated on a global random stream, if no other stream is explicitly specified by the `RandomStream` pointer argument of the first constructor.

CONSTRUCTORS AND INITIALIZATION

The class has three constructors. The first, taking a pointer to a `RandomStream` as argument, constructs a random number generator operating on this local random stream. The two other constructors initialize a global random stream. Using the default constructor implies that the seeds for the stream are set to the point of time and zero respectively. Alternatively, one or both seeds can be specified by using the second constructor.

See also the documentation of class `RandomGen`.

MEMBER FUNCTIONS

`operator()` - virtual function that is to generate a random number from a continuous distribution.

FILES

rand.C

EXAMPLE

```
#include <iostream.h>
#include <stdlib.h>
#include <matrix.h>
#include <rand.h>

main(int argc, char* argv[])
{
  int ns = atoi(argv[1]);          // number of simulations
  int seed = atoi(argv[2]);        // seed for random generator
  char* type = argv[3];            // type of distribution

  cout << type << endl;

  RandomCont* randn;
  if (!strcmp(type,"n"))
    randn = new RandStdNormal();   // attached to global stream
  else if (!strcmp(type,"u"))
    randn = new RandUnif();        // attached to global stream

  nVector randvec(ns);

  for (int i=1;i<=ns;i++)
    randvec(i) = (*randn)();

  cout << randvec;
  cout << endl;

  delete randn;
}
```

SEEALSO

class RandomGen, class RandomStream

AUTHOR

Jon Helgeland, NR

4.2.3 RandomDisc

NAME

RandomDisc - a base class for generation of random numbers from discrete distributions

INCLUDE

```
include "rand.h"
```

SYNTAX

```
class RandomDisc : public RandomGen
{
 public:
  RandomDisc(RandomStream* r)
    : RandomGen(r) { }
  RandomDisc(int i1, int i2=0)
    : RandomGen(i1,i2) { };
  RandomDisc()
    : RandomGen() { };

  virtual int operator()() = 0;
};
```

KEYWORDS

discrete distribution, random numbers, random stream

DESCRIPTION

The class is a base class for random number generators for discrete distributions, and is inherited from the general random generator class `RandomGen`.

The numbers are generated on a global random stream, if no other stream is explicitly specified by the `RandomStream` pointer argument of the first constructor.

CONSTRUCTORS AND INITIALIZATION

The class has three constructors. The first, taking a pointer to a `RandomStream` as argument, constructs a random number generator operating on this local random stream. The two other constructors initialize a global random stream. Using the default constructor implies that the seeds for the stream are set to the point of time and zero respectively. Alternatively, one or both seeds can be specified by using the second constructor.

See also the documentation of class `RandomGen`.

MEMBER FUNCTIONS

`operator()` - virtual function that is to generate a random number from a discrete distribution.

FILES

rand.C

EXAMPLE

See class RandPoisson

SEEALSO

class RandomGen, class RandomStream

AUTHOR

Turid Follestad, NR

4.3 Random number generators for different distributions

4.3.1 RandUnif

NAME

RandUnif - a class for generating uniformly distributed random numbers

INCLUDE

```
include "rand.h"
```

SYNTAX

```
class RandUnif : public RandomCont
{
 public:
  RandUnif(RandomStream* r)
    : RandomCont(r) { }
  RandUnif(int i1, int i2=0)
    : RandomCont(i1,i2) { };
  RandUnif()
    : RandomCont() { };

  virtual double operator()(){ return unif();}
};
```

KEYWORDS

continuous distribution, random numbers, random stream, uniform distribution

DESCRIPTION

The class is a random number generator for the uniform [0,1] distribution, as well as a base class for classes generating random numbers from several other continuous distributions. It is inherited from the base class for continuous random generators, `RandomCont`.

The numbers are generated on a global random stream, if no other stream is explicitly specified by the `RandomStream` pointer argument of the first constructor.

CONSTRUCTORS AND INITIALIZATION

The class has three constructors. The first, taking a pointer to a `RandomStream` as argument, constructs a random number generator operating on this local random stream. The two other constructors initialize a global random stream. Using the default constructor implies that the seeds for the stream are set to the point of time and zero respectively. Alternatively, one or both seeds can be specified by using the second constructor.

See also the documentation of class `RandomGen`.

MEMBER FUNCTIONS

`operator()` - generates a random number from the uniform distribution.

FILES

rand.C

EXAMPLE

```
#include <iostream.h>
#include <stdlib.h>
#include <matrix.h>
#include <rand.h>

main(int argc, char* argv[])
{
  int ns = atoi(argv[1]);     // number of simulations
  int seed = atoi(argv[2]);   // seed for random generator

  RandUnif randu(seed);       // global stream initialized, seeds  seed  and zero
  nVector randvec(ns);

  for (int i=1;i<=ns;i++)
    randvec(i) = randu();

  cout << "Uniform distribution:\n";
  cout << randvec;

  RandUnif randu1(seed);      // argument ignored, global stream already
                              // initialized, attached to global stream

  RandUnif randu2();          // OK, attached to global stream

  // ....

  RandomStream* rstr = new RandomStream(seed);
                              // creating a local random stream

  RandUnif randlocal(rstr);   // OK, attached to local stream

  for (i=1;i<=ns;i++)
    randvec(i) = randlocal();

  cout << "Uniform distribution, local stream:\n";
  cout << randvec;

  delete rstr;
}
```

SEEALSO

class RandomCont, class RandomGen, class RandomStream

AUTHOR

Jon Helgeland, NR

4.3.2 RandStdNormal

NAME

RandStdNormal - a class for generating normally distributed random numbers

INCLUDE

```
include "rand.h"
```

SYNTAX

```
class RandStdNormal: public RandUnif
{
 public:
  RandStdNormal(RandomStream* r)
    : RandUnif(r){}
  RandStdNormal(int i1, int i2=0)
    : RandUnif(i1,i2){}
  RandStdNormal()
    : RandUnif(){}

  double operator()();
};
```

KEYWORDS

normal distribution, random numbers

DESCRIPTION

The class is a random number generator, generating random numbers from the standard normal distribution on a random stream.

The numbers are generated on a global random stream, if no other stream is explicitly specified by the `RandomStream` pointer argument of the first constructor.

CONSTRUCTORS AND INITIALIZATION

The class has three constructors. The first, taking a pointer to a `RandomStream` as argument, constructs a random number generator operating on this local random stream. The two other constructors initialize a global random stream. Using the default constructor implies that the seeds for the stream are set to the point of time and zero respectively. Alternatively, one or both seeds can be specified by using the second constructor.

See also the documentation of class `RandomGen`.

MEMBER FUNCTIONS

`operator()` - generates a random number from the standard normal distribution.

FILES

rand.C

EXAMPLE

See class RandUnif. Classes `RandUnif` and `RandStdNormal` are used equivalently.

SEEALSO

class RandomGen, class RandNormal, class RandomStream, class RandUnif

AUTHOR

Jon Helgeland, NR

4.3.3 RandNormal

NAME

RandNormal - a class for generating normally distributed random numbers

INCLUDE

```
include "rand.h"
```

SYNTAX

```
class RandNormal: public RandStdNormal
{
  double meanval;  // mean value
  double stddev;   // standard deviance
 public:
  RandNormal(RandomStream* r, double mval, double var)
    : RandStdNormal(r), meanval(mval), stddev(sqrt(var)) {}
  RandNormal(int i1, double mval=0, double var=1, int i2=0)
    : RandStdNormal(i1,i2), meanval(mval), stddev(sqrt(var)) {}
  RandNormal (double mval, double var)
    : RandStdNormal(), meanval(mval), stddev(sqrt(var)) {}

  double operator()() { return meanval + stddev*RandStdNormal::operator()(); }
};
```

KEYWORDS

normal distribution, random numbers

DESCRIPTION

The class is a random number generator, generating normally distributed random numbers on a random stream.

The numbers are generated on a global random stream, if no other stream is explicitly specified by the `RandomStream` pointer argument of the first constructor.

If random numbers from the standard normal distribution, N(0,1), are to be generated, the class `RandStdNormal` could be used, to minimize the number of arithmetic operations in the member funtion generating the random numbers.

CONSTRUCTORS AND INITIALIZATION

The class has three constructors, all taking the distribution parameters mean and variance as two double arguments. The first, taking in addition a pointer to a `RandomStream` argument, constructs a random number generator operating on this local random stream. The two other constructors initialize a global random stream. Using the constructor with no integer arguments implies that the seeds for the stream are set to the point of time and zero respectively. Alternatively, one or both seeds can be specified by using the second constructor.

See also the documentation of class `RandomGen`.

MEMBER FUNCTIONS

`operator()` - generates a random number from the normal distribution.

FILES

rand.C

EXAMPLE

```
#include <iostream.h>
#include <stdlib.h>
#include <matrix.h>
#include <rand.h>

main(int argc, char* argv[])
{
  int ns = atoi(argv[1]);     // number of simulations
  double mv = atof(argv[2]);  // mean value
  double var = atof(argv[3]); // variance

  RandNormal rand(mv,var);    // default initialization of global random stream
  nVector randvec(ns);

  for (int i=1;i<=ns;i++)
    randvec(i) = rand();

  cout << "Normal distribution:\n";
  cout << randvec;
}
```

SEEALSO

class RandomGen, class RandStdNormal, class RandomStream, class RandUnif

AUTHOR

Turid Follestad, NR

4.3.4 RandExp

NAME

RandExp - a class for generating exponentially distributed random numbers

INCLUDE

```
include "rand.h"
```

SYNTAX

```
class RandExp: public RandUnif
{
  double lambda;  // distribution parameter
 public:
  RandExp(RandomStream* r, double l=1.0);
  RandExp(int i1, double l=1.0, int i2=0);
  RandExp(double l=1.0);

  double operator()();
};
```

KEYWORDS

exponential distribution, random numbers

DESCRIPTION

The class is a random number generator, generating exponentially distributed random numbers on a random stream.

The numbers are generated on a global random stream, if no other stream is explicitly specified by the `RandomStream` pointer argument of the first constructor.

CONSTRUCTORS AND INITIALIZATION

The class has three constructors, all having a double argument indicating the parameter of the distribution. By default, the parameter value is set to one. The first constructor, taking a pointer to `RandomStream` argument, constructs a generator operating on this local random stream. The two other constructors initialize a global random stream. Using the constructor with no integer arguments implies that the seeds for the stream are set to the point of time and zero respectively. Alternatively, one or both seeds can be specified by using the second constructor.

See also the documentation of class `RandomGen`.

MEMBER FUNCTIONS

`operator()` - generates a random number from the exponential distribution.

FILES

rand.C

EXAMPLE

```
#include <iostream.h>
#include <stdlib.h>
#include <matrix.h>
#include <rand.h>

main(int argc, char* argv[])
{
  int ns = atoi(argv[1]);         // number of simulations
  double lambda = atof(argv[2]);  // parameter of distribution

  RandExp rand(lambda);     // default initialization of global random stream
  nVector randvec(ns);

  for (int i=1;i<=ns;i++)
    randvec(i) = rand();

  cout << "Exponential distribution:\n";
  cout << randvec;
}
```

SEEALSO

class RandomGen, class RandomStream, class RandUnif

AUTHOR

Jon Helgeland, NR

4.3.5 RandGamma

NAME

RandGamma - a class for generating random numbers from the gamma distribution

INCLUDE

```
include "rand.h"
```

SYNTAX

```
class RandGamma: public RandUnif
{
  double lambda;          // scaling parameter
  double alpha;

 public:
  RandGamma(RandomStream* r, double alpha=1.0, double lambda=1.0);
  RandGamma(int i1, double alpha=1.0, double lambda=1.0, int i2=0);
  RandGamma(double alpha=1.0, double lambda=1.0);

  double operator()();
};
```

KEYWORDS

gamma distribution, random numbers

DESCRIPTION

The class is a random number generator, generating gamma distributed random numbers on a random stream. The distribution function is defined as

```
                                          alpha-1
f(x) = lambda/gamma(alpha) * (lambda*x)       exp(-lambda*x),
```

where `alpha` and the scaling parameter `lambda` are the two parameters of the distribution, and `gamma(alpha)` is the gamma-function.

The random numbers are generated by the method in Ripley (1987).

The numbers are generated on a global random stream, if no other stream is explicitly specified by the `RandomStream` pointer argument of the first constructor.

REFERENCE:

Ripley, B.D.: "Stochastic Simulation", Wiley, 1987, p.88.

CONSTRUCTORS AND INITIALIZATION

The class has three constructors, all having two double arguments specifying the parameters `alpha` and `lambda`, the scaling parameter, of the gamma distribution. The first constructor, taking a pointer to a `RandomStream` argument, creates a random number generator operating on this local random stream. The two other constructors initialize a global random stream. Using the constructor with no integer arguments implies that the seeds for the stream are set to the point of time and zero respectively. Alternatively, one or both seeds can be specified by using the second constructor.

See also the documentation of class `RandomGen`.

MEMBER FUNCTIONS

`operator()` - generates a random number from the gamma distribution.

FILES

rand.C

EXAMPLE

```
#include <iostream.h>
#include <stdlib.h>
#include <matrix.h>
#include <rand.h>

main(int argc, char* argv[])
{
  int ns = atoi(argv[1]);           // number of simulations

  double alpha = atof(argv[2]);     // parameters of distribution
  double lambda = atof(argv[3]);

  RandGamma rand(alpha, lambda);
                      // default initialization of global random stream
  nVector randvec(ns);

  for (int i=1;i<=ns;i++)
    randvec(i) = rand();

  cout << "Gamma distribution:\n";
  cout << randvec;
}
```

SEEALSO

class RandomGen, class RandomStream, class RandUnif

AUTHOR

Jon Helgeland and Tove Andersen, NR

4.3.6 RandPoisson

NAME

RandPoisson - a class for generating random numbers from the Poisson distribution

INCLUDE

```
include "rand.h"
```

SYNTAX

```
class RandPoisson : public RandomDisc
{
   double C;          // distribution parameter

 public:
   RandPoisson(RandomStream* r, double lambda=1.0);
   RandPoisson(int seed1, double lambda=1.0, int seed2=0);
   RandPoisson(double lambda=1.0);

   ~RandPoisson () { if (Rand!=NULL) delete Rand; }

   int operator()();
};
```

KEYWORDS

Poisson distribution, random numbers

DESCRIPTION

The class is a random number generator, generating random numbers from the Poisson distribution on a random stream, by the *Centred search algorithm* in Kemp and Kemp (1991).

The numbers are generated on a global random stream, if no other stream is explicitly specified by the `RandomStream` pointer argument of the first constructor.

REFERENCE:

Kemp, C.D. and Kemp, A.W.: "Poisson Random Variate Generation", Applied Statistics, `40`(1), 1991, pp. 143-158.

CONSTRUCTORS AND INITIALIZATION

The class has three constructors, all having a double argument indicating the parameter of the distribution. By default, the parameter value is set to one. The first constructor, taking a pointer to `RandomStream` argument, constructs a generator operating on this local random stream. The two other constructors initialize a global random stream. Using the constructor with no integer arguments, implies that the seeds for the stream are set to the point of time and zero respectively. Alternatively, one or both seeds can be specified by using the second constructor.

See also the documentation of class `RandomGen`.

MEMBER FUNCTIONS

`operator()` - generates a random number from the Poisson distribution.

FILES

rand.C

EXAMPLE

```
#include <iostream.h>
#include <stdlib.h>
#include <matrix.h>
#include <rand.h>

main(int argc, char* argv[])
{
  int ns = atoi(argv[1]);            // number of simulations
  int seed = atoi(argv[2]);          // seed for random generator

  double lambda = atof(argv[3]);     // parameter of distribution

  RandPoisson rand(seed,lambda);
        // initialization of global random stream with seeds seed and zero
  nIntVector randvec(ns);

  for (int i=1;i<=ns;i++)
    randvec(i) = rand();

  cout << "Poisson distribution:\n";
  cout << randvec;
}
```

SEEALSO

class RandomDisc, class RandomGen, class RandomStream

AUTHOR

Ingvar Koppervik and Tove Andersen, NR

Chapter 5

Some general tools

5.1 Functions

5.1.1 nfac

NAME

nfac - a function computing the factorial of an integer

INCLUDE

```
include "bincoef.h"
```

SYNTAX

```
int nfac(int n);
```

KEYWORD

factorial

DESCRIPTION

The function computes the factorial of the integer argument by computing the exponential of the sum of logarithms of numbers from 1 up to `n`. As the size of integers are restricted, this function, returning an integer, can be used only for values of `n` less than or equal to 12. For higher values of `n`, the function `dnfac`, returning the factorial funtion as a double value, can be used.

FILES

bincoef.C

SEEALSO

dnfac

AUTHOR

Turid Follestad, NR, translated from a Splus function written by Gro Hagen, NR

5.1.2 binCoef

NAME

binCoef - a function computing the binomial coefficient

INCLUDE

```
include "bincoef.h"
```

SYNTAX

```
int binCoef(int n, int r);
```

KEYWORD

binomial coefficient

DESCRIPTION

The function computes the binomial coefficient `n` over `r` by computing sums of logarithms for the series `n-r+1` to `n` (sumlog(n)) and 1 to `r` (sumlog(r)), returning the value exp(sumlog(n)-sumlog(r)). As the size of integers are restricted, this function, returning an integer, can be used only for values of `n` and `r` giving a value for the binomial coefficient less than the integer size limit. For values not satisfying this constraint, the function `dbinCoef`, returning a double value, might be used.

FILES

bincoef.C

SEEALSO

dbinCoef

AUTHOR

Turid Follestad, NR, based on a Splus function written by Gro Hagen, NR

5.1.3 dnfac

NAME

dnfac - a function computing factorial for large numbers

INCLUDE

```
include "bincoef.h"
```

SYNTAX

```
double dnfac(int n);
```

KEYWORD

factorial

DESCRIPTION

The function computes the factorial of the integer argument by computing the exponential of the sum of logarithms of numbers from 1 up to `n`. As the size of integers are restricted, this function, returning double and not integer, should be used when the result is expected to be greater than the integer size limit. The function `nfac`, returning an integer, can be used for values of `n` less than or equal 12.

FILES

bincoef.C

SEEALSO

nfac

AUTHOR

Turid Follestad, NR, translated from a Splus function written by Gro Hagen, NR

5.1.4 dbinCoef

NAME

dbinCoef - function computing the binomial coefficient for large numbers

INCLUDE

```
include "bincoef.h"
```

SYNTAX

```
double dbinCoef(int n, int r);
```

KEYWORD

binomial coefficient

DESCRIPTION

The function computes the binomial coefficient for `n` over `r` by computing sums of logarithms for the series `n-r+1` to `n` (sumlog(n)) and 1 to `r` (sumlog(r)), returning the value exp(sumlog(n)-sumlog(r)).

As the size of integers are restricted, this function, returning double and not integer, should be used when the result is expected to be a large integer value.

FILES

bincoef.C

SEEALSO

binCoef

AUTHOR

Turid Follestad, NR, based on a Splus function written by Gro Hagen, NR

5.1.5 countNumbers

NAME

countNumbers - counts the numbers on a file.

INCLUDE

```
include "countNumbers.h"
```

SYNTAX

```
int countNumbers(char* file);
```

KEYWORDS

file

DESCRIPTION

The function returns the number of numbers on a file with the name specified by the argument to the function. The file is expected to contain numerical entries only.

FILES

countNumbers.C

AUTHOR

Magne Aldrin, NR

5.2 Abstract base class for data sets

5.2.1 DataSet

NAME

DataSet - a base class for sample data sets

INCLUDE

```
include "DataSet.h"
```

SYNTAX

```
class DataSet
{
 public:
  virtual ~DataSet () { };

  virtual BooLean ok () = 0;       // returns dpTRUE if object status ok
  virtual int getNobs() const = 0; // returns number of observations
  virtual void cleanUp () = 0;     // editing tha data

  virtual void extract(const nIntVector& ind, DataSet& ds) const = 0;
       // Extracts observations indexed by  ind  returning a new DataSet

  virtual void extract(int f, int t, DataSet& ds) const = 0;
       // Extracts observations indexed from f to t (obs. no f and t included)
       // returning a new DataSet

  virtual void remove (const nIntVector& ind) = 0;
       // Removes observations indexed by  ind

  virtual void remove (int f, int t) = 0;
       // Removes observations indexed f to t (obs. no f and t included)

  virtual void insert (const DataSet& ds, int from) = 0;
       // Inserts DataSet ds after observation number  from-1

  virtual void scan (Is in) = 0;
  virtual void print (Os out) const = 0;
};
```

KEYWORDS

data set, data editing

DESCRIPTION

The class provides an interface for a general sample data set, with member functions representing operations that is supposed to be useful for several kinds of data sets.

CONSTRUCTORS AND INITIALIZATION

No constructors.

MEMBER FUNCTIONS

All member functions are pure virtual. Their intended use is indicated by the comments in the SYNTAX section. The `nIntVector` arguments represents a vector of indices indicating which observations are to be extracted, removed or inserted. `remove`, `insert` and `append` all modify the object, and `extract` creates a new one, which is returned through the `DataSet`-argument.

`scan` should read the contents of the data set from an input source, and `print` should print the contents of the data set to an output source.

FILES

None.

EXAMPLE

See class SpatialData in NSPACE and class RegData in NMODEL.

SEEALSO

class SpatialData in NSPACE, class RegData in NMODEL

AUTHOR

Turid Follestad, NR

Appendix A

Some functions and classes from Diffpack

A.1 Boolean variables

A.1.1 BooLean

NAME

BooLean - enum variable for boolean (logical) variables

INCLUDE

```
include "BooLean.h"
```

SYNTAX

```
enum BooLean  // the boolean variable
{
  dpFALSE = 0,
  dpTRUE = 1,
  OFF = 0,
  ON  = 1,
  dpFAILURE = 0,
  dpSUCCESS = 1
};
```

KEYWORDS

boolean, logical

DESCRIPTION

Usually, the `int` variable is used for boolean values in C and C++ programs. For most purposes, it is safer to use a distinct type `boolean` that is not automatically convertable with `int`. To avoid name conflicts with other libraries, this boolean variable is spelled as `BooLean`. It is implemented as an `enum`. Conversion to `int` is automatic (or by use of `operator int`), while conversion from an int (e.g., the return value of C library functions like `strcmp`) to a `BooLean` is performed explicitly by the function `getBooLean`. As usual in Diffpack, there are conversion functions `assignEnum` from int or a `String` to the enum variable, as well as `getEnumValue` and `getEnumDescription` that returns the name of the enum value and a short description of the meaning of that value, respectively.

DEVELOPED BY

SINTEF Applied Mathematics, Oslo, Norway, and University of Oslo, Dept. of Mathematics, Norway

AUTHOR

Hans Petter Langtangen, SINTEF/UiO

A.2 Simple vector templates

A.2.1 VecSimplest

NAME

VecSimplest(Type) - very simple vector, only allocation and subscripting

INCLUDE

```
include "VecSimplest.h"
```

SYNTAX

```
class VecSimplest(Type)  //: public virtual HandleId
{
protected:

  Type*   A;        // vector representation
  int     length;   // vector length
  BooLean internal; // dpTRUE: internal allocation, dpFALSE: do not delete A
                    // (A is borrowed from the user)

public:
  static long nbytes;         // total number of allocated bytes (all objects)
  static long nbytes_dealloc; // no of deallocated bytes
  static long narrays;        // no of allocated arrays
  static long narrays_dealloc;// no of deallocated arrays

protected:
  BooLean makeNew (int length); // used by redim for efficiency

  Type*   allocate (int length);
  void    deallocate ();
  Type*   borrow (Type* a, int n, int base = 0);

public:

  VecSimplest(Type) ();
  VecSimplest(Type) (int length);
  VecSimplest(Type) (Type* a, int n, int base = 0);
 ~VecSimplest(Type) ();

  BooLean redim (int length);
  BooLean redim (Type* a, int n, int base = 0);
  BooLean ok () const;

  int  size () const     { return length; }

  BooLean compatible (const VecSimplest(Type)& X) const;

  BooLean indexOk (int i, const char* message = NULL) const;
  const Type& operator () (int i) const;
        Type& operator () (int i);

  Type* getPtr0 ();
  Type* getPtr1 ();

  // illegal functions (included here with just an error message):
  VecSimplest(Type) (const VecSimplest(Type)& v);
```

```
  void operator =   (const VecSimplest(Type)& v);

  // CLASS_INFO  // removed (goal: as little data as possible in the class
};

//-------------------------------------------------------------------------
inline Type& VecSimplest(Type):: operator () (int i)
//-------------------------------------------------------------------------
{
#ifdef ARRAY_RANGECHECK
  indexOk(i);
#endif

  return  A[i];
}


//-------------------------------------------------------------------------
inline const Type& VecSimplest(Type):: operator () (int i) const
//-------------------------------------------------------------------------
{
#ifdef ARRAY_RANGECHECK
  indexOk(i);
#endif

  return  A[i];
}


//-------------------------------------------------------------------------
inline Type* VecSimplest(Type):: getPtr0 ()
//-------------------------------------------------------------------------
{
  return A+1;
}

//-------------------------------------------------------------------------
inline Type* VecSimplest(Type):: getPtr1 ()
//-------------------------------------------------------------------------
{
  return A;
}


//-------------------------------------------------------------------------
inline BooLean VecSimplest(Type):: redim (int length_)
//-------------------------------------------------------------------------
{
  return length == length_ ? dpFALSE : makeNew (length_);
}
```

KEYWORDS

vector class, array, base class

DESCRIPTION

The class implements a standard vector type. The public interface consists only of two constructors, a subscripting operator and some size information. In addition, the pointer representation of the vector can be returned from a member function (e.g. when using this vector in a C function). The base is unity. More operations, included `print`, `fill`, `operator=`, copy-constructors, `operator<<` etc., can be found in the derived class `VecSimple(Type)`.

If arithmetic operations have meanings (e.g. when `Type` is `float`, `double` or `dpComplex`) one can use the standard class `Vec(Type)` which is derived from `VecSimple(Type)`. A general base for the subscripting operator is provided in `ArrayGenSimplest(Type)`.

Since no operations, execpt a constructor without arguments, is assumed to exist for `Type`, this matrix class can be used as a general class for collecting (complicated) objects in a matrix.

CONSTRUCTORS AND INITIALIZATION

Three constructors are available. The constructor without parameters allocates no memory. To allocate the proper amount of memory at a later stage, the `redim` function can be used. Another constructor takes the number of entries in the vector as parameter and allocates memory for the corresponding vector representation. The third constructor takes a C array and its length and base index as parameters. This makes it easy for the vector to borrow its contents from an already existing C array. The C array will not be deleted by the `VecSimplest` class, but the content of the array may be changed. Note that `VecSimplest` cannot take a copy of the C array since `operator=` is not required for class `Type`. However, one may create a `VecSimple` object from a C array and then copy this class object.

To initialize the vector entries one must use the member function `operator()`.

MEMBER FUNCTIONS

Most member functions are self-explanatory.

`ok` - returns `dpTRUE` if the object is in an ok state, that is, if memory is allocated for the vector. In an ok state one can make the call `operator()(1)` (at least one entry is present in the vector).

`redim` - changes the dimension (length) of the vector (if necessary). There is also a `redim` function for borrowing a C-array.

`size` - returns the length of the vector.

`compatible` - checks if the object has compatible size with the object given as argument.

`operator()` - subscripting operator, can be used on the left side of an assignment. The function assumes that the base of the array is 1. Notice that there are two versions of this function, one is const and one is non-const. This is necessary to ensure that the compiler issues an error message if const arrays are altered by the subscript operator. The reader is encouraged to seek information such that a const/non-const pair of functions are thoroughly understood. For example, construction of a toy program may be helpful in experimenting with const and non-const.

`indexOk` - returns `dpFALSE` and an error message if the index is illegal, legal indices leads to a `dpTRUE` return value. There is an optional argument that can be used to give extra information, for example, where `indexOk` is called.

`getPtr0` - return access to the internal C array and assume that the base is 0 (note that internally in this class, the base is 1). This is the common function for sending the vector class object to a C function requiring a pointer.

`getPtr1` - return access to the internal C array and assume that the base is 1. A common use is when the object is to be transferred to C functions, e.g., Numerical Recipes routines.

Be aware that `VecSimplest` does not require the `Type` class to have an `operator=`. Hence, the copy constructor or the assignment operator has no meaning for `VecSimplest`. Since C++ automatically makes such functions if they are not explicitly declared (and such

automatically generated functions lead to serious errors in the present case), the functions are declared, but the content is just an error message.

EXAMPLES

```
VecSimplest(MyTool) g(4); // (1:4) vector of class MyTool objects
g(2) = MyTool (n,m,s);    // entry (2) is assigned a MyTool object

typedef MyTool* MyToolPtr;
VecSimplest(MyToolPtr)* gp;
gp = new VecSimplest(MyToolPtr)(5);
(*gp)(3) = MyTool (n,m,s);
int l = gp->size();       // l=5 (length of gp)

VecSimplest(MyTool) a;    // no memory is allocated
a.redim(10);              // allocates memory for 10 MyTool objects
```

SEE ALSO

class `MatSimplest(Type)`, `VecSimple(Type)`, `Vec(Type)`, `ArrayGenSimplest(Type)`

DEVELOPED BY

SINTEF Applied Mathematics, Oslo, Norway, and University of Oslo, Dept. of Mathematics, Norway

AUTHOR

Hans Petter Langtangen, SINTEF/UiO

A.2.2 VecSimple

NAME

VecSimple(Type) - simple vector, base=1, no arithmetic operations

INCLUDE

```
include "VecSimple.h"
```

SYNTAX

```
class VecSimple(Type) : public virtual VecSimplest(Type)
{

public:

  VecSimple(Type) ();
  VecSimple(Type) (int length);
  VecSimple(Type) (Type* a, int n, int base = 0);
  VecSimple(Type) (const VecSimple(Type)& X);
 ~VecSimple(Type) ();

  void fill (const Type& value);
  void operator = (const Type& value)     { fill(value); }
  void operator = (const VecSimple(Type)& X);
  void copy (const VecSimple(Type)& v);

  void print (Os os, const char* header = NULL,
              int nentries_per_line = 3) const;
  void scan  (Is is);

  void printAsIndex (Os os) const;
  void printAscii (Os os, const char* header = NULL,
                   int nentries_per_line = 3) const;
  void scanFromFile (const String& filename); // numbers on file -> empty vec

  friend Os& operator << (Os& os, const VecSimple(Type)& x);
  friend Is& operator >> (Is& is , VecSimple(Type)& x);

  // CLASS_INFO
};
```

KEYWORDS

vector class, array

DESCRIPTION

The class implements a standard matrix type derived from `VecSimplest(Type)`. The public interface consists of various operations such as subscripting (w/array-range-check if desired), assignment operators, fill functions, print and scan functions, etc. The subscript operator assumes that the base equals 1. If other bases are desired, the `ArrayGenSimplest(Type)` or `ArrayGen(Type)` classes can be used.

No arithmetic operations are used in any of the member functions. Hence this class is suited as a general vector class for various objects. The only requirement is that the object has a constructor with no arguments so that an array of the class' objects can be allocated by `new`. In addition, the class must define `operator=`, `operator<<` and `operator>>`. Macros for generating default versions of these operators are collected in the file `default_op.h`.

CONSTRUCTORS AND INITIALIZATION

Several constructors are available. The constructor without parameters allocates no memory. To allocate the proper amount of memory at a later stage, the `redim` function can be used. Another constructor takes the number of entries in the vector as parameter and allocates memory for the corresponding vector representation. Finally, a copy-constructor is available.

To initialize the vector entries one can use the member functions `fill` or `operator()`.

MEMBER FUNCTIONS

Most member functions are self-explanatory. Some member functions are inherited from and documented in class `VecSimplest(Type)`. New member functions are described below.

`copy` - makes the object a copy of the matrix argument, that is, the object is redimensioned according to the dimensions of `v` and thereafter `operator=` is called.

`print` - prints the contents of the vector. See documentation of the function `scan`.

`scan` - reads the vector from an `Is` object. The entries of an array can be written and read in two main formats, the binary format or the ascii format. When `print` is invoked, the format is determined by the state of the `Os` object (the result of the `getFormat` function). Each of the ascii/binary formats can have a header or not. The header is present if `print` is called with a non-`NULL` `header` string. In `scan` one detects the header if the first non-space character is `[`. If the header is present, its syntax is like this: ! [size]X text $ where `size` is the size of the array (for a vector it is the length, for a matrix it is a string on the form `rxc`, where `r` is the number of rows and `c` is the number of columns), `X` is a character that equals `b` in case of a binary format or space in case of an ascii format. `text` is the `header` string given to `print`. In `scan` this string is ignored, but it is available in the internal `Is` buffer so the programmer can in principle extract the header after `scan` is called.

After the header a character @ follows if the format is binary, if not, a space appears. Hence one can detect whether the format is binary or ascii even when the header does not appear. After the @, the binary format has two binary numbers: The number of array entries and the size of each entry.

If the header is present and the format is ascii, there will be `nentires_per_line` array entries on each output line. There array index will also be written. In all other cases, the array entries are written with no index, but in the ascii format there will be an extra space.

With a header the `scan` functions can read the dimensions of the array and call `redim` before reading the entries. Without a header the dimensions of the array must be correct BEFORE `scan` is called. In other words, a file containing the array entries only can only be read when the number of entries is known. The function `scanFromFile` can, however, read an unknown number of entries from file into a vector with wrong dimensions (see documentation below). If the format is binary, the `scan` function detects the array length and redimensions the array, but a warning is issued. The reason is that, without a header, the main rule is that the programmer must redimensioned the array on beforehand.

After the header the data appears. If they are written in binary mode, the first non-white character (after the header) is @. Then, in binary format, the number of items must appear and after that the size of each item. Then the data appear in binary format. If the @ character is not present, the data appear in ascii format.

Let us present some examples on valid data files. First the most comprehensive form:

```
[3]  Example of a data file with much additional information such as
indices for each vector entry. This file is in ascii format.$

(1)=1.2
(2)=3.4
(3)=1.8
```

Here is an ascii format file with as little information as possible (a NULL header was given when calling `print`:

```
1.2 3.4 1.8
```

An equivalent form may be

```
1.2 3.4
1.8
```

Finally we show a binary file without header. To indicate a binary number (f.ex. 8.1) we use the notation [B8.1].

```
@[B3][B4][B1.2][B3.4][B1.8]
```

Observe that the item is a float and that the size of the item is `sizeof(float)` which equals 4. There is a comprehensive demonstration and test program for the various ascii and binary file formats of arrays in the directory `$DPR/src/app/class-verify/category2/arrayprint`.

`scanFromFile` - enables an unknown numbers of `Type` objects to be read from file into a possibly empty vector. The objects must appear on the file without any text. Hence, this function cannot read a file written by `print` with a comment. First, the number of objects on the file is found, then the vector is correctly redimensioned and then an ordinary `scan` function is called. Only ascii files can be read by the present implementation (that is also the most useful application of a function like this).

`print` - writes the content of the vector in a format that can be read by `scan`. See the documentation for `VecSimple::scan`. A header string can be given (empty by default). Also the number of characters in each entry (when the entry is written in ascii format) can be given. If this parameter is missing the `print` function assumes that there can be no more than one entry per line. For example, if one wishes to write an `int` vector `v` with heading "Improved values by inner iterations" on standard output, where each `int` has width 5, one can make the calls:

```
s_o->setIntFormat("%5d");
v.print (s_o, "Improved values by inner iterations",
         s_o->getFormatWidth(v(1)));
s_o->resetIntFormat();
```

`<<>>` - `operator<<` and `operator>>` are defined in terms of the `print` and `scan` functions.

EXAMPLES

```
VecSimple(int) a(4);// allocates a vector of integers, length=4.
a.fill (5);         // sets all entries in  a  equal to 5.
a(2) = 6;           // entry (2) is set equal to 6.
a.print(cout);      // prints  a  on cout
cout << a(2);       // prints entry (2).
a.scan (cin);       // reads the vector from standard input, entry by entry,
                    // a total 4 numbers.

Vec(dpComplex) c;     // no size is given, no memory is allocated.
c.redim (m);        // c  becomes a vector of dpComplex of length m.
c.fill (dpComplex(2,7); // all entries in  c  is set equal to 2 + 7i.
```

SEE ALSO

class `ArrayGenSimplest(Type)`, class `Vec(Type)`, class `MatSimple(Type)`

DEVELOPED BY

SINTEF Applied Mathematics, Oslo, Norway, and University of Oslo, Dept. of Mathematics, Norway

AUTHOR

Hans Petter Langtangen, SINTEF/UiO

A.2.3 ArrayGenSimplest

NAME

ArrayGenSimplest(Type) - general array with variable no. of indices

INCLUDE

```
include "ArrayGenSimplest.h"
```

SYNTAX

```
class ArrayGenSimplest(Type) : public virtual VecSimplest(Type)
{
protected:

  int      ndim;   // number of dimensions
  Ptv(int) nm;     // length of each dimensions, multiple index
  Ptv(int) bm;     // base of each dimensions, multiple index
  int      current_iterator_index;  // for iterating over the array entries

  int   totalLength (const Ptv(int)& n);

  void  init (int n1);
  void  init (int n1, int n2);
  void  init (int n1, int n2, int n3);
  void  init (const Ptv(int)& n);

  BooLean indexOk (const Ptv(int)& index) const; // rangecheck for operator()
  void indexOk1 (int i) const;  // for singleIndex1
  void indexOk (int i) const;
  void indexOk (int i, int j) const;
  void indexOk (int i, int j, int k) const;

public:
  ArrayGenSimplest(Type) ();
  ArrayGenSimplest(Type) (int n1);
  ArrayGenSimplest(Type) (int n1, int n2);
  ArrayGenSimplest(Type) (int n1, int n2, int n3);
  ArrayGenSimplest(Type) (const Ptv(int)& n);         // multiple index
 ~ArrayGenSimplest(Type) () {}

  // NOTE: these redim functions are not inline and less efficient
  // than the redim functions in VecSimplest!!
  BooLean redim (int n1);                    // one-dim. array
  BooLean redim (int n1, int n2);            // two-dim. array
  BooLean redim (int n1, int n2, int n3);    // three-dim. array
  BooLean redim (const Ptv(int)& n);         // multiple index (arbitrary-dim.)

  BooLean compatible (const ArrayGenSimplest(Type)& a,
                      BooLean error_message = dpTRUE);

  void setBase  (int b1);
  void setBase  (int b1, int b2);
  void setBase  (int b1, int b2, int b3);
  void setBase  (const Ptv(int)& b);

  // with getBase and getMaxI one can easily determine lower and
  // upper bounds on loops involving ArrayGenSimplest objects, while
  // getDim finds the length of the loops

  // lower index:
```

```
  void getBase  (int& b1) const;
  void getBase  (int& b1, int& b2) const;
  void getBase  (int& b1, int& b2, int& b3) const;
  void getBase  (Ptv(int)& b) const;

  // upper index:
  void getMaxI  (int& n1) const;
  void getMaxI  (int& n1, int& n2) const;
  void getMaxI  (int& n1, int& n2, int& n3) const;
  void getMaxI  (Ptv(int)& n) const;

  // length of each dimension:
  int  getDim   () const;       // the dimension of the array (1D,2D,3D,..)
  void getDim   (int& n1) const;  // length of 1D array
  void getDim   (int& n1, int& n2) const;  // length of 2D array
  void getDim   (int& n1, int& n2, int& n3) const;  // length of 3D array
  void getDim   (Ptv(int)& n) const; // length of general dD array

  Type& singleIndex1 (int i);  // single index, 1 to total length
  const Type& singleIndex1 (int i) const;

  String arraySize () const;

  Type& operator () (int i);
  Type& operator () (int i, int j);
  Type& operator () (int i, int j, int k);
  Type& operator () (const Ptv(int)& index);

  const Type& operator () (int i) const;
  const Type& operator () (int i, int j) const;
  const Type& operator () (int i, int j, int k) const;
  const Type& operator () (const Ptv(int)& index) const;

  void startIterator ();
  BooLean nextEntry ();   // is there a next entry? if yes, move to it
  Type&   thisEntry ();             // enables assignment
  const Type& thisEntry () const;   // enables reading only

  // illegal functions (included here with just an error message):
  ArrayGenSimplest(Type) (const ArrayGenSimplest(Type)& v);
  void operator =        (const ArrayGenSimplest(Type)& v);

  // CLASS_INFO: not used, no virtual functions
};

//---------------------------------------------------------------------------
inline Type& ArrayGenSimplest(Type):: singleIndex1 (int i)
//---------------------------------------------------------------------------
{
#ifdef ARRAY_RANGECHECK
  indexOk1(i);
#endif

  return A[i];
}

//---------------------------------------------------------------------------
inline const Type& ArrayGenSimplest(Type):: singleIndex1 (int i) const
//---------------------------------------------------------------------------
{
#ifdef ARRAY_RANGECHECK
  indexOk1(i);
#endif

  return A[i];
}
```

```
//---------------------------------------------------------------------------
inline Type& ArrayGenSimplest(Type):: operator () (int i)
//---------------------------------------------------------------------------
{
#ifdef ARRAY_RANGECHECK
  indexOk(i);
#endif

  return A[i-bm(1)+1];
}

//---------------------------------------------------------------------------
inline const Type& ArrayGenSimplest(Type):: operator () (int i) const
//---------------------------------------------------------------------------
{
#ifdef ARRAY_RANGECHECK
  indexOk(i);
#endif

  return A[i-bm(1)+1];
}

//---------------------------------------------------------------------------
inline Type& ArrayGenSimplest(Type):: operator () (int i, int j)
//---------------------------------------------------------------------------
{
#ifdef ARRAY_RANGECHECK
  indexOk(i,j);
#endif

  return A[(j-bm(2))*nm(1) + i-bm(1)+1];
}

//---------------------------------------------------------------------------
inline const Type& ArrayGenSimplest(Type):: operator () (int i, int j) const
//---------------------------------------------------------------------------
{
#ifdef ARRAY_RANGECHECK
  indexOk(i,j);
#endif

  return A[(j-bm(2))*nm(1) + i-bm(1)+1];
}


//---------------------------------------------------------------------------
inline Type& ArrayGenSimplest(Type):: operator () (int i, int j, int k)
//---------------------------------------------------------------------------
{
#ifdef ARRAY_RANGECHECK
  indexOk(i,j,k);
#endif

  return A[(k-bm(3))*nm(1)*nm(2) + (j-bm(2))*nm(1) + i-bm(1)+1];
}

/* NOTE: 3-dimensional indexing is not very efficient since it implies
   look up in Ptv-objects and several multiplications. If extreme efficiency
   is required, one should use an array structure with a pointer for each
   dimension (classes with Type*** are not supperted in Diffpack).
   Here is a suggested implementation: Derive the class from Vector,
   take care of the details and support the Vector interface. Then the
   vector/array can be used in all vector computations in Diffpack (e.g.
   in linear system solvers).
*/
```

```
//--------------------------------------------------------------------------
inline const Type& ArrayGenSimplest(Type):: operator () (int i, int j, int k)
//--------------------------------------------------------------------------
 const
{
#ifdef ARRAY_RANGECHECK
  indexOk(i,j,k);
#endif

  return A[(k-bm(3))*nm(1)*nm(2) + (j-bm(2))*nm(1) + i-bm(1)+1];
}


//--------------------------------------------------------------------------
inline BooLean ArrayGenSimplest(Type):: nextEntry ()
//--------------------------------------------------------------------------
{
  current_iterator_index++;
  return (current_iterator_index >= 1 && current_iterator_index <= length) ?
    dpTRUE : dpFALSE;
}

//--------------------------------------------------------------------------
inline const Type& ArrayGenSimplest(Type):: thisEntry () const
//--------------------------------------------------------------------------
{
#ifdef ARRAY_RANGECHECK
  if (!(current_iterator_index >= 1 && current_iterator_index <= length))
    errorFP("ArrayGenSimplest(Type)::thisEntry",
"current_iterator_index is out of bounds, probably incorrect use of nextEntry()");
#endif
  return VecSimplest(Type)::operator()(current_iterator_index);
}


//--------------------------------------------------------------------------
inline Type& ArrayGenSimplest(Type):: thisEntry ()
//--------------------------------------------------------------------------
{
#ifdef ARRAY_RANGECHECK
  if (!(current_iterator_index >= 1 && current_iterator_index <= length))
    errorFP("ArrayGenSimplest(Type)::thisEntry",
"current_iterator_index is out of bounds, probably incorrect use of nextEntry()");
#endif
  return VecSimplest(Type)::operator()(current_iterator_index);
}
```

KEYWORDS

array, general array, multi-dimensional array

DESCRIPTION

The class implements a multi-dimensional array in terms of a standard, one-dimensional C array. The multi-dimensional feature is created by offering subscript operators for one, two, three and `Ptv(int)` indices. The bases of the indices can be arbitrary (see the example below). The array is parameterized and can contain any built-in or user defined type, cf. class `VecSimplest(Type)`. Further features, such as arithmetic operations, are enabled in the derived class `ArrayGen(Type)`.

CONSTRUCTORS AND INITIALIZATION

There are several constructors. All constructors allocate the proper amount of memory and initialize the various internal data structure needed for administrating the multi-dimensional array. The base of each index must be set manually. The default base is 1. The only user required initialization is to assign values to the entries in the array.

There is a default constructor which coincides with the default constructor of the base class `VecSimplest(Type)`. The other constructors take the number of entries in each dimension of the array as arguments. If there are more than 3 dimensions, a `Ptv(int)` object is given as argument.

MEMBER FUNCTIONS

If the documentation of class `VecSimplest` is known and the example below is studied, most of the member functions should be self-explanatory.

`redim` - redimensions the array. With this function one can change the dimension of the array, and the number of entries of each dimension. The base is set to 1.

`setBase` - enables the programmer to choose an arbitrary base for the index of each dimension.

`getBase` - returns the base of the index of each dimension.

`getMaxI` - returns the upper index value of each dimension. If one wants a loop over the array entries and need to extract the lower and upper loop limits, `getBase` will give the lower limits, while `getMaxI` will give the correct upper limits.

`getDim` - returns the number of array entries in each dimension. Note that if the base is unity, the number of entries equals the return values of `getMaxI`. To redimension another array, `getDim` extracts the correct size. If the bases should also coicide, one must extract the bases (`getBase`) and set them in the new array (`setBase`) after the declaration.

`singleIndex1` - enables the programmer to index the possibly multi-dimensional array by using a single index. This is convenient if the programmer wants to avoid separate loops over each dimension (hence there is no need for `getBase` or `getMaxI` or knowledge of the number of dimensions).

`arraySize` - returns a string containing the array size, e.g., for a 2-dimensional array [0:3]x[-1:4] it simply returns the string "[0:3]x[-1:4]".

`operator()` - enables subscripting of the array. The number of dimensions must be known and the index must be correct with respect to the base of each dimension. NOTE: 2- and 3-dimensional indexing are not very efficient since it implies look up in Ptv-objects and several multiplications. If extreme efficiency is required, one should use an array structure with a pointer for each dimension (classes with `Type[][][]` are not supperted in Diffpack, but for 2-dimensional arrays one can use `MatSimplest` and its subclasses). Here is a suggested implementation: Derive the class from Vector, take care of the details and support the Vector interface. Then the vector/array can be used in all vector computations in Diffpack (e.g. in linear system solvers).

`startIterator` - starts an iteration over all the array entries.

`nextEntry` - moves a pointer to the next entry in the array. Returns a true value of there is a next entry, if not, a false value is returned.

`thisEntry` - returns the value of the current entry. Actually, a reference is returned so that the function can be used for assigning values to entries (only true for a `const` object).

Be aware that `ArrayGenSimplest` does not require the `Type` class to have an `operator=`. Hence, the copy constructor or the assignment operator has no meaning for `ArrayGenSimplest`.

Since C++ automatically makes such functions if they are not explicitly declared (and such automatically generated functions lead to serious errors in the present case), the functions are declared, but the content is just an error message.

EXAMPLES

```
// two-dimensional array with each index starting at 0:
ArrayGenSimplest(real) v (10, 10);
v.setBase (0, 0);
// valid indices: v(i,j), where 0 <= i,j <= 9

// four-dimensional array with first index starting at 0, the others
// starting at 1:
Ptv(int) b(4), n(4), i(4);
n(1) = 10;  n(2) = 5;  n(3) = n(4) = 8;   // dimensions
b(1) = 0;   b(2) = b(3) = b(4) = 1;       // base
ArrayGenSimplest(real) w (n);
w.setBase (b);
i(1) = 1; i(2) = i(3) = i(4) = 3;         // 4-tuple index
w(i) = 6;                                 // assignment, one entry

// three-dimensional array of class Thing objects, each index base
// equals -1, each dimension of the array equals 3:
ArrayGenSimplest(Thing) t (3,3,3);
t.setBase (-1,-1,-1);
t(0,-1,1).scan (cin);

// copy a to b manually (ArrayGenSimplest(Type) has no operator=
// function, cf. class VecSimplest(Type):

int n1, n2; a.getDim (n1,n2);
ArrayGenSimplest(real) b(n1,n2);
int j,k,j1,k1,jn,kn;
a.getBase (j1,k1);  a.getMaxI (jn,kn);
for (j=j1; j<=jn; j++)
  for (k=k1; k<=kn; k++)
    b(j,k) = a(j,k);
```

SEE ALSO

class `ArrayGen(Type)`, class `VecSimplest(Type)`

DEVELOPED BY

SINTEF Applied Mathematics, Oslo, Norway, and University of Oslo, Dept. of Mathematics, Norway

AUTHOR

Hans Petter Langtangen, SINTEF/UiO

A.2.4 ArrayGenSimple

NAME

ArrayGenSimple(Type) - general array with operator= and printing

INCLUDE

```
include "ArrayGenSimple.h"
```

SYNTAX

```
class ArrayGenSimple(Type) : public virtual VecSimple(Type),
                             public ArrayGenSimplest(Type)
{
// recall: virtual base class VecSimplest(Type)

public:

  //---------- constructors and destructor:

  ArrayGenSimple(Type) ();
  ArrayGenSimple(Type) (int n1);
  ArrayGenSimple(Type) (int n1, int n2);
  ArrayGenSimple(Type) (int n1, int n2, int n3);
  ArrayGenSimple(Type) (const Ptv(int)& n);         // multiple index
 ~ArrayGenSimple(Type) () {}

  //---------- redim :

  BooLean redim (int n1)
    { return ArrayGenSimplest(Type)::redim(n1); }

  BooLean redim (int n1, int n2)
    { return ArrayGenSimplest(Type)::redim(n1,n2); }

  BooLean redim (int n1, int n2, int n3)
    { return ArrayGenSimplest(Type)::redim(n1,n2,n3); }

  BooLean redim (const Ptv(int)& n)
    { return ArrayGenSimplest(Type)::redim(n); }


  //---------- indexing operators :

  Type& operator () (int i)
    { return ArrayGenSimplest(Type)::operator()(i); }

  Type& operator () (int i, int j)
    { return ArrayGenSimplest(Type)::operator()(i,j); }

  Type& operator () (int i, int j, int k)
    { return ArrayGenSimplest(Type)::operator()(i,j,k); }

  Type& operator () (const Ptv(int)& index)
    { return ArrayGenSimplest(Type)::operator()(index); }

  const Type& operator () (int i) const
    { return ArrayGenSimplest(Type)::operator()(i); }

  const Type& operator () (int i, int j) const
    { return ArrayGenSimplest(Type)::operator()(i,j); }
```

```
  const Type& operator () (int i, int j, int k) const
    { return ArrayGenSimplest(Type)::operator()(i,j,k); }

  const Type& operator () (const Ptv(int)& index) const
    { return ArrayGenSimplest(Type)::operator()(index); }


  //----------- various standard functions :

  void operator = (Type a)    { VecSimple(Type)::fill(a); }
  void operator = (const ArrayGenSimple(Type)& a);
  void fill (const Type& a)   { VecSimple(Type)::fill(a); }

  void print      (Os os, const char* header = NULL,
                   int nentries_per_line = 3) const;
  void printAscii (Os os, const char* header = NULL) const;
  void scan (Is is);

  //CLASS_INFO: not used
};
```

KEYWORDS

array, finite difference methods, point operators, general array, multi-dimensional array

DESCRIPTION

The class implements a multi-dimensional array in terms of a standard, one-dimensional C array. The multi-dimensional feature is created by offering subscript operators for one, two, three and `Ptv(int)` indices. The bases of the indices can be arbitrary (see the example below). The only requirement of the array entries is that class `Type` must have a constructor without arguments and `operator=`, `operator>>` and `operator<<`. There are no assumptions that the entries in the array can be used in arithmetic operations like +, -, $*$ and /.

The present class has class `VecSimplest(Type)` as a virtual base class.

CONSTRUCTORS AND INITIALIZATION

See documentation for class `ArrayGenSimplest(Type)`.

MEMBER FUNCTIONS

The member functions should be self-explanatory if the documentation of the classes `VecSimplest(Type)`, `VecSimple(Type)` and `ArrayGenSimplest(Type)` has been studied.

EXAMPLES

See documentation for class `ArrayGenSimplest(Type)`.

SEE ALSO

class `ArrayGenSimplest(Type)`, class `VecSimple(Type)`

DEVELOPED BY

SINTEF Applied Mathematics, Oslo, Norway, and University of Oslo, Dept. of Mathematics, Norway

AUTHOR

Hans Petter Langtangen, SINTEF/UiO

A.3 Error handling functions

A.3.1 errors

NAME

errors.h - definition of error and warning message functions

INCLUDE

```
include "errors.h"
```

SYNTAX

```
// function pointers for error/warning message functions:
typedef void (*messageFP)(char*,...);

extern messageFP errorFP;
extern messageFP fatalerrorFP;
extern messageFP warningFP;

// as errorFP but returns an int or int*
typedef BooLean (*messageFPbool)(char*,...);
extern messageFPbool errorFPret;
typedef int* (*messageFPintPtr)(char*,...);
extern messageFPintPtr errorFPptr;

extern void setMaxWarnings(int);
```

KEYWORDS

error messages, warning messages, fatal error

DESCRIPTION

Error and warning messages in a program should be reported at the places where they occur. This is performed by calling an error or warning message function through a function pointer. There are three such pointers: 1) `errorFP`, reporting an error, 2) `fatalerrorFP`, reporting a fatal error and 3) `warningFP`, reporting a warning. An error leads to a prompt for further execution (the user must hit carriage return or click on an ok-button), a fatal error leads to abortion of the program while a warning message makes no interruption of the execution.

The programmer must decide which error/warning message function to use, according to the expected consequences for further execution of the program. The main guidelines are that a fatal error is required if further execution is expected to lead to core dump, segmentation fault or meaningless results, an error is issued if the results are likely to be wrong but execution is technically possible, and a warning is issued if further execution may yield acceptable results.

The error/warning message functions take a `char*` as the first argument. This is the name of the function in which the error or warning has occured. If the function is a member

function of a class, the complete name should be gien (see ghe example below). The next arguments follow the same syntax as the arguments to the standard `printf` C function. That is, a string, with formatting indications, and possibly some variables can be given. This allows the programmer to report the contents of variables. Examples are given below. In parameterized classes it may be convenient to use `oform` type of function to format the name of the function where the error was found. However, the error functions also use `oform` so care must be taken. The programmer should use the `eform` function, which is equivalent to `oform` except that the resulting formatted string is allocated separately and not a part of the `oform` buffer. In other words, a `grep` on `errorFP(oform)` should not result in any output! The `aform` function could be used, but then a cast and a call to the `String::chars()` function are required - it is simpler to use `eform` (acutally, the `e` stands for error and indicates that this is the version of `oform` that is suited in conjuction with the error functions).

In the future a tool will be developed that collects all calls to warning and error functions in a separate file. This information can be automatically be formatted for documentation of possible error messages from a module or the messages can be translated no other languages than English (tools for re-inserting the translated messages will be provided). Hence, at the present stage it is only necessary to report errors/warnings at the places where they occur, using the functions described above, and future tools will be based on this convention and will offer sophisticated documentation of the messages.

The function pointers `errorFP`, `fatalerrorFP` and `warningFP` are at compile time set to default values which corresponds to functions that writes the messages to standard output. Other choices are possible, for example, the pointers can be set to functions that report the messages in windows. The functions `setErrorMessageInWindows`() and `setErrorMessageOnStdout`() set the function pointers to the two possible choices, window messages or messages on standard output. In addition, the function `setMaxWarnings` is used to force program exit after a given number of warnings is issued.

It should be mentioned that empty functions are often equipped with an error message to tell the user that the body of the function is not implemented. In cases where the function returns a value or a pointer it is convenient to return a call to an error function that has `int` or `int*` as return value. This is provided by the two function pointers `errorFPret` (returns `int`) and `errorFPptr` (returns `int*`).

EXAMPLES

Suppose that `i` is an index in an array. If `i` is less than zero, it is out of bounds and a fatal error message with program abortion should be issued. On the other hand, other programmers would perhaps consider an index out of bounds to be an error, further execution is allowed if the user wants.

```
if (i < 0)
  fatalerrorFP("myfunction","i is negative (i=%d)",i);

void MyClass:: test ()
  {
    real r; int s;
    // some code ....
    if (i)
      errorFP("MyClass::test","wrong status, r=%5.2f, s=%d",r,s);
  }
```

DEVELOPED BY

SINTEF Applied Mathematics, Oslo, Norway, and University of Oslo, Dept. of Mathematics, Norway

AUTHOR

Hans Petter Langtangen, SINTEF/UiO. The basic structure of the code was provided by Per Oyvind Hvidsten, SINTEF.

Index