LA: a C++ linear algebra library

User manual

Jiří Pittner

October 30, 2026

Contents

1 Introduction **10**
- 1.1 Purpose . . . 10
- 1.2 Overview . . . 11
- 1.3 A minimal example . . . 12
- 1.4 Conventions used in this manual . . . 13
- 1.5 Authors and licence . . . 13

2 Building and using the library **14**
- 2.1 Requirements . . . 14
- 2.2 Configuration and installation . . . 14
  - 2.2.1 Configure options . . . 15
  - 2.2.2 Preprocessor macros . . . 15
- 2.3 Compiling and linking user programs . . . 16
- 2.4 Test programs . . . 16
- 2.5 Documentation . . . 17

3 Common concepts **18**
- 3.1 Reference counting and copy on demand . . . 18
  - 3.1.1 Shared data . . . 18
  - 3.1.2 Copy on demand . . . 18
  - 3.1.3 The DEBUG check of the reference count . . . 19
  - 3.1.4 Objects referring to external memory . . . 20
- 3.2 Shallow and deep copy . . . 20
- 3.3 Traits and element types . . . 21
  - 3.3.1 Element types and instantiation . . . 22
  - 3.3.2 Nested types . . . 22
- 3.4 Storage conventions . . . 23
- 3.5 GPU support (experimental) . . . 23
- 3.6 Error handling and DEBUG checks . . . 24
- 3.7 Input and output . . . 24

4 Vectors: NRVec **27**
4.1 Purpose and main idea . . . 27
4.2 Storage and class-specific concepts . . . 27
4.3 Construction, assignment, conversion . . . 28
4.4 Element access . . . 30
4.5 Arithmetic and BLAS-backed operations . . . 31
4.6 Norms, reductions, extrema . . . 34
4.7 Other member functions . . . 35
4.8 Related free functions . . . 37
4.9 Vectors indexed from one: NRVec_from1 . . . 38
4.10 Examples . . . 38
4.11 Pitfalls . . . 39

5 Dense matrices: NRMat **41**
5.1 Purpose . . . 41
5.2 Storage and class-specific concepts . . . 41
5.3 Construction, assignment, conversion . . . 43
5.4 Element access . . . 45
5.5 Arithmetic and BLAS/LAPACK-backed operations . . . 46
5.6 Other member functions . . . 48
5.6.1 Rows, columns, submatrices . . . 48
5.6.2 Transposition and conjugation . . . 49
5.6.3 Reordering, permutations, rotations, scaling . . . 49
5.6.4 Reductions, norms, properties . . . 50
5.6.5 Orthonormalization, inverse, random fill . . . 51
5.6.6 Input/output . . . 51
5.6.7 Sparse matrix interoperability . . . 52
5.7 Related free functions . . . 52
5.8 NRMat_from1: indexing from 1 . . . 53
5.9 Examples . . . 53
5.10 Pitfalls . . . 55

6 Packed symmetric and hermitian matrices: NRSMat **57**
6.1 Purpose . . . 57
6.2 Storage and concepts . . . 57
6.2.1 Packed lower triangle . . . 57
6.2.2 Index functions . . . 58
6.2.3 Hermitian interpretation of complex matrices . . . 58
6.2.4 Reference counting . . . 59
6.3 Construction, assignment, conversion . . . 60
6.4 Element access . . . 61
6.5 Arithmetic and BLAS/LAPACK-backed operations . . . 62
6.6 Other member functions . . . 64
6.6.1 Submatrices and permutations . . . 64

6.6.2 Input/output 65
6.7 Related free functions 65
6.8 NRSMat_from1 66
6.9 Examples 67
6.10 Pitfalls 67

7 Sparse matrices **69**
7.1 Overview 69
7.2 SparseMat 70
7.2.1 Storage 70
7.2.2 Reference counting 71
7.2.3 Construction, assignment, conversion 71
7.2.4 Element insertion and access 72
7.2.5 Arithmetic 73
7.2.6 Products 75
7.2.7 Transposition, permutations, direct sums 75
7.2.8 Input/output 76
7.2.9 Related free functions 77
7.3 SemiSparseMat 77
7.4 SparseSMat 78
7.4.1 Storage and concepts 78
7.4.2 Construction, assignment, conversion 79
7.4.3 Element insertion and access 80
7.4.4 Arithmetic and products 81
7.4.5 Input/output and free functions 83
7.5 CSRMat 84
7.6 Examples 84
7.7 Pitfalls 85

8 Dense linear algebra functions **86**
8.1 Purpose and general conventions 86
8.2 Linear systems, inverse, determinant 87
8.3 Eigenvalue problems 91
8.3.1 Symmetric and hermitian matrices 91
8.3.2 General (non-symmetric) matrices 91
8.4 Singular value decomposition 92
8.5 Matrix functions 93
8.6 Matrix polynomials and exponentials (matexp.h) 94
8.7 Products, traces, transformations 96
8.8 Norms and condition number 97
8.9 Real and imaginary parts 98
8.10 Miscellaneous 98
8.11 Examples 99
8.12 Pitfalls 100

9 Iterative methods **101**
9.1 Purpose and main idea . . . 101
9.2 Requirements on the matrix class . . . 102
9.3 Davidson diagonalization . . . 104
9.4 Lanczos diagonalization . . . 107
9.5 Conjugate gradients . . . 108
9.6 GMRES . . . 109
9.7 DIIS convergence acceleration . . . 110
9.8 Search in sorted arrays . . . 111
9.9 Examples . . . 112
9.9.1 Davidson with a user-defined direct matrix . . . 112
9.9.2 Linear systems and DIIS . . . 113
9.10 Pitfalls . . . 114

10 Optimization **116**
10.1 Problem interface and capabilities . . . 117
10.1.1 OptProblem . . . 117
10.1.2 Capabilities . . . 117
10.1.3 Least-squares problems: OptLeastSquares . . . 119
10.1.4 Adapters . . . 120
10.2 Optimizers, hyperparameters and results . . . 122
10.2.1 Optimizer and optresult . . . 122
10.2.2 Hyperparameter hierarchy . . . 123
10.2.3 How parameters are taken: setfrom() . . . 126
10.2.4 The achieved values . . . 127
10.3 Infrastructure and common drivers . . . 128
10.3.1 Evaluation and iteration bookkeeping . . . 128
10.3.2 One-dimensional searches . . . 129
10.3.3 Line-search methods . . . 129
10.3.4 First-order methods . . . 130
10.3.5 Trust-region methods . . . 131
10.3.6 Writing a new method . . . 132
10.4 Methods . . . 133
10.4.1 Quasi-Newton methods: BFGS, DFP, SR1 . . . 133
10.4.2 L-BFGS . . . 135
10.4.3 Nonlinear conjugate gradients . . . 135
10.4.4 GDIIS . . . 136
10.4.5 Trust-region Newton and quasi-Newton . . . 137
10.4.6 P-RFO: minima and saddle points . . . 137
10.4.7 Levenberg-Marquardt . . . 138
10.4.8 Nelder-Mead downhill simplex . . . 138
10.4.9 Powell's conjugate direction method . . . 139
10.4.10 COBYLA . . . 140
10.4.11 Rotosolve . . . 140

10.4.12 SGD with momentum . . . 141
10.4.13 Quantum natural gradient . . . 141
10.4.14 Adam, AdamW, AMSGrad . . . 142
10.4.15 SPSA . . . 142
10.5 Examples . . . 143
10.5.1 A user-defined problem and BFGS . . . 143
10.5.2 Freezing parameters . . . 144
10.5.3 COBYLA with constraints . . . 144
10.5.4 Least-squares fit by Levenberg-Marquardt . . . 145
10.5.5 Further examples in the distribution . . . 146
10.6 Pitfalls . . . 146

11 Tensors **148**
11.1 Purpose and main idea . . . 148
11.2 Storage and concepts . . . 149
11.2.1 Index groups . . . 149
11.2.2 Storage order . . . 149
11.2.3 Public data members . . . 150
11.2.4 Index types . . . 150
11.2.5 Reference counting and copy on demand . . . 151
11.3 Construction, assignment, conversion . . . 151
11.3.1 Conversions . . . 152
11.3.2 NumPy files . . . 154
11.4 Element access . . . 154
11.4.1 Looping over all elements . . . 154
11.4.2 Index bookkeeping . . . 155
11.5 Arithmetic . . . 156
11.6 Contractions . . . 157
11.7 Restructuring . . . 158
11.8 Permutation algebra, decompositions and transformations . . . 160
11.9 Names and shape queries . . . 161
11.10 Input and output . . . 162
11.11 Helpers in the header . . . 162
11.12 Examples . . . 162
11.13 Pitfalls . . . 163

12 Four-index quantities **165**
12.1 Purpose and main idea . . . 165
12.2 Symmetry types . . . 166
12.3 Element records . . . 167
12.4 In-core sparse storage: `fourindex<I,T>` . . . 168
12.4.1 Reference counting . . . 168
12.4.2 Construction and basic operations . . . 168
12.4.3 Iterators . . . 169

12.4.4 Input and output . . . 170
12.5 Disc storage: `fourindex_ext<I,T>` . . . 171
12.6 Dense storage: `fourindex_dense<S,T,I>` . . . 172
12.6.1 twoelectronrealmullikanAB . . . 172
12.6.2 twoelectronrealmullikan . . . 173
12.6.3 nosymmetry . . . 173
12.6.4 T2IjAb_aces . . . 173
12.6.5 T2ijab_aces . . . 174
12.6.6 antisymtwoelectronrealdiracAB . . . 175
12.6.7 antisymtwoelectronrealdirac . . . 175
12.6.8 Other symmetry types . . . 176
12.7 Examples . . . 176
12.8 Pitfalls . . . 177

13 Permutations and the symmetric group 178
13.1 Conventions . . . 178
13.2 NRPerm: permutations in one-line notation . . . 179
13.2.1 Construction and conversion . . . 179
13.2.2 Basic operations . . . 180
13.2.3 Ranking and inversion tables . . . 181
13.2.4 Generating all permutations . . . 181
13.3 CyclePerm: cycle notation . . . 182
13.4 Weighted permutations and the group algebra . . . 184
13.4.1 WeightPermutation . . . 184
13.4.2 PermutationAlgebra . . . 185
13.5 Partitions . . . 187
13.5.1 CompressedPartition . . . 187
13.5.2 Partition . . . 187
13.6 Young tableaux . . . 188
13.7 Characters of the symmetric group . . . 189
13.8 Applying permutations to vectors and matrices . . . 190
13.9 Small number-theoretic helpers . . . 191
13.10Sorting . . . 191
13.11Examples . . . 193
13.12Pitfalls . . . 194

14 Polynomials and continued fractions 195
14.1 Polynomial . . . 195
14.1.1 Purpose and storage . . . 195
14.1.2 Construction . . . 196
14.1.3 Arithmetic . . . 196
14.1.4 Calculus and transformations . . . 197
14.1.5 Evaluation . . . 198
14.1.6 Roots, interpolation, gcd . . . 198

14.1.7 Orthogonal polynomials . . . 199
14.1.8 Example . . . 199
14.2 Rational numbers and continued fractions . . . 200
14.2.1 Rational . . . 200
14.2.2 ContFrac . . . 201
14.2.3 Gosper's arithmetics . . . 202
14.2.4 Example . . . 203
14.2.5 Pitfalls . . . 203

15 3D vectors and matrices: Vec3, Mat3 **204**
15.1 Purpose . . . 204
15.2 Storage . . . 205
15.3 Vec3 reference . . . 205
15.4 Mat3 reference . . . 206
15.5 Free functions: I/O, Euler angles, projection . . . 208

16 Quaternions **210**
16.1 Purpose . . . 210
16.2 Storage . . . 210
16.3 Reference . . . 211
16.4 Example . . . 214
16.5 Pitfalls . . . 214

17 Bit vectors and polynomials over GF(2) **215**
17.1 Purpose and storage . . . 215
17.2 Copy on demand . . . 215
17.3 Reference: construction, access . . . 216
17.4 Reference: bitwise and integer operations . . . 217
17.5 Reference: polynomials over GF(2) and GF(2^n) . . . 218
17.6 bitmatrix, bitvector_from1 and helper functions . . . 219
17.7 Example . . . 220
17.8 Pitfalls . . . 221

18 Regression **222**
18.1 Kernel functions . . . 222
18.2 REG: kernel regression in $\mathbb{R}^d$ . . . 223
18.2.1 Model and parameters . . . 223
18.2.2 Construction . . . 224
18.2.3 Fitting and prediction . . . 224
18.2.4 Example . . . 225
18.3 REGSURF: regression of molecular surfaces and ML-NEA spectra . . . 225
18.3.1 Model . . . 226
18.3.2 Construction and data files . . . 226

18.3.3 Fitting, prediction and parameters 227
18.3.4 ML-NEA absorption spectra 228
18.3.5 Example 229
18.4 Pitfalls 229

19 Utilities **230**
19.1 Scratch-file storage: AuxStorage 230
19.1.1 Purpose and storage 230
19.1.2 Reference 231
19.1.3 Example 232
19.1.4 Pitfalls 232
19.2 Random numbers 232
19.2.1 Replacing the generator 233
19.2.2 Seeding 233
19.3 Special functions and combinatorial numbers 234
19.4 Simple algorithms for embedded use 235
19.5 Error handling reference 237
19.6 The single include file la.h 238
19.7 Version information 239
19.8 Number-theoretic utilities 239
19.9 Graph algorithms 240

Bibliography **241**

List of open questions for the author

Chapter 1

Introduction

1.1 Purpose

LA is a C++ class library for numerical linear algebra and related tasks. Its core are the class templates `NRVec<T>` (vector), `NRMat<T>` (dense matrix) and `NRSMat<T>` (symmetric or hermitian matrix in packed storage), which provide a convenient and efficient interface to the BLAS and LAPACK libraries. Around them a number of further classes and algorithms has grown, mostly motivated by the needs of quantum chemistry: sparse matrices, tensors with arbitrary index symmetry, four-index quantities (two-electron integrals), iterative eigensolvers and linear solvers working with matrices that are never stored explicitly, convergence acceleration, numerical optimization, permutations and the symmetric group, polynomials, quaternions and others.

The design follows a few principles, which are explained in detail in chapter 3:

- *Reference counting with copy on demand.* Copying and assigning objects is cheap, data are shared until they are modified. Objects can therefore be passed and returned by value. The price is that writing to individual elements requires an explicit `copyonwrite()` call (section 3.1).
- *No expression templates.* The library does not optimize matrix expressions. Operators like `A*x+y` are available for convenience, but performance critical code should call the BLAS-like member functions (`gemv`, `gemm`, `axpy`, ...) directly, which avoid temporaries.
- *Generic algorithms.* Iterative methods (Davidson, Lanczos, conjugate gradients, GMRES, the matrix exponential, ...) are templates which only need a few operations of the matrix class (typically `nrows()`, `ncols()`, `diagonalof()` and `gemv()`), so that they work with user classes which compute the matrix–vector product on the fly ("direct" methods).

- *Nesting.* The element type of the containers can itself be a container, e.g. `NRVec<NRMat<double> >`; copy on demand and I/O work recursively (section 3.3).

The manual is written for users who know both C++ and linear algebra. It explains the concepts and gives a compact but complete reference of the public interface. Many further usage examples are in the test programs `t.cc`, `t2.cc` and `test.cc` of the distribution; the regression test of the optimization module is `testoptimize.cc`.

The class names `NRVec`, `NRMat` etc. are historical: the classes were first used together with the Numerical Recipes library. They can be used with it, but are completely independent of it.

1.2 Overview

Header	**Content**
`la.h`	the single include file for the user
`vec.h`	`NRVec<T>`, `NRVec_from1<T>` (chapter 4)
`mat.h`	`NRMat<T>`, `NRMat_from1<T>` (chapter 5)
`smat.h`	`NRSMat<T>`, `NRSMat_from1<T>` (chapter 6)
`sparsemat.h`, `sparsesmat.h`	sparse matrices `SparseMat<T>`, `SparseSMat<T>` (chapter 7)
`csrmat.h`	compressed row storage sparse matrix (unfinished)
`nonclass.h`, `matexp.h`	linear systems, eigenvalue problems, SVD, matrix functions (chapter 8)
`davidson.h`, `lanczos.h`, `conjgrad.h`, `gmres.h`, `diis.h`, `bisection.h`	iterative methods, DIIS convergence acceleration (chapter 9)
`optimize.h`	numerical optimization (chapter 10)
`tensor.h`	tensors with index symmetries (chapter 11)
`fourindex.h`	four-index quantities, two-electron integrals (chapter 12)
`permutation.h`, `qsort.h`	permutations, partitions, Young tableaux, sorting (chapter 13)
`polynomial.h`, `contfrac.h`	polynomials, rationals and continued fractions (chapter 14)
`vecmat3.h`	3D vectors and matrices (chapter 15)
`quaternion.h`	quaternions (chapter 16)
`bitvector.h`	bit vectors, polynomials over GF(2) (chapter 17)
`numbers.h`, `graph.h`	number theory, graph algorithms (chapter 19)

Header	Content
`reg.h`, `regsurf.h`	regression (chapter 18)
`auxstorage.h`, `la_random.h`, `miscfunc.h`, `simple.h`, `laerror.h`	utilities (chapter 19)
`la_traits.h`	traits classes, the basis of the generic programming (section 3.3)

Everything is declared in the namespace `LA`, with the exception of a few headers designed to be usable also without the rest of the library (`vecmat3.h`: namespace `LA_Vecmat3`, `quaternion.h`: namespace `LA_Quaternion`, `simple.h`). `la.h` includes all headers and contains `using namespace LA;` together with a few typedefs (`NRDVec`, `NRCVec`, `NRIVec`, `NRDMat`, `NRCMat`, `NRIMat` for `double`, `std::complex<double>` and `int` elements).

1.3 A minimal example

```
#include "la.h"
using namespace LA;
using namespace std;

int main()
{
const int n=4;
NRSMat<double> h(n);
h.randomize(1.);          //symmetric matrix with random elements
NRSMat<double> hsave;
hsave |= h;               //deep copy, diagonalize() destroys its
    argument
NRVec<double> e(n);
NRMat<double> v(n,n);
diagonalize(h,e,&v);      //eigenvalues e (ascending),
    eigenvectors in columns of v
NRMat<double> r = hsave*v;  //residual H V - V diag(e)
r.copyonwrite();          //always before writing via operator()
    or []
for(int i=0; i<n; ++i) for(int j=0; j<n; ++j) r(i,j) -= v(i,j)*e[
    j];
cout << e << "residual norm = " << r.norm() << endl;
}
```

It is compiled e.g. by

```
g++ -O3 -I$PREFIX/include example.cc -L$PREFIX/lib -lla -llapack
    -lcblas -lblas
```

with the same preprocessor options and libraries the library was configured with (chapter 2).

1.4 Conventions used in this manual

- `T` denotes the element type of a class template. Unless said otherwise, BLAS and LAPACK backed operations are available for `double` and `std::complex<double>`; other element types (integer types, nested classes) support the generic operations.
- Indices run from 0, unless a `_from1` variant of a class is used.
- In the reference lists, "this" denotes the object the member function is called for. Functions that modify the object call `copyonwrite()` themselves unless stated otherwise.
- Boxes marked **Caution** point to common pitfalls; yellow boxes are open questions for the author, collected in the list at the beginning.

1.5 Authors and licence

LA is copyright © 2008– Jiří Pittner. The complex versions of the basic classes were written by Roman Curik, the CUDA interface was contributed by Miroslav Sulc; the regression module has its own author (chapter 18). The library is free software, distributed under the GNU General Public License, version 3 or later. Since 2026, the author started to employ an AI assistant for code review, documentation, and some new implementations.

Chapter 2

Building and using the library

2.1 Requirements

- a C++11 capable compiler (the library is developed with GNU g++),
- BLAS and LAPACK (Fortran interface); optionally the C interfaces CBLAS and CLAPACK, or Intel MKL, or ATLAS,
- GNU autotools (autoconf, automake, libtool) when building from the git repository,
- optionally NVIDIA CUBLAS (experimental, section 3.5),
- optionally doxygen for the generated source code reference, which is independent of this manual.

2.2 Configuration and installation

```
autoreconf --install       # only if ./configure does not exist
    yet
./configure [options] CXXFLAGS=-I<path> LDFLAGS=-L<path>
make                       # library, and this manual and the
    doxygen pages if pdflatex/doxygen were found
make check                 # builds the test programs, runs
    testoptimize
make install
```

`CXXFLAGS` and `LDFLAGS` should point to the preferred BLAS/LAPACK installation; after running `configure`, check in the generated `Makefile` that the intended library was found. `./configure -disable-optimize CXXFLAGS=" LDFLAGS="` gives a fast compilation for development.

`configure` looks for the libraries in the following order and sets the preprocessor macros accordingly:

1. Intel MKL (`mkl_intel_lp64`, `mkl_sequential`, `mkl_core` and the header `mkl_cblas.h`); if found, `-DHAS_MKL -DNONCLAPACK` is used;
2. otherwise ATLAS (optional) and the reference BLAS (mandatory), CBLAS (if `cblas_ddot` and `cblas.h` are not found, `-DNONCBLAS` is set) and LAPACK (mandatory; if the C interface `clapack_dgesv` with `clapack.h` is not found, `-DNONCLAPACK` is set);
3. CUBLAS: if `cublasInit` and `cublas.h` are found, the GPU support is switched on (`-DCUDALA`) and the C interfaces of BLAS and LAPACK are not used.

2.2.1 Configure options

Option	Effect
`-disable-debug`	do not define `DEBUG`; by default range checks and other consistency checks are compiled in (section 3.6)
`-enable-optimize=<flags>`	compiler optimization flags, default `-O3 -finline-limit=1000`; `-disable-optimize` for none
`-enable-profile`	compile with `-pg`
`-enable-matptr`	define `MATPTR`: `NRMat` stores an array of row pointers (`T**`), see chapter 5; incompatible with CUDA
`-enable-fortran64int`	define `LONG_FORTRAN_INT` to link with BLAS and LAPACK using 64-bit integers; implies `NONCBLAS` and `NONCLAPACK` (the old misspelling `-enable-fotran64int` is accepted as an alias)

2.2.2 Preprocessor macros

The macros below are set by `configure` in the `Makefile`. Since most of the library consists of templates compiled in the user's code, *a program using the library must be compiled with the same macros* (at least `DEBUG`, `MATPTR`, `CUDALA`, `NONCBLAS`, `NONCLAPACK`, `HAS_MKL`, `LONG_FORTRAN_INT`, `FORTRAN_`), otherwise the object layouts or the called functions do not match.

Macro	Meaning
DEBUG	range checks of indices, dimension checks, the check of the reference count in non-const element access
NONCBLAS	no CBLAS library: the cblas_* functions used by the library are provided by the wrappers in noncblas.cc, which call the Fortran BLAS
NONCLAPACK	the same for the C interface of LAPACK (the library calls the Fortran LAPACK routines)
HAS_MKL	use the headers of Intel MKL
FORTRAN_	Fortran routine names have a trailing underscore (always set in Makefile.am)
LONG_FORTRAN_INT	Fortran integers are 64-bit (long)
MATPTR	row pointer representation of NRMat
CUDALA	GPU support via CUBLAS (experimental)
NO_STRASSEN	do not compile the experimental Strassen matrix multiplication interface (strassen.cc, needs an external library); always set in Makefile.am

The headers fortran.h, noncblas.h, cuda_la.h, config.h and LA_version.h are internal to the library and are not documented in this manual.

2.3 Compiling and linking user programs

Include la.h (or the individual headers) and link with -lla, followed by the libraries found by configure, e.g.

```
g++ -O3 -DDEBUG -DFORTRAN_ -DNONCLAPACK -I$PREFIX/include prog.cc
     \
    -L$PREFIX/lib -lla -llapack -lcblas -lblas
```

The variables AM_CXXFLAGS and AM_LDFLAGS in the generated Makefile show the exact options used for the library itself.

The string LA::version (declared in LA_version.h) contains the git version, the commit date and the time and place of compilation of the library; it is generated by the script get_git_version at every make.

2.4 Test programs

make check builds the programs

t a large collection of tests and usage examples of nearly all parts of the library (t.cc, t2.cc); the individual tests are switched on and off by if(0) / if(1) blocks in the source, many of them read their input from the standard input;

test further small tests (`test.cc`);

tX a test with X11 graphics (`tX.cc`);

test_reg, test_regsurf tests of the regression module;

testoptimize the regression test of the optimization module (chapter 10); it prints PASS/FAIL lines and its exit code is the number of failed checks.

`make check` builds all these programs and runs `testoptimize` (automake `TESTS`); its result is in `test-suite.log`.

2.5 Documentation

This manual is in the subdirectory `doc`. `make manual` (alias `make doc`) compiles it by running `pdflatex` three times in the scratch directory `doc/.build` and copies the result to `doc/manual.pdf`; this is done only when a `.tex` file is newer than the PDF, and a LaTeX error stops the build with the end of the log. A source code reference is generated by doxygen with the configuration file `doxygen.cfg` by `make doxygen` (output in `doxygen-doc`, rebuilt when a source file changes); it is independent of this manual. `make` (i.e. `make all`) builds both in addition to the library, each only if `configure` found `pdflatex` resp. `doxygen`; `make clean` removes `doc/.build` and `doxygen-doc` (not the PDF).

Chapter 3

Common concepts

This chapter describes the design common to all classes of the library. The class chapters refer to it and describe only what is specific to the class.

3.1 Reference counting and copy on demand

3.1.1 Shared data

An object of `NRVec`, `NRMat`, `NRSMat` and of all other classes holding large data (`SparseMat`, `SparseSMat`, `Tensor`, `fourindex`, `Polynomial`, the permutation classes, `bitvector`, ...) consists of a small header (dimensions, a pointer to the data and a pointer to a reference counter) and the data block. The copy constructor and the assignment operator between objects of the same type copy only the header and increment the reference counter, so that both objects share the same data:

```
NRVec<double> a(1.,1000);   //1000 elements equal to 1
NRVec<double> b(a);         //no copy of the data, a.getcount()
    ==2
NRVec<double> c;
c = a;                      //no copy, count is 3 now
```

The data are deallocated when the last object referring to them is destroyed or assigned a different value. The current value of the counter is returned by `getcount()`.

Consequently, objects can be passed to and returned from functions by value at low cost, and expressions like `x = A*x` are safe, since the result is computed into a new temporary object which is then assigned to `x`.

3.1.2 Copy on demand

Before shared data are modified, the object must obtain its own copy of them. This is done by the member function

`void copyonwrite(bool detachonly=false, bool deep=true)`

If the reference count is larger than one, allocate a private copy of the data (copy them unless `detachonly`, which only allocates uninitialized memory) and decrement the count of the shared block. For nested element types (see section 3.3) it is applied recursively to the elements when `deep` is true, also when the outer object is not shared. For an object with count one it does nothing (apart from the recursion).

The rules are:

1. *Every member function and operator of the library which modifies the data of an object as a whole calls* `copyonwrite()` *itself* (`axpy`, `gemv`, `gemm`, `+=`, `*=`, assignment of a scalar, `randomize`, `storesubmatrix`, `resize`, ...). Explicit calls before these functions are not needed.

2. *Before writing to individual elements through* `operator[]`, `operator()`, *a raw pointer* (`operator T*()`, `&x[0]`, ...) *or a member function accessing individual elements* (such as `swap(i,j)` of `NRVec`, used in sorting loops) the user has to call `copyonwrite()` explicitly. In the first cases the library cannot distinguish a write access from a read access; element-level members do not call it by design, since a `copyonwrite()` inside inner loops over individual elements would be an unacceptable overhead:

   ```
   NRVec<double> b(a);   //b shares data with a
   b.copyonwrite();      //b gets its own copy
   b[0] = 5.;            //a is not changed
   ```

   Without the `copyonwrite()`, the assignment would silently change `a` as well. The rule applies to every object which might share its data: objects obtained by copying or assignment, objects received as function arguments by reference or pointer (they may share data with other copies in the caller), objects stored in other containers. A freshly constructed object (by a constructor taking dimensions) has count one, but calling `copyonwrite()` for it costs nothing.

3. `resize()` detaches an object from shared data: a shared object always gets newly allocated memory (with the old contents only if their preservation is requested), even when the size does not change. An object which has just been resized can be written to. `resize(0)` deallocates.

3.1.3 The DEBUG check of the reference count

In programs compiled with `DEBUG`, the non-const versions of `operator[]` and `operator()` of the basic classes call `laerror` (section 3.6) when the reference count is larger than one, since this is a possible write to shared data:

```
NRVec<double> c = a;
c[1] = 7.;   //DEBUG: LA:ERROR - possible use of NRVec[] with
    count>1 as l-value
```

The check cannot distinguish reading from writing. Reading elements of a possibly shared object should therefore be done through a const reference, which selects the const version of the operator:

```
void f(NRVec<double> &x)   //non-const, x may share data with
    another vector
{
const NRVec<double> &xc = x;
double s = xc[0] + xc[1];  //no false alarm
x.copyonwrite();
x[0] = s;                  //write
}
```

The check can be switched off at run time by setting the global variable `LA::_LA_count_check=false`. It is a safety net only: writes through raw pointers and writes in programs compiled without `DEBUG` are not checked.

Caution: A missing `copyonwrite()` does not cause a crash, it silently modifies other objects which share the data. It is the most frequent error when using the library. A library member function which modifies data without calling `copyonwrite()` is a bug of the library.

3.1.4 Objects referring to external memory

The constructor `NRVec(T *a, int n, bool skeleton)` with `skeleton=true` (and analogous constructors of other classes, see the class chapters) creates an object which refers to memory owned by somebody else, without copying. Its reference count is set to two, so that the memory is never deallocated by the library and any modification by a library function first makes a private copy (writes through `operator[]` go to the external memory, but are reported by the DEBUG check). It is useful for applying library functions to data stored elsewhere. Such an object is intended as a *read-only view* of the external data.

3.2 Shallow and deep copy

`X(const X &rhs); X &operator=(const X &rhs)`
: shallow copy: share the data with `rhs` (section 3.1).

`X &operator|=(const X &rhs)`
: deep copy: `*this = rhs` followed by `copyonwrite()`; this object gets its own copy of the data, recursively for nested types.

`X &operator=(const T &a)`
: assignment of a scalar. For vectors every element is set to `a`; for square matrices (`NRMat`, `NRSMat`) the matrix is set to $a\,\mathbf{1}$, i.e. the diagonal is set to `a` and the off-diagonal elements to zero. In the same spirit `+=` and `-=` with a scalar add to the diagonal of a matrix.

The deep copy is needed when the copy will be modified by element access, or when a function destroys its argument (e.g. `diagonalize`):

```
NRMat<double> a(n,n);
...
NRMat<double> asave;
asave |= a;          //a will be overwritten
```

`x.swap(y)` exchanges the contents of two objects without any copying.

Conversions between different classes (e.g. `NRSMat` → `NRMat`, `NRMat` → `NRVec`, real → complex) are done by explicit constructors and always create new data, unless stated otherwise in the class chapters.

3.3 Traits and element types

Header: `la_traits.h`

The algorithms of the library are written generically with the help of the traits class template `LA_traits<T>`, which is defined for all scalar types and for all container classes of the library. Its main members are:

`typedef ... elementtype`
: type of the elements (for a scalar the type itself)

`typedef ... normtype`
: type of the norm, e.g. `double` for `std::complex<double>` and for `NRMat<std::complex<double> >`

`typedef ... realtype; typedef ... complextype`
: the corresponding real and complex type, e.g. `NRVec<double>` and `NRVec<std::complex<double> >`

`typedef ... producttype`
: type of the product of two objects (for `NRSMat` it is `NRMat`)

`static bool is_plaindata()`
: true for scalar types, which can be copied by `memcpy`, false for the classes

`static bool is_complex()`
: whether the (innermost) element type is complex

`static normtype norm(const T &x); static void axpy(T &s, const T &x, const elementtype c)`
: norm and $s \leftarrow s + c\,x$, used by generic algorithms such as `matexp.h`

`static void copyonwrite(T &x); static void clearme(T &x); static void deallocate(T &x)`
: generic versions of the member functions (no operation for scalars)

`static void put(int fd, const T &x, ...); static void get(int fd, T &x, ...); multiput; multiget`
: binary I/O, recursive for nested types

`static T conjugate(const T &x); realpart(x); imagpart(x); setrealpart(x,y); setimagpart(x,y)`
: complex helpers, trivial for real types

`static bool bigger(const T &x, const T &y); smaller(x,y)`
: comparison used for sorting; for complex types it is an error

The auxiliary template `isscalar<T>` distinguishes scalars from classes, `LA_traits_complex<T>` gives for a complex type the component type and the corresponding real vector and matrix types (used by the mixed real–complex functions such as `gemv` of a complex vector with a real matrix), and `LA_traits_io<T>` makes `char` types be read and written as numbers by the stream operators. The functions `LA::real(c)` and `LA::imag(c)` return references to the real and imaginary part of a `std::complex`, which can be used as l-values.

3.3.1 Element types and instantiation

The class templates are explicitly instantiated in the library for `double`, `std::complex<double>` and the integer types `char`, `short`, `int`, `long`, `long long` and their unsigned variants. The BLAS and LAPACK backed functions are implemented for `double` and `std::complex<double>` (some also for mixed real–complex arguments); calling them for other types results in a run-time `laerror` ("not implemented for this type") or a link error. Functions implemented in the headers work for any type with the necessary arithmetic operations.

3.3.2 Nested types

The element type can be a class of the library, e.g. `NRVec<NRMat<double> >` (a list of matrices) or `NRMat<NRVec<double> >`. Copy on demand is then applied on both levels: `copyonwrite()` of the outer object copies the inner objects shallowly and calls their `copyonwrite()`, so that `|=` makes a complete deep copy; `put` and `get` write and read the inner objects including their dimensions. Arithmetic operations of the outer class (e.g. `axpy`) work elementwise via the traits.

Caution: Writing into an inner object obtained through `operator[]` of the outer object (`v[i](j,k)=x`) requires both objects to be unshared: call `v.copyonwrite()` (which recursively detaches the inner objects as well) before.

3.4 Storage conventions

- Indices run from 0. The classes `NRVec_from1`, `NRMat_from1` and `NRSMat_from1` are derived from the basic classes and differ only in their element access operators, which take indices from 1. They convert to and from their base classes without copying.
- `NRVec`: contiguous array of `size()` elements.
- `NRMat`: row major (C) order, element (i, j) of an $n \times m$ matrix is at position $i\,m + j$; `a[i]` is a pointer to the i-th row. Note that BLAS and LAPACK use column major order, so the library internally works with the transposed matrices; the user does not need to care, except when calling Fortran routines directly.
- `NRSMat`: lower triangle of a symmetric or hermitian matrix, packed by rows, element (i, j) with $i \geq j$ is at position $i(i+1)/2+j$ (which is also the upper triangle packed by columns, the format of the BLAS/LAPACK packed routines). The macros/functions `SMat_index` compute these positions.
- `Tensor`: the leftmost index changes fastest (chapter 11).

All classes can return a raw pointer to their data (e.g. `operator T*()` of `NRVec` and `NRMat`), for passing them to external routines; the copy on demand rule for writes through pointers applies.

3.5 GPU support (experimental)

When compiled with `CUDALA` (configure finds CUBLAS), the data of `NRVec`, `NRMat` and `NRSMat` can reside either in the main memory or in the memory of a GPU. The location is described by the enumeration

```
typedef enum {undefined=-1, cpu=0, gpu1=1, gpu2=2, gpu3=3, gpu4
    =4} GPUID;
```

Constructors take an optional location argument (default `undefined`, which means the global default location `DEFAULT_LOC`, settable by `set_default_loc()`); `getlocation()` returns the location and `moveto(loc)` moves the data. BLAS-like operations of objects on the GPU are performed by CUBLAS; operands of a binary operation must reside in the same

location. Operations which are not implemented for the GPU call `laerror` ("Operation not implemented on GPU (yet). Use .moveto(0) first."); element access by `operator[]` and `operator()` is possible only in the main memory. Without `CUDALA`, `getlocation()` always returns `cpu` and `moveto()` does nothing, so that code written for the GPU compiles also without it.

Caution: The GPU support is experimental and has not been thoroughly tested. It is not documented in detail in this manual; the class chapters at most mention operations restricted to the CPU.

The GPU code is currently not maintained: it uses the legacy CUBLAS API (`cublasInit`) and a compute capability (`-arch sm_20`) not supported by current CUDA versions, and the `CUDALA` build has known compile errors. It will be revisited with current NVIDIA hardware.

3.6 Error handling and DEBUG checks

Header: `laerror.h, laerror.cc`

Errors are reported by the macro `laerror(message)`, which calls `LA::laerror2(message, function)` with the name of the calling function. `laerror2` prints the message and the function name to both `std::cout` and `std::cerr`, the system error message if `errno` is set, the command line, the user name, host name and working directory (and the list of files in the working directory if `LA_error_list_files` is true), and then throws an exception of the class

```
class LAerror : public std::exception {public: std::string msg;
    ... const char *what() const noexcept;};
```

which can be caught and printed by `operator«`:

```
try { ... }
catch(LAerror &e) {std::cerr << "error: " << e << std::endl;}
```

Errors detected by BLAS, CBLAS and ATLAS (via `xerbla` and its variants) are converted to `laerror` as well. Users may call `laerror` in their own code. The function `sigtraceback(signal,flags)` installs a handler printing a stack trace (by running `gdb` on the process) when the given signal arrives, e.g. `sigtraceback(SIGSEGV,1)`; see chapter 19.

With `DEBUG` defined, the library checks index ranges, dimensions of operands, allocation of objects and the reference count in element access (section 3.1). Without `DEBUG`, these errors lead to undefined behaviour. Programs should be developed and tested with `DEBUG`.

3.7 Input and output

All classes support three kinds of I/O:

Stream operators `operator«` and `operator»` write and read a human readable text format: the dimensions followed by the elements (a matrix by rows, one row per line). The input operator resizes the object. `char` elements are written as numbers.

`fprintf, fscanf` formatted output with a `printf` format for the elements and input from a C `FILE*` (see the class chapters).

`put, get` unformatted binary I/O to a POSIX file descriptor:

```
void put(int fd, bool dimensions=1, bool transp=0, bool
    orca_format=false) const;
void get(int fd, bool dimensions=1, bool transp=0, bool
    orca_format=false);
```

With `dimensions` true, a header with the dimensions is written before the data and `get` resizes the object accordingly; otherwise nothing but the elements is written and the object must already have the right size. The format is the binary representation of the machine (no conversion of endianity). Per class:

- `NRVec`: header n and a padding `int` 1 (8-byte alignment), then the n elements; `transp` is ignored.
- `NRMat`: header `nrows`, `ncols`, then the elements by rows. With `transp` the transposed matrix is written (dimensions swapped, elements column by column); `get` with `transp` reads such data back into the original orientation, i.e. it reads a file stored by columns.
- `NRSMat`: header n, n, then the packed lower triangle; `transp` is ignored.
- `SparseMat`: header `nrows`, `ncols` (`SPMatindex`) and the `int` flags `symmetric` and `nonzero`, then triples (row, column, element) and a terminating pair of indices -1; `SparseSMat`: header `nrows`, `ncols`, triples and the terminating pair. With `transp` rows and columns are swapped.
- `Tensor` and `fourindex` have their own `put`/`get` (see their chapters).

`orca_format` (dense classes only; `laerror` for the sparse ones) writes the header of the ORCA program: the dimensions followed by `sizeof(T)` instead of the padding; ORCA matrices are stored by columns, so they are read by `get(fd,true,true,true)`. Elements of class type (e.g. a vector of matrices) are written recursively by their own `put`, with the same `dimensions` flag; `transp` is passed on to them only by the transposed `NRMat` and the sparse matrices.

For storing many objects of the same size in a scratch file, see `AuxStorage` (chapter 19).

Chapter 4

Vectors: NRVec

Header: `vec.h, vec.cc`

4.1 Purpose and main idea

`NRVec<T>` is the basic dense vector class template of LA: a contiguous array of `nn` elements of type `T`, indexed from zero, with reference-counted storage (copy on demand). It is used directly as the vector type of linear algebra (BLAS-1 and BLAS-2 operations for `double` and `std::complex<double>`), as a general resizable container of arbitrary elements (including nested LA objects such as `NRVec<NRVec<int> >`), and as the storage backbone of other classes (`NRMat<T>` and `NRSMat<T>` can be viewed as vectors without copying, see Sec. 4.3). `NRVec_from1<T>` (Sec. 4.9) is a thin derived class with indexing from one.

The class is explicitly instantiated in `vec.cc` for the element types `double`, `char`, `short`, `int`, `long`, `long long`, their `unsigned` variants, and for `std::complex<double>`. Members defined in `vec.h` can be used with any other element type (user classes, nested LA types, `float`, ...); members defined only in `vec.cc` (`NRVec(const NRMat&)`, `fprintf`, `fscanf`, unary `operator-`, `operator<`, `operator>`, `find`, `findthr`, `permuteme(const CyclePerm<int>&)`, `conjugateme`, `negateme`, `negateconjugateme`, `sum`) are available only for the instantiated types. Numerical members based on BLAS are intended for floating-point element types only (real and complex), not for integers; at present they exist for `double` and `std::complex<double>` (see the remarks in the reference below).

4.2 Storage and class-specific concepts

The object holds the size `nn`, a data pointer `v` and a pointer `count` to the reference counter (plus a location tag in GPU builds, see Sec. 3.5). A default-constructed vector has `nn==0`, `v==0`, `count==0`.

NRVec follows the copy-on-demand scheme of Sec. 3.1: copy construction and operator= share the data, operator|= makes a deep copy (Sec. 3.2), member functions that modify the data call copyonwrite() themselves, resize() detaches shared data, and an explicit copyonwrite() is needed only before writing through operator[] or a raw pointer. In DEBUG builds the non-const operator[] raises laerror if the reference count is larger than one and _LA_count_check is set. For element types that are not plain data (LA_traits<T>::is_plaindata() false, e.g. nested vectors), copyonwrite() recursively calls copyonwrite() on the elements, see Sec. 3.3.

The element-level member swap(int,int) does not call copyonwrite() by design, like operator[]: call copyonwrite() once before using it on a vector that may share its data (Sec. 3.1).

Elements are stored contiguously, so (T*)x or &x[0] can be passed to C/Fortran routines expecting an array. The nested type ROWTYPE is T.

4.3 Construction, assignment, conversion

`NRVec()`
: Empty vector (size 0, nothing allocated).

`explicit NRVec(const int n, const GPUID loc = undefined)`
: Vector of n elements, *not initialized* (for plain data). loc is the memory location in GPU builds, ignored otherwise.

`NRVec(const T &a, const int n, const GPUID loc = undefined)`
: Vector of n elements all equal to a. Note the order of arguments: value first, size second.

`template<int SIZE> NRVec(const T (&a)[SIZE])`
: Copy of a C array; also accepts a braced list, e.g. NRVec<double> x({1.,2.,3.}).

`NRVec(const T *a, const int n)`
: Copy of n elements from memory at a.

`NRVec(T *a, const int n, bool skeleton)`
: With skeleton==false same as the previous one. With skeleton==true the vector *wraps* the external memory a without copying; the reference count is set to 2 so that the memory is never freed by LA. Consequently any modifying member function makes a private copy first (the external memory is *not* modified), and DEBUG-mode operator[] for writing raises an error. Use it for read-only views of foreign data.

`NRVec(const Vec3<T> &rhs)`
: Copy of the three components of a Vec3 (from vecmat3.h, see Chapter 15).

`NRVec(const NRVec &rhs)`
: Copy constructor, shares data.

`explicit NRVec(const std::list<T> l)`
: Copy of the elements of a `std::list` (the list is passed by value).

`NRVec(const typename LA_traits_complex<T>::NRVec_Noncomplex_type &rhs, bool imagpart=false)`
: Complex vector from a real one: `rhs` becomes the real part (or the imaginary part if `imagpart`), the other part is zero. Implemented only for `NRVec<std::complex<double> >` from `NRVec<double>`.

`explicit NRVec(const NRSMat<T> &S)`
: View of the packed storage of a symmetric matrix of order n as a vector of $n(n+1)/2$ elements. *Shares* the data with `S` (no copy).

`explicit NRVec(const NRMat<T> &rhs)`
: View of the row-major storage of an $n \times m$ matrix as a vector of nm elements. *Shares* the data with `rhs`; if the library is compiled with `MATPTR`, the data are copied instead.

`explicit NRVec(const SparseMat<T> &rhs)`
: Dense vector from a sparse matrix having one row or one column (otherwise `laerror`); duplicate entries are summed. Defined in `sparsemat.cc` for `double` and `std::complex<double>`.

`~NRVec()`
: Decrements the reference count, frees the data when it reaches zero.

`NRVec& operator=(const NRVec &rhs)`
: Shallow assignment (shares data), see Sec. 3.2.

`NRVec& operator|=(const NRVec &rhs)`
: Deep copy; `rhs` must be allocated (checked in DEBUG).

`NRVec& operator=(const T &a)`
: Sets all elements to `a` (keeps the size); calls copyonwrite.

`void copyonwrite(bool detachonly=false, bool deep=true)`
: If the data are shared, allocate a private copy. `detachonly=true` allocates private storage without copying the contents (they are undefined afterwards). With `deep` and non-plain-data `T`, elements are made unique recursively even if this object is not shared. Calling it on an empty vector is harmless (prints a warning if `_LA_warn_empty_copyonwrite` is set).

`int getcount() const`
: Current reference count (0 for an empty vector).

`void resize(const int n, const bool preserve=false, const bool`

`preserve_at_end=false)`

Change size; detaches from shared data. Without `preserve` the new contents are undefined (no-op if the size does not change and the data are not shared). With `preserve` the first $\min(n_{\text{old}}, n)$ elements are kept and new elements are zeroed (by `memset`); with `preserve_at_end` the last elements are kept and aligned to the end, the new ones being at the beginning. `resize(0)` deallocates.

`void dealloc(void)`

Same as `resize(0)`.

`void clear()`

Set all elements to zero (`LA_traits<T>::clear`); detaches shared plain data.

`void swap(NRVec &rhs)`

Exchange contents (size, data, counter) of two vectors in $O(1)$.

`template<typename U> NRVec<std::complex<U> > complexify(const NRVec<U> &rhs)`

(free function) Complex vector with real part `rhs`. CPU only (NOT_GPU) in the generic version; specialized for `double` via BLAS.

`template<typename T, typename S> void NRVec_convert(NRVec<T> &a, const NRVec<S> &b)`

(free function) Resize `a` and set `a[i] = (T)b[i]`.

4.4 Element access

`T & operator[](const int i); const T & operator[](const int i) const`

Element i, $0 \le i < n$. DEBUG builds check the range, allocation, and (non-const version) the reference count, see Sec. 4.2.

`operator T*(); operator const T*() const`

Pointer to the data (DEBUG: error if unallocated). Call `copyonwrite()` before writing through the pointer.

`int size() const`

Number of elements.

`const NRVec subvector(const int from, const int to) const`

Copy of elements `from`...`to` (inclusive). If this is the whole vector, a shallow copy is returned. Plain data only.

`const NRVec subvector(const NRVec<int> &selection) const`

Gather into a new vector: `r[i] = x[selection[i]]`. Index range is always checked.

`void subvector_from(const NRVec<int> &selection, const NRVec<T> &rhs)`

Gather into `*this`: `x[i] = rhs[selection[i]]`; `x.size()` must equal `selection.size()` (otherwise `laerror`); calls copyonwrite.

`void storesubvector(const int from, const NRVec &rhs)`
: Copy `rhs` into positions `from...from+rhs.size()-1`; plain data only; calls copyonwrite.

`void storesubvector(const NRVec<int> &selection, const NRVec &rhs)`
: Scatter into `*this`: `x[selection[i]] = rhs[i]`; sizes of `selection` and `rhs` must agree; calls copyonwrite.

`void storesubvector_to(const NRVec<int> &selection, NRVec &rhs) const`
: Scatter `*this` into `rhs`: `rhs[selection[i]] = x[i]`; `x.size()` must equal `selection.size()`; calls `rhs.copyonwrite()`.

`NRVec reverse() const`
: Copy with reversed order of elements.

`NRVec concat(const NRVec &rhs) const; NRVec concatscaled(const NRVec &rhs, const T alpha=1) const`
: New vector $(x, \texttt{rhs})$, resp. $(x, \alpha\,\texttt{rhs})$. If one of the vectors is empty, the other one is returned (shared; for `concatscaled` with an empty `*this` and $\alpha \neq 1$ a scaled copy of `rhs`).

`void concatme(const NRVec &rhs); void concatmescaled(const NRVec &rhs, const T alpha=1)`
: Append `rhs` (resp. $\alpha\,\texttt{rhs}$) to `*this` in place (via `resize` with preservation).

`void append(const T &a); void prepend(const T &a)`
: Add one element at the end / beginning. Each call reallocates; inefficient when repeated.

`const int find(const T &val) const`
: Index of the first element equal to `val`, -1 if none.

`const int findthr(const T &val, const typename LA_traits<T>::normtype &thr=0) const`
: Index of the first element with $|x_i - \texttt{val}| < \texttt{thr}$, -1 if none (note the strict inequality: the default `thr=0` never matches).

4.5 Arithmetic and BLAS-backed operations

All compound operators call `copyonwrite()`. Dimensions of vector operands are checked only in DEBUG builds. For `double` and `std::complex<double>` the operations marked BLAS are specialized to call CBLAS; the generic versions are element-wise loops (CPU only).

`NRVec& operator+=(const NRVec &rhs); NRVec& operator-=(const NRVec &rhs)`
: Element-wise sum/difference (BLAS axpy).

`NRVec& operator*=(const NRVec &rhs); NRVec& operator/=(const NRVec &rhs)`
: Element-wise (Hadamard) product/quotient.

`const NRVec operator+(const NRVec &rhs) const; const NRVec operator-(const NRVec &rhs) const`

Sum/difference. There are no binary element-wise `*` and `/` of two vectors; `x*y` is the scalar product.

`NRVec& operator+=(const T &a); NRVec& operator-=(const T &a)`

Add/subtract `a` to/from every element (BLAS).

`NRVec& operator*=(const T &a); NRVec& operator/=(const T &a)`

Scale every element (BLAS scal; division as multiplication by $1/a$ for the specialized types).

`const NRVec operator+(const T &a) const; ... operator-, operator*, operator/`

`x+a`, `x-a`, `x*a`, `x/a` element-wise with a scalar.

`const NRVec operator+(const T &a, const NRVec &x); const NRVec operator*(const T &a, const NRVec &x)`

(free functions) Scalar on the left: $a+x$ and ax. The noncommutative `a-x` and `a/x` are deliberately not available (declared `= delete`, a compile error); write `-x+a` explicitly.

`const NRVec operator-() const`

Negated copy.

`NRVec& negateme()`

$x \leftarrow -x$ in place.

`NRVec& conjugateme(); NRVec conjugate() const`

Complex conjugation in place / conjugated copy (no-op for real types).

`NRVec& negateconjugateme()`

$x \leftarrow -x^*$ in place (specialized for `std::complex<double>` and `std::complex<float>`).

`const T operator*(const NRVec &rhs) const`

Scalar product $\sum_i x_i^* y_i$ (conjugating `*this` for complex types; BLAS dot/dotc).

`const T dot(const T *a, const int stride = 1, bool conjugate=true) const`

$\sum_i \tilde{x}_i \, a_{i\cdot\mathrm{stride}}$, where $\tilde{x}_i = x_i^*$ if `conjugate` and `T` is complex, else x_i; `a` must have sufficient length.

`const T dot(const NRVec &rhs, bool conjugate=true) const`

Same with a vector (no dimension check).

`void axpy(const T alpha, const NRVec &x)`

`this` $\leftarrow$ `this` $+ \alpha$ `x` (BLAS axpy). Only `double` and `std::complex<double>`.

`void axpy(const T alpha, const T *x, const int stride=1)`

Same with strided raw data; CPU only. Only `double` and `std::complex<double>`.

`void gemv(const T beta, const NRMat<T> &a, const char trans, const T alpha, const NRVec &x)`

$y \leftarrow \alpha\,\mathrm{op}(A)x + \beta y$ with $y =$ `*this`; `trans` is `'n'` (A), `'t'` (A^{T}) or for complex `'c'` ($A^{\dagger}$), case insensitive. `*this` must already have the correct size (checked in DEBUG builds) and defined contents if $\beta \neq 0$.

`void gemv(const T beta, const NRSMat<T> &a, const char trans, const T alpha, const NRVec &x)`

Same for a packed symmetric (real) or Hermitian (complex) matrix (BLAS spmv/hpmv); `trans`: for complex `'t'` gives $a^{\mathsf{T}}x = a^{*}x$, `'n'` and `'c'` give ax; ignored for real matrices.

`void gemv(const T beta, const SparseMat<T> &a, const char trans, const T alpha, const NRVec &x, const bool treat_as_symmetric = false)`

Same for a sparse matrix (defined in `sparsemat.cc`, see Chapter 7); with `treat_as_symmetric` only one triangle is assumed to be stored.

`void gemv(const Component_type beta, const NRMat_Noncomplex_type &a, const char trans, const Component_type alpha, const NRVec &x);`
`void gemv(const Component_type beta, const NRSMat_Noncomplex_type &a, const char trans, const Component_type alpha, const NRVec &x)`

(Types from `LA_traits_complex<T>`.) Complex vector times *real* dense or packed-symmetric matrix with real coefficients; real and imaginary parts are processed separately.

`const NRVec operator*(const NRMat<T> &mat) const`

Row vector times matrix, $x^{\mathsf{T}}A$ (gemv with `'t'`; no conjugation). For Ax see `NRMat::operator*` in Chapter 5.

`const NRVec operator*(const NRSMat<T> &mat) const`

Row vector times packed symmetric/hermitian matrix, $x^{\mathsf{T}}S$ (for hermitian S computed as $S^{\mathsf{T}}x = S^{*}x$ by `gemv` with `'t'`).

`const NRVec operator*(const SparseMat<T> &mat) const`

Row vector times sparse matrix; CPU only.

`const NRMat<T> otimes(const NRVec<T> &rhs, const bool conjugate = false, const T &scale = 1) const`

Outer product $A_{ij} = \mathrm{scale} \cdot x_i\, y_j$, with y_j^{*} if `conjugate` (complex only); BLAS ger/geru/gerc.

`const NRMat<T> operator|(const NRVec<T> &rhs) const`

Same as `otimes(rhs,true)`, i.e. $xy^{\dagger}$.

`const NRVec otimes2vec(const NRVec<T> &rhs, const bool conjugate = false, const T &scale = 1) const`

Outer product stored as a vector of length $n_x n_y$ (row-major, index $in_y + j$).

The real-matrix `gemv`, `normalize`, `otimes` and `otimes2vec` exist only for `double` and `std::complex<double>`; for the instantiated integer types (and complex integer types) they are defined to raise `laerror`.

4.6 Norms, reductions, extrema

`const typename LA_traits<T>::normtype norm(int length= -1, int offset=0, int stride=1) const`
: Euclidean norm (BLAS nrm2) of `length` elements (default all) starting at `offset` with `stride`. Only `double` and `std::complex<double>`.

`NRVec& normalize(typename LA_traits<T>::normtype* norm = 0)`
: Divide by the Euclidean norm, optionally storing the norm in `*norm`; DEBUG: error for a zero vector. Only `double` and `std::complex<double>`.

`const NRVec unitvector() const`
: Normalized copy.

`const typename LA_traits<T>::normtype asum() const`
: $\sum_i |x_i|$ (BLAS asum; for complex $\sum_i |\mathrm{Re}\, x_i| + |\mathrm{Im}\, x_i|$). Only `double` and `std::complex<double>`.

`inline int iamax() const`
: Zero-based index of the element of largest absolute value (BLAS i?amax; for complex the 1-norm |Re| + |Im|). Only `double` and `std::complex<double>`. `laerror` for an empty vector.

`inline const T amax() const; inline const T amin() const`
: The element (with sign) of largest / smallest absolute value (complex: 1-norm). Only `double` and `std::complex<double>`. `laerror` for an empty vector.

`const T max() const; const T min() const`
: Largest / smallest element (0 for an empty vector). Only real types (instantiated for the integer types, `float`, `double`).

`const T sum() const`
: $\sum_i x_i$.

`const T sumsqr() const`
: $\sum_i x_i^2$, 0 for an empty vector. No conjugation for complex types; use `norm()` for $\sqrt{\sum_i |x_i|^2}$.

`const T prod() const`
: $\prod_i x_i$, 1 for an empty vector.

`NRVec<typename LA_traits<T>::normtype> abs() const`
: Vector of $|x_i|$.

`NRVec<typename LA_traits<T>::normtype> diffabs(const NRVec &rhs)`

`const`
: Vector of $|x_i| - |y_i|$.

4.7 Other member functions

Comparison

`const bool operator==(const NRVec &rhs) const; const bool operator!=(const NRVec &rhs) const`
: Equal size and elements. For plain data the comparison is bytewise (`memcmp`), so e.g. `0.0` and `-0.0` differ.

`const bool operator<(const NRVec &rhs) const; ... operator>, operator<=, operator>=`
: Lexicographic order using `LA_traits<T>::smaller/bigger`; a proper prefix is smaller. Raises `laerror` for complex elements.

Sorting and permutations

`int sort(int direction = 0, int from = 0, int to = -1, int *perm = NULL, bool stable=false)`
: Quicksort of elements `from...to` (`-1`: last) in place, ascending for `direction==0`, descending otherwise; returns the parity (0/1) of the applied permutation. If `perm` is given, `perm[from..to]` (initialized by the caller) is permuted along with the elements. `stable` selects a stable variant. Calls copyonwrite.

`int sort(int direction, NRPerm<int> &perm, bool stable=false)`
: Sort the whole vector; `perm` (of the same size) is set to the identity and returned with `perm[i]` = original (1-based) position of the element now at position i (1-based).

`void printsorted(std::ostream &s, int direction=1, bool absvalue=false, bool stable=false, typename LA_traits<T>::normtype thr=0, int topmax=0, const NRMat<int> *indexmap=NULL) const`
: Print lines "*index value*" in sorted order (default descending) without modifying the vector. `absvalue`: sort by $|x_i|$; `thr`: skip elements with $|x_i| <$ thr; `topmax`: print at most this many lines (0 = all); `indexmap`: print `(*indexmap)[i][0]` and `(*indexmap)[i][1]` instead of the index i.

`bool bigger(int i, int j) const; bool smaller(int i, int j) const; void swap(int i, int j)`
: Element comparison and exchange used by the sorting templates (`qsort.h`). `swap(i,j)` is element-level access and, like `operator[]`, does not call copyonwrite: call it before, outside the loop (Sec. 4.2).

`const NRVec permuted(const NRPerm<int> &p, const bool inverse=false) const`

Permuted copy: `r[p[i]-1] = x[i-1]`, or with `inverse` `r[i-1] = x[p[i]-1]` (p is 1-based, see Chapter 13).

`void permuteme(const NRPerm<int> &p, const bool inverse=false); void permuteme(const CyclePerm<int> &p)`

The same in place; for a cycle $(c_1 \dots c_L)$ the element at c_k moves to c_{k+1} and the one at c_L to c_1.

Element-wise function application

`NRVec& call_on_me(T (*_F)(const T &))`

$x_i \leftarrow F(x_i)$ for a plain function pointer. This non-template overload is kept on purpose: when the name of an *overloaded* function is passed, the fixed parameter type selects the right overload (the template below cannot deduce its argument from an overload set). Note that the function must take `const T &`, so e.g. `std::sqrt` does not match it.

`template<typename F> NRVec& call_on_me(F _F)`

$x_i \leftarrow F(x_i)$ for any callable (functor, lambda, function with another compatible signature). For a single non-overloaded function both overloads match and the non-template one is chosen.

`template<typename F> NRVec& call_on_me(F _F, const NRVec &rhs)`

Binary version $x_i \leftarrow F(x_i, \texttt{rhs}_i)$; sizes checked in DEBUG.

All three are CPU only and call copyonwrite.

Random numbers

`void randomize(const typename LA_traits<T>::normtype &x)`

Fill with uniform random numbers from $[-x, x)$ (real and imaginary parts independently for complex), using `RANDDOUBLESIGNED` (Chapter 19). Only `double` and `std::complex<double>`.

Input/output

See Sec. 3.7 for the general conventions.

`void put(int fd, bool dimensions=1, bool transp=0, bool orca_format=false) const`

Binary output to file descriptor. With `dimensions`: header `int n`, then `int 1` (padding), or with `orca_format` `int sizeof(T)`; followed by the elements (`LA_traits<T>::multiput`, recursive for nested types). `transp` is unused.

`void get(int fd, bool dimensions=1, bool transp=0, bool orca_format=false)`
: Binary input matching `put`; with `dimensions` the vector is resized, otherwise the current size is used (and copyonwrite called). `orca_format` checks the element size.

`void fprintf(FILE *f, const char *format, const int modulo) const`
: Formatted output via `lawritemat` as a $1 \times n$ matrix (Chapter 8).

`void fscanf(FILE *f, const char *format)`
: Read the dimension and then the elements with the given `scanf` format (e.g. `"%lf"` for `double`); resizes.

Miscellaneous

`GPUID getlocation() const; void moveto(const GPUID dest)`
: Memory location (always `cpu` without `CUDALA`, then `moveto` is a no-op); see Sec. 3.5.

`void setcoldim(int i); void simplify()`
: Dummies for interface compatibility with other classes.

4.8 Related free functions

`std::ostream & operator«(std::ostream &s, const NRVec<T> &x)`
: Text output: the size on one line, then the elements separated by spaces on the next line (converted to `LA_traits_io<T>::IOtype`, so e.g. `char` is printed as a number).

`std::istream & operator»(std::istream &s, NRVec<T> &x)`
: Reads the format written by `«`; resizes `x`.

`template<typename T> void NRVec_simplify(NRVec<T> &x, const typename LA_traits<T>::normtype thr=0, bool alwayskeepfirst=false)`
: For vectors of "coefficient $\times$ element" terms (`T` must provide sorting traits, `operator==` comparing only the element, and `LA_traits<T>::coefficient`, `abscoefficient`, `elementtype`): sort, merge equal elements by adding coefficients, drop terms with zero coefficient or $|\text{coefficient}| < \text{thr}$ (the first term is kept if `alwayskeepfirst`), and shrink the vector. Public API; within the library it is presently used by `PermutationAlgebra::simplify()`.

`template<typename T> typename LA_traits<T>::normtype MYABS(const T &x)`
: `std::abs` with pass-through specializations for unsigned integer types; auxiliary.

`complexify` and `NRVec_convert` are listed in Sec. 4.3. The header also defines constants `CONE`, `CMONE`, `CZERO` (complex $1, -1, 0$ for CBLAS calls)

and the internal macros `NRVECMAT_OPER`, `NRVECMAT_OPER_NONCOMMUTATIVE`, `NRVECMAT_OPER2` generating the binary operators from the compound ones.

4.9 Vectors indexed from one: NRVec_from1

`NRVec_from1<T>` is derived publicly from `NRVec<T>` and differs only in indexing: `operator[]` takes $1 \le i \le n$ (same DEBUG checks as `NRVec`), and `iamax()` returns a 1-based index. It is used e.g. for permutations (Chapter 13). Everything else is inherited unchanged and *remains zero-based* (e.g. `subvector`, `find`, `sort(direction,from,to,...)`, the raw pointer).

`NRVec_from1(); explicit NRVec_from1(const int n); NRVec_from1(const T &a, const int n); NRVec_from1(const T *a, const int n)`
: As for `NRVec`.

`template<int SIZE> NRVec_from1(const T (&a)[SIZE]); explicit NRVec_from1(const std::list<T> l)`
: As for `NRVec`.

`NRVec_from1(const NRVec<T> &rhs)`
: Implicit conversion from the base class (shares data).

`const T& operator[](const int i) const; T& operator[](const int i)`
: Element i, 1-based.

`int iamax() const`
: 1-based index of the element of largest absolute value.

Caution: `NRVec_from1` does not redeclare `operator=(const T&)`; the implicitly declared copy assignment hides it, so `f = 0.;` does not compile for `NRVec_from1<double> f`. Use the `(value, size)` constructor or assign through a base class reference.

4.10 Examples

```
#include "la.h"
using namespace LA;

double sq(const double &a) {return a*a;}
float sq(const float &a) {return a*a;}

int main()
{
NRVec<double> x({1.,2.,3.});
NRVec<double> y(x);       // shares data with x
y *= 2.;                  // y gets its own copy, x unchanged
double d = x*y;           // scalar product = 28
NRVec<double> z = x + y*0.5;
```

```
z.axpy(-1.,x);            // z = x
std::cout << d << " " << z.norm() << std::endl;

NRVec<double> w(0.,5);    // five zeros
NRVec<int> sel({4,0,2});
w.storesubvector(sel,x);  // scatter: w[4]=x[0], w[0]=x[1], w[2]=
    x[2]
NRVec<double> g(3);
g.subvector_from(sel,w);  // gather: g[i]=w[sel[i]], i.e. g == x
std::cout << w << g;

x.call_on_me(sq);                                    // overloaded
    name: non-template overload
x.call_on_me([](const double &a){return a+1.;});   // lambda:
    template overload
x.call_on_me([](const double &a, const double &b){return a>b?a:b
    ;}, y); // binary

x.copyonwrite();          // needed before writing via operator[]
x[0] = -1.;

NRVec_from1<double> f(1.,3);
f[3] = 5.;                // last element
std::cout << f << f.iamax() << std::endl;   // prints 3

NRVec<std::complex<double> > c(x,true);     // purely imaginary
std::cout << c << complexify(x);
}
```

Sorting with the permutation returned:

```
NRVec<double> v(12);
v.randomize(1.);
NRPerm<int> p(v.size());
NRVec<double> s(v);
s.sort(0,p);              // ascending; s copied before sorting,
    v unchanged
// s[i-1] == v[p[i]-1] for i=1..n
v.printsorted(std::cout,1,true);   // descending by absolute
    value
```

4.11 Pitfalls

- `NRVec<double> x(n)` is not initialized; use `NRVec<double> x(0.,n)` or `x.clear()`. The argument order of the filling constructor is *(value, size)*.
- After `y=x` the data are shared: call `y.copyonwrite()` before `y[i]=...`. In DEBUG builds reading an element of a shared non-const vector via

`operator[]` also triggers the error; read through a const reference.

- `NRVec(const NRMat&)` and `NRVec(const NRSMat&)` share the data with the matrix (unless `MATPTR`); writing into the vector after `copyonwrite()` does not affect the matrix, but the matrix and the vector are not independent before that.
- A skeleton vector `NRVec(ptr,n,true)` never writes into the foreign memory through modifying member functions; they silently make a copy.
- `x*y` is the (conjugated) scalar product, not an element-wise product; use `x*=y` for that.
- `a-x` and `a/x` with a scalar `a` on the left do not compile (deleted); write `-x+a`.
- `resize()` without `preserve` leaves the contents undefined.
- `norm`, `normalize`, `asum`, `amax`, `amin`, `iamax`, `axpy`, `randomize`, `gemv`, `otimes` are intended for floating-point types only and are presently available for `double` and `std::complex<double>` (link error or `laerror` otherwise, by design for integer types); `max`/`min` only for real types.
- `NRVec_from1` indexing is 1-based only in `operator[]` and `iamax()`.

Chapter 5

Dense matrices: NRMat

Header: `mat.h, mat.cc, strassen.cc (sparse-related members: sparsemat.h, sparsemat.cc, sparsesmat.h, sparsesmat.cc)`

5.1 Purpose

`NRMat<T>` is the general dense rectangular matrix of the `LA` library, the workhorse for everything that is passed to BLAS level 3 and LAPACK. It is a reference counted handle to a contiguous row-major array of $n \times m$ elements of type `T`. For symmetric/Hermitian matrices stored in packed form see `NRSMat` (chapter 6), for sparse matrices chapter 7, for vectors `NRVec` (chapter 4). Decompositions, eigenvalue problems, linear solvers and matrix functions (exponential etc.) are free functions described in chapter 8. `NRMat_from1<T>` (section 5.8) is a thin derived class with 1-based `operator()`.

5.2 Storage and class-specific concepts

Layout. Element A_{ij} ($0 \le i < n$, $0 \le j < m$) is stored at offset $i\,m + j$ of one contiguous block, i.e. C row-major order (sec. 3.4). BLAS calls use `CblasRowMajor`; where Fortran routines are called, the transposition implied by the column-major convention is handled internally. Dimensions are `int`; offsets are mostly computed in `size_t`.

Reference counting. `NRMat` follows the general copy-on-demand scheme of section 3.1: the copy constructor and `operator=` share data, `|=` makes a deep copy (sec. 3.2), all modifying member functions call `copyonwrite()` themselves, and an explicit `copyonwrite()` is needed only before writing through `operator[]`, `operator()` or the raw pointer `operator T*()`. In DEBUG builds the non-const `operator[]` and `operator()` raise

laerror when the reference count is larger than one (controlled by _LA_count_check); read elements of a possibly shared matrix through a const reference.

Element types. mat.cc explicitly instantiates NRMat<T> for double, std::complex<double>, and the integer types char, short, int, long, long long and their unsigned variants. Many numerical members are only specialized (BLAS-backed) for double and complex<double> and have no generic definition, so using them with an integer type results in a link error. Table 5.1 summarizes this. Nested element types (e.g. NRMat<NRVec<double> >) are supported by the reference counting (type-nested copyonwrite, sec. 3.3), but several members are restricted to plain data (submatrix(int,int,int,int), storesubmatrix(int,int,const NRMat&), operator&).

Members	Available for
gemm, norm, axpy(T,const NRMat&), dot, transpose, diagmultl, diagmultr, operator*(const NRSMat&), diagonalset, inverse, svdinverse, absadjoint, randomize, amax, amin	double, complex<double>
operator*(const NRMat&)	double, complex<double>, int
diagonalof, orthonormalize, strassen, gemm with SparseMat	double only
nonhermiticity	complex<double> only
everything else	all instantiated types

Table 5.1: Availability of NRMat<T> members for the instantiated element types.

The MATPTR representation. configure -enable-matptr adds -DMATPTR, which switches the internal pointer from T *v to T **v: an array of row pointers into the same single contiguous block (v[0] points to the beginning). For the user the interface is the same: operator[](i) returns a T* to the beginning of row i in both cases, operator()(i,j) and operator T*() (pointer to the whole contiguous row-major block) behave identically, so code passing the matrix to BLAS is unaffected. Differences:

- NRMat(const NRVec<T>&, n, m, offset) and NRVec(const NRMat<T>&) make deep copies with MATPTR, whereas without it they share the data with the source.
- It is incompatible with CUDA (MATPTR is undefined when CUDALA is set).

MATPTR is rarely used and less tested than the default representation; the library compiles with it and passes the optimization regression test (`testoptimize`).

Strassen multiplication. Unless `NO_STRASSEN` is defined, `NRMat` declares `strassen()` and `s_cutoff()` (specialized for `double` only, in `strassen.cc`). They call the external Fortran-compatible routines `fmm` and `strassen_cutoff`, which are *not* part of LA (the test code in `t.cc` mentions linking `-lstrassen`). Strassen multiplication is never used implicitly: `operator*` and `gemm` always call BLAS `gemm`. Note that `Makefile.am` unconditionally adds `-DNO_STRASSEN`, so in the standard build these members do not exist.

Caution: The Strassen interface is *experimental*: it has not been used or tested for a long time. It is kept for possible future work, in particular if fast (sub-$O(n^3)$) matrix multiplication algorithms become practically useful.

5.3 Construction, assignment, conversion

`NRMat()`
: Empty matrix (0×0, no data, no reference counter).

`NRMat(const int n, const int m, const GPUID loc = undefined)`
: $n \times m$ matrix, elements *not* initialized.

`NRMat(const T &a, const int n, const int m); NRMat(const T &a, const int n, const int m, const GPUID loc)`
: $n \times m$ matrix with *all* elements equal to `a` (contrast with `operator=(const T&)`).

`NRMat(const T *a, const int n, const int m)`
: Deep copy of nm elements from a row-major C array.

`template<int R, int C> NRMat(const T (&a)[R][C])`
: From a two-dimensional C array; also allows brace initialization `NRMat<int> x({{1,2,3},{4,5,6}});`.

`NRMat(const Mat3<T> &rhs)`
: 3×3 matrix from the small fixed-size class `Mat3` (chapter 15).

`explicit NRMat(const int n) = delete`
: A single integer argument is forbidden (it would otherwise silently create a 3×3 matrix through `Mat3`); use `NRMat(n,m)`.

`NRMat(const NRMat &rhs)`
: Shallow copy (shares data).

`NRMat(const typename LA_traits_complex<T>::NRMat_Noncomplex_type`

`&rhs, bool imagpart = false)`

Complexifying (non-explicit) constructor, defined for `T = complex<double>`: `rhs` (`NRMat<double>`) becomes the real part (or the imaginary part if `imagpart`), the other part is zero.

`explicit NRMat(const NRMat<std::complex<T> > &rhs, bool imagpart = false)`

Decomplexifying constructor, defined for `T = double`: real part (or imaginary part if `imagpart`) of `rhs`.

`explicit NRMat(const NRSMat<T> &rhs)`

Full square matrix from a packed symmetric one ($A_{ij} = A_{ji}$; for complex types hermitian, $A_{ij} = A^*_{ji}$).

`explicit NRMat(const NRVec<T> &rhs, const int n, const int m, const int offset = 0)`

View of the vector data from position `offset` as an $n \times m$ matrix. Without MATPTR the data are *shared* (reference count of the vector is incremented); with MATPTR they are copied. A nonzero offset gives a view into the middle of the vector without copying (a deliberate feature); its condition of use is that the vector outlives the matrix, see Pitfalls. The constructor prints a reminder to `std::cout`.

`explicit NRMat(const NRPerm<int> &p, const bool direction, const bool parity = false)`

Permutation matrix: $A_{i,p(i)} = 1$ if `direction`, else $A_{p(i),i} = 1$ (permutation 1-based, matrix 0-based); if `parity`, the nonzero elements are the parity of p instead of 1.

`explicit NRMat(const WeightPermutation<int,T> &p, const bool direction)`

Weight times permutation matrix.

`explicit NRMat(const PermutationAlgebra<int,T> &p, const bool direction, const int nforce = 0)`

Sum of weighted permutation matrices of the elements of a permutation algebra; size taken from the first element unless `nforce>0`. (The header remarks that the permutation-matrix representation of S_n is reducible to only the irreps $[n]$ and $[n-1,1]$, so e.g. Young projectors cannot be represented this way.) See chapter 13.

`explicit NRMat(const SparseMat<T> &rhs); explicit NRMat(const SparseSMat<T> &rhs); explicit NRMat(const SemiSparseMat<T> &rhs)`

Dense matrix from sparse ones (chapter 7).

`explicit NRMat(const CSRMat<T> &rhs)`

Not supported: `CSRMat` is unfinished, the declaration is kept for possible future work (no definition, a link error if used).

`~NRMat()`

Decrements the reference count, frees the data when it reaches zero.

`NRMat & operator=(const NRMat &rhs)`
: Shallow assignment (sec. 3.2).

`NRMat & operator|=(const NRMat &rhs)`
: Deep copy.

`NRMat & operator=(const T &a)`
: Sets the matrix to $a\,\mathbf{1}$ (diagonal a, off-diagonal 0); the matrix must already be square (checked in DEBUG only), it is not resized.

`void identity()`
: `*this = (T)1`.

`void resize(int n, int m)`
: Change dimensions; if either is 0 the matrix is deallocated. If the data are shared, the matrix is detached and new (uninitialized) storage allocated; if the size changes, the contents are lost (not preserved); if the size is unchanged and unshared, nothing happens.

`void dealloc(void)`
: `resize(0,0)`.

`void copyonwrite(bool detachonly = false, bool deep = true)`
: Make the data unshared (sec. 3.1). With `detachonly` new storage is allocated but the data are not copied (contents undefined); `deep` controls the recursive `copyonwrite` of nested element types.

`int getcount() const`
: Current reference count (0 for an empty matrix).

`void swap(NRMat &rhs)`
: Exchange contents of two matrices in $O(1)$ (pointers, sizes, counters).

`GPUID getlocation() const; void moveto(const GPUID dest)`
: GPU support, experimental, see sec. 3.5; without `CUDALA` they return `cpu` resp. do nothing.

5.4 Element access

`T* operator[](const int i); const T* operator[](const int i) const`
: Pointer to the beginning of row i (0-based), so that `a[i][j]` is A_{ij}. DEBUG: row range check, and the shared-data check for the non-const version (the column index is not checked).

`T& operator()(const int i, const int j); const T& operator()(const int i, const int j) const`
: Element A_{ij}, 0-based; DEBUG: range checks for both indices, and shared-data check for the non-const version. CPU only (`NOT_GPU`).

`const T get_ij(const int i, const int j) const`
: Element by value (works also for data in GPU memory).

`operator T*(); operator const T*() const`
: Pointer to the contiguous row-major data block (no reference count check even in DEBUG).

`int nrows() const; int ncols() const; size_t size() const`
: Number of rows, columns, elements.

5.5 Arithmetic and BLAS/LAPACK-backed operations

Unless stated otherwise, dimension mismatches are detected only in DEBUG builds. Note that scalar addition and subtraction act on the *diagonal* only, i.e. they add $a\,\mathbf{1}$.

`NRMat & operator+=(const T &a); NRMat & operator-=(const T &a)`
: $A \leftarrow A \pm a\,\mathbf{1}$ (square matrices only).

`inline const NRMat operator+(const T &a) const; inline const NRMat operator-(const T &a) const`
: $A \pm a\,\mathbf{1}$ by value (the scalar is added to the diagonal).

`NRMat & operator*=(const T &a); inline const NRMat operator*(const T &a) const`
: Scaling by a scalar (BLAS `scal` for double/complex).

`const NRMat operator-() const`
: Unary minus.

`NRMat & operator+=(const NRMat &rhs); NRMat & operator-=(const NRMat &rhs)`
: $A \leftarrow A \pm B$ (BLAS `axpy` for double/complex).

`inline const NRMat operator+(const NRMat &rhs) const; inline const NRMat operator-(const NRMat &rhs) const`
: $A \pm B$ by value.

`NRMat & operator+=(const NRSMat<T> &rhs); NRMat & operator-=(const NRSMat<T> &rhs); inline const NRMat operator+(const NRSMat<T> &rhs) const; inline const NRMat operator-(const NRSMat<T> &rhs) const`
: Addition/subtraction of a packed symmetric matrix (square A).

`NRMat & operator^=(const NRMat<T> &rhs)`
: Hadamard (element-wise) product $A_{ij} \leftarrow A_{ij}B_{ij}$.

`void axpy(const T alpha, const NRMat &x)`
: $A \leftarrow A + \alpha X$ (BLAS `axpy`).

`const NRMat operator*(const NRMat &rhs) const`
: Matrix product AB (BLAS `gemm`; simple loops for `int`).

`void gemm(const T &beta, const NRMat &a, const char transa, const NRMat &b, const char transb, const T &alpha)`

$C \leftarrow \alpha\,\mathrm{op}(A)\,\mathrm{op}(B) + \beta C$ with $C =$ `*this`, which must already have the correct dimensions (it is not resized; if $\beta = 0$ its contents may be uninitialized). `trans`: `'n'` no transposition; for `double` any other character means transposition; for `complex<double>` `'t'` transposition and `'c'` conjugate transposition (case insensitive). Returns immediately when $\alpha = 0$ and $\beta = 1$. `*this` must not share storage with `a` or `b`.

`void gemm(const T &beta, const SparseMat<T> &a, const char transa, const NRMat &b, const char transb, const T &alpha)`

The same with a sparse first factor; `double` only (the complex version calls `laerror("not implemented yet")`), see chapter 7.

`const NRMat operator*(const NRSMat<T> &rhs) const`

AS with packed symmetric (real) or hermitian (complex) S (BLAS `dspmv`/`zhpmv` per row).

`SparseSMat<T> operator*(const SparseSMat<T> &rhs) const`

Dense times sparse, the result is sparse (chapter 7).

`const NRVec<T> operator*(const NRVec<T> &rhs) const; const NRVec<std::complex<T> > operator*(const NRVec<std::complex<T> > &rhs) const`

Matrix–vector product via `NRVec::gemv` (the second form: real matrix times complex vector).

`void gemv(const T beta, NRVec<T> &r, const char trans, const T alpha, const NRVec<T> &x) const; void gemv(const T beta, NRVec<std::complex<T> > &r, const char trans, const T alpha, const NRVec<std::complex<T> > &x) const`

$r \leftarrow \alpha\,\mathrm{op}(A)x + \beta r$; forwards to `r.gemv(beta,*this,trans,alpha,x)`, see chapter 4.

`const T dot(const NRMat &rhs) const`

$\sum_{ij} A^*_{ij} B_{ij}$ (Frobenius inner product, conjugating `*this`).

`const NRSMat<T> transposedtimes() const`

$A^\mathsf{T} A$ ($A^\dagger A$ for complex) as packed matrix.

`const NRSMat<T> timestransposed() const`

AA^T ($AA^\dagger$ for complex) as packed matrix. For complex matrices both results are hermitian (lower triangle stored).

`void diagmultl(const NRVec<T> &rhs); void diagmultr(const NRVec<T> &rhs)`

$A \leftarrow DA$ (scale rows) resp. $A \leftarrow AD$ (scale columns), $D = \mathrm{diag}($`rhs`$)$.

`const NRMat operator&(const NRMat &rhs) const; const NRMat`

`oplus(const NRMat &rhs) const`

Direct sum $A \oplus B$ (block-diagonal matrix). `oplus` returns the other operand when one is empty; `operator&` is plain-data only.

`const NRMat operator|(const NRMat<T> &rhs) const`

Direct (Kronecker) product $A \otimes B$, $(A \otimes B)_{in_B+k,\, jm_B+l} = A_{ij}B_{kl}$.

`const NRMat otimes(const NRMat &rhs, bool reversecolumns = false) const`

Kronecker product as `operator|`; with `reversecolumns` the column index is ordered the other way, $R_{in_B+k,\, lm_A+j} = A_{ij}B_{kl}$. Returns the empty operand when one is empty.

`void strassen(const T beta, const NRMat &a, const char transa, const NRMat &b, const char transb, const T alpha); void s_cutoff(const int, const int, const int, const int) const`

Only without `NO_STRASSEN`, `double` only: Strassen multiplication with `gemm` semantics (dimensions always checked) and setting of the cutoffs of the external Strassen library (section 5.2).

5.6 Other member functions

5.6.1 Rows, columns, submatrices

`const NRVec<T> row(const int i, int len = -1, int offset = 0) const`

Copy of `len` elements of row i starting at column `offset` (`len<0`: `ncols()`).

`void rowset(const NRVec<T> &r, const int i, int len = -1, int offset = 0)`

Inverse of `row`: stores `len` elements of `r` into row i from column `offset`.

`const NRVec<T> column(const int j, int l = -1) const; void columnset(const NRVec<T> &r, const int j, int l = -1)`

Copy of the first `l` (default all) elements of column j, resp. store them.

`const NRMat submatrix(const int fromrow, const int torow, const int fromcol, const int tocol) const`

Copy of the block of rows `fromrow..torow` and columns `fromcol..tocol` (inclusive). Plain data only.

`const NRMat submatrix(const NRVec<int> &rows, const NRVec<int> &cols) const`

Copy of the elements $A_{r_i c_j}$ for given (arbitrary) lists of row and column indices; indices are always range-checked.

`void storesubmatrix(const int fromrow, const int fromcol, const NRMat &rhs)`

Store `rhs` as a block with the upper left corner at (`fromrow`,`fromcol`). Plain data only.

`void storesubmatrix(const NRVec<int> &rows, const NRVec<int> &cols, const NRMat &rhs)`

Scatter: $A_{r_i c_j} \leftarrow R_{ij}$.

`const T* diagonalof(NRVec<T> &r, const bool divide = 0, bool cache = false) const`

`double` only. For a square matrix copy the diagonal into `r`, or with `divide` divide `r` element-wise by the (nonzero) diagonal elements. For a non-square matrix the diagonal of $A^\mathsf{T}A$ (squared column norms) is used instead; `r` must have `ncols()` elements. Returns pointer to `r`'s data, or NULL if `divide`. `cache` is unused. Same interface as in other matrix classes, so that it can be used by iterative solvers as a preconditioner (chapter 9).

`void diagonalset(const NRVec<T> &r)`

Set the diagonal; `r` has $\min(n, m)$ elements.

5.6.2 Transposition and conjugation

`const NRMat transpose(bool conj = false) const`

A^T, or $A^\dagger$ if `conj` (complex). `double` and `complex<double>` only.

`NRMat& transposeme(const int n = 0)`

In-place transposition of a square matrix (no conjugation); with `n>0` only the leading $n \times n$ block is transposed (this also works for a non-square matrix with $n \leq \min(\text{rows}, \text{cols})$). Dimensions are not changed, so a non-square matrix cannot be transposed in place.

`NRMat& conjugateme(); const NRMat conjugate() const`

Complex conjugation in place resp. by value; no-op for real types.

5.6.3 Reordering, permutations, rotations, scaling

`NRMat & swap_rows(); NRMat & swap_cols()`

Reverse the order of rows (columns).

`NRMat & swap_rows_cols()`

$A_{ij} \leftarrow A_{n-1-i,\,m-1-j}$ (reverse both, i.e. the whole element array).

`NRMat & swap_rows(const int i, const int j); NRMat & swap_cols(const int i, const int j)`

Exchange two rows (columns).

`NRMat & rotate_rows(const int i, const int j, const T phi); NRMat & rotate_cols(const int i, const int j, const T phi)`

Plane rotation of rows (columns) i, j: $x_i \leftarrow c\,x_i + s\,x_j$, $x_j \leftarrow c\,x_j - s\,x_i$ with $c = \cos\phi$, $s = \sin\phi$ (evaluated in type `T`).

`void scale_row(const int i, const T f); void scale_col(const int i, const T f)`

Multiply row (column) i by f.

`const NRMat permuted_rows(const NRPerm<int> &p, const bool inverse = false) const`

Row i of A becomes row $p(i)$ of the result (indices of p are 1-based); with `inverse` row i of the result is row $p(i)$ of A.

`const NRMat permuted_cols(const NRPerm<int> &p, const bool inverse = false) const; const NRMat permuted_both(const NRPerm<int> &p, const NRPerm<int> &q, const bool inverse = false) const`

The same for columns, resp. rows by p and columns by q simultaneously.

`void permuteme_rows(const CyclePerm<int> &p); void permuteme_cols(const CyclePerm<int> &p)`

In-place permutation given in cycle notation: in each cycle $(c_1\, c_2 \ldots c_L)$ row (column) c_k moves to c_{k+1} and c_L to c_1.

`void axpy(const T alpha, const NRPerm<int> &p, const bool direction)`

Add α times the permutation matrix (same convention as the constructor from `NRPerm`).

5.6.4 Reductions, norms, properties

`const typename LA_traits<T>::normtype norm(const T scalar = (T)0) const`

Frobenius norm $\|A - s\,\mathbf{1}\|_F$ (BLAS `nrm2` when $s = 0$).

`const T trace() const`

$\sum_i A_{ii}$ (square matrices).

`const T sum() const; const NRVec<T> rsum() const; const NRVec<T> csum() const`

Sum of all elements; vector of row sums (length `nrows()`, sum over the column index); vector of column sums (length `ncols()`).

`inline const T amax() const; inline const T amin() const`

The element (not its index) with the largest resp. smallest absolute value; for complex the measure is |Re| + |Im| (BLAS `i?amax` convention).

`const typename LA_traits<T>::normtype nonsymmetry() const`

$\sum_{i>j} |A_{ij} - A_{ji}|^2$ (square of the norm, no square root).

`const typename LA_traits<T>::normtype nonhermiticity() const`

`complex<double>` only: $\sum_{i \geq j} |A_{ij} - A^*_{ji}|^2$.

`NRMat<typename LA_traits<T>::normtype> abs() const`

Matrix of element-wise absolute values.

`NRMat<typename LA_traits<T>::normtype> diffabs(const NRMat &rhs) const`

Element-wise $|A_{ij}| - |B_{ij}|$.

`const bool operator==(const NRMat &rhs) const; const bool`

`operator!=(const NRMat &rhs) const`

Equality of dimensions and contents; for plain types a bitwise (`memcmp`) comparison, element-wise for nested types.

`bool issymmetric() const; inline void simplify()`

Always `false` resp. no-op; they exist for interface compatibility with the sparse matrix classes, so that generic code can use `NRMat` in their place.

5.6.5 Orthonormalization, inverse, random fill

`void orthonormalize(const bool rowcol, const NRSMat<T> *metric = NULL)`

`double` only. Modified Gram–Schmidt orthonormalization of the rows (`rowcol==true`) or columns (`false`) of the matrix in place, optionally with respect to the inner product $x^{\mathsf{T}} S y$ given by `metric`. A zero vector (or non-positive metric) raises `laerror`.

`NRMat inverse() const`

A^{-1} via `calcinverse` (LAPACK linear solve, chapter 8).

`NRMat svdinverse(const typename LA_traits<T>::normtype thr = 0) const`

Pseudo-inverse via SVD, singular values below `thr` are dropped.

`NRMat absadjoint(const typename LA_traits<T>::normtype thr = 0) const`

Adjugate (transposed cofactor) matrix computed via SVD up to the factor $\det U \det V$ (a sign, or a phase in the complex case), cf. `calcsvdinverse` in chapter 8. `thr` is not used in this mode.

`void randomize(const typename LA_traits<T>::normtype &x)`

Fill with uniform pseudorandom numbers from $[-x, x)$ (`RANDDOUBLESIGNED`); for complex, real and imaginary parts independently. (The source comment saying $[0, x]$ is incorrect.)

`void clear()`

Set all elements to zero (detaches shared data without copying).

5.6.6 Input/output

See section 3.7 for the general conventions.

`void put(int fd, bool dimensions = 1, bool transposed = false, bool orcaformat = false) const`

Binary output to a file descriptor: optionally the two dimensions (as `int`), then the elements row by row. With `transposed` the matrix is written as its transpose (dimensions swapped, element by element). With `orcaformat sizeof(T)` is written as a third `int` after the dimensions (format of the ORCA program).

`void get(int fd, bool dimensions = 1, bool transposed = false, bool orcaformat = false)`

Binary input, counterpart of `put`; with `dimensions` the matrix is resized, otherwise it must already have the right size. With `orcaformat` the stored element size is checked.

`void fprintf(FILE *f, const char *format, const int modulo) const`

Formatted output via `lawritemat` with a printf-like `format`.

`void fscanf(FILE *f, const char *format)`

Formatted input: reads the two dimensions, resizes, then the elements with the scanf-like `format`.

`void printsorted(std::ostream &s, int direction = 1, bool absvalue = false, bool stable = false, typename LA_traits<T>::normtype thr = 0, int topmax = 0) const`

Print the elements sorted by value (or absolute value), each line containing row index, column index and value; delegates to `NRVec::printsorted` (chapter 4), where `direction`, `stable`, threshold `thr` and maximum count `topmax` are described.

5.6.7 Sparse matrix interoperability

Besides the constructors, `operator*(const SparseSMat<T>&)` and `gemm` with a `SparseMat` factor listed above, `NRMat` declares

`NRMat & operator+=(const SparseMat<T> &rhs); NRMat & operator-=(const SparseMat<T> &rhs)`

Add/subtract the sparse matrix (defined in `sparsemat.h`; a symmetric `SparseMat` is unfolded, without conjugation).

5.7 Related free functions

`template<class T> const NRMat<T> operator+(const T &a, const NRMat<T> &rhs); template<class T> const NRMat<T> operator*(const T &a, const NRMat<T> &rhs)`

$a\,\mathbf{1} + A$ and aA.

`template<class T> const NRMat<T> operator-(const T &a, const NRMat<T> &rhs) = delete`

Deliberately not available (compile error); write `-A+a` for $a\,\mathbf{1} - A$.

`template<typename T> NRMat<std::complex<T> > complexify(const NRMat<T> &rhs)`

Complex matrix with real part `rhs`.

`template<typename T> NRMat<std::complex<T> > complexify(const NRMat<T> &rhsr, const NRMat<T> &rhsi)`

Complex matrix from real and imaginary parts.

`template <typename T, typename S> void NRMat_convert(NRMat<T> &a, const NRMat<S> &b)`
: Element type conversion: resizes `a` and assigns `(T)b(i,j)`.

`template <typename T> std::ostream& operator«(std::ostream &s, const NRMat<T> &x)`
: Text output: a line with the dimensions, then one line per row.

`template <typename T> std::istream& operator»(std::istream &s, NRMat<T> &x)`
: Text input in the same format (resizes `x`).

5.8 NRMat_from1: indexing from 1

`NRMat_from1<T>` is derived publicly from `NRMat<T>` and differs only in `operator()` and `get_ij`, which take 1-based indices ($1 \le i \le n$, $1 \le j \le m$); it is meant for transcribing algorithms written with Fortran-like indexing. Everything else is inherited unchanged, in particular `operator[]` remains 0-based (`a[i][j]` is element $(i{+}1, j{+}1)$) and member functions taking indices (`row`, `submatrix`, ...) use 0-based indices. Inherited members return `NRMat<T>`, which converts back implicitly.

`NRMat_from1(); NRMat_from1(const int n, const int m); NRMat_from1(const T &a, const int n, const int m); NRMat_from1(const T *a, const int n, const int m)`
: As the corresponding `NRMat` constructors.

`template<int R, int C> explicit NRMat_from1(const T (&a)[R][C])`
: From a two-dimensional C array / brace initializer.

`NRMat_from1(const NRMat<T> &rhs)`
: Implicit conversion from the base class (shallow, shares data).

`explicit NRMat_from1(const NRSMat_from1<T> &rhs)`
: Full matrix from a packed symmetric one.

`explicit NRMat_from1(const int n) = delete`
: A single integer argument is forbidden (as for `NRMat` and `Mat3`); use `NRMat_from1(n,m)`.

`inline T& operator()(const int i, const int j); inline const T& operator()(const int i, const int j) const; inline const T get_ij(const int i, const int j) const`
: 1-based element access, with the same DEBUG checks as in `NRMat`.

5.9 Examples

Products, reference counting, orthonormalization and submatrices:

```
#include "la.h"
```

```
using namespace LA;
using namespace std;

int main()
{
NRMat<double> a(4,3), b(3,5);
a.randomize(1.); b.randomize(1.);

NRMat<double> c = a*b;                  // dgemm
NRMat<double> d(3,3);                   // result must have the
    right shape
d.gemm(0., a, 't', a, 'n', 1.);         // d = a^T a
NRSMat<double> s = a.transposedtimes();// the same, packed
    symmetric
cout << (d - NRMat<double>(s)).norm() << endl;

NRMat<double> e = c;                    // shares data with c
e.copyonwrite();                        // needed: we write
    through operator()
e(0,0) = 1.;
const NRMat<double> &cc = c;            // read-only access via
    const reference
cout << cc(0,0) << " " << e(0,0) << endl;

NRMat<double> u(3,3); u.randomize(1.);
u.orthonormalize(false);                // orthonormal columns
cout << (u.transpose()*u - 1.).norm() << endl;   // "- 1."
    subtracts the unit matrix

NRMat<double> blk = c.submatrix(1,2,0,4);        // rows 1..2,
    all columns
NRVec<double> r = c.row(3);
NRMat<double> big(0., 6, 6);
big.storesubmatrix(2, 1, blk);
cout << blk << r << big.trace() << endl;
return 0;
}
```

Complex matrices, 1-based indexing, Kronecker product and direct sum:

```
NRMat_from1<int> m({{1,2,3},{4,5,6}}); // 2x3, indices from 1
const NRMat_from1<int> &mc = m;
cout << mc(2,3) << " " << mc[1][2] << endl;  // both print 6:
    operator[] stays 0-based

NRMat<double> re(2,2), im(2,2);
re.randomize(1.); im.randomize(1.);
NRMat<complex<double> > z = complexify(re, im);
NRMat<complex<double> > h(2,2);
h.gemm(0., z, 'c', z, 'n', 1.);           // h = z^dagger z
```

```
cout << h.nonhermiticity() << endl;       // 0
NRMat<double> rp(z);                      // real part (explicit
    constructor)
NRMat<double> ip(z, true);                // imaginary part

NRMat<double> k = re.otimes(im);          // Kronecker product, 4
    x4
NRMat<double> ds = re.oplus(im);          // direct sum, 4x4
    block diagonal
```

5.10 Pitfalls

Caution: Scalars mean multiples of the unit matrix: `A = 1.` makes a unit matrix (of the current, square size), `A + a` and `A -= a` change only the diagonal. In contrast `NRMat<double> A(a, n, m)` fills *all* elements with `a`. `a - A` does not compile; write `-A + a`.

Caution: `NRMat<T>(n,m)` does not initialize the elements; call `clear()` or use the fill constructor. Likewise `resize()` does not preserve the contents when the size changes.

Caution: `gemm` does not resize the target; allocate it with the correct dimensions first. The target must not share storage with an operand (e.g. `c.gemm(0.,c,'n',b,'n',1.)` is wrong); after `c = a` the call `c.gemm(...,a,...)` is fine because `gemm` first detaches `c`.

Caution: Writing through `operator[]`, `operator()` or `operator T*()` into a matrix obtained by copy construction or `=` modifies all sharing matrices unless `copyonwrite()` is called first. In DEBUG builds even *reading* through the non-const operators of a shared matrix raises an error; use a const reference.

Caution: `NRMat(const NRVec<T>&, n, m, offset)` shares the vector data (without MATPTR). With a nonzero offset, if the matrix happens to be the last owner of the data, it will `delete[]` a pointer that is not the beginning of the allocated block; the constructor therefore prints a warning. This is by design (a view without copying, avoiding the cost of a copy); the condition of use is: keep the vector alive longer than the matrix, or make an independent copy (`|=`) if the matrix has to outlive it.

Caution: Many numerical members (`gemm`, `transpose`, `norm`, ...) exist only for `double` and `complex<double>` (Table 5.1); with other element types the program fails to link.

Caution: `transposeme()` does not conjugate and works only on square matrices (or square leading blocks); use `transpose(true)` for $A^\dagger$ or for a non-square matrix.

Caution: In the non-DEBUG build no dimension checks are done in most operations (`operator+=`, `gemm`, `operator=(const T&)`, ...); mismatches lead to silent memory corruption.

Chapter 6

Packed symmetric and hermitian matrices: NRSMat

Header: `smat.h, smat.cc`

6.1 Purpose

`NRSMat<T>` stores a square symmetric (real `T`) or hermitian (complex `T`) matrix of order n in packed form, i.e. only $n(n+1)/2$ elements of one triangle are kept. It halves the memory compared to `NRMat<T>` and gives access to the packed BLAS/LAPACK routines (`dspmv`, `zhpmv`, `dspr2`, `zhpr2`, `dspev`, ...). Typical uses are Hamiltonians, overlap and density matrices, Hessians (see 10) and as input for `diagonalize()` (see 8). Results of operations which are not symmetric in general (e.g. the product of two symmetric matrices) are returned as `NRMat<T>`.

`NRSMat_from1<T>` (section 6.8) is the same class with the two-index `operator()` counting from 1.

The file `smat.cc` contains explicit instantiations for the element types `double`, `std::complex<double>` and the integer types `char`, `short`, `int`, `long`, `long long` and their unsigned variants (`float` is not instantiated). Most numerical members have BLAS-based specializations only for the two types `double` and `std::complex<double>`. Some members are defined only for these two types: `amax`, `amin`, `norm`, `dot`, `axpy`, `randomize`, `inverse` and the products with `NRMat` and `NRSMat`.

6.2 Storage and concepts

6.2.1 Packed lower triangle

The elements A_{ij} with $i \geq j$ (lower triangle) are stored row by row, i.e. the row-major packed lower triangle, which is the same memory layout as the

column-major packed upper triangle (Fortran `UPLO='U'`):

$$A_{ij} \;\to\; \mathtt{v}[k], \qquad k = \frac{i(i+1)}{2} + j \quad (0 \le j \le i < n), \tag{6.1}$$

so the order is $A_{00}, A_{10}, A_{11}, A_{20}, A_{21}, A_{22}, \ldots$ `size()` returns the length of the packed data $n(n+1)/2$, *not* n; `nrows()` and `ncols()` both return n. The packed data can be accessed linearly by `operator[]` and by the conversion to a raw pointer. The data of an `NRSMat<T>` can be viewed as an `NRVec<T>` of length $n(n+1)/2$ and vice versa without copying (see section 6.3).

`smat.h` defines the macro `NN2` (`(size_t)nn*(nn+1)/2`) which it uses internally and does not undefine; avoid using this name in your own code.

6.2.2 Index functions

The following inline function templates (namespace `LA`, template parameter = integer type of the indices) compute the packed index and return `size_t`. They are used by the class itself and are useful for code working directly with packed data (e.g. two-electron integrals in 12).

`size_t SMat_index(T i, T j)`
: Index (6.1) for $0 \le i, j < n$, either order: (i, j) and (j, i) give the same result.

`size_t SMat_index_igej(T i, T j); size_t SMat_index_ilej(T i, T j)`
: As `SMat_index`, faster, assuming $i \ge j$ resp. $i \le j$ (not checked).

`size_t SMat_index_1(T i, T j)`
: Index for indices counting from 1 ($1 \le i, j \le n$): $i(i-1)/2 + j - 1$ for $i \ge j$; the resulting packed index counts from 0.

`size_t SMat_index_1igej(T i, T j); size_t SMat_index_1ilej(T i, T j)`
: As `SMat_index_1`, assuming $i \ge j$ resp. $i \le j$.

`size_t ASMat_index(T i, T j)`
: Packed index of the strictly lower triangle (antisymmetric matrices, used by 12): $i(i-1)/2 + j$ for $i > j$, symmetric in i, j; returns `(size_t)-1` for $i = j$.

`size_t ASMat_index_1(T i, T j)`
: The same for indices counting from 1: $(i-2)(i-1)/2 + j - 1$ for $i > j$; `(size_t)-1` for $i = j$. Computed in the arithmetic of `T` (no cast to `size_t`), so it may overflow for large `int` indices.

6.2.3 Hermitian interpretation of complex matrices

For complex `T` an `NRSMat` is a *hermitian* matrix: the lower triangle is stored and the upper triangle is its complex conjugate, $A_{ij} = A^*_{ji}$ for $i < j$. For real `T` this is just the symmetric matrix. The arithmetic of the library follows this convention:

- products: `NRSMat*NRVec`, `NRSMat*NRMat`, `NRMat*NRSMat`, `NRVec*NRSMat` ($x^{\mathsf{T}}H$), `NRSMat*NRSMat`, `NRVec::gemv` (`zhpmv`, `trans`='t' gives $H^{\mathsf{T}}x = H^*x$), `rank2update` (`zhpr2`), `diagonalize()`, `linear_solve()`, `trace2()` and the other functions of chapter 8;

- conversions: `NRMat<T>(const NRSMat<T>&)` conjugates the upper triangle and `NRSMat(const NRMat<T>&)` takes the hermitian part $(A + A^\dagger)/2$;

- `fprintf()`, `sum()` and `NRMat<T>::operator+=/-=` (also `+`, `-`) with an `NRSMat` operand use the full hermitian matrix; `operator»` and `fscanf` read the full square matrix, the elements of the lower triangle (read later) are the ones stored;

- `operator«`, `permuted()`, `submatrix(const NRVec<int>&)`, `rsum()` and `csum()` are built on `operator()` and therefore use the stored element also for $i < j$ (no conjugation);

- `randomize()` produces a hermitian matrix (real diagonal); `conjugateme()`/`conjugate()` conjugate all stored elements, which is the transposition of a hermitian matrix.

Caution: Element access does *not* follow the hermitian convention: `operator()(i,j)` (both the const and the non-const version, also of `NRSMat_from1`) and `operator[]` just access the storage. (i, j) and (j, i) refer to the same stored element $A_{\max(i,j),\min(i,j)}$, and *no* complex conjugation is performed for $i < j$: `s(0,1)` returns A_{10}, not $A_{01} = A_{10}^*$. Code reading the upper triangle of a complex hermitian matrix must conjugate itself, e.g. `std::conj(s(j,i))` for $i < j$, and must write only the elements with $i \geq j$.

Planned extension. `NRSMat` might in the future be extended by a `bool` member `hermitian` which would toggle the behaviour for complex types between the hermitian and the complex symmetric interpretation (element access, conversions, products, `operator«`, `permuted()`, `submatrix()`, `rsum()`, `csum()`). The conversion `Tensor(const NRSMat&)` (chapter 11), which now marks complex matrices as hermitian (symmetry 2), would then have to respect the flag.

6.2.4 Reference counting

`NRSMat` follows the copy-on-demand scheme of section 3.1: the copy constructor and `operator=` share the data, `operator|=` copies deeply, all modifying members call `copyonwrite()` (I found no member modifying data without it, except the broken constructor from a raw pointer, see below), `resize()` detaches, and in DEBUG builds the non-const `operator[]` and

`operator()` raise an error if the data are shared. Note that the constructor `NRSMat(const NRVec<T>&, int)` and `NRVec<T>(const NRSMat<T>&)` also share the data (the reference count is common to the vector and the matrix).

GPU support (`CUDALA`) is experimental, see section 3.5; many members are CPU only (`NOT_GPU`).

6.3 Construction, assignment, conversion

`NRSMat()`
: Empty matrix (no data, no reference count).

`explicit NRSMat(const int n, const GPUID loc = undefined)`
: Order n, elements uninitialized.

`NRSMat(const T &a, const int n)`
: Order n, all $n(n+1)/2$ stored elements (i.e. also the off-diagonal ones) set to `a`.

`NRSMat(const T *a, const int n)`
: Copy of $n(n+1)/2$ packed elements (lower triangle by rows) from `a`.

`NRSMat(const NRSMat &rhs)`
: Shallow copy.

`NRSMat(const typename LA_traits_complex<T>::NRSMat_Noncomplex_type &rhs, bool imagpart = false)`
: Complex matrix from a real one: `rhs` becomes the real part (`imagpart=false`) or the imaginary part (`imagpart=true`), the other part is zero. Defined only for `T=std::complex<double>`.

`explicit NRSMat(const NRMat<T> &rhs)`
: Symmetric part $(A + A^{\mathsf{T}})/2$ of a square dense matrix (deep copy; nonsquare input is checked only in DEBUG).

`explicit NRSMat(const NRVec<T> &rhs, const int n)`
: Interprets a vector of length $n(n+1)/2$ as packed matrix of order n. *Shares* the data with `rhs` (no copy); length checked only in DEBUG.

`explicit NRSMat(const SparseMat<T> &rhs); explicit NRSMat(const SparseSMat<T> &rhs)`
: Dense from sparse, see 7.

`~NRSMat()`
: Destructor, frees the data when the last reference disappears.

`NRSMat & operator=(const NRSMat &rhs)`
: Shallow copy, see section 3.2.

`NRSMat & operator|=(const NRSMat &rhs)`
: Deep copy.

`NRSMat & operator=(const T &a)`
: Scalar matrix: zero all elements (by `memset`), set the diagonal to `a`. Dimension must be set before.

`void identity()`
: `*this = (T)1`.

`void clear()`
: Set all elements to zero (keeps the dimension).

`void resize(const int n)`
: Change the order; data are not preserved and new elements are uninitialized. A shared matrix is detached; an unshared one of the same order is left untouched; `n=0` deallocates.

`void dealloc()`
: `resize(0)`.

`void copyonwrite(bool detachonly=false, bool deep=true)`
: Make the data private, see section 3.1.

`void swap(NRSMat &rhs)`
: Exchange contents without copying.

`void moveto(const GPUID dest); GPUID getlocation() const`
: GPU location (experimental), see section 3.5.

`int getcount() const`
: Current reference count (0 for an empty matrix).

Conversions in other classes: `explicit NRMat<T>(const NRSMat<T>&)` expands to a full square matrix (hermitian for complex types, see section 6.2.3); `explicit NRVec<T>(const NRSMat<T>&)` views the packed data as a vector (shared data). `NRMat<T>::transposedtimes()` and `timestransposed()` return $A^\dagger A$ and $AA^\dagger$ as `NRSMat` (see 5).

6.4 Element access

`T& operator()(const int i, const int j); const T& operator()(const int i, const int j) const`
: Element (i, j), $0 \le i, j < n$, either order; (i, j) and (j, i) refer to the same storage; plain storage access, no complex conjugation for $i < j$ (see section 6.2.3). Range checked in DEBUG.

`T& operator[](const size_t ij); const T& operator[](const size_t ij) const`
: Packed element number `ij`, $0 \le$ `ij` $< n(n+1)/2$, see (6.1).

`operator T*(); operator const T*() const`
: Pointer to the packed data.

`int nrows() const; int ncols() const`
: Order n.

`size_t size() const`
: Number of stored elements $n(n+1)/2$.

`bool transp(const int i, const int j) const`
: Returns `i>j`; helper for code that uses packed storage for a matrix one triangle of which is redundant but not equal to the other (e.g. antisymmetric), to know whether the stored element must be transformed.

`bool issymmetric() const`
: Always true (interface compatibility with sparse matrices).

`void simplify()`
: Does nothing (interface compatibility with sparse matrices).

Caution: The non-const `operator()` and `operator[]` do not call `copyonwrite()`. Call `x.copyonwrite()` before writing through them; in DEBUG builds they raise an error for a shared matrix even when only reading, so read elements of a possibly shared matrix through a const reference.

6.5 Arithmetic and BLAS/LAPACK-backed operations

All in-place operators call `copyonwrite()`. The binary operators returning `NRSMat` are generated from the in-place ones.

`NRSMat & operator*=(const T &a)`
: Multiply all elements by `a` (`dscal`/`zscal`).

`NRSMat & operator+=(const T &a); NRSMat & operator-=(const T &a)`
: Add/subtract $a \cdot 1$, i.e. `a` on the diagonal only.

`NRSMat & operator+=(const NRSMat &rhs); NRSMat & operator-=(const NRSMat &rhs)`
: Elementwise sum/difference (`daxpy`/`zaxpy`).

`const NRSMat operator-() const`
: Unary minus.

`const NRSMat operator*(const T &a) const; const NRSMat operator+(const T &a) const; const NRSMat operator-(const T &a) const`
: As the in-place versions (`+`/`-` act on the diagonal).

`const NRSMat operator+(const NRSMat &rhs) const; const NRSMat operator-(const NRSMat &rhs) const`
: Sum, difference.

`const NRMat<T> operator+(const NRMat<T> &rhs) const; const NRMat<T>`

`operator-(const NRMat<T> &rhs) const`
: Sum/difference with a dense matrix, dense result.

`const NRMat<T> operator*(const NRSMat &rhs) const`
: Matrix product (not symmetric in general). Real: dedicated BLAS-1/2 code; complex: `*this * NRMat(rhs)`, see section 6.2.3.

`const NRMat<T> operator*(const NRMat<T> &rhs) const`
: SA by `dspmv`/`zhpmv` column by column.

`const NRVec<T> operator*(const NRVec<T> &rhs) const`
: Sx (`dspmv`/`zhpmv`).

`const NRVec<std::complex<T> > operator*(const NRVec<std::complex<T> > &rhs) const`
: Real symmetric matrix times complex vector (real and imaginary parts separately).

`void gemv(const T beta, NRVec<T> &r, const char trans, const T alpha, const NRVec<T> &x) const`
: $r \leftarrow \beta r + \alpha S x$, forwards to `r.gemv(beta,*this,trans,alpha,x)`; `trans` is ignored.

`void gemv(const T beta, NRVec<std::complex<T> > &r, const char trans, const T alpha, const NRVec<std::complex<T> > &x) const`
: The same, real matrix and complex vectors.

`void axpy(const T alpha, const NRSMat &x)`
: $S \leftarrow S + \alpha x$. For hermitian complex matrices the result is hermitian only for real α (not checked).

`void rank2update(const T alpha, const NRVec<T> &x, const NRVec<T> &y)`
: Rank-2 update $S \leftarrow S + \alpha x y^\dagger + \bar{\alpha} y x^\dagger$ (real: $\alpha(xy^\mathsf{T} + yx^\mathsf{T})$); `dspr2`/`zhpr2` for `double`/`complex<double>`, generic loop otherwise. A symmetric rank-1 update $S \leftarrow S + \beta x x^\dagger$ (real β) is obtained as `rank2update(beta/2,x,x)`; there is no separate rank-1 member.

`const T dot(const NRSMat &rhs) const`
: $\sum_{i \geq j} \bar{A}_{ij} B_{ij}$, i.e. the dot product of the *packed* vectors (off-diagonal elements counted once; `ddot`/`zdotc`).

`const T dot(const NRVec<T> &rhs) const`
: Dot product of the packed data with a vector of length $n(n+1)/2$.

`const typename LA_traits<T>::normtype norm(const T scalar = (T)0) const`
: Euclidean norm of the packed data of $S - \texttt{scalar} \cdot 1$; off-diagonal elements are counted *once*, so this is not the Frobenius norm of the full matrix.

`const T trace() const`
: Trace.

`const T sum(bool onlytriangle=false) const`
: Sum of all elements of the full matrix, or of the stored triangle only.

`const NRVec<T> rsum() const; const NRVec<T> csum() const`
: Row sums and column sums of the full matrix (equal for real, complex conjugate of each other for hermitian matrices).

`inline const T amax() const; inline const T amin() const`
: Stored element with the largest/smallest absolute value (complex: $|\Re| + |\Im|$); `double` and `complex<double>` only.

`const T* diagonalof(NRVec<T> &r, const bool divide = 0, bool cache = false) const`
: `divide=false`: store the diagonal into `r` and return `&r[0]`; `divide=true`: divide `r` elementwise by the nonzero diagonal elements and return `NULL`. `cache` is unused. Interface used by iterative solvers (9).

`NRSMat inverse()`
: Inverse via `NRMat<T>(*this).inverse()`, symmetrized back. `double` and `complex<double>` only.

`NRSMat& conjugateme(); const NRSMat conjugate() const`
: Complex conjugation of all stored elements (no-op for real types).

`NRSMat<typename LA_traits<T>::normtype> abs() const`
: Elementwise absolute values.

`NRSMat<typename LA_traits<T>::normtype> diffabs(const NRSMat &rhs) const`
: Elementwise $|A_{ij}| - |B_{ij}|$.

`void randomize(const typename LA_traits<T>::normtype &x)`
: Uniformly distributed elements in $(-x, x)$; complex: real and imaginary parts independent, imaginary part of the diagonal set to zero. `double` and `complex<double>` only.

`const bool operator==(const NRSMat &rhs) const; const bool operator!=(const NRSMat &rhs) const`
: Exact elementwise comparison.

6.6 Other member functions

6.6.1 Submatrices and permutations

`const NRSMat submatrix(const int from, const int to) const`
: Diagonal block with rows/columns `from`...`to` (inclusive).

`const NRSMat submatrix(const NRVec<int> &selection) const`
: Matrix $B_{ij} = A_{s_i s_j}$ for the index list $s =$ `selection`.

`void storesubmatrix(const int from, const NRSMat &rhs)`
: Store `rhs` as diagonal block starting at `from`.

`void storesubmatrix(const NRVec<int> &selection, const NRSMat &rhs)`
$A_{s_i s_j} = B_{ij}$.

`const NRSMat permuted(const NRPerm<int> &p, const bool inverse=false) const`
Symmetric permutation of rows and columns by `p` (indices from 1, see 13): `inverse=false` gives $B_{p_i p_j} = A_{ij}$, `inverse=true` gives $B_{ij} = A_{p_i p_j}$.

6.6.2 Input/output

See section 3.7 for the general conventions.

`void put(int fd, bool dimensions = 1, bool transp = 0, bool orcaformat=false) const`
Binary output: if `dimensions`, n is written twice (as for a square `NRMat`), followed by `sizeof(T)` if `orcaformat`; then the $n(n+1)/2$ packed elements. `transp` is ignored.

`void get(int fd, bool dimensions = 1, bool transp = 0, bool orcaformat=false)`
Binary input matching `put`; with `dimensions=false` the current order is used.

`void fprintf(FILE *f, const char *format, const int modulo) const`
Formatted output of the full square matrix by `lawritemat` (as `NRMat::fprintf`, see 5).

`void fscanf(FILE *f, const char *format)`
Reads the dimensions (must be equal) and then the *whole square* matrix row by row; this is intentional, the data are assumed symmetric. Since (i, j) and (j, i) share the storage, the element read later, i.e. the lower triangle one, wins.

`void printsorted(std::ostream &s, int direction=1, bool absvalue=false, bool stable=false, typename LA_traits<T>::normtype thr=0, int topmax=0) const`
Print the stored elements sorted with their indices, see `NRVec::printsorted` in 4.

6.7 Related free functions

`std::ostream& operator«(std::ostream &s, const NRSMat<T> &x)`
Text output: `n n`, then the full square matrix row by row.

`std::istream& operator»(std::istream &s, NRSMat<T> &x)`
Text input of the same format; the full square is read, the lower triangle element wins (as in `fscanf`). Error if the dimensions differ.

`const NRSMat<T> operator+(const T &a, const NRSMat<T> &rhs); const`

`NRSMat<T> operator*(const T &a, const NRSMat<T> &rhs)`
Scalar from the left: $a \cdot 1 + S$, aS.

`const NRSMat<T> operator-(const T &a, const NRSMat<T> &rhs) = delete`
Deliberately not available (compile error); write `-S+a` for $a \cdot 1 - S$.

`NRSMat<std::complex<T> > complexify(const NRSMat<T> &rhs)`
Complex copy of a real matrix.

`void NRSMat_convert(NRSMat<T> &a, const NRSMat<S> &b)`
Element type conversion (`a` is resized, elements converted by a cast).

Functions in other headers taking `NRSMat` (described in 8): `diagonalize()` (`dspev`/`zhpev`, also generalized problem), `linear_solve()`, `trace2()`, `twoside_transform()` ($C^\dagger SC$), `matrixfunction()`, `sqrt()`, `sqrtinv()`, `log()`, `realpart()`, `imagpart()`. `NRMat<T>` has `operator+=`, `operator-=`, `operator+`, `operator-` and `operator*` with an `NRSMat<T>` right operand, and `NRVec<T>` has `gemv` with an `NRSMat` and `operator*(const NRSMat<T>&)` (vector from the left); see 5 and 4.

6.8 NRSMat_from1

`NRSMat_from1<T>` is derived from `NRSMat<T>` and differs only in `operator()`, whose indices run from 1 to n (`SMat_index_1`). `operator[]` and everything else are inherited and still count from 0; arithmetic operators return `NRSMat<T>`, which converts back implicitly.

`NRSMat_from1(); explicit NRSMat_from1(const int n);`
`NRSMat_from1(const T &a, const int n); NRSMat_from1(const T *a, const int n)`
As for `NRSMat` (the last one inherits the bug of the base constructor).

`NRSMat_from1(const NRSMat<T> &rhs)`
Implicit conversion from the base class (shallow).

`explicit NRSMat_from1(const NRMat<T> &rhs); explicit NRSMat_from1(const NRVec<T> &rhs, const int n)`
As for `NRSMat`.

`const T& operator()(const int i, const int j) const; T& operator()(const int i, const int j)`
Element (i, j), $1 \le i, j \le n$. Range checked in DEBUG; unlike the base class the non-const version does not check the reference count. Plain storage access as in the base class (no conjugation, section 6.2.3).

6.9 Examples

```
#include "la.h"
using namespace LA;

int main()
{
const int n=4;
NRSMat<double> s(n);
for(int i=0; i<n; ++i) for(int j=0; j<=i; ++j) s(i,j) = 1./(1.+i+
    j); //fill lower triangle
NRVec<double> x(1.,n);
NRVec<double> y = s*x;                    //dspmv
NRMat<double> full(s);                    //full square copy
NRSMat<double> t = s;                     //shares data with s
t.copyonwrite(); t(0,3) = 5.;             //same storage as t(3,0)
    ; s unchanged
t.rank2update(0.5,x,y);                   //t += 0.5*(x y^T + y x^
    T)
const NRSMat<double> &sc = s;
double sum=0; for(size_t k=0; k<sc.size(); ++k) sum += sc[k]; //
    packed loop
NRVec<double> w(n); NRMat<double> v(n,n);
NRSMat<double> work(s);                   //diagonalize() detaches
     work itself
diagonalize(work,w,&v);                   //eigenvalues/vectors,
    work is overwritten
std::cout << s << w;
}
```

6.10 Pitfalls

- `size()` is $n(n+1)/2$, use `nrows()` for the order.
- `dot()` and `norm()` work on the packed triangle; for the Frobenius quantities of the full matrix use e.g. `NRMat<T>(s).norm()` or `trace2()`.
- For complex types read section 6.2.3 carefully.
- `NRSMat(const NRVec<T>&,int)` and `NRVec<T>(const NRSMat<T>&)` share data with their argument.
- `NRSMat(const NRMat<T>&)` silently symmetrizes (takes the hermitian part for complex types), it does not check symmetry.
- Complex matrices: `operator()(i,j)` with $i < j$ does not conjugate (plain storage access, section 6.2.3).

- Products of two symmetric matrices return `NRMat<T>`; converting back with `NRSMat(...)` symmetrizes.
- `operator=(const T&)` produces a scalar (diagonal) matrix, whereas the constructor `NRSMat(a,n)` fills all elements.

Chapter 7

Sparse matrices

Header: `sparsemat.h, sparsemat.cc, sparsesmat.h, sparsesmat.cc`

7.1 Overview

LA offers two independent sparse matrix classes with different storage schemes, plus a small helper class:

- `SparseMat<T>` (section 7.2): an *unordered linked list* of (row, column, value) triplets. Appending elements is cheap, duplicates are allowed and summed, sorting is done on demand. A flag marks symmetric matrices of which only one triangle is stored. Products with dense vectors and matrices are its main purpose (e.g. CI-type Hamiltonians in Davidson/Lanczos solvers, 9, and the matrix exponential).
- `SparseSMat<T>` (section 7.4): an array of rows, each row a `std::map` from column index to value. It allows efficient random access and accumulation and sparse×sparse products, and offers a pivoted sparse Cholesky decomposition. It was written for exponentiating symmetric matrices (e.g. the unitary complex symmetric $\exp(\mathrm{i}H)$ with real symmetric H), but despite its name it is a general (also nonsymmetric and rectangular) sparse matrix; symmetry (hermiticity for complex types) is only a convention of some of its operations.
- `SemiSparseMat<T>` (section 7.3): a dense diagonal plus a `SparseMat` off-diagonal part, just enough to be used in iterative eigensolvers.

`CSRMat<T>` (section 7.5) is unfinished.

Common definitions in `sparsemat.h`:

`typedef unsigned int SPMatindex; typedef int SPMatindexdiff`
: Index type and index difference type. All row and column indices count from 0.

`const double SPARSEEPSILON=1e-15`

: Threshold: elements with absolute value below (or equal to) it are dropped by `simplify()` and by most constructors and accumulating operations.

`template<typename T> struct matel {T elem; SPMatindex row; SPMatindex col; matel *next;}`

: List element of `SparseMat`; also the value type returned by the iterators of both classes.

Both classes are explicitly instantiated for `double` and `std::complex<double>`. Several operations are implemented for `double` only. `SparseMat` nowhere conjugates when it mirrors elements of a symmetric matrix (source comments: "not OK for complex hermitean"), so its complex symmetric matrices are complex symmetric, not hermitian. `SparseSMat` stores both triangles and its mirroring operations follow the hermitian convention of `NRSMat` (section 6.2.3). Neither class has GPU support.

The header `sparsemat_traits.h` is an obsolete leftover: it specializes a template `NRMat_traits` which no longer exists, is not included by any file and is not installed; it is kept in the source tree.

7.2 SparseMat

Header: `sparsemat.h, sparsemat.cc`

7.2.1 Storage

The nonzero elements form a singly linked list of `matel<T>` (pointer `list`); new elements are prepended. The list is in no particular order and may contain several entries with the same (i, j), which represent their sum; `simplify()` merges such duplicates, removes elements with $|a_{ij}| \leq$ `SPARSEEPSILON` and leaves the list sorted by rows. Operations needing an ordering (`simplify`, sparse products, `dot`) build private arrays of pointers sorted by rows or by columns (member `sort()`); these arrays and the cached element count `nonzero` are kept until the next operation which calls the private `unsort()`: `add()` (and everything built on it), `simplify`, `resize`, `transposeme`, the permutation members, `get`, assignment, and `copyonwrite()` of a shared matrix. Writing *indices* through the iterator or `getlist()` after a sort is not detected.

Symmetric flag. If `issymmetric()` is true, the matrix is square and each stored off-diagonal element a_{ij} represents both a_{ij} and a_{ji} (no conjugation); diagonal elements are stored once. Which triangle is used is arbitrary: the constructor from `NRSMat` stores $i \geq j$, `simplify()` moves all elements to $i \leq j$, and user code may mix them, but it must not store both a_{ij} and a_{ji} unless their sum is meant. `setsymmetric()` only sets the

flag, `defineunsymmetric()` only clears it (the data are then interpreted as a triangular matrix), and `setunsymmetric()` unfolds the matrix by adding the mirror images of the off-diagonal elements. Operations not supporting symmetric storage (sparse×sparse products, `sort()`) unfold it themselves or raise an error.

7.2.2 Reference counting

The list is reference counted: the copy constructor, `operator=` and `transpose()` of a symmetric matrix share it (sorted arrays are never shared). There is *no* `operator|=`; a deep copy is obtained by `SparseMat<T> b(a); b.copyonwrite();`. `copyonwrite()` copies the list (in reversed order) when it is shared.

Most modifying members call `copyonwrite()`: `simplify`, `operator+=/-=` with a matrix, `addtriangle`, `join`, `axpy`, `operator*=`, `transposeme`, `setunsymmetric`, `incsize`, `addsafe`, the permutation members, `gemm`, `get`; `operator=(const T&)`, `resize` and `gemm` with `beta=0` detach by reallocation. Exceptions:

- `add()` is a minimal inline function: no `copyonwrite()`, no range check (it only invalidates the sorted arrays and the cached `nonzero` if they exist). By design, user code must call `copyonwrite()` before `add()` on a possibly shared matrix (as done in the examples in `t.cc`). Adding to a shared list does not corrupt the other copies (elements are prepended), but the added elements are leaked or, after a later `copyonwrite()`, silently included only in this copy.
- `operator+=(const T&)` and `operator-=(const T&)` call `add()` before the `copyonwrite()` inside `simplify()` (result correct, memory leaked when shared).
- The iterator and `getlist()` give writable access to the list elements without `copyonwrite()`.
- `norm()` is const but calls `simplify()` (which detaches a shared list).

7.2.3 Construction, assignment, conversion

`SparseMat()`
: Undefined empty matrix (0×0, no reference count).

`SparseMat(const SPMatindex n, const SPMatindex m)`
: Defined empty (all zero) $n \times m$ matrix, not symmetric.

`SparseMat(const SparseMat &rhs)`
: Shallow copy; an empty matrix gets its own reference count.

`explicit SparseMat(const NRMat<T> &rhs)`
: From a dense matrix, elements with $|a_{ij}| >$ `SPARSEEPSILON`.

`explicit SparseMat(const NRSMat<T> &rhs)`
: From a packed symmetric matrix; symmetric flag set, lower triangle stored.

`~SparseMat()`
: Destructor.

`SparseMat & operator=(const SparseMat &rhs)`
: Shallow copy (also copies the symmetric flag).

`SparseMat & operator=(const T &a)`
: Scalar matrix $a \cdot 1$; the matrix must be square and dimensioned, old contents are discarded.

`void identity()`
: `*this = (T)1`.

`void resize(const SPMatindex n, const SPMatindex m)`
: New empty $n \times m$ matrix; data are discarded (detaches if shared), the symmetric flag is cleared.

`void dealloc()`
: `resize(0,0)`.

`void incsize(const SPMatindex n, const SPMatindex m)`
: Enlarge the dimensions by n and m keeping the elements (used for direct sums).

`void copyonwrite(bool detachonly=false, bool deep=true)`
: Make the list private, see section 3.1.

`void clear()`
: Zero matrix of the same dimensions: as `resize(nrows(),ncols())`, but the symmetric flag is kept (detaches if shared).

`void swap(SparseMat &rhs)`
: Exchange contents without copying.

`int getcount() const`
: Reference count (0 if undefined).

Dense from sparse: `explicit NRMat<T>(const SparseMat<T>&)` (symmetric matrices are unfolded, duplicates summed), `explicit NRSMat<T>(const SparseMat<T>&)` (requires the symmetric flag, duplicates summed), `explicit NRVec<T>(const SparseMat<T>&)` (for a $1 \times m$ or $n \times 1$ matrix).

7.2.4 Element insertion and access

`inline void add(const SPMatindex n, const SPMatindex m, const T`

elem)

Prepend element (n, m); it is summed with existing entries at the same position. No checks, no `copyonwrite()`, see section 7.2.2. The matrix must have been dimensioned (constructor with dimensions or `resize`).

`void addsafe(const SPMatindex n, const SPMatindex m, const T elem)`

As `add`, but ignores $|\texttt{elem}| \leq \texttt{SPARSEEPSILON}$, calls `copyonwrite()` and creates the reference count of an undefined matrix (range check only with the lowercase `debug` macro).

`void simplify(const double sparseepsilon=SPARSEEPSILON)`

Sum duplicates, remove elements with $|a_{ij}| \leq$ `sparseepsilon`, sort by rows; symmetric matrices are first moved to the upper triangle ($i \leq j$).

`unsigned int length() const`

Number of list elements (cached until the next modification, see section 7.2.1).

`unsigned int sort(int type) const`

Build the internal array sorted by rows (`type=0`) or columns (`type=1`) and return the number of elements. Logically const, but unfolds a symmetric matrix (`setunsymmetric`). Normally called internally only.

`inline matel<T> *getlist() const; void setlist(matel<T> *l)`

Raw access to the list head (no reference count handling; `setlist` does not free the old list).

`iterator begin() const; iterator end() const`

Forward iterator over the raw list (unsorted, duplicates possible, one triangle for symmetric matrices). `*it` is `matel<T>&`, `it->row`, `it->col`, `it->elem` are writable; `++` (pre- and postfix), `==`, `!=`.

`inline SPMatindex nrows() const; inline SPMatindex ncols() const`

Dimensions.

`inline bool issymmetric() const`

Symmetric flag.

`inline void setsymmetric(); inline void defineunsymmetric(); void setunsymmetric()`

Set the flag (square only) / clear the flag / unfold the stored triangle, see section 7.2.1.

7.2.5 Arithmetic

Binary operators with a matrix operand must not combine a symmetric left operand with a general right one.

`SparseMat & operator*=(const T &a)`

Scale all elements (`a=0` calls `clear()`).

`SparseMat & operator+=(const T &a); SparseMat & operator-=(const T &a)`

Add/subtract $a \cdot 1$ (square matrices), then `simplify()`.

`SparseMat & operator+=(const SparseMat &rhs); SparseMat & operator-=(const SparseMat &rhs)`

Append `rhs` (unfolded if `rhs` is symmetric and `*this` is not; error for symmetric `*this` and general `rhs`), then `simplify()`.

`const SparseMat operator+(const T &rhs) const; const SparseMat operator-(const T &rhs) const; const SparseMat operator*(const T &rhs) const`

Out-of-place versions.

`const SparseMat operator+(const SparseMat &rhs) const; const SparseMat operator-(const SparseMat &rhs) const`

Out-of-place versions.

`SparseMat & addtriangle(const SparseMat &rhs, const bool lower, const char sign)`

Append the elements of `rhs` with $i \le j$ (`lower=true`) or $i \ge j$ (`lower=false`), all of them if `rhs` is symmetric; `sign` is `'+'` or `'-'`. No `simplify()`.

`SparseMat & join(SparseMat &rhs)`

Fast `+=`: moves the list of `rhs` (must be unshared, same dimensions and symmetry) to the end of this list; `rhs` is left empty. No `simplify()`.

`void axpy(const T alpha, const SparseMat &x, const bool transp=0)`

$A \leftarrow A + \alpha x$ or $A + \alpha x^{\mathsf{T}}$; no `simplify()` (except for $\alpha = \pm 1$, which uses `+=`/`-=`).

`const T trace() const`

Trace.

`const typename LA_traits<T>::normtype norm(const T scalar=(T)0) const`

Frobenius norm of $A -$ `scalar` $\cdot 1$ of the full matrix (symmetric off-diagonal elements counted twice). Calls `simplify()`.

`const T dot(const SparseMat &rhs) const`

$\sum_{ij} a_{ij} b_{ij}$ without conjugation (also for complex types), general (not symmetric) matrices only; merge of the two lists sorted by rows.

`const T dot(const NRMat<T> &rhs) const`

$\sum_{ij} a_{ij} b_{ij}$ with a dense matrix; `double` only.

`const T* diagonalof(NRVec<T> &r, const bool divide=0, bool`

`cache=false) const`

Square matrix: diagonal into `r` (`divide=false`, returns `&r[0]`) or divide `r` by the nonzero diagonal elements (`divide=true`, returns `NULL`). Rectangular matrix: the diagonal of $A^\mathsf{T}A$ (assuming a simplified matrix), for preconditioning. `cache` is unused.

7.2.6 Products

`inline const NRVec<T> operator*(const NRVec<T> &rhs) const`

Ax.

`void gemv(const T beta, NRVec<T> &r, const char trans, const T alpha, const NRVec<T> &x, bool treat_as_symmetric=false) const`

$r \leftarrow \beta r + \alpha\,\mathrm{op}(A)x$; forwards to `NRVec<T>::gemv`. Any `trans` other than `'n'` means transposition (no conjugation). With `treat_as_symmetric` a general matrix is used as if it had the symmetric flag. The sizes of `x` and (in DEBUG) of the result are checked.

`inline const NRMat<T> operator*(const NRMat<T> &rhs) const`

AB via `NRMat<T>::gemm` with a sparse first operand; `double` only (`complex<double>` raises "not implemented yet").

`const SparseMat operator*(const SparseMat &rhs) const`

Sparse product of two general matrices (error if one is symmetric); the result is simplified.

`void gemm(const T beta, const SparseMat &a, const char transa, const SparseMat &b, const char transb, const T alpha)`

$C \leftarrow \beta C + \alpha\,\mathrm{op}(A)\,\mathrm{op}(B)$ for general `a`, `b`; if `*this` is symmetric (possible only with $\beta \neq 0$) only elements with $i \leq j$ are added. Result simplified. `a` and `b` must not be `*this` (checked in DEBUG).

Related functions in other classes: `NRVec<T>::gemv(const T beta, const SparseMat<T> &a, const char trans, const T alpha, const NRVec &x, const bool treat_as_symmetric=false)`, `NRVec<T>::operator*(const SparseMat<T>&)` ($x^\mathsf{T}A$), `NRMat<T>::gemm(const T &beta, const SparseMat<T> &a, const char transa, const NRMat &b, const char transb, const T &alpha)` (`double` only) and `NRMat<T>::operator+=`, `operator-=(const SparseMat<T>&)`.

7.2.7 Transposition, permutations, direct sums

`void transposeme(); const SparseMat transpose() const`

Transpose in place / out of place (no conjugation); trivial for symmetric matrices.

`void permuteindices(const NRVec<SPMatindex> &p)`

Square matrix: $(i, j) \to (p_i, p_j)$, indices from 0.

`void permuterows(const NRVec<SPMatindex> &p); void permutecolumns(const NRVec<SPMatindex> &p)`

$i \to p_i$ resp. $j \to p_j$ (for symmetric matrices both indices are permuted), indices from 0.

`void permuteme_rows(const NRPerm<int> &p, const bool inverse=false); void permuteme_cols(const NRPerm<int> &p, const bool inverse=false)`

The same with a permutation object (13), optionally inverted. The matrix indices count from 0, the permutation (as always for `NRPerm`) from 1: element (i, j) moves to row $p_{i+1} - 1$ resp. column $p_{j+1} - 1$ (for symmetric matrices both indices are permuted), the same direction as `NRMat::permuted_rows/permuted_cols`.

`void permuteme_both(const NRPerm<int> &p, const NRPerm<int> &q, const bool inverse=false)`

Rows by `p`, columns by `q` (same index convention).

`const SparseMat permuted_rows(const NRPerm<int> &p, const bool inverse=false) const; const SparseMat permuted_cols(const NRPerm<int> &p, const bool inverse=false) const; const SparseMat permuted_both(const NRPerm<int> &p, const NRPerm<int> &q, const bool inverse=false) const`

Out-of-place versions.

`SparseMat & oplusequal(const SparseMat &rhs); const SparseMat oplus(const SparseMat &rhs) const`

Direct sum $A \oplus B$ (block diagonal); both must have the same symmetry.

`SparseMat & oplusequal(const NRMat<T> &rhs); const SparseMat oplus(const NRMat<T> &rhs) const`

Direct sum with a dense matrix (general `*this` only).

`SparseMat & oplusequal(const NRSMat<T> &rhs); const SparseMat oplus(const NRSMat<T> &rhs) const`

Direct sum with a packed symmetric matrix (symmetric `*this` only; the lower triangle of `rhs` is added).

`const SparseMat otimes(const SparseMat &rhs) const; const SparseMat otimes(const NRMat<T> &rhs) const; const SparseMat otimes(const NRSMat<T> &rhs) const`

Direct (Kronecker) product: declared, but not implemented anywhere (link error).

7.2.8 Input/output

`void put(int fd, bool dimensions=1, bool transposed=false, bool`

`orcaformat=false) const`

Binary output: if `dimensions`, n, m (`SPMatindex`), the symmetric flag and the cached element count (`int`); then for each list element row, column and the value (written by `LA_traits<T>::put`, so nested types work); terminated by the pair $(-1, -1)$. `transposed` swaps rows and columns.

`void get(int fd, bool dimensions=1, bool transposed=false, bool orcaformat=false)`

Binary input matching `put`. With `dimensions=false` the read list replaces the current one (which is freed, or detached if shared); the dimensions are kept.

The argument `orcaformat` exists for the uniform `LA_traits` interface (section 3.7), so that e.g. `NRVec<SparseMat<T>>` can be written and read; the ORCA format is defined for dense data only and `orcaformat=true` raises `laerror`. The same holds for `SparseSMat`.

7.2.9 Related free functions

`std::ostream& operator<<(std::ostream &s, const SparseMat<T> &x)`

Text output: `n m`, then lines `row col value` in list order, terminated by `-1 -1`. The symmetric flag is not written.

`std::istream& operator>>(std::istream &s, SparseMat<T> &x)`

Text input of the same format; the result is a general (not symmetric) matrix.

`inline const SparseMat<T> commutator(const SparseMat<T> &x, const SparseMat<T> &y, const bool trx=0, const bool tryy=0)`

$[\mathrm{op}(x), \mathrm{op}(y)] = xy - yx$, via two `gemm` calls (general matrices only).

`inline const SparseMat<T> anticommutator(const SparseMat<T> &x, const SparseMat<T> &y, const bool trx=0, const bool tryy=0)`

$xy + yx$.

In 8: `realpart()` and `imagpart()` of a `SparseMat<complex<double>>`. The generic matrix functions in `matexp.h` (e.g. `exp()`) work with `SparseMat`, and the iterative solvers of 9 use its `gemv`, `diagonalof`, `issymmetric`, `nrows` interface.

7.3 SemiSparseMat

Header: `sparsemat.h`

A rudimentary class ("implemented just enough to call davidson etc. on it") holding a dense diagonal and a sparse off-diagonal part, both public members. If `diagonal` is nonempty, the diagonal elements are assumed to be stored *only* there. It is not reference counted by itself; its members are.

`NRVec<T> diagonal; SparseMat<T> offdiagonal`
: The two parts (public data members).

`SemiSparseMat(); SemiSparseMat(const SPMatindex n, const SPMatindex m)`
: Empty object / $n \times m$; for $n = m$ the diagonal vector is allocated and zeroed (zero matrix).

`SPMatindex nrows() const; SPMatindex ncols() const; SPMatindex issymmetric() const; void setsymmetric()`
: Forwarded to `offdiagonal`.

`void add(const SPMatindex n, const SPMatindex m, const T elem)`
: Diagonal elements go to `diagonal` (if allocated), others to `offdiagonal.add()`; no `copyonwrite()`.

`void gemv(const T beta, NRVec<T> &r, const char trans, const T alpha, const NRVec<T> &x, bool treat_as_symmetric=false) const; NRVec<T> operator*(const NRVec<T> &rhs) const`
: Matrix-vector product.

`const T* diagonalof(NRVec<T> &x, const bool divide=0, bool cache=false) const`
: As for `SparseMat`, using `diagonal` if allocated.

`void clear(); void copyonwrite(); void resize(const SPMatindex n, const SPMatindex m)`
: Applied to both parts (`resize` gives a zero matrix and allocates the diagonal only for $n = m$).

`std::ostream& operator«(std::ostream &s, const SemiSparseMat<T> &x)`
: Prints both parts.

`explicit NRMat<T>(const SemiSparseMat<T> &rhs)`
: Dense conversion.

7.4 SparseSMat

Header: `sparsesmat.h, sparsesmat.cc`

7.4.1 Storage and concepts

The data member `v` is an array of `nrows()` pointers to `std::map<SPMatindex,T>`; row i maps column indices to values and is `NULL` if empty. Each position occurs at most once, rows are sorted by column automatically.

Despite its name, `SparseSMat` is a *general* sparse matrix: it need not be square ($n \times m$), and the elements (i, j) and (j, i) are stored independently, so nonsymmetric matrices are represented exactly. `operator*` (the general product AB), `gemv`, `axpy`, the iterator, `transpose()`, `submatrix`/`storesubmatrix`, `put`/`get`, the text I/O and the conversions to

and from `NRMat` work for general matrices. The symmetry (hermiticity for complex types) survives only as a convention of some operations:

- `add(i,j,x)` with the default `both=true` inserts x at (i,j) and x^* at (j,i); use `add(i,j,x,false)` for a single element;
- the constructor from `NRSMat` and `otimes_sparse` produce hermitian (real: symmetric) matrices;
- `gemm` computes $A^\mathsf{T}B$ (or $A^\dagger B$), i.e. AB only for $A = A^\mathsf{T}$ ($A = A^\dagger$), and is restricted to square matrices;
- `NRSMat(const SparseSMat&)` takes the lower triangle and assumes a hermitian matrix;
- `issymmetric()` always returns `true` (needed by the Davidson interface, chapter 9).

In particular complex symmetric matrices such as $\exp(\mathrm{i}H)$ with real symmetric H (the original purpose of the class, built e.g. by `imagmatrix` and `exp`), hermitian and general matrices are all handled correctly by `operator*` and `exp`.

Planned extension. In the future `SparseSMat` may be handled as a general matrix class with a member `symmetry_type` (0 for no symmetry, 1 symmetric, 2 hermitian); the conversion from `NRSMat`, `otimes_sparse`, `gemm` (and `issymmetric()`) would then be implemented according to it, in a backward compatible way.

Reference counting. The row array is shared by the copy constructor and `operator=`; there is no `operator|=`. `copyonwrite()` (no arguments) copies all maps (element type copied by its copy constructor, i.e. no nested deep copy). It is called by `operator*=`, `axpy` (hence `+=`, `-=`), `operator+=(const T&)`, `operator-=(const T&)`, `gemm` and `storesubmatrix`; `resize`, `clear`, `get(...,false,...)` and `operator=(const T&)` detach by reallocation. Not calling it: `add()` (by design: element-level access, like `operator[]` of the dense classes and `SparseMat::add()`; call `copyonwrite()` before adding to a possibly shared matrix), `operator[]` (raw row access, no reference count check even in DEBUG) and `simplify()` (removes only near-zero elements, which then disappear from all sharing copies).

7.4.2 Construction, assignment, conversion

`SparseSMat()`

Undefined empty matrix.

`explicit SparseSMat(const SPMatindex n); explicit SparseSMat(const SPMatindex n, const SPMatindex m)`

Empty (zero) $n \times n$ or $n \times m$ matrix.

`SparseSMat(const SparseSMat &rhs)`
: Shallow copy.

`explicit SparseSMat(const NRSMat<T> &rhs)`
: From a packed symmetric (complex: hermitian) matrix (both triangles inserted, the upper one conjugated, $|a_{ij}| >$ `SPARSEEPSILON`).

`explicit SparseSMat(const NRMat<T> &rhs)`
: From a dense $n \times m$ matrix, nonzero elements copied as they are (no symmetrization).

`explicit SparseSMat(const SparseMat<T> &rhs)`
: From a `SparseMat` (same dimensions); duplicates are summed, a symmetric `SparseMat` is unfolded without conjugation (complex symmetric, as `SparseMat` defines it).

`explicit SparseSMat(const CSRMat<T> &rhs)`
: Declared but not implemented (`CSRMat` is not supported).

`~SparseSMat()`
: Destructor.

`SparseSMat & operator=(const SparseSMat &rhs)`
: Shallow copy.

`SparseSMat & operator=(const T &a)`
: Scalar matrix $a \cdot 1$ (diagonal elements are created even for $a = 0$).

`void copyonwrite()`
: Make the data private.

`void resize(const SPMatindex nn, const SPMatindex mm)`
: Empty matrix of the new size (detaches if shared); `resize(0,0)` frees everything and leaves an undefined (empty) matrix.

`void dealloc(); void clear()`
: `resize(0,0)`; `resize(nrows(),ncols())`.

`inline void setcoldim(int i)`
: Set the number of columns without touching the data.

`inline int getcount() const`
: Reference count.

Dense from sparse: `explicit NRMat<T>(const SparseSMat<T>&)` copies all stored elements; `explicit NRSMat<T>(const SparseSMat<T>&)` (square only) takes the lower triangle $i \geq j$ (as stored in `NRSMat`; the matrix is assumed hermitian).

7.4.3 Element insertion and access

`inline void add(const SPMatindex n, const SPMatindex m, const T`

`elem, const bool both=true)`
: Add `elem` to element (n, m) and, if `both` and $n \neq m$, its complex conjugate to (m, n). Creates the row map if needed; no `copyonwrite()`; range check in DEBUG only. Resulting zeros remain until `simplify()`.

`inline const std::map<SPMatindex,T> *line(const SPMatindex n) const`
: Read-only access to row n (`NULL` if empty).

`inline ROWTYPE & operator[](const SPMatindex i)`
: Row i as a reference to the pointer `std::map<SPMatindex,T>*` (`typedef ROWTYPE`), possibly `NULL`. Non-const only.

`unsigned long long simplify()`
: Remove elements with $|a_{ij}| <$ `SPARSEEPSILON` and empty rows; returns the number of remaining elements.

`inline unsigned long long length()`
: `simplify()` (non-const).

`int nrows() const; int ncols() const; inline bool issymmetric() const`
: Dimensions; `issymmetric()` is always true.

`iterator begin() const; iterator end() const`
: Iterator over all stored elements, row by row in increasing column order. `*it` is a `matel<T>` *copy* held in the iterator (writing to it does not change the matrix). It is meant for comparison with `end()` or via `it.notend()`; `iterator(const SparseSMat&)` starts at the first element.

7.4.4 Arithmetic and products

`SparseSMat & operator*=(const T &a)`
: Scale (`a=0` clears the matrix).

`SparseSMat & operator+=(const T &a); SparseSMat & operator-=(const T &a)`
: Add/subtract $a \cdot 1$.

`void axpy(const T alpha, const SparseSMat &x, const bool transp=0)`
: $A \leftarrow A + \alpha x$, then `simplify()`; `transp` is ignored.

`inline SparseSMat & operator+=(const SparseSMat &rhs); inline SparseSMat & operator-=(const SparseSMat &rhs)`
: `axpy(`± 1`,rhs)`.

`inline const SparseSMat operator*(const T &rhs) const; inline const SparseSMat operator+(const T &rhs) const; inline const SparseSMat operator-(const T &rhs) const`
: Out-of-place scalar operations.

`inline const SparseSMat operator+(const SparseSMat &rhs) const;`

`inline const SparseSMat operator-(const SparseSMat &rhs) const`
: Sum, difference.

`void gemm(const T beta, const SparseSMat &a, const char transa, const SparseSMat &b, const char transb, const T alpha)`
: $C \leftarrow \beta C + \alpha \tilde{A}^{\mathsf{T}} \tilde{B}$ where $\tilde{A} = \bar{A}$ if `transa=='c'` and $\tilde{B} = \bar{B}$ if `transb=='c'` (other values: no conjugation); all matrices square of equal order. For symmetric A this is $\mathrm{op}(A)\,\mathrm{op}(B)$; it is *not* a general product. Rows k of A and B are multiplied as an outer product by BLAS and scattered into the maps; the result is simplified.

`const SparseSMat operator*(const SparseSMat &rhs) const`
: General product AB of the stored matrices (hermitian, complex symmetric, nonsymmetric or rectangular): row i of the result is accumulated as $\sum_k A_{ik}$ (row k of B); simplified.

`void gemv(const T beta, NRVec<T> &r, const char trans, const T alpha, const NRVec<T> &x) const`
: $r \leftarrow \beta r + \alpha A x$ (general A); only `trans='n'` is implemented.

`inline const NRVec<T> operator*(const NRVec<T> &rhs) const`
: Ax.

`typename LA_traits<T>::normtype norm(const T scalar=(T)0) const`
: Frobenius norm of $A - \texttt{scalar} \cdot 1$ over all stored elements.

`const T* diagonalof(NRVec<T> &r, const bool divide=0, bool cache=false) const`
: As for `SparseMat` (square matrices only).

`SparseSMat transpose(bool conj=false) const`
: Transposed (optionally conjugated) copy.

`void transposeme() const`
: Raises an error (not implemented).

`SparseSMat<T> cholesky(void) const`
: Pivoted Cholesky decomposition of a positive semidefinite symmetric/hermitian matrix (not checked): returns R with $R^\dagger R = A$ (`result.transpose(true)*result` reproduces the input). The pivoting is done a priori by sorting the diagonal; pivots with real part below `CHOLESKYEPSILON`$= 10^{-16}$ are skipped (rank-deficient case), elements below `SPARSEEPSILON` are dropped. R is upper triangular only in the permuted basis.

`SparseSMat submatrix(const int fromrow, const int torow, const int fromcol, const int tocol) const`
: Block (inclusive bounds).

`void storesubmatrix(const int fromrow, const int fromcol, const`

`SparseSMat &rhs)`

Store `rhs` into the block starting at (`fromrow`, `fromcol`): the old elements of the block are removed first (as `NRMat::storesubmatrix`); calls `copyonwrite()`.

7.4.5 Input/output and free functions

`void put(int fd, bool dimen=1, bool transp=0, bool orcaformat=false) const`

Binary output: optionally n, m; then row, column, value of each element; terminated by $(-1,-1)$. With `transp` the transposed matrix is written (dimensions and indices swapped), as for `SparseMat`.

`void get(int fd, bool dimen=1, bool transp=0, bool orcaformat=false)`

Binary input matching `put`; with `transp` the transposed matrix is stored. Elements are accumulated by `add(...,false)`; with `dimen=false` the read matrix replaces the old contents (`clear()`), the dimensions are kept.

`std::ostream & operator«(std::ostream &s, const SparseSMat<T> &x)`

Text output: `n m`, lines `row col value` (all stored elements), terminator `-1 -1`.

`std::istream& operator»(std::istream &s, SparseSMat<T> &x)`

Text input of the same format (square only); every element is inserted only at its own position (`add(i,j,x,false)`), so both triangles must be present in the input.

`SparseSMat<T> otimes_sparse(const NRVec<T> &lhs, const NRVec<T> &rhs, const bool conjugate=false, const T &scale=1)`

Symmetrized outer product, symmetric (complex: hermitian) regardless of the input vectors (unlike `SparseMat`): with $M = \texttt{scale} \cdot l\,r^{\mathsf{T}}$ (or $l\,r^{\dagger}$ if `conjugate`) the result is $S = M + M^{\dagger}$ ($= l\,r^{\mathsf{T}} + r\,l^{\mathsf{T}}$ for real types), including the doubled diagonal $S_{ii} = M_{ii} + M_{ii}^{*}$. Note that `otimes_sparse(x,x,true)` is $2\,xx^{\dagger}$; use `scale=0.5` for $xx^{\dagger}$. Elements with $|S_{ij}| \le$ `SPARSEEPSILON` are omitted. Both vectors must have the same length (`laerror` otherwise).

`SparseSMat<T> NRMat<T>::operator*(const SparseSMat<T> &rhs) const`

Declared in mat.h: dense times sparse, general product AS with a sparse result (row i accumulated as $\sum_k A_{ik}$ (row k of S), zero A_{ik} skipped; simplified). Instantiated for `double` and `std::complex<double>`.

In 8: `complexmatrix()`, `realmatrix()` and `imagmatrix()` build a `SparseSMat<complex<double»` from real ones. The generic `exp()` of `matexp.h` works with `SparseSMat`.

7.5 CSRMat

Header: `csrmat.h`

`CSRMat<T>` is meant to hold a matrix in compressed sparse row format (row pointers, column indices, values, optionally on the GPU via cuSPARSE) for fast products. The class is unfinished: only the declarations exist, `csrmat.cc` contains no implementation and no instantiation. It is *not supported* and must not be used; the declarations (also `NRMat(const CSRMat&)` and `SparseSMat(const CSRMat&)`) are kept for possible future work.

7.6 Examples

```
#include "la.h"
using namespace LA;

int main()
{
//general sparse matrix as a list of triplets
SparseMat<double> a(4,4);
a.add(0,0,2.); a.add(1,2,1.); a.add(3,0,-1.);
a.add(1,2,0.5);                          //duplicate position:
    values are summed
a.simplify();                            //merge duplicates, drop
     zeros
NRVec<double> x(1.,4);
NRVec<double> y = a*x;                   //sparse * dense vector
NRMat<double> d(a);                      //dense copy
for(SparseMat<double>::iterator p=a.begin(); p!=a.end(); ++p)
        std::cout << p->row <<" "<< p->col <<" "<< p->elem <<"\n"
    ;

//symmetric storage: only one triangle is stored
SparseMat<double> s(4,4);
s.setsymmetric();
s.add(2,0,1.5); s.add(3,3,1.);
NRVec<double> z = s*x;                   //uses s(2,0) and s(0,2)
NRSMat<double> sd(s);                    //dense packed copy

//shared matrices: detach before add()
SparseMat<double> b(a);                  //shares the list with a
b.copyonwrite();
b.add(2,2,3.);

//map-based sparse matrix
SparseSMat<double> h(4);
h.add(0,1,0.5);                          //adds (0,1) and (1,0)
```

```
h.add(2,2,1.);
SparseSMat<double> h2 = h*h;              //result stays symmetric
NRVec<double> w = h*x;
std::cout << y << z << w << NRMat<double>(h2) << sd;
}
```

7.7 Pitfalls

- `add()` of both classes does not call `copyonwrite()`; call it yourself on a matrix that may be shared.
- `SparseMat` may contain duplicates; call `simplify()` before relying on the list contents (iteration, `NRSMat` conversion, `diagonalof` for rectangular matrices).
- In a symmetric `SparseMat` never store both a_{ij} and a_{ji}; `setsymmetric()` does not check this.
- `SparseSMat::gemm` computes $A^{\mathsf{T}}B$ (or $A^{\dagger}B$); use `operator*` for the general product.
- Complex `SparseMat` matrices are complex symmetric, never hermitian, in all mirroring operations, whereas `SparseSMat` (`add` with `both=true`, conversion from `NRSMat`, `otimes_sparse`) is hermitian.

Chapter 8

Dense linear algebra functions

Header: `nonclass.h, nonclass.cc, matexp.h`

8.1 Purpose and general conventions

This chapter describes the free (non-member) functions operating on the dense classes `NRVec`, `NRMat` and `NRSMat`: linear systems, determinants, inverses, Cholesky decomposition, symmetric/hermitian and general eigenvalue problems, singular value decomposition, matrix functions, and a few auxiliary products and conversions. Most of them are thin wrappers of LAPACK routines (`nonclass.cc`); they exist for `T = double` and `std::complex<double>` only. The templates in `matexp.h` (matrix polynomials, matrix exponential, $\exp(A)\,x$, BCH expansion) are generic: they work for every class that provides the few operations listed in section 8.6, including sparse matrices (chapter 7) and user-defined matrix classes which only implement `gemv`. Iterative (Krylov) methods for large matrices are in chapter 9.

Destroyed arguments. LAPACK works in place. All functions taking a matrix by non-const reference (`linear_solve`, `diagonalize`, `gdiagonalize`, `singular_decomposition`, `cholesky`, `determinant_destroy`, ...) overwrite it, and the generalized eigenvalue routines overwrite also the metric `b`. These functions call `copyonwrite()` on every object they modify (except where noted as a bug below), so passing a shallow copy is enough to preserve the original (sec. 3.1):

```
NRMat<double> work = a;   // shallow copy, no data copied yet
diagonalize(work, w);     // copyonwrite() inside detaches work;
    a is intact
```

Functions which take the matrix by value (`calcinverse`, `calcsvdinverse`, `determinant`, `matrixfunction`, ...) do this themselves. Output vectors and matrices must be allocated by the caller with the correct dimensions (they are checked, not resized), with the exceptions noted.

C versus Fortran order. LA matrices are row-major (sec. 3.4), LAPACK is column-major, so a Fortran routine called on the raw data sees the transposed matrix. The wrappers take this into account; the visible consequence is the parameter `corder`, which selects whether eigenvectors are returned in the *columns* (`corder=1`, the default, the usual mathematical convention $AV = V\Lambda$) or in the *rows* (`corder=0`, the natural Fortran result, saves a transposition). For complex matrices the row form is the "bra" form, i.e. the rows are the complex conjugated eigenvectors; table 8.2 summarizes the conventions as verified numerically.

Active dimension. Several routines have an argument `n` (sometimes `m`): if positive, only the leading $n \times n$ submatrix of the (larger) arrays is processed, otherwise the whole matrix. This is used by the iterative methods, which keep a large work matrix and diagonalize its growing leading block.

Errors. LAPACK error codes (info $\neq 0$) are turned into `laerror` (sec. 3.6), e.g. "singular matrix" or "convergence problem". Exceptions: `linear_solve_x` and `multiply_by_inverse` return `info`.

Element types and availability. The functions are declared for `double` and `std::complex<double>` by the macro `declare_la`, but not all declared complex variants are defined in `nonclass.cc`. Calling an undefined one gives a link error (table 8.1).

8.2 Linear systems, inverse, determinant

The basic solver uses the LU decomposition with partial pivoting (`dgesv`/`zgesv`) for `NRMat` and the Bunch–Kaufman factorization (`dspsv`) for packed symmetric `NRSMat`. **The right-hand sides are the rows of `B`**: `B` has dimension $n_{\mathrm{rhs}} \times n$ and each row b is overwritten by the solution of $Ax = b$ (i.e. $AX^{\mathsf{T}} = B^{\mathsf{T}}$). `A` is overwritten by its factorization. With `B==NULL` only the factorization (and determinant) is computed.

Function	Available for
`linear_solve(NRMat&, NRMat*, ...)`	`double`, `complex<double>`
`linear_solve(NRMat&, NRVec&, ...)`, `linear_solve(NRSMat&, ...)`	`double`, `complex<double>` (complex `NRSMat`: hermitian)
`determinant`, `determinant_destroy`	`NRMat`, `NRSMat`, real and complex
`diagonalize`, `singular_decomposition`, `cholesky`, `diagofproduct`, `trace2(NRMat,NRMat)`, `linear_solve_x`, `multiply_by_inverse`, `MatrixNorm`, `CondNumber`	`double`, `complex<double>`
`gdiagonalize`	real matrix (real or complex eigenvector storage), complex matrix
`trace2` with `NRSMat`	`double`, `complex<double>` (hermitian)
`diagonalize3`, `realmatrixfunction`, `complexmatrixfunction`, `adjustphases`, `xcopy`, `xaxpy`, `xdot`	`double` only
templates of `matexp.h`, `power`, `commutator`, `twoside_transform`, ...	any type providing the needed operations

Table 8.1: Element types for which the functions of this chapter are available.

`void linear_solve(NRMat<T> &a, NRMat<T> *b, T *det=0, int n=0)`
: Solve $Ax = b$ for all rows b of `*b` (`b->ncols()` must be n). If `det` is non-NULL, the determinant of A is returned there (0 for a singular matrix, then no error is raised if `b==NULL`); a singular matrix with `b!=NULL` raises `laerror`. `n>0` restricts the problem to the leading $n \times n$ block of `a` and the first n columns of `b`. Prints a warning when `b` is `NULL` (some LAPACK libraries skip the factorization for $n_{\mathrm{rhs}} = 0$).

`void linear_solve(NRMat<T> &a, NRVec<T> &b, T *det=0, int n=0)`
: The same with a single right-hand side `b`.

`void linear_solve(NRSMat<T> &a, NRMat<T> *b, T *det=0, int n=0);`
`void linear_solve(NRSMat<T> &a, NRVec<T> &b, T *det=0, int n=0)`
: Symmetric (real) or hermitian (complex), possibly indefinite A in packed storage, `dspsv`/`zhpsv` (Bunch–Kaufman factorization; the determinant is computed from its 1×1 and 2×2 blocks and is real in the hermitian case). `n` selects a leading block, which is contiguous in the packed storage.

`int linear_solve_x(NRMat<T> &A, T *B, const int rhsCount, const int`

Call	`corder=1` (default)	`corder=0`
`diagonalize(NRMat&,...)`, real	columns of `a`: $AV = V\Lambda$	rows of `a`
`diagonalize(NRMat&,...)`, complex	columns of `a`: $AV = V\Lambda$	rows of `a` are $v_k^\dagger$: $V^\dagger A = \Lambda V^\dagger$
`diagonalize(NRSMat&,...)`, real	columns of `*v`	rows of `*v`
`diagonalize(NRSMat&,...)`, complex	columns of `*v`: $HV = V\Lambda$	rows of `*v` are $v_k^\dagger$
`diagonalize3`	columns of `*v`	rows of `*v`
`gdiagonalize`, complex matrix	columns of `*vr`: $Ax = \lambda x$; columns of `*vl`: $y^\dagger A = \lambda y^\dagger$ (LAPACK convention)	rows of `*vr` are $\bar{x}$; rows of `*vl` are $\bar{y}$, i.e. $\bar{y}^\mathsf{T} A = \lambda \bar{y}^\mathsf{T}$
`gdiagonalize`, real matrix, complex storage	as for a complex matrix	as for a complex matrix
`gdiagonalize`, real storage	columns; for a complex pair $(k, k+1)$ the eigenvector of w_k is column k + i column $k+1$ (as `dgeev`)	rows, analogously
`singular_decomposition`	$A = U\,\mathrm{diag}(s)\,V^\dagger$; `*u` $= U$ (columns = left singular vectors), `*v` $= V^\dagger$ (rows), or V if `vnotdagger`	

Table 8.2: Storage of eigenvectors and singular vectors (verified numerically). x denotes right, y left eigenvectors, bar complex conjugation.

`eqCount, const bool eq, const bool saveA, double *rcond)`

Expert solver (`dgesvx`/`zgesvx`): solves $Ax = b$ for `rhsCount` right-hand sides stored consecutively (each of length `eqCount`) in the raw array `B`, using the leading `eqCount` × `eqCount` block of `A`. `eq`: equilibrate A (for badly conditioned matrices); then `A` is overwritten unless `saveA`. The reciprocal condition number is stored in `*rcond` if non-NULL. Returns LAPACK `info` (0 success, > 0 singular or ill-conditioned). The solution replaces `B`.

`int multiply_by_inverse(NRMat<T> &A, NRMat<T> &B, bool useEq=false, double *rcond=NULL)`

$B \leftarrow AB^{-1}$ for square A, B via `dgesvx`/`zgesvx`; `useEq` as `eq` above. `A` is not modified (with `useEq` a temporary copy is scaled). Returns `info`.

`const NRMat<T> calcinverse(NRMat<T> a, T *det=NULL)`

A^{-1} by `linear_solve` with unit right-hand sides; optionally the determinant. `NRMat::inverse()` (chapter 5) calls it.

`const NRMat<T> calcsvdinverse(NRMat<T> a, double thr=0, bool do_absadjoint=false)`

Inverse via SVD, $V \operatorname{diag}(1/s_i)\, U^\dagger$, singular values $s_i <$`thr` are treated as zero (pseudoinverse). With `do_absadjoint` returns $V \operatorname{diag}(\prod_{j\neq i} s_j)\, U^\dagger$, the adjugate matrix up to the phase $\det U \det V$. Square matrices only.

`NRVec<T> svd_solve(NRMat<T> &a, const NRVec<T> &b, double thres=1e-12, int *nullity=NULL); NRMat<T> svd_solve(NRMat<T> &a, const NRMat<T> &b, double thres=1e-12, int *nullity=NULL)`

Least-squares solution $x = V \operatorname{diag}(1/s_i)\, U^\dagger b$ discarding singular values $\leq$`thres` (no contamination by the null space); the number of discarded ones is stored in `*nullity`. Here (unlike `linear_solve`) the right-hand sides are the *columns* of the matrix `b` ($m \times n_{\mathrm{rhs}}$); `a` ($m \times n$, $m \geq n$) is destroyed.

`const LA_traits<MAT>::elementtype determinant(MAT a)`

Determinant by LU (`NRMat`) or Bunch–Kaufman (`NRSMat`) factorization of a copy.

`const LA_traits<MAT>::elementtype determinant_destroy(MAT &a)`

The same, overwriting `a`. Unless `HAS_MKL` is defined, a dummy right-hand side is passed to `linear_solve` (work-around for OpenBLAS, which skips the factorization without right-hand sides).

`void cholesky(NRMat<double> &a, bool upper=1); void cholesky(NRMat<std::complex<double> > &a, bool upper=1)`

Cholesky factorization of a positive definite matrix in place (`dpotrf`/`zpotrf`). `upper=1`: `a` becomes the upper triangular U with $A = U^\dagger U$; `upper=0`: lower triangular L with $A = LL^\dagger$. The other triangle is zeroed. Raises `laerror` if A is not positive definite.

8.3 Eigenvalue problems

8.3.1 Symmetric and hermitian matrices

Eigenvalues are always returned in **ascending order** (LAPACK). Only the **lower triangle** (row index $\geq$ column index) of an `NRMat` is referenced (verified); `NRSMat` stores only that triangle and is interpreted as hermitian (sec. 6.2.3).

```
void diagonalize(NRMat<T> &a, NRVec<double> &w, const bool eivec=1,
    const bool corder=1, int n=0, NRMat<T> *b=NULL, const int
    itype=1)
```

Eigenvalues `w` (size must equal `a.nrows()`) and, if `eivec`, eigenvectors stored in `a` (columns for `corder=1`, see table 8.2) of a real symmetric or complex hermitian matrix (`dsyev`/`zheev`). If `b` is given, the generalized problem with positive definite B is solved (`dsygv`/`zhegv`): `itype=1`: $Ax = \lambda Bx$, `itype=2`: $ABx = \lambda x$, `itype=3`: $BAx = \lambda x$; eigenvectors are B-normalized ($x^\dagger Bx = 1$ for `itype` 1, 2), `b` is overwritten by its Cholesky factor.

```
void diagonalize(NRSMat<T> &a, NRVec<double> &w, NRMat<T> *v, const
    bool corder=1, int n=0, NRSMat<T> *b=NULL, const int itype=1)
```

The same for packed storage (`dspev`/`zhpev`, `dspgv`/`zhpgv`); eigenvectors go to `*v` (square, at least $n \times n$) or are not computed if `v==NULL`. `a` (and `b`) are destroyed.

```
void diagonalize3(NRVec<double> &d, NRVec<double> &d1, NRMat<double>
    *v, const bool corder=1, int n0=0)
```

Real symmetric tridiagonal matrix (`dsteqr`) with diagonal `d` and sub-diagonal `d1` (size $\geq n-1$). On return `d` holds the eigenvalues (ascending), `d1` is destroyed, `*v` (exactly $n \times n$) the eigenvectors. `n0>0` uses only the first `n0` elements. Used by `lanczos` (chapter 9).

8.3.2 General (non-symmetric) matrices

`gdiagonalize` wraps `dgeev`/`zgeev` and, if `b` and `beta` are given, the generalized problem $Ax = \lambda Bx$ (`dggev`/`zggev`), where the eigenvalues are $\lambda_i = w_i/\beta_i$ (`beta` may contain zeros for infinite eigenvalues). Right (`vr`) and left (`vl`) eigenvectors are computed only if the pointers are non-NULL; storage according to table 8.2. Without sorting, the eigenvalues come in LAPACK order (unsorted, complex conjugate pairs adjacent with positive imaginary part first).

```
void gdiagonalize(NRMat<double> &a, NRVec<double> &wr, NRVec<double>
    &wi, NRMat<double> *vl=NULL, NRMat<double> *vr=NULL, const bool
    corder=1, int n=0, const int sorttype=0, const int
    biorthonormalize=0, NRMat<double> *b=NULL, NRVec<double>
```

`*beta=NULL)`

Real matrix, eigenvalues $w_r + \mathrm{i}\, w_i$, eigenvectors in the real storage of `dgeev`: for a real eigenvalue the vector is real; for a complex pair $(j, j{+}1)$ the vectors j and $j{+}1$ hold the real and imaginary part. Verified: the right eigenvector belonging to $w_{r,j} + \mathrm{i}w_{i,j}$ $(w_{i,j} > 0)$ is $\mathrm{Re} - \mathrm{i}\,\mathrm{Im}$, and $\mathrm{Re} + \mathrm{i}\,\mathrm{Im}$ of `vl` satisfies $y^{\mathsf{T}}A = \lambda y^{\mathsf{T}}$ – note that this sign is opposite to the `dgeev` documentation. `sorttype`: 0 no sorting; 1 ascending real part (for equal real parts descending imaginary part; with `beta`, by w/β, infinite eigenvalues last); 2 real eigenvalues first, then complex ones, each group ordered as for 1. Eigenvectors are permuted accordingly. `biorthonormalize` (needs `vl` and `vr`, not with `b`): 1 scales the left, 2 the right eigenvectors so that $y_k^{\mathsf{T}} x_k = 1$; complex pairs are rotated to biorthonormality. `a` is destroyed (and used as scratch space).

```
void gdiagonalize(NRMat<T> &a, NRVec<std::complex<double> > &w,
    NRMat<std::complex<double> > *vl=NULL,
    NRMat<std::complex<double> > *vr=NULL, const bool corder=1, int
    n=0, const int sorttype=0, const int biorthonormalize=0,
    NRMat<T> *b=NULL, NRVec<T> *beta=NULL)
```

Template specialized for `T=double` (calls the previous routine and converts to complex eigenvalues and eigenvectors) and `T=complex<double>` (`zgeev`/`zggev`). For the complex matrix, `sorttype>0` raises `laerror` (not implemented) and `biorthonormalize` scales so that $\bar{y}_k^{\mathsf{T}} x_k = 1$ in the row storage.

```
void gdiagonalize(NRMat<std::complex<double> > &a, NRVec<double>
    &wr, NRVec<double> &wi, NRMat<std::complex<double> > *vl=NULL,
    NRMat<std::complex<double> > *vr=NULL, const bool corder=1, int
    n=0, const int sorttype=0, const int biorthonormalize=0,
    NRMat<std::complex<double> > *b=NULL, NRVec<std::complex<double>
    > *beta=NULL)
```

Complex matrix with real and imaginary parts of the eigenvalues in separate vectors; exists to make `davidson` compile for complex `T`.

8.4 Singular value decomposition

```
void singular_decomposition(NRMat<T> &a, NRMat<T> *u, NRVec<double>
```

`&s, NRMat<T> *v, const bool vnotdagger=0, int m=0, int n=0)`
SVD $A = U\,\mathrm{diag}(s)\,V^\dagger$ of the $m \times n$ matrix `a` (`dgesvd`/`zgesvd`; `m`, `n` select a leading block, default whole matrix). `s` (size $\geq \min(m,n)$) receives the singular values in descending order. `u`: `NULL` (not computed), $m \times m$ (all columns) or $m \times \min(m,n)$ (economy size, chosen by `u` being non-square). `v`: `NULL`, $n \times n$ or $\min(m,n) \times n$, receives $V^\dagger$ (right singular vectors as conjugated rows), or V if `vnotdagger` (square `v` only). `a` is destroyed.

8.5 Matrix functions

Functions of matrices computed via the eigendecomposition $f(A) = V f(\Lambda) V^{-1}$. For general matrices the eigenvalue problem is solved in complex arithmetic and the result converted back to the element type of the argument; for a real argument the imaginary part is *silently dropped* (its size is not checked). The matrix must be diagonalizable; nothing is done about defective or ill-conditioned eigenvector matrices.

`NRMat<T> matrixfunction(NRSMat<T> a, double (*f)(double))`
$f(A)$ for a real symmetric or complex hermitian packed matrix, real function `f` of the (real) eigenvalues.

`NRMat<T> matrixfunction(NRMat<T> a, std::complex<double> (*f)(const std::complex<double> &))`
$f(A)$ for a general real or complex matrix by `gdiagonalize`, $f(A) = \sum_k f(\lambda_k)\, x_k y_k^{\mathsf{T}} / (y_k^{\mathsf{T}} x_k)$.

`NRMat<double> realmatrixfunction(NRMat<double> a, double (*f)(double))`
$f(A)$ for a real symmetric matrix in full storage (only the lower triangle is used).

`NRMat<std::complex<double> > complexmatrixfunction(NRMat<double> a, double (*fre)(double), double (*fim)(double))`
Complex function $f = f_{\mathrm{re}} + \mathrm{i} f_{\mathrm{im}}$ of a real symmetric matrix, e.g. $\exp(\mathrm{i}A)$ with `cos`, `sin`.

`NRMat<double> sqrt(const NRSMat<double> &a); NRMat<double> sqrtinv(const NRSMat<double> &a)`
$A^{1/2}$, $A^{-1/2}$ of a symmetric matrix (e.g. symmetric orthogonalization).

`NRMat<double> realsqrt(const NRMat<double> &a); NRMat<double> realsqrtinv(const NRMat<double> &a)`
The same for a symmetric matrix in full storage.

`NRMat<double> log(const NRSMat<double> &a)`
Logarithm of a symmetric positive definite matrix.

`NRMat<double> log(const NRMat<double> &a); NRMat<std::complex<double> > log(const`

`NRMat<std::complex<double> > &a)`

Principal logarithm of a general matrix via complex eigendecomposition (for a real matrix the imaginary part of the result is dropped).

`NRMat<double> exp0(const NRMat<double> &a);`
`NRMat<std::complex<double> > exp0(const`
`NRMat<std::complex<double> > &a)`

exp(A) via eigendecomposition, an alternative to the Taylor-series `exp()` of section 8.6.

`NRMat<double> copytest(const NRMat<double> &a);`
`NRMat<std::complex<double> > copytest(const`
`NRMat<std::complex<double> > &a)`

`matrixfunction` with the identity function, i.e. reconstructs A from its eigendecomposition (a check of `gdiagonalize` available to users).

`double sqrtinv(const double); std::complex<double> sqrtinv(const`
`std::complex<double> &)`

Scalar $1/\sqrt{x}$ (used as function argument above).

`void adjustphases(NRMat<double> &v)`

Changes signs of rows of an orthogonal matrix (eigenvectors in rows) so that the diagonal elements are positive where possible and $\det v = +1$, to obtain a real logarithm of a rotation. The source comments say "not rigorous yet".

8.6 Matrix polynomials and exponentials (matexp.h)

Header: `matexp.h`

These templates use only generic operations and therefore work for `NRMat`, `NRSMat`, `SparseMat`, `SparseSMat` and user classes. `exp` and the polynomials need for the matrix type `T`: copy and assignment, `T*T`, `T*scalar`, `T+scalar` and `T+=T` where adding a scalar means adding it to the *diagonal* (this is how LA matrix classes define it), `norm()`, and the traits `LA_traits<T>::elementtype`, `normtype`, `deallocate` and `axpy` (sec. 3.3). The "`times`" variants never form the matrix: they need only `nrows()`, `ncols()`, `norm()` and `gemv(beta, NRVec &r, char trans, alpha, const NRVec &x)` computing $r \leftarrow \beta r + \alpha\,\mathrm{op}(A)x$, so they can be used with implicitly defined ("direct") matrices as in chapter 9. The vector type `V` needs `size()`, `operator*=`, `axpy` and `resize()`.

The exponential uses scaling and squaring: the power p is chosen so that $\|A\|/2^p \lesssim 0.75$ (`mynextpow2`), the Taylor series of the scaled argument is truncated when the next term would be below 10^{-14} relative to the norm, and the result is squared p times. The scaling is put into the Taylor coefficients, so the argument itself is not copied. `maxpower` (if ≥ 0) limits p,

`maxtaylor` (if ≥ 0) limits the order of the Taylor polynomial, which is useful and exact for nilpotent matrices (e.g. coupled-cluster excitation operators).

`const T polynom0(const T &x, const NRVec<R> &c)`
: $\sum_{i=0}^{N} c_i x^i$ by the Horner scheme (N=`c.size()-1`).

`const T polynom(const T &x, const NRVec<R> &c)`
: The same with fewer matrix multiplications at the cost of storing powers $x, \ldots, x^m$ (block summation of Paterson and Stockmeyer, $\sum_i s_i \, (x^m)^i$ with $s_i = \sum_{j<m} c_{im+j} x^j$): m is chosen to minimize the number $m - 2 + 2\lfloor N/m \rfloor - [m \,|\, N]$ of multiplications, e.g. 5, 8, 13 instead of 8, 16, 32 for $N = 8, 16, 32$; for $N \leq 4$ calls `polynom0`.

`const T exp(const T &x, bool horner=false, int maxpower= -1, int maxtaylor= -1)`
: $\exp(x)$ by scaling and squaring; `horner` selects `polynom0` or `polynom` (default) for the Taylor polynomial. Both give the same accuracy; `polynom` is 1.3–1.5 times faster for dense matrices (tests with $n = 100$–800), while for sparse matrices `horner=true` is faster, since the stored powers fill in.

`void sincos(T &s, T &c, const T &x, bool horner=false, int maxpower= -1, int maxtaylor= -1)`
: $s = \sin x$, $c = \cos x$ simultaneously in real arithmetic (doubling formulas instead of squaring); `T` also needs `operator*=(double)`.

`NRMat<std::complex<T> > expi(const NRMat<T> &x)`
: $\exp(\mathrm{i}x) = \cos x + \mathrm{i} \sin x$ of a real matrix via `sincos`.

`const T ipow(const T &x, int i)`
: x^i, $i \geq 0$, by binary powering; x^0 is the unit matrix of the size of x.

`const V exptimes(const M &mat, V rhs, bool transpose=false, const LA_traits<V>::elementtype scale=1., int maxpower= -1, int maxtaylor= -1, bool mat_is_0=false)`
: $\exp(s \, \mathrm{op}(A))$ `rhs` with $\mathrm{op}(A) = A$ or A^{T} (`transpose`), using only `mat.gemv`. The scaled Taylor polynomial is applied 2^p times instead of squaring. `scale` s may be complex for a real matrix and complex vectors (e.g. time evolution $\exp(-\mathrm{i}Ht)x$). `mat_is_0` short-cuts to a (deep) copy of `rhs`. Raises `laerror` for an unrealistically large p (use `maxpower`).

`const V exptimesreal(const M &mat, V rhs, bool transpose=false, const LA_traits<V>::normtype scale=1., int maxpower= -1, int maxtaylor= -1, bool mat_is_0=false)`
: The same with a real `scale` for a complex vector (overload resolution helper).

`void exptimesdestructive(const M &mat, V &result, V &rhs, bool transpose, const MEL scale, int maxpower= -1, int maxtaylor= -1,`

`bool mat_is_0=false)`

Worker of `exptimes`: `result` $= \exp(s\,\mathrm{op}(A))$ `rhs`, `rhs` is used as scratch and destroyed.

`void sincostimes(const M &mat, V &si, V &co, const V &rhs, bool transpose=false, const LA_traits<V>::normtype scale=1., int maxpower= -1, int maxtaylor= -1, bool mat_is_0=false)`

`si` $= \sin(sA)$ `rhs`, `co` $= \cos(sA)$ `rhs` with `gemv` only. The doubling is done by recursion with 3^p evaluations; the source comment recommends complex `exptimes` instead.

`const T ncommutator(const T &x, const T &y, int nest=1, const bool right=1); const T nanticommutator(const T &x, const T &y, int nest=1, const bool right=1)`

Nested (anti)commutator: `right`: $[\ldots[[x,y],y]\ldots,y]$, otherwise $[x,[x,\ldots[x,y]]]$, `nest` levels.

`const T BCHexpansion(const T &h, const T &t, const int nmax, const bool verbose=0, const bool right=1)`

Truncated Baker–Campbell–Hausdorff series $\sum_{n=0}^{n_{\max}} \frac{1}{n!}[\ldots[h,t],\ldots,t]_n = e^{-t}h\,e^{t}$ (`right`) or $e^{t}h\,e^{-t}$ (`right=0`); `verbose` prints the norm of each order to `std::cerr`.

`const V BCHtimes(const M1 &H, const char transH, const M2 &T, const char transT, const V &x, const int nmax, const bool right=1)`

$e^{-T}He^{T}x$ (`right`) or $e^{T}He^{-T}x$ truncated at order `nmax`, using only `gemv` of H and T (`transH`, `transT` select op): $HT^{j}x$ is precomputed for $j \le n_{\max}$ and the nested commutators are built recursively in place, so $n_{\max}+1$ vectors are stored.

The helpers `exp_aux`, `sincos_aux`, `sincostimes_aux`, `sincostimes_simple`, `mynextpow2` and the coefficient table `exptaylor` are internal.

8.7 Products, traces, transformations

`const NRVec<T> diagofproduct(const NRMat<T> &a, const NRMat<T> &b, bool trb=0, bool conjb=0)`

Diagonal of AB without forming the product; `trb`: of AB^{T} (or $AB^{\dagger}$ with `conjb`, complex only).

`T trace2(const NRMat<T> &a, const NRMat<T> &b, bool trb=0)`

$\mathrm{Tr}(AB)$, or $\mathrm{Tr}(AB^{\mathsf{T}}) = \sum_{ij} a_{ij}b_{ij}$ with `trb` (complex: $\mathrm{Tr}(AB^{\dagger})$).

`T trace2(const NRSMat<T> &a, const NRSMat<T> &b, const bool diagscaled=0)`

$\mathrm{Tr}(AB) = \sum_{ij} a_{ij}b_{ij}$ of symmetric matrices, $\sum_{ij} a_{ij}\bar{b}_{ij}$ (real) of hermitian ones. `diagscaled`: returns $2\sum_{i\ge j} a_{ij}b_{ij}$, i.e. the correct trace if the diagonal of one argument has been pre-multiplied by $1/2$.

`T trace2(const NRSMat<T> &a, const NRMat<T> &b, const bool diagscaled=0); T trace2(const NRMat<T> &a, const NRSMat<T> &b, const bool diagscaled=0)`

$\mathrm{Tr}(AB)$ of a symmetric (hermitian for complex) and a general matrix; `diagscaled` counts the diagonal twice.

`const NRSMat<T> twoside_transform(const NRSMat<T> &S, const NRMat<T> &C, bool transp=0); const NRMat<T> twoside_transform(const NRMat<T> &S, const NRMat<T> &C, bool transp=0)`

$C^\dagger SC$, or $CSC^\dagger$ with `transp` (similarity/congruence transformation).

`const NRMat<T> diagonalmatrix(const NRVec<T> &x)`

Square matrix with diagonal `x`.

`const NRMat<T> commutator(const NRMat<T> &x, const NRMat<T> &y, const bool trx=0, const bool tryy=0); const NRMat<T> anticommutator(...)`

$\mathrm{op}(x)\mathrm{op}(y) \mp \mathrm{op}(y)\mathrm{op}(x)$ with two `gemm` calls; `trx`, `tryy` transpose the factors (no conjugation).

`T positive_power(const T &x, const int n)`

x^n, $n \geq 1$, by binary powering (any type with `operator*`).

`T power(const T &x, const int n)`

x^n for any integer n; needs `identity()` for $n = 0$ and `inverse()` for $n < 0$ (e.g. `NRMat`, permutations).

8.8 Norms and condition number

`typename LA_traits<MAT>::normtype MatrixNorm(const MAT &A, const char norm)`

Matrix norm by `dlange`/`zlange`, implemented for `NRMat<double>` and `NRMat<std::complex<double> >` (explicit specializations of the template, which can be added for other types). `'1'` or `'O'`: 1-norm (maximum column sum); `'I'`: ∞-norm (maximum row sum); `'F'` or `'E'`: Frobenius norm; `'M'`: largest absolute value of an element (lower case accepted). Works for non-square matrices.

`typename LA_traits<MAT>::normtype CondNumber(const MAT &A, const char norm)`

Condition number $\|A\|\,\|A^{-1}\|$ of a square matrix in the 1-norm (`'1'`, `'O'`) or ∞-norm (`'I'`), from the LU factorization of a copy of `A` (`dgetrf`/`zgetrf`) and `dgecon`/`zgecon`; infinity for a singular matrix. It is LAPACK's estimate, a lower bound that is usually exact. Same types as `MatrixNorm`.

8.9 Real and imaginary parts

`const LA_traits<T>::realtype realpart(const T &a); const LA_traits<T>::realtype imagpart(const T &a)`
: Real or imaginary part as a real object; specialized for `NRVec`, `NRMat`, `NRSMat` and `SparseMat` over `complex<double>`.

`const LA_traits<T>::complextype realmatrix(const T &a); const LA_traits<T>::complextype imagmatrix(const T &a)`
: Complex object A resp. iA from a real one; for `NRMat<double>` and `SparseSMat<double>`.

`const LA_traits<T>::complextype complexmatrix(const T &re, const T &im)`
: Complex object re + i im; for `NRMat<double>` and `SparseSMat<double>`.

For the same purpose the classes provide `complexify()` (chapters 4, 5). The real and imaginary part functions are CPU only (sec. 3.5).

8.10 Miscellaneous

`const NRMat<LA_traits<MAT>::elementtype> submatrix(const MAT a, const int nrows, const INDEX rows, const int ncols, const INDEX cols, int indexshift=0, bool ignoresign=false, bool equalsigns=false)`
: General (non-contiguous) submatrix $r_{ij} = a($`rows`$[i] + s,$ `cols`$[j] + s)$, $s=$`indexshift`; `INDEX` is anything indexable (`NRVec<int>`, `int*`). `ignoresign`: use |`rows`$[i]$|, |`cols`$[j]$|; `equalsigns`: $r_{ij} = 0$ if the two indices have opposite signs. Application from the source: overlap of Slater determinants with signed spin-orbital indices (`indexshift=-1`, both flags true). `MAT` needs only `operator()(int,int)`.

`NRMat<T> reconstructmatrix<M,T>(const M &implicitmat)`
: Explicit matrix of an implicit one (applying `gemv` to unit vectors), for debugging; both template arguments must be given. Equivalent to `explicit_matrix<T>` of `davidson.h` (chapter 9).

`void xcopy(int n, const T *x, int incx, T *y, int incy); void xaxpy(int n, const T &a, const T *x, int incx, T *y, int incy); T xdot(int n, const T *x, int incx, const T *y, int incy)`
: Type-generic names of the CBLAS routines (to ease switching between `float`/`double`/complex in user code); at present specialized only for `double`.

8.11 Examples

Symmetric eigenproblem, eigenvectors in columns, and a linear system with two right-hand sides (rows of `b`):

```
#include "la.h"
using namespace LA;

int main()
{
const int n=4;
NRSMat<double> s(n);
for(int i=0; i<n; ++i) for(int j=0; j<=i; ++j) s(i,j) = (i==j? i
    +2. : 0.5/(1+i+j));

NRSMat<double> work = s;      //shallow copy, diagonalize
    detaches it
NRVec<double> w(n);
NRMat<double> v(n,n);
diagonalize(work, w, &v);     //w ascending, columns of v are
    eigenvectors
NRMat<double> sv = NRMat<double>(s)*v;
for(int k=0; k<n; ++k) std::cout << (sv.column(k) - v.column(k)*w
    [k]).norm() << std::endl;

NRMat<double> a(s), b(2,n);   //solve a x = b_i for the 2 rows of
     b
for(int i=0; i<2; ++i) for(int j=0; j<n; ++j) b(i,j) = i+j;
double det;
linear_solve(a, &b, &det);    //a now holds the LU factors, rows
    of b the solutions
NRMat<double> s2 = sqrtinv(s); //S^{-1/2}
std::cout << det << std::endl << s2;
}
```

Exponential of a sparse matrix and its action on a vector (`t.cc` contains similar tests):

```
SparseMat<double> h(n,n);  //... filled with h.add(i,j,x)
NRMat<double> e = exp(NRMat<double>(h));    //dense exponential
SparseMat<double> es = exp(h);              //sparse arithmetics
NRVec<double> y = exptimes(h, x);           //exp(h) x, only gemv
     used
NRVec<std::complex<double> > xc(complexify(x));
NRVec<std::complex<double> > psi = exptimes(h, xc, false, std::
    complex<double>(0,-0.1)); //exp(-0.1 i h) x
```

General matrix: left and right eigenvectors in complex storage, rows for `corder=0`:

```
NRMat<double> g(n,n);   //... filled
```

```
NRMat<double> scratch = g;
NRVec<std::complex<double> > lambda(n);
NRMat<std::complex<double> > vl(n,n), vr(n,n);
gdiagonalize(scratch, lambda, &vl, &vr, false, 0, 0, 0, (NRMat<
    double>*)NULL, (NRVec<double>*)NULL);
//rows: vl[k]^T g = lambda[k] vl[k]^T,  g conj(vr[k]) = lambda[k]
     conj(vr[k])
```

In the last call the null pointers must be cast, otherwise the template argument of `b` cannot be deduced.

8.12 Pitfalls

- Right-hand sides of `linear_solve` are the *rows* of `B` ($n_{\mathrm{rhs}} \times n$), those of `svd_solve` the *columns*.
- Eigenvectors are in columns only for `corder=1`; in row storage (`corder=0`) complex eigenvectors are conjugated (table 8.2).
- `diagonalize(NRMat&, ...)` uses only the lower triangle; a non-symmetric `NRMat` is silently treated as symmetric.
- Unsorted output of `gdiagonalize` unless `sorttype` is set (real matrices only).
- `matrixfunction`, `log` etc. of a real general matrix drop the imaginary part of the result without any check.
- The arguments passed by non-const reference are destroyed; keep a shallow copy if the original is needed (it is sufficient because of internal `copyonwrite()`).
- Several declared complex variants are missing (link errors), see section 8.1.

Chapter 9

Iterative methods

Header: `davidson.h, lanczos.h, conjgrad.h, gmres.h, diis.h, bisection.h`

9.1 Purpose and main idea

This chapter describes the header-only iterative algorithms of LA:

- `davidson()` (`davidson.h`): a few lowest (or closest to a target) eigenpairs of a large real symmetric or complex hermitian matrix, or right eigenpairs with real eigenvalues of a real non-symmetric matrix;
- `lanczos()` (`lanczos.h`): a few lowest (or closest to a target) eigenpairs of a large hermitian matrix;
- `conjgrad()` (`conjgrad.h`) and `gmres()` (`gmres.h`): solution of large linear systems $Ax = b$, or of the normal equations $A^{\mathsf{T}}Ax = A^{\mathsf{T}}b$ for rectangular A (least squares);
- the class `DIIS` (`diis.h`): Pulay's DIIS convergence acceleration of a general fixed-point iteration;
- `bisection_find()` and `interpolation_find()` (`bisection.h`): search in a sorted array.

Dense (LAPACK based) diagonalization, linear solvers and matrix functions are in chapter 8; the small subspace problems of `davidson` and `lanczos` are solved by the routines described there (`diagonalize`, `gdiagonalize`, `diagonalize3`, `linear_solve`). The quasi-Newton GDIIS optimizer built on the `DIIS` class is in chapter 10.

All functions are templates defined in the headers (all except `bisection.h` are included by `la.h`) and are instantiated by the user's code. The matrix is a template parameter `Matrix` which is accessed only

through a few member functions, so that the matrix need not be stored at all: the matrix-vector product may be computed "directly" from other data (direct CI in quantum chemistry, operators given by a formula, a product of several matrices, ...). Any of the LA matrix classes providing these functions (`NRMat`, `NRSMat`, `SparseMat`, `SparseSMat`, `CSRMat`, `SemiSparseMat`, chapters 5, 6, 7) can also be used directly.

9.2 Requirements on the matrix class

The member functions used are listed in table 9.1; they must be `const`. Their semantics follow the LA matrix classes:

`int nrows() const; int ncols() const`
: Dimensions (any integer type). The eigensolvers require a square matrix (`laerror` otherwise).

`bool issymmetric() const`
: `true` if the matrix is real symmetric or complex hermitian. `davidson` chooses the symmetric or non-symmetric algorithm by it; `lanczos` calls `laerror` if it is `false`. Note that `NRMat::issymmetric()` always returns `false`, `NRSMat` and `SparseSMat` return `true`, `SparseMat` returns its symmetric flag.

`const T* diagonalof(NRVec<T> &r, const bool divide=0, bool cache=false) const`
: With `divide==false`: store the diagonal A_{ii} into `r` (allocated by the caller with `nrows()` elements) and return a pointer to the n diagonal elements, normally `&r[0]`. The eigensolvers call it with `cache=true`; a direct matrix may then compute the diagonal once, keep it, and return a pointer to its own copy (the comment in davidson.h recommends this "for efficiency in a direct CI case"); the library classes ignore `cache`. With `divide==true`: divide `r` element-wise by the nonzero diagonal elements (Jacobi preconditioning) and return `NULL`. For a rectangular matrix ($m \times n$, $m \neq n$) the library classes use the diagonal of $A^\mathsf{T}A$ (squared column norms, `r` with $n =$`ncols()` elements) instead, which is the right (Jacobi) preconditioner for the normal equations used by `conjgrad` and `gmres` in that case; user matrix classes should follow the same convention.

`void gemv(const T beta, NRVec<T> &r, const char trans, const T alpha, const NRVec<T> &x) const`
: $r \leftarrow \beta r + \alpha\,\mathrm{op}(A)\,x$ with $\mathrm{op}(A) = A$ for `trans=='n'` and A^T for `'t'`; `r` is allocated by the caller with the right size. With $\beta = 0$ the previous contents of `r` must be ignored (it can be uninitialized).

	issymmetric	diagonalof	gemv
davidson	yes	divide=false	'n', $\beta = 0$
lanczos	yes (must be true)	divide=false, only without initguess	'n', $\beta = 0$ and $\beta = 1$
conjgrad	–	divide=true, if precondition	'n'; 't' if doguess or non-square
gmres	–	divide=true, if precondition	'n'; 't' if doguess or non-square
explicit_matrix	–	–	'n', $\beta = 0$

Table 9.1: Member functions of Matrix used by the iterative solvers; all of them also use nrows() and ncols().

Copy on demand. The solvers keep the Krylov vectors as shallow copies of their work vectors (sec. 3.1), so the vectors r passed to gemv and diagonalof frequently share their data with stored vectors. A user implementation must therefore not write elements of r through operator[] or a raw pointer before calling r.copyonwrite(); whole-object operations (r=0., r*=beta, r|=d, axpy, and the gemv of the library classes) call it themselves. Forgetting it silently corrupts the Krylov space (in DEBUG builds the non-const operator[] of a shared vector raises laerror). The input vector x is a const reference and can be read freely. The library classes satisfy this.

Out-of-core storage. davidson, lanczos, gmres and DIIS can keep their vectors in scratch files instead of memory (parameter incore). They use AuxStorage<T> (auxstorage.h, chapter 19): a binary file with fixed-length records of raw elements, created by mkostemp as AUX_XXXXXX in \$TMPDIR (if set, otherwise in the current working directory) and unlinked at once, so no file remains, even after a crash or an exception. Only one n-vector (two in davidson) is then kept in memory per stored set besides the work vectors, at the price of reading all stored vectors in every iteration. If a solver is left by an exception, the storage objects themselves are not freed (raw pointers), but the disk space is released when the process ends.

Template argument deduction. The element type T of davidson and lanczos is deduced from the vector arguments. A literal NULL passed for one of the pointer parameters depending on T (eivecs, initguess, target) makes the deduction fail; either cast it ((NRVec<double> *)NULL, as in

t.cc) or give `T` explicitly: `davidson<double>(a,e,NULL,NULL,2)` (verified). Arguments left at their default values are not a problem.

9.3 Davidson diagonalization

Header: `davidson.h`

```
template <typename T, typename Matrix> void davidson(const Matrix
    &bigmat, NRVec<T> &eivals, NRVec<T> *eivecs, const char
    *eivecsfile, int nroots=1, const bool verbose=0, const double
    eps=1e-6, const bool incore=1, int maxit=100, const int
    maxkrylov = 500, void (*initguess)(NRVec<T> &)=NULL, const
```

`typename LA_traits<T>::normtype *target=NULL)`

Davidson's method [1] for the `nroots` lowest eigenvalues of `bigmat`.

- `eivals`: on return the eigenvalues in its first `nroots` elements (ascending, or by increasing distance from `*target`); it must have at least `nroots` elements (`laerror`), it is not resized.
- `eivecs`: `NULL` or an array of at least `nroots` vectors receiving the normalized eigenvectors (by deep copy, `|=`, so their previous size does not matter).
- `eivecsfile`: `NULL` or a file name; the eigenvectors are then also written into this file as binary records of n elements of type `T`, record i = root i (an existing file is removed first), also if `eivecs==NULL`.
- `nroots`: number of roots; reduced silently to `maxkrylov-1` if larger.
- `verbose`: print the eigenvalue estimates in each iteration and the convergence of each root to `std::cout`.
- `eps`: convergence threshold (see below).
- `incore`: keep the Krylov vectors v_k and Av_k in memory (`true`) or in two `AuxStorage` scratch files.
- `maxit`: maximum number of iterations, i.e. of matrix-vector products after the initial one (restarts included).
- `maxkrylov`: maximum dimension of the subspace; the subspace matrices are allocated with this dimension. When it is reached, the method restarts (see below).
- `initguess`: function filling its argument (allocated with n elements, uninitialized) with the start vector, which need not be normalized. By default the unit vector e_j with j the index of the lowest (real part of the) diagonal element is used.
- `target`: if not `NULL`, the subspace eigenpairs are sorted by $|\lambda - *\texttt{target}|$ instead of ascending order, i.e. the roots closest to the target value are sought.

Throws `laerror` if not converged within `maxit` iterations.

Algorithm. The method keeps the subspace vectors v_k (not orthonormalized exactly, therefore the overlap matrix $S_{jk} = v_j \cdot v_k$ is kept too), the products Av_k, and the subspace matrix $H_{jk} = v_j \cdot Av_k$. In each iteration

the generalized subspace eigenproblem $Hc = \lambda Sc$ is solved by `diagonalize` (symmetric case) or by `gdiagonalize` with `sorttype=2` (real eigenvalues first, in ascending order; non-symmetric case, right eigenvectors; the imaginary parts of complex eigenvalues are discarded). The roots are converged one after another: for the current root λ with the Ritz vector $y = \sum_k c_k v_k$ the residual $\rho = \sum_k c_k(Av_k - \lambda v_k)$ is formed and the correction vector is $\delta_i = \rho_i/\mathrm{d}_i$ with $\mathrm{d}_i = A_{ii} - \lambda$, where $|\mathrm{d}_i|$ is limited from below by 0.1 (keeping its sign). The correction is normalized and orthogonalized by Gram-Schmidt to all previous subspace vectors (the projection on each v_j is repeated while it removes more than 1 % of the norm) and appended to the subspace. Each iteration costs one `gemv` with `'n'` and one `diagonalof`.

Convergence. A root is considered converged when the coefficient of the most recently added subspace vector in its subspace eigenvector, $|c_{\mathrm{last}}|$, is below `eps`; this is *not* the residual norm, and the error of the eigenvalue is typically much smaller than `eps`. After each iteration the method continues with the lowest root not yet converged, and returns to a lower root if its $|c_{\mathrm{last}}|$ exceeds `eps` again. The method finishes successfully when all `nroots` roots are converged, or (with a warning on `std::cout`) when the new correction vector vanishes before or during the orthogonalization; in this case the current Ritz values are returned even if not all roots satisfy the criterion (intended for tiny matrices). If $|c_{\mathrm{last}}| > 100$ for the current root, "Divergence in Davidson" is printed and `laerror` is called.

Restart. When the subspace dimension reaches `maxkrylov`, the method restarts from the current approximation of the lowest root alone, but only if no root has converged yet; otherwise it fails with `laerror`. For several roots `maxkrylov` has therefore to be large enough to converge all of them without restart.

Element types. `T=double` is the main case. For complex hermitian matrices (`T=std::complex<double>`) the method works, but according to the source comment "does not converge so well for more than 1 root"; non-hermitian complex matrices are not supported. The non-symmetric real case yields right eigenvectors belonging to real eigenvalues only.

The file has no header; it is read back with the `get` methods without dimensions, e.g. all roots as the rows of a matrix, `NRMat<T> V(nroots,n); V.get(fd,false);`, or root by root with `NRVec<T>::get(fd,false)` (`fd` from `open(eivecsfile,O_RDONLY)`). `AuxStorage(name)` cannot be used for reading, it recreates the file.

```
template <typename T, typename Matrix> NRMat<T>
```

`explicit_matrix(const Matrix &bigmat)`

The explicit dense matrix of an implicitly given one, built column by column by `gemv` on unit vectors (for debugging). `T` must be given: `explicit_matrix<double>(a)`. Same as `reconstructmatrix` in chapter 8.

9.4 Lanczos diagonalization

Header: `lanczos.h`

```
template <typename T, typename Matrix> void lanczos(const Matrix
    &bigmat, NRVec<T> &eivals, NRVec<T> *eivecs, const char
    *eivecsfile, int nroots=1, const bool verbose=0, const double
    eps=1e-6, const bool incore=1, int maxit=100, const int
    maxkrylov = 1000, void (*initguess)(NRVec<T> &)=NULL, const
    typename LA_traits<T>::normtype *target=NULL)
```

Lanczos method for the `nroots` lowest (or closest to `*target`) eigenvalues of a hermitian matrix. The parameters have the same meaning as for `davidson`, with these differences: `bigmat.issymmetric()` must be `true` (`laerror` otherwise); the start vector is normalized by the function; `incore=false` keeps only the Lanczos vectors v_k in one scratch file; the default `maxkrylov` is 1000.

Algorithm. Plain three-term Lanczos recurrence $w = Av_j - \beta_{j-1}v_{j-1}$, $\alpha_j = v_j \cdot w$, $w \leftarrow w - \alpha_j v_j$, $\beta_j = |w|$, $v_{j+1} = w/\beta_j$, one `gemv` per iteration (with $\beta = 1$ in the `gemv` sense, accumulating into $-\beta_{j-1}v_{j-1}$). The vectors are not reorthogonalized, so that in long runs copies of converged eigenvalues may appear (a known property of the method in finite precision). If $\beta_j = 0$ (invariant subspace), a random vector orthonormalized to all previous v_k is used to continue. The tridiagonal matrix is diagonalized in each iteration by `diagonalize3`. All v_k are stored (in core or in the scratch file) to build the eigenvectors at the end. If the diagonal elements $v_j \cdot Av_j$ have an imaginary part larger than 10^{-6}, `laerror` reports a non-hermitian matrix.

Convergence. Converged when the changes of the `nroots` lowest (or closest to the target) eigenvalues between two iterations are all $\leq$ `eps`; this is checked once the dimension exceeds `nroots`. There is no restart: the method fails with `laerror` when more than `maxit` iterations are needed or when the dimension reaches `maxkrylov` (the effective limit is min(`maxit`, `maxkrylov` − 1) iterations).

9.5 Conjugate gradients

Header: `conjgrad.h`

```
template<typename T, typename Matrix> bool conjgrad(const Matrix
    &bigmat, const NRVec<T> &b, NRVec<T> &x, const bool
    doguess=true, const double tol=1e-8, const int itmax=1000, const
    bool verbose=true, bool issquare=1, const bool precondition=1)
```

Solves $Ax = b$ by the (Jacobi preconditioned) conjugate gradient method (the source refers to the book of Stoer and Bulirsch). Returns `true` if converged, `false` after `itmax`+1 iterations without convergence; `x` then holds the last iterate.

- `b`: right-hand side with `nrows()` elements; `x`: initial guess and solution, must have `ncols()` elements (`laerror` otherwise).
- `doguess`: overwrite `x` by the start vector $-A^\mathsf{T}b$ (divided by the diagonal if `precondition`), normalized to 1. With `false` the contents of `x` are the start vector.
- `tol`: convergence threshold, see below.
- `itmax`: maximum number of iterations; `verbose`: print the error in each iteration (default `true`).
- `issquare`: `true` solves $Ax = b$ directly, which requires a symmetric positive definite matrix; `false` applies CG to the normal equations $A^\mathsf{T}Ax = A^\mathsf{T}b$ (two `gemv` per iteration), usable for non-symmetric matrices. It is set to `false` automatically for a rectangular matrix.
- `precondition`: divide the residual by the diagonal (`diagonalof(r,true)`) in each iteration. The diagonal must then be positive. For a square matrix with `issquare=false` the diagonal of A (not of $A^\mathsf{T}A$) is used, since `diagonalof` cannot know that the normal equations are solved; this is not the proper preconditioner for $A^\mathsf{T}A$ (the Jacobi preconditioner would be the squared column norms) and may slow down or break the convergence if A has non-positive diagonal elements; use `precondition=false` then.

Convergence. The criterion is $|p| \le$ `tol` for the current search direction p (an absolute criterion; in the first iteration p is the preconditioned residual). The function calls `laerror` ("singular matrix") if $p \cdot Ap$ (or $|Ap|^2$) or the residual product $r \cdot z$ vanishes. Only real `T` is supported (the scalars are `double`).

9.6 GMRES

Header: `gmres.h`

`template<typename T, typename Matrix> bool gmres(const Matrix &bigmat, const NRVec<T> &b, NRVec<T> &x, const bool doguess=1, const double eps=1e-7, const int MAXIT=50, const bool verbose=1, bool square=1, const bool precondition=1, int neustart=0, const int incore=1)`

Solves $Ax = b$ by restarted GMRES with Givens rotations (after H. R. Schwarz, original implementation by M. Warken). Returns `true` if converged; otherwise `false`, with `x` holding the last iterate.

- `b`, `x`, `doguess`, `precondition`: as for `conjgrad`; the preconditioner D^{-1} (Jacobi) is applied from the left, i.e. GMRES minimizes $|D^{-1}(Ax - b)|$.
- `eps`: relative threshold; converged when the estimated (preconditioned) residual norm is below `eps`$\cdot\beta_0$, β_0 being the preconditioned residual norm of the start vector.
- `MAXIT`: dimension of the Krylov space in one cycle; `MAXIT+1` vectors are stored.
- `verbose`: print the residual estimate in each step (default `true`).
- `square`: `false` solves the normal equations $A^{\mathsf{T}}Ax = A^{\mathsf{T}}b$; set automatically for a rectangular matrix.
- `neustart`: maximum number of cycles (restarts); 0 means `nrows()/10`, and at least 10 cycles are always allowed.
- `incore`: 0 stores the Krylov vectors in an `AuxStorage` scratch file.

On convergence the minimizer over the whole current Krylov space is returned. A breakdown ($|h_{k+1,k}| < 0.01\,$`eps`, the Krylov space is invariant) ends the current cycle with the solution in that space; if it is not converged (singular matrix), the next cycle starts from the new residual. Calls `laerror` for a 1×1 system (`ncols()==1`), for incompatible sizes and for an exactly singular triangular factor. The scalars are `double`, so only real `T` is supported.

`template<class T> void gmres_backsubstitute(const NRMat<T> &R, NRVec<T> &c, const NRVec<T> &d, const int k)`

Auxiliary: solves the upper triangular system $R_{0..k,0..k}\,c = d$ (calls `c.copyonwrite()`; `c` must be allocated).

`template<typename T> void gmres_update(NRMat<T> &H, const NRVec<T> &cci, const NRVec<T> &ssi, NRVec<T> &d, NRVec<T> &c, const int k, NRVec<T> *v, AuxStorage<T> *st, const int incore, NRVec<T> &x)`

Auxiliary: applies the Givens rotations $0..k$ to the Hessenberg matrix `H` (in place), solves $Rc = -d$ and adds $\sum_{i=0}^{k} c_i v_i$ to `x`.

9.7 DIIS convergence acceleration

Header: `diis.h`

Direct inversion in the iterative subspace [2, 3]: given the last m iterates x_i of some iterative procedure and their error vectors e_i, the extrapolated vector is $\sum_i c_i x_i$ with the coefficients minimizing $|\sum_i c_i e_i|$ subject to $\sum_i c_i = 1$, obtained from the linear system

$$\begin{pmatrix} 0 & -1 & \cdots & -1 \\ -1 & B_{11} & \cdots & B_{1m} \\ \vdots & \vdots & & \vdots \\ -1 & B_{m1} & \cdots & B_{mm} \end{pmatrix} \begin{pmatrix} \lambda \\ c_1 \\ \vdots \\ c_m \end{pmatrix} = \begin{pmatrix} -1 \\ 0 \\ \vdots \\ 0 \end{pmatrix}, \qquad B_{ij} = e_i \cdot e_j. \tag{9.1}$$

`template<typename T, typename U> class DIIS`

`T` is the type of the iterated quantity, `U` the type of the error vector, typically `NRVec`, `NRMat` or `NRSMat` (e.g. `T=NRSMat<double>` for a Fock or density matrix and `U=NRMat<double>` for the commutator error). Both template arguments must be given (there is no default). Requirements: `T` and `U` need `operator|=`, `clear()`, `axpy()`, `copyonwrite()` and the traits `LA_traits<>::elementtype` and `normtype`; `U` also `dot()` and `norm()`. The element types of `T` and `U` must be the same (the matrix of eq. (9.1) is built over the element type of `U` and converted to that of `T`). For out-of-core storage `T` and `U` must be `NRVec`, `NRMat` or `NRSMat` of plain data (`AuxStorage`) and keep their size.

`DIIS()`

Uninitialized object (for arrays of `DIIS`); `setup()` must be called before use, otherwise `extrapolate` calls `laerror`.

`DIIS(const int n, const bool core=1)`

Subspace of at most `n` stored iterates, kept in memory (`core=true`) or in two `AuxStorage` scratch files.

`void setup(const int n, const bool core=1)`

(Re)initialize as by the constructor; the previous storage is released and all stored vectors are forgotten. Can be called repeatedly.

`void reset()`

Forget all stored vectors (the next `extrapolate` starts a new subspace) but keep the allocated storage and the dimension. Used to restart DIIS, e.g. after a step judged unreasonable (`GDIIS_Optimizer`, chapter 10). After `reset()` the object behaves exactly as a freshly constructed one (tested in testoptimize.cc).

`typename LA_traits<U>::normtype extrapolate(T &vec, U &errvec, bool verbose=false, const Un diiseps=0, bool errvecout=false)`

Stores deep copies of `vec` and `errvec` as the newest pair (when `n` pairs are already stored, the oldest is dropped: circular buffer), updates B with the products `errvec.dot(`e_i`)` and solves eq. (9.1) for the current number m of stored pairs (`linear_solve` for `NRSMat`, chapter 8). `vec` is overwritten by $\sum_i c_i x_i$, and with `errvecout=true` also `errvec` by $\sum_i c_i e_i$ (the estimated error of the extrapolated vector). `diiseps` (type `LA_traits<U>::normtype`) is added to the diagonal element B_{mm} of the new vector as a regularization. `verbose` prints the solution vector $(\lambda, c_1, \ldots, c_m)$. Returns |`errvec`| of the input error vector. On the first call after construction, `setup` or `reset` the pair is only stored, `vec` and `errvec` are unchanged, and the return value is 1.

Typical use: in each iteration compute the new iterate x and its error e (e.g. $e = x_{\text{new}} - x_{\text{old}}$ for a fixed-point iteration, or the commutator $FDS - SDF$ in SCF), call `diis.extrapolate(x,e)` and continue with the extrapolated x. The stored copies are independent of the user's objects (deep copies by `|=`), so the user may modify `vec` and `errvec` freely afterwards.

For complex `U` the matrix $B_{ij} = e_i^\dagger e_j$ is hermitian: `errvec.dot(`e_i`)`$= e_{\text{new}}^\dagger e_i$ is stored in the lower triangle of the complex `NRSMat`, which is hermitian in arithmetic (chapter 6), and the coefficients c_i are complex (tested on a complex fixed-point iteration, in core and out of core).

9.8 Search in sorted arrays

Header: `bisection.h`

The header is included by `la.h`; included alone it needs `laerror` declared before (it does not include `laerror.h`).

```
template<typename INDEX, typename COMPAR, typename SUBJECT> INDEX
    bisection_find(INDEX dm, INDEX hm, const SUBJECT *key, const
    SUBJECT *base, unsigned int lead_dimension_base, COMPAR
```

`(*cmp)(const SUBJECT *, const SUBJECT *))`

Bisection search of `*key` among the elements `base[i*lead_dimension_base]`, $i = \texttt{dm}, \ldots, \texttt{hm}$ (inclusive), which must be sorted in ascending order with respect to `cmp`. `cmp(a,b)` returns 0 if equal, a positive value if `*a` > `*b`, negative otherwise; it is always called with the array element as the first argument. `lead_dimension_base` allows searching e.g. the first column of a row-major matrix (`SUBJECT` is the element type, the key is compared with one element only). Returns the index i of a matching element, or `dm-1` if not found (also for `dm>hm`). `INDEX` should be a signed type (for `dm=0` the failure value is -1).

```
template<typename INDEX, typename DISTANCE, typename SUBJECT> int
    interpolation_find(INDEX dm0, INDEX high, const SUBJECT *x,
    const SUBJECT *base, unsigned int lead_dimension_base, DISTANCE
    (*dist)(const SUBJECT *, const SUBJECT *))
```

Interpolation search with the same layout and result convention (index or `dm0-1`). `dist(a,b)` is a signed distance $a - b$ (positive if `*a` > `*b`, zero if equal); the next probe is placed by linear interpolation, so the speed depends on how well `dist` reflects the distribution of the keys. Calls `laerror` if an interpolated index falls outside the current interval (inconsistent `dist`).

9.9 Examples

9.9.1 Davidson with a user-defined direct matrix

The matrix $H_{ij} = d_i\delta_{ij} + c\,(\delta_{i,j+1} + \delta_{i+1,j})$ is never stored; the class provides only the functions of table 9.1. The program was compiled and run; the three eigenvalues agree with dense diagonalization and the residual norms $|Hv - \lambda v|$ are 10^{-12}–10^{-10} for `eps=1e-8`.

```
#include "la.h"
using namespace LA;

//H = diag(d) + c*(nearest neighbour coupling), never stored
class Chain {
        NRVec<double> d;
        double c;
public:
        Chain(int n, double cc) : d(n), c(cc) {for(int i=0; i<n;
    ++i) d[i]=i+0.1*i*i;}
        int nrows() const {return d.size();}
        int ncols() const {return d.size();}
        bool issymmetric() const {return true;}
        const double *diagonalof(NRVec<double> &r, const bool
    divide=false, bool cache=false) const
                {
```

```
                if(divide) {r.copyonwrite(); for(int i=0; i<r.
    size(); ++i) r[i]/=d[i]; return NULL;}
                r|=d; return &r[0]; //deep copy, r may share data
     with a stored Krylov vector
                }
           void gemv(const double beta, NRVec<double> &r, const char
     trans, const double alpha, const NRVec<double> &x) const
                {
                int n=d.size();
                if(beta==0) r=0.; else r*=beta;
                r.copyonwrite(); //before writing r[i]
                for(int i=0; i<n; ++i)
                        {
                        double t=d[i]*x[i];
                        if(i>0) t+=c*x[i-1];
                        if(i<n-1) t+=c*x[i+1];
                        r[i]+=alpha*t;
                        }
                }
};

int main()
{
const int n=2000, nroots=3;
Chain h(n,0.5);
NRVec<double> e(nroots);
NRVec<double> *v = new NRVec<double>[nroots];
davidson(h,e,v,NULL,nroots,false,1e-8);            //vectors in
    memory
std::cout << "eigenvalues "<<e;
davidson(h,e,v,NULL,nroots,false,1e-8,false);      //vectors in
    scratch files
lanczos(h,e,v,NULL,nroots,false,1e-10);            //the same by
    Lanczos
NRMat<double> hd = explicit_matrix<double>(h);     //for checking
     by dense diagonalization
delete[] v;
}
```

The `gemv` above ignores `trans` since the matrix is symmetric; for `conjgrad` or `gmres` on a non-symmetric direct matrix, `'t'` has to be implemented as well (or `doguess=false` used with a square matrix).

9.9.2 Linear systems and DIIS

```
NRMat<double> a(n,n); ...          //non-symmetric, diagonally
    dominant
NRVec<double> b(n), x(n); ...
bool ok=gmres(a,b,x,true,1e-10,50,false);          //x: solution
```

```
NRSMat<double> s(n); ...           //symmetric positive definite
ok=conjgrad(s,b,x,true,1e-10,1000,false);

//DIIS acceleration of a fixed-point iteration x=f(x) (from
    testoptimize.cc)
DIIS<NRVec<double>,NRVec<double> > diis(4);
NRVec<double> x(0.,n);
for(int it=0; it<maxit; ++it)
        {
        NRVec<double> xn=f(x);
        NRVec<double> err=xn-x;
        diis.extrapolate(xn,err);      //xn becomes the
    extrapolated vector
        x=xn;
        }
```

t.cc contains further calls: `davidson` on `NRMat`, `NRSMat`, `SparseMat` (symmetric and non-symmetric, out of core), `lanczos` and `davidson` compared with dense diagonalization, and `gmres` on dense and sparse matrices. testoptimize.cc tests the in-core and out-of-core `DIIS` and `reset()`.

9.10 Pitfalls

- A user `gemv` or `diagonalof` writing into `r` via `operator[]` without `r.copyonwrite()` corrupts the stored Krylov vectors (sec. 9.2).
- A literal `NULL` for `eivecs`, `initguess` or `target` breaks template argument deduction of `davidson`/`lanczos`; cast it or write `davidson<double>(...)`.
- `NRMat::issymmetric()` is `false`: `davidson` on an `NRMat` always uses the non-symmetric algorithm, and `lanczos` refuses it. Use `NRSMat` (or a `SparseMat` with the symmetric flag) for symmetric matrices.
- The Davidson threshold `eps` bounds the last expansion coefficient, the Lanczos one the change of the eigenvalues, the CG one the norm of the search direction (absolute) and the GMRES one the residual relative to the initial one.
- `davidson` with several roots fails if `maxkrylov` is reached after the first root has converged; `lanczos` has no restart at all.
- The diagonal used for preconditioning must have no small elements of the wrong sign (zero elements are skipped by the library classes); in `davidson` the denominators $|A_{ii} - \lambda|$ are bounded by 0.1 from below.
- `doguess=true` (the default of `conjgrad` and `gmres`) discards the contents of `x`.

- Scratch files of the out-of-core variants are created in the current directory and survive an exception.
- `DIIS` is not copyable (copy constructor and assignment are deleted, it owns raw storage); use references or pointers, or default-construct it in arrays (`NRVec<DIIS<...> >`) and call `setup()`.

Chapter 10

Optimization

Header: `optimize.h, optimize.cc`

The optimization module provides local minimization of a function $f : \mathbb{R}^n \to \mathbb{R}$ (and, for two methods, the search for saddle points) by a family of methods sharing one interface: quasi-Newton methods with line search (BFGS, DFP, SR1, L-BFGS), nonlinear conjugate gradients, GDIIS, trust-region Newton and quasi-Newton methods, partitioned rational function optimization (P-RFO) for saddle points, Levenberg-Marquardt for least squares, the derivative-free methods of Nelder and Mead, Powell and COBYLA (with nonlinear inequality constraints), and the first-order and stochastic methods SGD, Adam, SPSA, the quantum natural gradient and Rotosolve used in machine learning and for variational quantum algorithms.

The design separates three things:

- the *problem*: a class derived from `OptProblem<T>` implementing `funcgrad()`, optionally also deriving from *capability* interfaces (Hessian, metric, trigonometric structure, constraints, residuals) which some methods require or exploit (section 10.1);
- the *method*: a class derived from `Optimizer<T>` with the single virtual function `minimize()`;
- the *hyperparameters*: plain value objects of a class hierarchy rooted in `hyperparameters<T>`, one derived class per method, passed to `minimize()` and returned filled with the values actually reached (section 10.2).

Methods are written on top of a small infrastructure (evaluation counter, iteration bookkeeping, line searches) and of three common drivers (line search, first order, trust region), so that a new method usually consists of a short class supplying the search direction, the update rule or the trust-region step (section 10.3).

All templates are parameterized by the real type `T`, but optimize.cc instantiates them for `T = double` only. The capability interfaces, adapters and the hyperparameter classes are header-only templates; the optimizers, the adapters with member functions in optimize.cc (`FrozenOptProblem`, `ParameterShift`, `FiniteDifference`, `OptLeastSquares`) and the auxiliary functions exist in the library only for `double`. The common drivers `linesearch_minimize`, `firstorder_minimize` and `trustregion_minimize`, the extension points for user-written methods, are explicitly instantiated for `double` as well.

The data members of all classes of this chapter are small, except vectors held by `FrozenOptProblem` and `optevaluator`; these are ordinary `NRVec` objects following the reference counting and copy-on-demand scheme of section 3.1, which has consequences for user code described in section 10.6. The library code of this module calls `copyonwrite()` (or works on freshly allocated vectors) before every element write; the only weak point found is noted in section 10.1.4.

10.1 Problem interface and capabilities

10.1.1 OptProblem

`template<typename T> class OptProblem`

Abstract base class of every optimization problem. A user problem derives from it (and possibly from capability classes) and stores any auxiliary data needed to evaluate f.

`virtual T funcgrad(const NRVec<T> &x, NRVec<T> *grad) const =0`

Returns $f(x)$. If `grad!=NULL`, also stores $\nabla f(x)$ in `*grad`. The gradient vector is allocated by the caller with `x.size()` elements, but it may share its data with another vector, so the implementation must call `grad->copyonwrite()` before writing its elements (or assign a whole new vector to `*grad`). Derivative-free methods always pass `NULL`; a problem not able to provide the gradient may then simply ignore `grad` (or call `laerror` if it is not `NULL`) and be wrapped in `FiniteDifference` or `ParameterShift` for gradient methods.

10.1.2 Capabilities

Further information about the problem is supplied by deriving the problem class also from one or more of the following abstract classes (multiple inheritance). Methods look them up at run time by `optcapability<C>(problem)`; a method requiring a capability calls `laerror` at the start if it is missing, a method that can only use it optionally falls back silently (with a verbose message in some cases). Matrices

and vectors passed to the capability functions are allocated by the caller with the right dimensions; implementations writing their elements have to call `copyonwrite()` first (or assign whole objects, e.g. `h=(T)0;` followed by element writes, since assignment of a scalar is a modifying member function).

`template<typename T> class OptHessian`
: Hessian of f. Used by `TrustRegion_Optimizer` and `PRFO_Optimizer`.

`virtual void hessian(const NRVec<T> &x, NRSMat<T> &h) const =0`
: Hessian at `x` into `h` ($n \times n$ symmetric, allocated by the caller).

`virtual NRVec<T> hessian_times(const NRVec<T> &x, const NRVec<T> &v) const`
: Hessian-vector product $H(x)\,v$; the default builds the full Hessian by `hessian()` and multiplies. It can be overridden for large problems. Currently no method calls it (the Steihaug subproblem works with the stored matrix B; `FrozenOptProblem` forwards it); it is kept for future Hessian-free methods.

`template<typename T> class OptMetric`
: Metric tensor of the parameter space: the Fubini-Study metric for the quantum natural gradient [27] or the Fisher information for the natural gradient in machine learning [28]. Used by `QNG_Optimizer` and by every first-order method with `my_natural=true`.

`virtual void metric(const NRVec<T> &x, NRSMat<T> &g) const =0`
: Metric $G(x)$ into `g` ($n \times n$).

`template<typename T> class OptTrigonometric`
: Declares that f as a function of the single parameter x_i is a trigonometric polynomial $a_0 + \sum_{k=1}^{R_i} \big(a_k \cos(k\omega_i x_i) + b_k \sin(k\omega_i x_i)\big)$, $\omega_i = 2\pi/\mathrm{period}_i$. For parameterized quantum circuits with gates $\exp(-\mathrm{i}x_i P/2)$, P a Pauli string, $R_i = 1$ and the period is 2π. Used by `Rotosolve_Optimizer` and `ParameterShift`.

`virtual int degree(const int i) const =0`
: Degree R_i; 0 means that f is not trigonometric in x_i (such parameters are treated by line minimization or central differences).

`virtual T period(const int i) const`
: Period of f in x_i; default 2π.

`template<typename T> class OptConstraints`
: Inequality constraints $c_i(x) \geq 0$, $i = 0, \ldots, m-1$, used by `Cobyla_Optimizer`. An equality constraint is given as two inequalities $c(x) \geq 0$, $-c(x) \geq 0$. Other methods ignore constraints.

`virtual int nconstraints() const =0`
: Number m of constraints.

`virtual void constraints(const NRVec<T> &x, NRVec<T> &c) const =0`
: Values $c_i(x)$ into `c`, allocated by the caller with `nconstraints()` elements.

`template<typename T> class OptResiduals`
: Least-squares problem $f(x) = \frac{1}{2}|r(x)|^2$ given by the residual vector $r \in \mathbb{R}^m$ and optionally its Jacobian. Required by `LM_Optimizer`.

`virtual int nresiduals() const =0`
: Number m of residuals.

`virtual void residuals(const NRVec<T> &x, NRVec<T> &r) const =0`
: Residuals into `r` (allocated by the caller with `nresiduals()` elements).

`virtual bool jacobian(const NRVec<T> &x, NRMat<T> &J) const`
: Jacobian $J_{ij} = \partial r_i/\partial x_j$ into `J` ($m \times n$, allocated by the caller); returns `true` if provided. The default returns `false`, and the callers then use forward differences.

`template<class C, typename T> const C *optcapability(const OptProblem<T> *o)`
: Returns the capability `C` (e.g. `OptHessian<T>`) of the problem, or `NULL`. It is found by `dynamic_cast` on the problem itself and, if the problem is an `OptProblemAdapter`, recursively on the wrapped problem. For an adapter with `transforms_capabilities()==true` the adapter's own implementation of `C` is returned only if the wrapped problem also provides `C`, and the untransformed capability of the wrapped problem is never passed on. Methods always use this function (not a plain `dynamic_cast`), so that capabilities remain available through adapters.

10.1.3 Least-squares problems: OptLeastSquares

`template<typename T> class OptLeastSquares : public OptProblem<T>, public OptResiduals<T>`
: Convenience base class for least-squares problems. The user derives from it and implements `nresiduals()`, `residuals()` and optionally `jacobian()`. Its `funcgrad()` provides $f = \frac{1}{2}|r|^2$ and $\nabla f = J^{\mathsf{T}} r$, so that every method, not only `LM_Optimizer`, can be applied to the problem.

`T fd_step`
: Relative step h of the forward-difference Jacobian used when `jacobian()` returns `false`.

`OptLeastSquares(const T h=1e-7)`
: Sets `fd_step`.

`void jacobian_fd(const NRVec<T> &x, const NRVec<T> &r, NRMat<T> &J)`

`const`
: Jacobian by forward differences, columns $(r(x + h_j e_j) - r(x))/h_j$ with $h_j = h \max(1, |x_j|)$; `r` are the residuals at `x` (n further residual evaluations).

`virtual T funcgrad(const NRVec<T> &x, NRVec<T> *grad) const`
: $\frac{1}{2}|r(x)|^2$; with `grad`, the gradient $J^\mathsf{T} r$ (from `jacobian()` or `jacobian_fd()`).

Note that `LM_Optimizer` does not call `funcgrad()`: it evaluates `residuals()` directly (counted as function evaluations) and uses `LM_hyperparameters::my_fd_step`, not `fd_step`, for its own finite-difference Jacobian.

10.1.4 Adapters

An adapter is a problem wrapping another one, e.g. to supply gradients computed from function values or to optimize in a subspace. Adapters can be nested.

`template<typename T> class OptProblemAdapter : public OptProblem<T>`
: Base class of adapters; `funcgrad()` is left to the derived class.

`const OptProblem<T> *p`
: The wrapped problem (not owned).

`OptProblemAdapter(const OptProblem<T> *pp)`
: Wraps `pp`.

`virtual bool transforms_capabilities() const`
: `false` (default): the capabilities of the wrapped problem are accessible unchanged through `optcapability()`. `true`: the adapter provides them in a transformed form (e.g. in a different dimension) by its own implementations, see `optcapability()`.

FiniteDifference.

`template<typename T> class FiniteDifference : public OptProblemAdapter<T>`
: Gradient by central differences $(f(x + he_i) - f(x - he_i))/(2h)$, i.e. $2n$ evaluations of the wrapped problem besides $f(x)$. Makes all gradient methods usable for a problem providing only f. Capabilities of the wrapped problem pass through.

`T h`
: Difference step (absolute).

`FiniteDifference(const OptProblem<T> *pp, const T hh=1e-6)`
: Wraps `pp`.

`virtual T funcgrad(const NRVec<T> &x, NRVec<T> *grad) const`
: f of the wrapped problem; the gradient by central differences if requested.

ParameterShift.

`template<typename T> class ParameterShift : public OptProblemAdapter<T>`
: Exact gradient of a problem providing `OptTrigonometric` by the general parameter-shift rule [24] (for degree 1 the classical two-point rule [25, 26]): for x_i of degree R, $\partial_i f = \omega \sum_{\mu=1}^{2R} (-1)^{\mu-1} f(x + s_\mu e_i/\omega)/(4R \sin^2(s_\mu/2))$, $s_\mu = (2\mu - 1)\pi/(2R)$, i.e. $2R_i$ evaluations per parameter. Parameters of degree 0 are differentiated by central differences with step `h`. Capabilities of the wrapped problem pass through.

`const OptTrigonometric<T> *trig`
: The trigonometric capability of the wrapped problem.

`T h`
: Central-difference step for parameters of degree 0.

`ParameterShift(const OptProblem<T> *pp, const T hh=1e-6)`
: Wraps `pp`; `laerror` if it does not provide `OptTrigonometric`.

`virtual T funcgrad(const NRVec<T> &x, NRVec<T> *grad) const`
: f of the wrapped problem and the parameter-shift gradient.

FrozenOptProblem. Optimization in a subset of the parameters, the others ("frozen") keeping their values.

`template<typename T> class FrozenOptProblem : public OptProblemAdapter<T>, public OptHessian<T>, public OptMetric<T>, public OptTrigonometric<T>, public OptConstraints<T>`
: Problem in the n_{free} free parameters. Its `transforms_capabilities()` returns `true`: the Hessian and metric (submatrices for the free parameters), the Hessian-vector product, degree and period (of the free parameters) and the constraints are provided if and only if the wrapped problem provides them. `OptResiduals` is not forwarded, so `LM_Optimizer` cannot be used through it (the other methods can, via `funcgrad()` of an `OptLeastSquares` problem).

`NRVec<T> xfull`
: Full parameter vector; the values of the frozen parameters are taken from it.

`NRVec<int> freeindex`
: Indices of the free parameters in the full vector (ascending).

`FrozenOptProblem(const OptProblem<T> *pp, const NRVec<T> &x, const bitvector &frozen)`
: Wraps `pp`; `xfull` shares its data with `x` (shallow copy); `frozen` has bit i set if x_i is frozen.

`void setfrozen(const bitvector &frozen)`
: New set of frozen parameters (`laerror` if `frozen.size()!=xfull.size()`).

`void setfull(const NRVec<T> &x)`
: New full vector (new values of the frozen parameters); shallow copy.

`NRVec<T> reduce(const NRVec<T> &full) const`
: The free parameters of a full vector, `full.subvector(freeindex)`.

`NRVec<T> expand(const NRVec<T> &reduced) const`
: Full vector: a copy of `xfull` with the free parameters replaced by `reduced`.

`virtual T funcgrad(const NRVec<T> &x, NRVec<T> *grad) const`
: f at `expand(x)`; the gradient is the free part of the full gradient.

`virtual void hessian(const NRVec<T> &x, NRSMat<T> &h) const; virtual NRVec<T> hessian_times(const NRVec<T> &x, const NRVec<T> &v) const`
: Free-free block of the wrapped Hessian; product with the free block.

`virtual void metric(const NRVec<T> &x, NRSMat<T> &g) const`
: Free-free block of the wrapped metric.

`virtual int degree(const int i) const; virtual T period(const int i) const`
: Of the free parameter i, i.e. of `freeindex[i]` of the wrapped problem.

`virtual int nconstraints() const; virtual void constraints(const NRVec<T> &x, NRVec<T> &c) const`
: Constraints of the wrapped problem at `expand(x)`.

Usage: `FrozenOptProblem<T> fp(&problem,x,frozen); NRVec<T> xr=fp.reduce(x); optimizer.minimize(&fp,xr); x=fp.expand(xr);` (see example 10.5.2).

10.2 Optimizers, hyperparameters and results

10.2.1 Optimizer and optresult

`template<typename T> class Optimizer`
: Abstract base class of all methods; the methods can be used through `Optimizer<T> *` or `&`. Each method class (`BFGS_Optimizer` etc.) has no data members and a virtual destructor.

`virtual optresult<T> minimize(const OptProblem<T> *o, NRVec<T> &x, const hyperparameters<T> *par=NULL, hyperparameters<T> *achieved=NULL) =0`

Minimizes `o` (for P-RFO: searches a stationary point of the requested order) starting from `x`, which is replaced by the result. `par==NULL` means the default parameters of the method; otherwise the parameters are taken from `*par` as described in section 10.2.3. If `achieved!=NULL`, it is filled with the parameters used and with the values actually reached (section 10.2.4).

`template<typename T> class optresult`

Result of `minimize()`.

`T f; bool converged`

Final function value (for SPSA an estimate); whether a convergence criterion was satisfied (`false` if the iterations stopped on `my_maxstep`, `my_maxfunc` or a failure).

`optresult(const T ff, const bool c); operator T() const`

Constructor; implicit conversion to `f`, so `double f=o.minimize(&p,x);` works.

10.2.2 Hyperparameter hierarchy

The hyperparameters form a class hierarchy; each method has its own class adding its specific parameters to those of the general classes. All members are public, named `my_...`, and initialized to the method's defaults by the constructor. The hierarchy (defaults in parentheses; a derived class may change inherited defaults, given at the method):

```
hyperparameters                    maxstep(200) maxfunc(0) f_thr
    (0) x_thr(0) saddle_order(0) verbose(1)
 +- gradient_hyperparameters       g_thr(1e-6)
 |   +- linesearch_hyperparameters    c1(1e-4) c2(0.9)
    maxlinesearch(30) maxsteplength(1e10)
 |   |   +- quasinewton_hyperparameters   sy_thr(1e-12)
 |   |   |   +- BFGS_hyperparameters         update(QN_BFGS)
    sr1_thr(1e-8)
 |   |   |   |   +- GDIIS_hyperparameters       diis_dim(5)
    diis_eps(0)
 |   |   |   +- LBFGS_hyperparameters        memory(10)
 |   |   +- CG_hyperparameters            beta(CG_PR) restart(0)
    powell_thr(0.2); c2=0.1
 |   +- firstorder_hyperparameters    lr(1e-3) lr_decay(0)
    lr_power(1) weight_decay(0) return_best(false)
 |   |   |                            natural(false) metric_reg(1
    e-3); maxstep=1000
 |   |   +- SGD_hyperparameters           momentum(0.9) nesterov(
    false); lr=1e-2
```

```
|   |   |   +- QNG_hyperparameters          momentum=0 natural=
   true lr=0.05
|   |   +- Adam_hyperparameters          beta1(0.9) beta2(0.999)
    eps(1e-8) amsgrad(false)
|   |   +- SPSA_hyperparameters          c(0.1) gamma(0.101)
   resamplings(1) calibrate(20) target_step(0.1);
|   |                                    lr_decay=0.01 lr_power
   =0.602 g_thr=0
|   +- trustregion_hyperparameters   delta0(1) deltamax(1e3) eta
   (1e-4) hessian(TR_AUTO) hessian_recompute(1)
|   |   |                              update_thr(1e-8)
|   |   +- TrustRegion_hyperparameters   subproblem(TR_EIGEN)
   maxcg(0)
|   |   +- PRFO_hyperparameters          delta0=0.3
|   +- LM_hyperparameters            tau(1e-3) scaling(false)
   fd_step(1e-7)
+- NelderMead_hyperparameters     initstep(0.1) adaptive(true)
   alpha(1) gamma(2) rho(0.5) sigma(0.5);
|                                  maxstep=5000 f_thr=1e-8 x_thr
   =1e-6
+- Powell_hyperparameters         initstep(0.1) linmin_tol(1e-4)
    maxlinemin(50); maxstep=1000 f_thr=1e-10 x_thr=1e-6
|   +- Rotosolve_hyperparameters     reevaluate(false)
+- Cobyla_hyperparameters         rhobeg(0.5) rhoend(1e-6)
   flexible_radius(true); maxstep=100000
```

Base classes.

`template<typename T> class hyperparameters`
: Parameters common to all methods.

`int my_maxstep`
: Maximum number of iterations (default 200). What an iteration is depends on the method (a line search step, a sweep over all directions or parameters, a simplex operation, a trust-region step including rejected ones).

`int my_maxfunc`
: Maximum number of function evaluations, ≤ 0 = unlimited (default 0). It counts every call of `funcgrad()` by the method (with or without gradient), and for LM every residual evaluation, including those of finite-difference Jacobians. Evaluations inside adapters (finite differences, parameter shifts) are not counted. Methods check before each iteration that the budget suffices and never exceed it, except for the unavoidable initial evaluations (the starting point, the initial simplex of Nelder-Mead and COBYLA, the first SPSA estimate).

`T my_f_thr; T my_x_thr`

Thresholds on $|\Delta f|$ and $|\Delta x|$ (Euclidean norm) in the last step (default 0). The general criterion is $|\Delta f| <$ `my_f_thr` *and* $|\Delta x| <$ `my_x_thr`, both strict, so with the default 0 it is switched off, and setting only one of them has no effect. Some methods reinterpret them (Nelder-Mead, Powell, Rotosolve, COBYLA, see there).

`int my_saddle_order`

0 = minimum (default), $k > 0$ = saddle point with k negative Hessian eigenvalues. Only `PRFO_Optimizer` accepts $k > 0$; all other methods call `laerror` if it is nonzero. For the trust-region methods it returns the Hessian index in `achieved`.

`int my_verbose`

0 = silent, 1 = one line per iteration and a final report (default), 2 = more details (line searches, step lengths, trust-region radii, restarts). Output goes to `std::cout`.

`hyperparameters()`

Defaults as above.

`virtual void setfrom(const hyperparameters &p)`

Copies the parameters from `p` which exist in both classes (section 10.2.3).

`virtual bool converged(const optstate<T> &s) const`

Convergence test: `s.stepdone && s.df<my_f_thr && s.dx<my_x_thr`.

`virtual void setreached(const optstate<T> &s)`

Stores the reached values: `my_maxstep` = iterations done, `my_maxfunc` = evaluations done, `my_f_thr`, `my_x_thr` = $|\Delta f|$, $|\Delta x|$ of the last step.

`template<typename T> class gradient_hyperparameters : public hyperparameters<T>`

Methods using the gradient.

`T my_g_thr`

Convergence threshold on the gradient norm (default `1e-6`). `converged()` is $|g| <$ `my_g_thr` *or* the criterion of the base class; `setreached()` stores the last $|g|$ (or -1 if the gradient norm is not available, as for SPSA).

`template<typename T> class linesearch_hyperparameters : public gradient_hyperparameters<T>`

Methods with the strong Wolfe line search.

`T my_c1; T my_c2`

Sufficient decrease (Armijo) and curvature parameters of the strong Wolfe conditions, $0 < c_1 < c_2 < 1$ (defaults `1e-4`, 0.9).

`int my_maxlinesearch`

Maximum number of function evaluations in one line search (30).

`T my_maxsteplength`
: Maximum norm of a single step in x (`1e10`).

`template<typename T> class firstorder_hyperparameters : public gradient_hyperparameters<T>`
: First-order methods with a learning rate instead of a line search, suitable also for noisy functions and gradients. Default `my_maxstep=1000`.

`T my_lr; T my_lr_decay; T my_lr_power`
: Learning rate in iteration $k = 0, 1, \ldots$ is $\mathrm{lr}/(1+\mathrm{lr_decay}\cdot k)^{\mathrm{lr_power}}$ (defaults `1e-3`, 0, 1).

`T my_weight_decay`
: Decoupled weight decay $x \mathrel{-}= \mathrm{lr}\cdot\mathrm{weight_decay}\cdot x$ [32] (default 0).

`bool my_return_best`
: Return the point with the lowest function value seen instead of the last one (default `false`); only meaningful for (nearly) deterministic f, the lowest of noisy values is biased.

`bool my_natural`
: Use the natural gradient $(G + \mathrm{metric_reg}\, I)^{-1} g$ with the metric G of the problem (`OptMetric` required) [28, 27]; $|g|$ in the convergence test and in the output then refers to the natural gradient (default `false`).

`T my_metric_reg`
: Regularization added to the diagonal of the metric (`1e-3`).

`template<typename T> class trustregion_hyperparameters : public gradient_hyperparameters<T>`
: Trust-region methods, see section 10.3.5.

The method-specific classes are described with their methods in section 10.4. Each of them has a default constructor and overrides `setfrom()`.

10.2.3 How parameters are taken: setfrom()

Every `minimize()` constructs the hyperparameter object of its own method (with the method's defaults) and calls `hp.setfrom(*par)`. Each class's `setfrom()` tries `dynamic_cast` of `par` to its own type; on success it assigns `*this=*q` at its own level, otherwise it delegates to its parent class. The effect is that all fields of the deepest class common to the method's class and the dynamic type of `*par` are copied from `*par`, while the method-specific fields below it keep the method's defaults. Thus one may pass

- the method's own class (all parameters given),
- a more general class, e.g. `gradient_hyperparameters<double>` to any gradient method, or `hyperparameters<double>` to any method,

- the class of another method: e.g. `CG_hyperparameters` passed to `BFGS_Optimizer` provides the line search parameters (including CG's default `my_c2=0.1`).

Caution: All fields of the common level are copied, including those the caller left at the defaults *of that level*, which may differ from the defaults of the method. For example, a plain `hyperparameters<double>` object passed to `NelderMead_Optimizer` replaces its `my_maxstep=5000`, `my_f_thr=1e-8`, `my_x_thr=1e-6` by 200, 0, 0 (no convergence test); `gradient_hyperparameters` passed to Adam replaces `my_maxstep=1000` by 200; `SGD_hyperparameters` passed to QNG brings `my_momentum=0.9`. Set those fields explicitly, or use the method's own class.

A derived hyperparameter class written for a new method must override `setfrom()` in the same way (one line, see the header), and `converged()`, `setreached()` if it adds parameters relevant for them.

10.2.4 The achieved values

If `achieved!=NULL`, at the end `achieved->setfrom(hp)` copies the parameters used (at the common level of the two classes, as above) and `achieved->setreached()` overwrites the limits and thresholds with the values reached:

- `my_maxstep`: number of iterations done; `my_maxfunc`: number of function evaluations done;
- `my_f_thr`, `my_x_thr`: $|\Delta f|$ and $|\Delta x|$ of the last step (0 if no step was done); for Nelder-Mead the spread of f and the diameter of the simplex, for COBYLA the change of f and the radius ρ;
- `my_g_thr` (if `achieved` is at least a `gradient_hyperparameters`): the last gradient norm (-1 if not available);
- `my_saddle_order` (trust-region methods only): the number of negative eigenvalues of the Hessian at the result (of the exact one if the problem provides `OptHessian`, otherwise of the approximation);
- `my_lr` (first-order methods): the initial learning rate actually used (after the SPSA calibration), if `achieved` is at least of the class `firstorder_hyperparameters`;
- the other fields: the parameters as they were used.

The type of `achieved` determines what is returned: an object of the class `hyperparameters<double>` suffices for the counts, the class `gradient_hyperparameters<double>` is needed to obtain $|g|$.

10.3 Infrastructure and common drivers

This section is needed only for writing new methods; the classes are public so that methods can be added outside the library.

10.3.1 Evaluation and iteration bookkeeping

`template<typename T> class optevaluator`
: Wraps the problem: counts evaluations and keeps the best point seen.

`const OptProblem<T> *o; int maxfunc; int nfunc; T fbest; NRVec<T> xbest`
: The problem, the evaluation budget (≤ 0 unlimited), the number of evaluations done, the lowest f found and its point.

`optevaluator(const OptProblem<T> *oo, const int maxf=0)`
: Constructor.

`T operator()(const NRVec<T> &x, NRVec<T> *grad=NULL)`
: `o->funcgrad(x,grad)`, counted; if f is the lowest so far, `xbest=x` (a shallow copy sharing the data of `x`, see section 10.6).

`T operator()(const OptResiduals<T> *res, const NRVec<T> &x, NRVec<T> &r)`
: Residuals at `x` into `r`, returns $\frac{1}{2}|r|^2$; counted and recorded as above.

`bool exhausted() const; int remaining() const`
: Budget used up; number of evaluations left (`std::numeric_limits<int>::max()` if unlimited).

`template<typename T> class optstate`
: Iteration bookkeeping common to all methods: convergence and limit tests, verbose output, result.

`const char *name; const hyperparameters<T> &hp; const optevaluator<T> &e`
: Method name for the output, the parameters used (their virtual `converged()` is called), the evaluator.

`int iter; T f; T gnorm; T df; T dx; bool stepdone; bool converged`
: Iterations done; current f; current $|g|$ (< 0 if not available); $|\Delta f|$, $|\Delta x|$ of the last step; whether `df`, `dx` are valid; whether converged.

`optstate(const char *nm, const hyperparameters<T> &h, const optevaluator<T> &ee)`
: Constructor.

`bool stop(const T ff, const T gn= -1, const int needed=1)`
: To be called at the beginning of each iteration with the current f and $|g|$: prints the iteration line (verbose ≥ 1) and returns `true` if converged (`hp.converged()`), if `iter>=my_maxstep`, or if fewer than `needed` evaluations (those of the next iteration) remain.

`void step(const T dff, const T dxx)`
: Records a step: $|\Delta f|$, $|\Delta x|$, `stepdone=true`.

`optresult<T> finish(hyperparameters<T> *achieved) const`
: Final report (verbose ≥ 1), fills `achieved` if not `NULL`, returns the result.

10.3.2 One-dimensional searches

`template<typename T> T linesearch_wolfe(optevaluator<T> &e, NRVec<T> &x, T &f, NRVec<T> &g, const NRVec<T> &p, const T alpha1, const linesearch_hyperparameters<T> &lp)`
: Line search along the descent direction `p` satisfying the strong Wolfe conditions (bracketing and zoom with safeguarded cubic interpolation, [4] alg. 3.5, 3.6). On input `x`, `f`, `g` at the starting point, `alpha1` the initial trial step; on success they are replaced by the new point and the step length α is returned. If the strong Wolfe conditions cannot be met within `my_maxlinesearch` evaluations (or the remaining budget), the best point satisfying sufficient decrease is accepted; 0 is returned on failure (also if `p` is not a descent direction), `x`, `f`, `g` then unchanged. The step is limited to $|\alpha p| \leq$ `my_maxsteplength`.

`template<typename T> T linemin_brent(optevaluator<T> &e, NRVec<T> &x, T &f, const NRVec<T> &u, const T t1, const T tol, const T abstol, const int maxiter, const int verbose=0)`
: Derivative-free minimization of $f(x + tu)$: bracketing starting with the trial step `t1` (the opposite direction is tried if uphill), then Brent's method [6] with relative and absolute tolerances `tol`, `abstol` on t, at most `maxiter` evaluations (and the remaining budget). On output `x`, `f` at the best point found; returns t, or 0 if no decrease was found (`x`, `f` unchanged).

10.3.3 Line-search methods

`template<typename T> class linesearch_direction`
: Search direction of a line-search method, abstract.

`virtual NRVec<T> direction(const NRVec<T> &g) =0`
: Search direction at the point with gradient `g`.

`virtual bool update(const NRVec<T> &s, const NRVec<T> &y, const NRVec<T> &p, const NRVec<T> &g) =0`
: After a step $s = \alpha p$ with gradient change y and new gradient g; returns `false` if the update was skipped (only reported).

`virtual void reset() =0`
: Forget the accumulated information (restart).

`virtual bool steepest() const =0`
: No information accumulated, `direction()` gives steepest descent.

`virtual bool indefinite() const`
: `true`: a non-descent direction is legitimate (SR1); the driver then takes a steepest-descent step without reset. Default `false`.

`virtual bool alpha_from_previous() const`
: `true`: initial trial step $\alpha_{k-1}\, g_{k-1} \cdot p_{k-1}/(g_k \cdot p_k)$ ([4] eq. 3.60, used by CG); otherwise 1 (the quasi-Newton step), or a unit step in x for steepest descent. Default `false`.

```
template<typename T> optresult<T> linesearch_minimize(const char
    *name, linesearch_direction<T> &dir, const OptProblem<T> *o,
    NRVec<T> &x, const linesearch_hyperparameters<T> &hp,
    hyperparameters<T> *achieved)
```
: Common driver: in each iteration `p=dir.direction(g)`; if $p \cdot g \geq 0$, steepest descent (with `dir.reset()` unless `indefinite()`); `linesearch_wolfe()`; on its failure a restart, or the end if it failed along steepest descent; then `dir.update()` and the step is recorded. One evaluation per line search trial point.

10.3.4 First-order methods

`template<typename T> class firstorder_step`
: Update rule of a first-order method, abstract.

`virtual NRVec<T> step(const NRVec<T> &g, const int k) =0`
: Direction d of the step $x \mathrel{-}= \mathrm{lr}\, d$ in iteration $k = 1, 2, \ldots$ with gradient `g`. The gradient vector passed is freshly allocated in each iteration, so the stepper may keep it.

`template<typename T> class gradient_estimator`
: Supplies f and the gradient to the driver; the default implementation is the exact evaluation, derived classes estimate them (SPSA).

```
virtual T operator()(optevaluator<T> &e, const NRVec<T> &x, NRVec<T>
    &g, const int k)
```
: Returns f at `x` and the gradient in `g`; `k=0,1,...` is the iteration.

`virtual int cost() const`
: Number of evaluations of one call (default 1).

`virtual bool exact() const`
: Default `true`. With estimates (`false`) no convergence test is done, since small gradients or changes can occur by chance, and the iterations run until `my_maxstep` or `my_maxfunc`.

```
virtual T calibrate(optevaluator<T> &e, const NRVec<T> &x, const
    firstorder_hyperparameters<T> &hp)
```
: Learning rate to be used; default `hp.my_lr`.

```
template<typename T> optresult<T> firstorder_minimize(const char
```

*name, firstorder_step<T> &stepper, const OptProblem<T> *o, NRVec<T> &x, const firstorder_hyperparameters<T> &hp, hyperparameters<T> *achieved, gradient_estimator<T> *est=NULL)

Common driver: $x \leftarrow x - \mathrm{lr}_k\, d - \mathrm{lr}_k$ weight_decay x with d from the stepper, `est==NULL` meaning exact gradients. With `my_natural` the gradient of the estimator is replaced by the natural gradient. Implements `my_return_best`, and returns the learning rate used in `achieved` if it is a `firstorder_hyperparameters`.

10.3.5 Trust-region methods

The trust-region methods work with the quadratic model $f + g \cdot p + \frac{1}{2} p \cdot Bp$ within $|p| \le \Delta$ [4, 5]. The matrix B is the exact Hessian (`OptHessian`) or an approximation updated by a secant formula.

`enum TR_hessian {TR_AUTO, TR_EXACT, TR_BFGS, TR_SR1, TR_PSB, TR_BOFILL}`

Choice of B. `TR_EXACT`: exact Hessian (`laerror` if the problem does not provide it). `TR_AUTO`: exact if available, otherwise BFGS for minima and Bofill for saddle points. `TR_BFGS`, `TR_SR1`, `TR_PSB`, `TR_BOFILL`: updated by the BFGS, SR1, Powell-symmetric-Broyden or Bofill [10] formula starting from the identity, scaled after the first step by $y \cdot y / s \cdot y$. Between exact Hessians (`my_hessian_recompute>1`) the updates are SR1 for minima (the exact Hessian may be indefinite) and Bofill for saddle points. BFGS is refused for saddle points.

`template<typename T> bool hessian_update(NRSMat<T> &B, const NRVec<T> &s, const NRVec<T> &y, const TR_hessian type, const T thr)`

Update of a Hessian approximation B (not of its inverse) by the step s and gradient change y, $\xi = y - Bs$: BFGS $B + yy^{\mathsf{T}}/(y \cdot s) - Bs(Bs)^{\mathsf{T}}/(s \cdot Bs)$ (skipped if $s \cdot y \le \mathrm{thr}\,|s||y|$ or $s \cdot Bs \le 0$); SR1 $B + \xi\xi^{\mathsf{T}}/(\xi \cdot s)$ (skipped if $|\xi \cdot s| \le \mathrm{thr}\,|\xi||s|$); PSB $B + (\xi s^{\mathsf{T}} + s\xi^{\mathsf{T}})/(s \cdot s) - (\xi \cdot s) ss^{\mathsf{T}}/(s \cdot s)^2$; Bofill $\varphi\,\mathrm{SR1} + (1-\varphi)\,\mathrm{PSB}$ with $\varphi = (\xi \cdot s)^2/(|\xi|^2|s|^2)$ (skipped if $\xi = 0$). Returns `false` if skipped. Only `TR_BFGS`, `TR_SR1`, `TR_PSB`, `TR_BOFILL` are valid.

`T my_delta0; T my_deltamax`

(`trustregion_hyperparameters`) Initial (1) and maximal (`1e3`) trust-region radius.

`T my_eta`

A minimization step is accepted if (actual reduction)/(predicted reduction) $> \eta$ (`1e-4`).

`TR_hessian my_hessian`

See above (`TR_AUTO`).

`int my_hessian_recompute`
: With an exact Hessian: recompute it every k-th accepted step and update in between; 1 = always exact (default).

`T my_update_thr`
: Threshold for skipping the updates (`thr` of `hessian_update()`, `1e-8`).

`template<typename T> class trustregion_step`
: Step of a trust-region method, abstract.

`virtual NRVec<T> step(const NRVec<T> &g, const NRSMat<T> &B, const T delta) =0`
: Step within the trust region.

`virtual bool saddle() const`
: Searching for a saddle point: steps are always taken and the radius is controlled by the quality of the model only. Default `false`.

`template<typename T> optresult<T> trustregion_minimize(const char *name, trustregion_step<T> &stepper, const OptProblem<T> *o, NRVec<T> &x, const trustregion_hyperparameters<T> &hp, hyperparameters<T> *achieved)`
: Common driver. For minimization: radius control by [4] alg. 4.1 (ratio < 0.25: $\Delta = |p|/4$; ratio > 0.75 and step on the boundary: $\Delta = \min(2\Delta, \Delta_{\max})$); if the predicted change is below the rounding errors of f, the step is judged by the decrease of $|g|$. For saddle points: every step is taken; ratio outside $[0.25, 1.75]$ gives $\Delta = |p|/4$, ratio in $(0.75, 1.25)$ with a boundary step doubles Δ. One evaluation (with gradient) per iteration; rejected steps count as iterations. At the end the Hessian (exact if available, otherwise B) is diagonalized and the number of its negative eigenvalues is printed and returned in `achieved->my_saddle_order`.

10.3.6 Writing a new method

A new line-search method needs a `linesearch_direction` and an `Optimizer` whose `minimize()` sets up the hyperparameters and calls the driver; the same pattern applies to `firstorder_step` with the driver `firstorder_minimize()` and to `trustregion_step` with the driver `trustregion_minimize()`. A complete (compile-tested) example, steepest descent with the strong Wolfe line search:

```
template<typename T>
class SD_direction : public linesearch_direction<T> {
public:
NRVec<T> direction(const NRVec<T> &g) {NRVec<T> p(g); p*= (T)-1;
    return p;}
bool update(const NRVec<T> &s, const NRVec<T> &y, const NRVec<T>
    &p, const NRVec<T> &g) {return true;}
```

```
void reset() {}
bool steepest() const {return true;}
};

template<typename T>
class SD_Optimizer : public Optimizer<T> {
public:
optresult<T> minimize(const OptProblem<T> *o, NRVec<T> &x, const
    hyperparameters<T> *par=NULL, hyperparameters<T> *achieved=
    NULL)
        {
        linesearch_hyperparameters<T> hp; //a derived class would
     set the method's defaults
        if(par) hp.setfrom(*par);
        if(hp.my_saddle_order) laerror("SD cannot search for
    saddle points");
        SD_direction<T> dir;
        return linesearch_minimize("SD",dir,o,x,hp,achieved);
        }
};
```

Methods with their own iteration loop use `optevaluator` for all evaluations and `optstate`: `stop()` at the beginning of each iteration, `step()` after each step, `finish(achieved)` at the end.

10.4 Methods

Table 10.1 summarizes the methods. "Gradient" means that `funcgrad()` is called with a gradient request; such methods can be applied to an f-only problem through `FiniteDifference` or `ParameterShift`. "Optional" capabilities are used if present.

Every `XXX_Optimizer` class below has only the member `minimize()` with the signature of `Optimizer<T>::minimize()` and a virtual destructor; only the hyperparameters are listed. All methods except P-RFO call `laerror` if `my_saddle_order!=0`.

10.4.1 Quasi-Newton methods: BFGS, DFP, SR1

`BFGS_Optimizer` keeps an explicit approximation H of the inverse Hessian, starting from the identity scaled after the first step by $s\cdot y/y\cdot y$ ([4] eq. 6.20), and uses the strong Wolfe line search with the initial trial step 1 (the quasi-Newton step). Memory and work are $O(n^2)$ per iteration; for large n use L-BFGS.

```
template<typename T> class quasinewton_hyperparameters :  public
    linesearch_hyperparameters<T>
```

Table 10.1: Methods, their requirements and properties.

Class	Derivatives	Capabilities	Constraints, saddles	Noisy f
`BFGS_Optimizer` (BFGS, DFP, SR1)	gradient	–	no	no
`LBFGS_Optimizer`	gradient	–	no	no
`CG_Optimizer`	gradient	–	no	no
`GDIIS_Optimizer`	gradient	–	no	no
`TrustRegion_Optimizer`	gradient	`OptHessian` optional (required for `TR_EXACT`)	no	no
`PRFO_Optimizer`	gradient	`OptHessian` optional	saddles of any order	no
`LM_Optimizer`	Jacobian optional	`OptResiduals` required	no	no
`NelderMead_Optimizer`	none	–	no	partly
`Powell_Optimizer`	none	–	no	no
`Rotosolve_Optimizer`	none	`OptTrigonometric` optional	no	with `my_re-evaluate`
`Cobyla_Optimizer`	none	`OptConstraints` optional	inequality constraints	no
`SGD_Optimizer`	gradient	`OptMetric` if `my_natural`	no	yes
`Adam_Optimizer`	gradient	`OptMetric` if `my_natural`	no	yes
`QNG_Optimizer`	gradient	`OptMetric` required	no	yes
`SPSA_Optimizer`	none	`OptMetric` if `my_natural`	no	yes

`T my_sy_thr`
: Skip the update if $s \cdot y \leq \text{sy_thr}\,|s||y|$ (curvature condition violated numerically; `1e-12`). Also used by L-BFGS.

`enum QN_update {QN_BFGS, QN_DFP, QN_SR1}`
: Update formula of H ([4] eqs. 6.17, 6.15, 6.25).

```
template<typename T> class BFGS_hyperparameters :  public
    quasinewton_hyperparameters<T>
```

`QN_update my_update`
: `QN_BFGS` (default), `QN_DFP` (needs an accurate line search, `my_c2`$\approx$ 0.1, otherwise it can stall), `QN_SR1` (H may become indefinite; a non-descent direction then leads to a steepest descent step without reset). The verbose name is BFGS, DFP or SR1.

`T my_sr1_thr`
: SR1: skip the update if $|(s - Hy) \cdot y| \leq \text{sr1_thr}\,|s - Hy||y|$ ([4] eq. 6.26; `1e-8`).

```
template<typename T> class BFGS_Optimizer :  public Optimizer<T>
```

10.4.2 L-BFGS

Limited-memory BFGS ([4] sec. 7.2): the last `my_memory` pairs (s, y) in a ring buffer, the direction by the two-loop recursion (alg. 7.4) with $H_0 = \gamma I$, $\gamma = s \cdot y / y \cdot y$ of the newest pair. Memory $O(mn)$. Pairs violating the curvature condition (`my_sy_thr`) are skipped.

```
template<typename T> class LBFGS_hyperparameters :  public
    quasinewton_hyperparameters<T>
```

`int my_memory`
: Number of stored pairs (10; `laerror` if < 1).

```
template<typename T> class LBFGS_Optimizer :  public Optimizer<T>
```
: Verbose name L-BFGS.

10.4.3 Nonlinear conjugate gradients

$p_{k+1} = -g_{k+1} + \beta p_k$ ([4] sec. 5.2, [9]), with the initial trial step of the line search from the previous step. Restarts with steepest descent every `my_restart` iterations, by Powell's test, after a failed line search, and if the curvature $p \cdot y \leq 0$ for the formulas dividing by it.

`enum CG_beta {CG_FR, CG_PR, CG_HS, CG_DY, CG_HZ}`

Formula for β ($y = g_{k+1} - g_k$): Fletcher-Reeves $|g_{k+1}|^2/|g_k|^2$; Polak-Ribière+ $\max(0, g_{k+1} \cdot y/|g_k|^2)$; Hestenes-Stiefel+ $\max(0, g_{k+1} \cdot y/p_k \cdot y)$; Dai-Yuan $|g_{k+1}|^2/p_k \cdot y$; Hager-Zhang $\max(\beta^{\mathrm{HZ}}, -1/(|p_k| \min(0.01, |g_k|)))$, $\beta^{\mathrm{HZ}} = (g_{k+1} \cdot y - 2(p_k \cdot g_{k+1})|y|^2/p_k \cdot y)/p_k \cdot y$. Verbose names CG-FR, CG-PR+, CG-HS+, CG-DY, CG-HZ.

`template<typename T> class CG_hyperparameters : public linesearch_hyperparameters<T>`

Default `my_c2=0.1` (CG needs a more accurate line search).

`CG_beta my_beta`

`CG_PR` (default), see above.

`int my_restart`

Restart every `my_restart` iterations, ≤ 0 = the dimension n (default 0).

`T my_powell_thr`

Restart if $|g_{k+1} \cdot g_k| \geq$ powell_thr $|g_{k+1}|^2$ ([4] eq. 5.52), ≤ 0 disables (0.2).

`template<typename T> class CG_Optimizer : public Optimizer<T>`

10.4.4 GDIIS

Geometry DIIS [7] with a quasi-Newton Hessian [8]. The error vectors are the quasi-Newton steps $e_i = -Hg_i$; the new point is $\sum_i c_i(x_i + e_i)$ with c minimizing $|\sum_i c_i e_i|$ subject to $\sum_i c_i = 1$ (the `DIIS` class of diis.h, chapter 9). One evaluation per iteration. If the DIIS step goes uphill ($s \cdot g \geq 0$), is longer than `my_maxsteplength`, or does not decrease f, the DIIS history is reset and a strong Wolfe line search along the quasi-Newton step is done; the first step after a (re)start is always a line search. H is updated as in `BFGS_Optimizer` (all its parameters apply, including `my_update`).

`template<typename T> class GDIIS_hyperparameters : public BFGS_hyperparameters<T>`

`int my_diis_dim`

Number of stored points (5; `laerror` if < 1).

`T my_diis_eps`

Regularization added to the diagonal of the DIIS matrix (0).

`template<typename T> class GDIIS_Optimizer : public Optimizer<T>`

10.4.5 Trust-region Newton and quasi-Newton

Minimization by `trustregion_minimize()` (section 10.3.5) with the subproblem $\min g \cdot p + \frac{1}{2} p \cdot Bp$, $|p| \leq \Delta$, solved either exactly in the eigenbasis of B [11] (Newton step if B is positive definite and the step lies inside, otherwise $p(\mu) = -(B + \mu I)^{-1} g$ with $|p(\mu)| = \Delta$ found by a safeguarded Newton iteration on $1/|p(\mu)|$, including the hard case; $O(n^3)$ per iteration, handles indefinite B), or approximately by truncated conjugate gradients [12] ([4] alg. 7.2), which avoids the diagonalization for large n. With the exact Hessian (default if `OptHessian` is provided) this is a globally convergent Newton method.

`enum TR_subproblem {TR_EIGEN, TR_STEIHAUG}`
: Subproblem solver.

`template<typename T> class TrustRegion_hyperparameters : public trustregion_hyperparameters<T>`
: All parameters of `trustregion_hyperparameters` (section 10.3.5) and:

`TR_subproblem my_subproblem`
: `TR_EIGEN` (default) or `TR_STEIHAUG`. Verbose names TR-Newton, TR-Steihaug.

`int my_maxcg`
: Maximum number of CG iterations of Steihaug's method, ≤ 0 = the dimension (0).

`template<typename T> class TrustRegion_Optimizer : public Optimizer<T>`
: Minimization only (`laerror` for `my_saddle_order!=0`, use P-RFO). `achieved->my_saddle_order` returns the Hessian index at the result, which is 0 for a true minimum.

Note that the Steihaug variant works with the stored matrix B (exact or updated), so it saves the $O(n^3)$ diagonalization but not the $O(n^2)$ storage.

10.4.6 P-RFO: minima and saddle points

Partitioned rational function optimization [13, 14]: search for a stationary point with `my_saddle_order`$= k$ negative Hessian eigenvalues. In the eigenbasis of B (λ_i ascending, $\tilde{g}_i$ the components of g), the step is $\tilde{p}_i = -\tilde{g}_i/(\lambda_i - \mu)$ with $\mu = \mu_{\max}$, the largest eigenvalue of the bordered $(k+1) \times (k+1)$ matrix $\left[\begin{smallmatrix} \Lambda_k & \tilde{g}_k \\ \tilde{g}_k^{\mathsf{T}} & 0 \end{smallmatrix}\right]$ of the k lowest modes (maximized), and $\mu = \mu_{\min}$, the lowest eigenvalue of the corresponding matrix of the other modes (minimized); the step is scaled down to Δ if longer. $k = 0$ gives RFO minimization. For $k > 0$ the trust-region driver works in its saddle mode (steps always taken). The modes to be maximized are always the k lowest ones (no mode following). The Hessian choice follows `my_hessian`: with `TR_AUTO` the exact Hessian if available, otherwise Bofill updates for $k > 0$ (`TR_BFGS` is refused

for $k > 0$). Convergence is normally by `my_g_thr`; the number of negative eigenvalues at the result is only reported (in the verbose output and in `achieved->my_saddle_order`), not checked, so the caller should compare it with k.

`template<typename T> class PRFO_hyperparameters : public trustregion_hyperparameters<T>`
: All parameters of `trustregion_hyperparameters`, with the default `my_delta0=0.3`; `my_saddle_order` in $[0, n]$.

`template<typename T> class PRFO_Optimizer : public Optimizer<T>`
: Verbose name P-RFO; `laerror` for an invalid saddle order.

10.4.7 Levenberg-Marquardt

For least-squares problems providing `OptResiduals` (typically derived from `OptLeastSquares`): the step h solves $(J^\mathsf{T}J + \mu D)h = -J^\mathsf{T}r$ (`linear_solve`), $D = I$ or $D = \mathrm{diag}(J^\mathsf{T}J)$ (Marquardt scaling); the damping starts at $\mu = \tau \max_i (J^\mathsf{T}J)_{ii}$ and is updated by the gain ratio ϱ [15] (alg. 3.16): for $\varrho > 0$ the step is accepted and $\mu \leftarrow \mu \max(1/3, 1-(2\varrho-1)^3)$, $\nu = 2$, otherwise $\mu \leftarrow \mu\nu$, $\nu \leftarrow 2\nu$. The Jacobian comes from `jacobian()` of the problem or from forward differences (relative step `my_fd_step`, n residual evaluations counted in `my_maxfunc`). Convergence: $|J^\mathsf{T}r| <$ `my_g_thr`, the general f/x criterion, or when a rejected step becomes negligible at machine precision ($|h| \le 10\epsilon(|x| + \epsilon)$). The function value returned is $f = \frac{1}{2}|r|^2$. The problem's `funcgrad()` is not used.

`template<typename T> class LM_hyperparameters : public gradient_hyperparameters<T>`

`T my_tau`
: Initial damping factor τ (`1e-3`).

`bool my_scaling`
: Marquardt scaling $D = \mathrm{diag}(J^\mathsf{T}J)$ (default `false`, $D = I$).

`T my_fd_step`
: Relative step of the forward-difference Jacobian (`1e-7`).

`template<typename T> class LM_Optimizer : public Optimizer<T>`
: `laerror` if the problem does not provide `OptResiduals`. Verbose name Levenberg-Marquardt.

10.4.8 Nelder-Mead downhill simplex

Derivative-free simplex method [16] with reflection, expansion, outside and inside contraction and shrink, starting from the simplex x_0, $x_0 + \mathrm{initstep}\, e_i$ ($n + 1$ evaluations). Convergence: `my_f_thr` is compared with the spread $f_{\mathrm{worst}} - f_{\mathrm{best}}$ of the function values in the simplex and `my_x_thr` with its

diameter (largest distance from the best vertex); both have to be satisfied. The best vertex is returned.

`template<typename T> class NelderMead_hyperparameters : public hyperparameters<T>`
: Defaults `my_maxstep=5000`, `my_f_thr=1e-8`, `my_x_thr=1e-6`.

`T my_initstep`
: Edge length of the initial simplex (0.1).

`bool my_adaptive`
: Dimension-dependent coefficients $\alpha = 1$, $\gamma = 1 + 2/n$, $\rho = 3/4 - 1/(2n)$, $\sigma = 1 - 1/n$ [17] (default `true`, recommended above about 10 dimensions); otherwise the following ones.

`T my_alpha; T my_gamma; T my_rho; T my_sigma`
: Reflection (1), expansion (2), contraction (0.5), shrink (0.5) coefficients.

`template<typename T> class NelderMead_Optimizer : public Optimizer<T>`

10.4.9 Powell's conjugate direction method

Derivative-free [18]: in each iteration (sweep) line minimizations by `linemin_brent()` along n directions, initially the coordinate axes; the overall displacement of the sweep replaces the direction of the largest decrease if Powell's test indicates it is worthwhile, followed by a line minimization along it. Convergence: `my_f_thr` and `my_x_thr` are compared with the change of f and x in one sweep, both have to be satisfied. The trial step of each direction is adapted to its last line minimum.

`template<typename T> class Powell_hyperparameters : public hyperparameters<T>`
: Defaults `my_maxstep=1000`, `my_f_thr=1e-10`, `my_x_thr=1e-6`.

`T my_initstep`
: Initial trial step of the line minimizations (0.1).

`T my_linmin_tol`
: Relative tolerance of the line minimizations (`1e-4`); the absolute one is 0.01 `my_x_thr`.

`int my_maxlinemin`
: Maximum number of evaluations in one line minimization (50).

`template<typename T> class Powell_Optimizer : public Optimizer<T>`

10.4.10 COBYLA

Constrained optimization by linear approximations [19, 20], derivative-free, minimizing f subject to the inequality constraints of `OptConstraints` (without them unconstrained). Linear models of f and of the constraints are interpolated on a simplex of $n+1$ points (initially x, $x+\text{rhobeg}\, e_j$). The trust-region step minimizes the linear model of f subject to the linearized constraints within $|d| \le \Delta$ (an active-set method on the small linear program; if the linearized constraints cannot be satisfied within the trust region, the largest violation is minimized first); the merit function is $f + \mu\cdot$(largest constraint violation) with an automatically increased penalty μ. Geometry steps keep the simplex acceptable. The resolution ρ decreases from `my_rhobeg` to `my_rhoend`; the iterations finish when ρ has reached `my_rhoend` and the trust-region step does not improve the point (`converged=true`), or by the general criterion, in which `my_f_thr` is compared with the change of f at the best vertex in an iteration and `my_x_thr` with ρ (both 0 by default, i.e. off). The best vertex with respect to the merit function is returned; with verbose output the largest remaining constraint violation is printed. COBYLA does not need a constraint to be satisfied at the starting point.

`template<typename T> class Cobyla_hyperparameters : public hyperparameters<T>`
: Default `my_maxstep=100000`.

`T my_rhobeg`
: Initial trust-region radius, about one tenth of the expected change of x (0.5).

`T my_rhoend`
: Final trust-region radius, the accuracy of x (`1e-6`).

`bool my_flexible_radius`
: `true` (default): the trust-region radius $\Delta \ge \rho$ is revised by the reduction ratio and ρ is reduced by the rules of NEWUOA [21] (eqs. 7.1, 7.6, 6.16), so that Δ can also increase again; geometry steps have the length $\max(\min(0.1\,\text{dist}, 0.5\Delta), \rho)$. `false`: Powell's original COBYLA with $\Delta = \rho$, ρ halved when reduced.

`template<typename T> class Cobyla_Optimizer : public Optimizer<T>`

10.4.11 Rotosolve

Sweeps of exact minimizations along single parameters in which f is trigonometric (`OptTrigonometric`) [22, 23], for degree > 1 [24]. For x_i of degree R, f along x_i is reconstructed by a discrete Fourier transform from $2R+1$ equidistant samples over its period (the current value and $2R$ new evaluations) and minimized without further evaluations (in closed form for $R = 1$, otherwise by a dense grid refined by Brent's method on the reconstruction);

the step taken is the shortest one modulo the period. Parameters of degree 0, or all parameters if the problem does not provide `OptTrigonometric`, are minimized by `linemin_brent()` as in Powell's method, whose parameters apply. Convergence as for Powell (change of f and x in one sweep). If the evaluation budget does not suffice for the next minimization, the sweep is interrupted and the method stops without convergence.

```
template<typename T> class Rotosolve_hyperparameters :  public
    Powell_hyperparameters<T>
```

`bool my_reevaluate`
: Evaluate f at the new point instead of taking the value of the reconstruction (for noisy f or doubtful degrees; one more evaluation per parameter; default `false`).

```
template<typename T> class Rotosolve_Optimizer :  public
    Optimizer<T>
```

10.4.12 SGD with momentum

By `firstorder_minimize()`: velocity $v \leftarrow \text{momentum} \cdot v + g$ ($v = g$ in the first iteration), step direction $d = v$, or $d = g + \text{momentum} \cdot v$ (Nesterov) [29], as in the PyTorch update formula.

```
template<typename T> class SGD_hyperparameters :  public
    firstorder_hyperparameters<T>
```
: Default `my_lr=1e-2`.

`T my_momentum; bool my_nesterov`
: Momentum (0.9; 0 gives plain gradient descent); Nesterov variant (`false`). Verbose names SGD, SGD-Nesterov.

```
template<typename T> class SGD_Optimizer :  public Optimizer<T>
```

10.4.13 Quantum natural gradient

$x \leftarrow x - \text{lr}\,(G + \text{metric_reg}\, I)^{-1} g$ [27]: SGD without momentum with the natural gradient; the problem has to provide the metric (`OptMetric`), the gradient may come e.g. from `ParameterShift`. Momentum, Adam or SPSA can be combined with the natural gradient by setting `my_natural=true` in their hyperparameters. Natural steps are longer than gradient steps with the same learning rate (the metric eigenvalues are typically $\leq 1/4$): use lr up to about 0.1 or a larger `my_metric_reg`.

```
template<typename T> class QNG_hyperparameters :  public
    SGD_hyperparameters<T>
```
: Defaults `my_momentum=0`, `my_natural=true`, `my_lr=0.05`.

`template<typename T> class QNG_Optimizer : public Optimizer<T>`

Forces `my_natural=true` also if other hyperparameters were passed; `laerror` if the problem does not provide `OptMetric`.

10.4.14 Adam, AdamW, AMSGrad

Adam [30] with bias-corrected moment estimates, $d = \hat{m}/(\sqrt{\hat{v}} + \epsilon)$; with `my_weight_decay>0` the decoupled weight decay of AdamW [32]; optionally AMSGrad [31] (the elementwise maximum of the past second moments). `my_beta1=0` gives RMSprop.

`template<typename T> class Adam_hyperparameters : public firstorder_hyperparameters<T>`

Default learning rate `1e-3` of the base class.

`T my_beta1; T my_beta2`

Decay rates of the first (0.9) and second (0.999) moment estimates.

`T my_eps`

Regularization of the division by the second moment (`1e-8`).

`bool my_amsgrad`

AMSGrad (`false`). Verbose names Adam, AdamW, AMSGrad.

`template<typename T> class Adam_Optimizer : public Optimizer<T>`

10.4.15 SPSA

Simultaneous perturbation stochastic approximation [33, 34]: derivative-free gradient estimate $\hat{g}_i = (f(x + c_k\Delta) - f(x - c_k\Delta))/(2c_k\Delta_i)$ with random $\Delta_i = \pm 1$ (from `randdouble()` of `la_random.h`), two evaluations per estimate independently of the dimension, robust to noise (e.g. shot noise in VQE). The function value reported is the average of the two evaluations. Gain sequences: $a_k = \mathrm{lr}/(1 + \mathrm{lr_decay} \cdot k)^{\mathrm{lr_power}}$, i.e. Spall's $a/(k+1+A)^\alpha$ with $\mathrm{lr} = a/(1+A)^\alpha$, $\mathrm{lr_decay} = 1/(1+A)$, $\mathrm{lr_power} = \alpha$; $c_k = c/(k+1)^\gamma$. Since f and g are estimates, no convergence test is done: the iterations run until `my_maxstep` or `my_maxfunc`, and `converged` is `false`. The first estimate at the starting point is always done, even if `my_maxfunc` is smaller than its 2·`my_resamplings` evaluations. Before it, the learning rate can be calibrated (evaluations counted; the value used is returned in `achieved->my_lr` if `achieved` is a `firstorder_hyperparameters`).

`template<typename T> class SPSA_hyperparameters : public firstorder_hyperparameters<T>`

Defaults `my_lr_decay=0.01`, `my_lr_power=0.602`, `my_g_thr=0`.

`T my_c; T my_gamma`

Perturbation size c, of the order of the noise of f (0.1), and its decay exponent γ (0.101).

`int my_resamplings`
: Number of gradient estimates averaged in one iteration (1; `laerror` if < 1).

`int my_calibrate`
: Number of gradient estimates at the starting point for the calibration of lr, 0 = no calibration (20).

`T my_target_step`
: Calibration: lr is set so that the first step has on average this length (0.1).

`template<typename T> class SPSA_Optimizer : public Optimizer<T>`

10.5 Examples

All examples were compiled and run; they assume `#include "la.h"` and `using namespace LA;`.

10.5.1 A user-defined problem and BFGS

The extended Rosenbrock function with its analytic gradient. Note the `copyonwrite()` on the gradient vector before its elements are written.

```
class Rosenbrock : public OptProblem<double> {
public:
double funcgrad(const NRVec<double> &x, NRVec<double> *g) const
        {
        const int n=x.size();
        double f=0;
        if(g) {g->copyonwrite(); *g=0.;} //g may share data with
    another vector
        for(int i=0; i<n-1; ++i)
                {
                double a=1-x[i], b=x[i+1]-x[i]*x[i];
                f+=a*a+100*b*b;
                if(g) {(*g)[i]+= -2*a-400*x[i]*b; (*g)[i+1]+=200*
    b;}
                }
        return f;
        }
};

Rosenbrock problem;
NRVec<double> x(-1.2,4); //starting point (-1.2,...,-1.2)
BFGS_hyperparameters<double> param;
param.my_g_thr=1e-8;
param.my_maxstep=1000;
param.my_verbose=0;
```

```
gradient_hyperparameters<double> achieved;
BFGS_Optimizer<double> o;
optresult<double> res=o.minimize(&problem,x,&param,&achieved);
std::cout <<"converged "<<res.converged<<" f= "<<res.f<<" x= "<<x
        <<" iterations "<<achieved.my_maxstep<<" evaluations "<<
    achieved.my_maxfunc
        <<" |g| "<<achieved.my_g_thr<<std::endl;
```

(Here `x` is read in `funcgrad()` through a const reference, so no DEBUG count check is triggered.) The same problem can be given to any gradient method by changing the optimizer and hyperparameter classes, and to derivative-free methods directly. With `FiniteDifference<double> fd(&problem);` an f-only problem can be minimized by BFGS as `BFGS_Optimizer<double>().minimize(&fd,x);`.

10.5.2 Freezing parameters

```
Rosenbrock problem;
NRVec<double> x(0.5,6);
bitvector frozen(6);
frozen.clear();
frozen.set(0); frozen.set(3); //x_0 and x_3 keep their values
FrozenOptProblem<double> fp(&problem,x,frozen);
NRVec<double> xr=fp.reduce(x); //the 4 free parameters
LBFGS_Optimizer<double> o;
double f=o.minimize(&fp,xr); //default parameters; optresult
    converts to f
NRVec<double> xopt=fp.expand(xr); //full vector, x_0 = x_3 = 0.5
```

`fp.xfull` shares its data with `x`; reading elements of `x` afterwards in a DEBUG build has to go through a const reference, writing them needs `x.copyonwrite()` first (which then also detaches `x` from `fp.xfull`).

10.5.3 COBYLA with constraints

```
//min (x0-1)^2+(x1-2)^2 subject to x0+x1 <= 1, solution (0,1)
class ConstrainedQuadratic : public OptProblem<double>, public
    OptConstraints<double> {
public:
double funcgrad(const NRVec<double> &x, NRVec<double> *g) const
        {return (x[0]-1)*(x[0]-1)+(x[1]-2)*(x[1]-2);} //COBYLA
    never asks for g
int nconstraints() const {return 1;}
void constraints(const NRVec<double> &x, NRVec<double> &c) const
        {c.copyonwrite(); c[0]=1-x[0]-x[1];} //c_i >= 0
};

ConstrainedQuadratic problem;
```

```
NRVec<double> x(0.,2);
Cobyla_hyperparameters<double> param;
param.my_rhobeg=0.5;
param.my_rhoend=1e-8;
param.my_verbose=0;
Cobyla_Optimizer<double> o;
optresult<double> res=o.minimize(&problem,x,&param);
NRVec<double> c(1);
problem.constraints(x,c); //check the feasibility
```

10.5.4 Least-squares fit by Levenberg-Marquardt

```
//fit of y = b0 exp(-b1 t) to data; residuals r_i = model - y_i
class ExpFit : public OptLeastSquares<double> {
public:
NRVec<double> t,y; //data
ExpFit(const NRVec<double> &tt, const NRVec<double> &yy) : t(tt),
     y(yy) {};
int nresiduals() const {return t.size();}
void residuals(const NRVec<double> &b, NRVec<double> &r) const
        {
        r.copyonwrite();
        for(int i=0; i<t.size(); ++i) r[i]=b[0]*std::exp(-b[1]*t[
    i])-y[i];
        }
bool jacobian(const NRVec<double> &b, NRMat<double> &J) const
        {
        J.copyonwrite();
        for(int i=0; i<t.size(); ++i)
                {
                const double e=std::exp(-b[1]*t[i]);
                J(i,0)=e; J(i,1)= -b[0]*t[i]*e;
                }
        return true; //false would mean: use finite differences
        }
};

const int m=10;
NRVec<double> t(m), y(m);
for(int i=0; i<m; ++i) {t[i]=0.5*i; y[i]=3*std::exp(-0.7*t[i])
    +0.01*((i%3)-1);}
ExpFit problem(t,y);
NRVec<double> b(2);
b[0]=1; b[1]=0.1;
LM_hyperparameters<double> param;
param.my_g_thr=1e-10;
param.my_verbose=0;
hyperparameters<double> achieved;
LM_Optimizer<double> o;
```

```
optresult<double> res=o.minimize(&problem,b,&param,&achieved);
//res.f = 1/2 |r|^2, i.e. half the residual sum of squares
```

Since `ExpFit` is an `OptLeastSquares`, the same object can also be passed to e.g. `BFGS_Optimizer`.

10.5.5 Further examples in the distribution

t.cc contains testbeds (blocks starting with `//testbed of the opt...`) for the methods on `testproblem`, `testproblem_vqe(_constrained)` (with `ParameterShift`), `FrozenOptProblem`, P-RFO on the Müller-Brown potential and LM on a NIST problem. The header testoptimize.h (in the source distribution, not installed) defines these test problems, which can serve as templates for user problems:

- `testproblem<T>`: a smooth function with analytic gradient;
- `testproblem_vqe<T>`: a VQE-like Ising chain in a product state providing `OptHessian`, `OptMetric` and `OptTrigonometric`; `testproblem_vqe_constrained<T>`: the same with one constraint (`OptConstraints`);
- `testproblem_mullerbrown<T>`: the Müller-Brown potential [35] with analytic Hessian, three minima and two saddle points;
- NIST StRD nonlinear regression problems (`OptLeastSquares`) with starting values, certified parameters and residual sums of squares: `testproblem_misra1a<T>`, `testproblem_mgh09<T>` (analytic Jacobians) and `testproblem_thurber<T>` (finite-difference Jacobian).

The regression program testoptimize.cc, run by `make check`, tests all methods, adapters, capabilities, the `maxfunc` limits, the hyperparameter inheritance and the copy-on-write behaviour of the functions used; it prints PASS/FAIL lines and its exit code is the number of failures.

10.6 Pitfalls

Caution: Gradient and other output arguments. The vector `*grad` passed to `funcgrad()`, and the arguments `h`, `g`, `c`, `r`, `J` of the capability functions, are allocated by the caller but may share their data with other objects. Call `copyonwrite()` before writing elements through `operator[]`/`operator()` (assignments of whole objects or scalars, e.g. `*grad=0.;`, are safe and detach by themselves). Forgetting it can silently corrupt vectors of the optimizer (e.g. the previous gradient).

Caution: Shared reference to the best point. `optevaluator` keeps `xbest=x`, a shallow copy sharing the data of the vector evaluated. Code calling an `optevaluator` (a new method, or a gradient estimator) and then modifying that vector element-wise must call `copyonwrite()` before each such modification, otherwise the recorded best point changes as well (see `forward_difference_jacobian` and `trig_minimize` in optimize.cc). Whole-vector operations (`axpy`, `+=`, assignment) are safe.

Caution: Reading possibly shared vectors. After `minimize()` the vector `x`, and the vectors given to `FrozenOptProblem` (`xfull`), may share data with other objects. In DEBUG builds the non-const `operator[]` raises `laerror` when the reference count is > 1 (section 3.6); read elements of a possibly shared non-const vector through a const reference, e.g. `const NRVec<double> &xc=x; ... xc[i]`. Inside `funcgrad()` the argument `x` is already const.

Caution: Convergence thresholds. The general criterion needs *both* `my_f_thr` and `my_x_thr` (strict inequalities); with the default 0 it is off, and gradient methods then converge only by `my_g_thr`. When passing a hyperparameter object of a more general class, its defaults replace those of the method (section 10.2.3). Methods print by default (`my_verbose=1`); set it to 0 for silent operation.

Caution: Saddle points. `converged` of P-RFO does not imply the requested saddle order; check `achieved->my_saddle_order`. Likewise a converged `TrustRegion_Optimizer` run may end at a saddle point if started at one with zero gradient components along the negative modes; the reported index tells.

Caution: Noisy functions. SPSA (and first-order methods with an inexact `gradient_estimator`) never report convergence; set `my_maxstep` or `my_maxfunc`. `my_return_best` should not be used with noisy f. SPSA uses the global random generator of `la_random.h`; seed it for reproducible runs.

Chapter 11

Tensors

Header: `tensor.h, tensor.cc`

11.1 Purpose and main idea

`Tensor<T>` is a dense tensor of arbitrary rank whose indices are organized in *index groups*. Each group has its own index range, offset and permutational symmetry (none, symmetric, antisymmetric, hermitean, antihermitean), and only the permutationally non-equivalent elements of a group are stored. Indices can optionally carry names (`INDEXNAME`) and most operations can address indices by name. The class provides element access with automatic sign/conjugation handling, contractions implemented via unwinding to matrices and BLAS `gemm`, reordering of indices and groups, (anti)symmetrization by a general permutation algebra, the HOSVD/Tucker decomposition, conversions to and from `NRVec`, `NRMat`, `NRSMat` and the dense four-index classes of chapter 12, and binary, text and NumPy I/O.

Typical uses are many-body quantities (amplitudes, integrals, density matrices) where antisymmetry in groups of indices should be exploited for storage. Limitations stated in the source:

- the implementation is "rudimentary"; only `Tensor<double>` and `Tensor<std::complex<double> >` are instantiated (tensor.cc);
- index types `LA_index` and `LA_largeindex` are `int`, so the total size is limited to 2^{31} elements (`NRVec` limitation);
- symmetry *between* index groups (e.g. $(ij|kl) = (kl|ij)$) is not supported; one would have to nest `Tensor<Tensor<T> >` or use the four-index classes;
- covariant and contravariant indices are not distinguished: the header `##undefs LA_TENSOR_INDEXPOSITION`, the code guarded by this macro is not active.

11.2 Storage and concepts

11.2.1 Index groups

```
typedef class INDEXGROUP {
public:
  int number;        // number of indices in the group (>0)
  int symmetry;      // -2,-1,0,1,2, see below
  LA_index offset;   // indices run offset ... offset+range-1
  LA_index range;
  bool operator==(const INDEXGROUP &rhs) const; bool operator!=(
    const INDEXGROUP &rhs) const;
} INDEXGROUP;
```

`INDEXGROUP` is a plain aggregate without constructor; all members must be set by the user. If the macro `LA_TENSOR_ZERO_OFFSET` is defined before including the header, `offset` becomes a `static const` zero (which saves some arithmetic); otherwise any offset, also negative, is allowed. Index groups are compared memberwise.

`symmetry`	meaning	stored elements	odd permutation gives
0	no symmetry	r^m	—
1	symmetric	$\binom{r+m-1}{m}$	$+x$
-1	antisymmetric	$\binom{r}{m}$	$-x$; zero for a repeated index
2	hermitean (complex `T` only)	$\binom{r+m-1}{m}$	x^*
-2	antihermitean (complex `T` only)	$\binom{r+m-1}{m}$	$-x^*$

Here m is `number` and r is `range`. For real `T` the symmetries ± 2 are illegal (`canonicalize_shape()` raises an error); a group with a single index always gets symmetry 0. For (anti)hermitean groups the diagonal elements (repeated indices) are stored including their real/imaginary part which should vanish; this is *not* checked automatically, see `enforce_hermiticity()` and `fulfills_hermiticity()`. The symmetry factors of different groups combine multiplicatively (for complex `T` via a small group multiplication table, so that e.g. two conjugations cancel).

11.2.2 Storage order

The element data are kept in the public member `NRVec<T> data`. The *leftmost* index is the least significant, i.e. it changes fastest: group 0 is the innermost loop, and inside a non-symmetric group the first index is the

fastest. For a group with m indices $I_0, \ldots, I_{m-1}$ and offset subtracted, the position inside the group is

$$\begin{aligned}
&\textstyle\sum_k I_k\, r^k && \text{(no symmetry)},\\
&\textstyle\sum_k \binom{I_k+k}{k+1}, \quad I_0 \le I_1 \le \cdots \le I_{m-1} && \text{(symmetric, (anti)hermitean)},\\
&\textstyle\sum_k \binom{I_k}{k+1}, \quad I_0 < I_1 < \cdots < I_{m-1} && \text{(antisymmetric)},
\end{aligned}$$

i.e. the canonical (stored) element has nondecreasing (increasing) indices; for $m = 2$ this is the usual lower-triangle packing `i1*(i1+1)/2+i0` of `NRSMat`. The total position is $\sum_g p_g$ `cumsizes`$[g]$, where `groupsizes[g]` is the number of stored elements of group g and `cumsizes[g]` the product of the sizes of groups $0 \ldots g-1$. The binomial coefficients are taken from the precomputed table of `simplicial()` (miscfunc.h), which limits symmetric groups to fewer than 16 indices and ranges below 1024.

11.2.3 Public data members

All data members are public:

`NRVec<INDEXGROUP> shape`
: the index groups; `shape.size()==0` for a scalar (rank 0).

`NRVec<T> data`
: the stored elements.

`int myrank`
: total number of indices (sum of `number` over groups), -1 for a default-constructed tensor.

`NRVec<LA_largeindex> groupsizes; NRVec<LA_largeindex> cumsizes`
: redundant, precomputed from `shape` by `calcsize()`.

`NRVec<INDEXNAME> names`
: index names in flat order (all indices of group 0, then group 1, ...); either empty (unnamed tensor) or of length `rank()`.

If `shape` is modified directly, `calcsize()`, `calcrank()` and possibly `data.resize()` must be called by the user to restore consistency.

11.2.4 Index types

`typedef int LA_index; typedef int LA_largeindex`
: an index value; a position in `data`.

`typedef NRVec<LA_index> FLATINDEX`
: all indices of an element in flat order.

`typedef NRVec<FLATINDEX> SUPERINDEX`
: indices structured by groups: `I[g][i]`.

`typedef NRVec<LA_largeindex> GROUPINDEX`
: one position (inside the compressed group storage) per group.

`typedef NRMat<LA_index> INDEXMATRIX`
: rows are the `FLATINDEX`es of all stored elements, in the order of `data`.

`struct INDEX {int group; int index;}`
: position of one index: group number and index number inside the group (both from 0); constructors `INDEX()` and `INDEX(int g, int i)`, `operator==`.

`typedef NRVec<INDEX> INDEXLIST`
: a list of indices.

`struct INDEXNAME {char name[N_INDEXNAME];}`
: index name, `N_INDEXNAME`=8; constructible from `const char *` (copied by `strncpy`), so a string literal can be passed wherever an `INDEXNAME` is expected; `operator=`, `==`, `!=` compare by `strncmp`. Use at most 7 characters so that the name stays NUL-terminated.

`LA_traits` are specialized for `INDEXNAME`, `INDEXGROUP` and `INDEX` as plain data, so that vectors of them can be compared and written/read in binary form (section 3.3, 3.7).

11.2.5 Reference counting and copy on demand

`Tensor` itself has no reference counter, but all its large members are `NRVec`s, so the scheme of section 3.1 applies member by member: the copy constructor and `operator=` share `data`, `shape`, `names` etc. (shallow copy), and `copyonwrite()` detaches all of them. There is *no* `operator|=`; a deep copy is made by `Tensor<T> b(a); b.copyonwrite();`. Arithmetic members delegate to `NRVec` methods which call `copyonwrite()`; contractions call `data.copyonwrite()`; results of restructuring operations are freshly allocated. An explicit `copyonwrite()` is needed before writing through `lhs()`, which, like a non-const `operator[]`, raises an error in DEBUG builds when the data are shared. The element reading `operator()` is `const` only, so reading a shared tensor is safe. `loopover()` and `grouploopover()` pass writable pointers to the callback and therefore detach the data first (one copy if shared, otherwise only a check); use `constloopover()`/`constgrouploopover()` for reading only, they never copy. `renameindex()`, `permute_inside_group()`, `enforce_hermiticity()` and `flatten()` also detach what they modify.

11.3 Construction, assignment, conversion

`Tensor()`
: empty tensor, `rank()==-1`.

`explicit Tensor(const T &x)`
: scalar (rank 0) with value `x`.

`Tensor(const NRVec<INDEXGROUP> &s); Tensor(const INDEXGROUP &g)`
: allocate a tensor of the given shape (data not initialized); a single group can be given directly. The shape is canonicalized (`canonicalize_shape()`).

`Tensor(const NRVec<INDEXGROUP> &s, const NRVec<INDEXNAME> &newnames); Tensor(const INDEXGROUP &g, const NRVec<INDEXNAME> &newnames)`
: the same with index names; `newnames` must be empty or of length `rank()`.

`Tensor(const NRVec<INDEXGROUP> &s, const NRVec<T> &mydata); Tensor(const NRVec<INDEXGROUP> &s, const NRVec<T> &mydata, const NRVec<INDEXNAME> &newnames)`
: use `mydata` as the data (shared, not copied); its size must match the shape.

`Tensor(int xrank, const NRVec<INDEXGROUP> &xshape, const NRVec<LA_largeindex> &xgroupsizes, const NRVec<LA_largeindex> xcumsizes, const NRVec<T> &xdata); Tensor(..., const NRVec<INDEXNAME> &xnames)`
: low-level constructor setting all members directly without any check.

`Tensor(const Tensor &rhs); Tensor& operator=(const Tensor &rhs)`
: shallow copy, see above.

`void resize(const NRVec<INDEXGROUP> &s)`
: set a new shape, reallocate data (not initialized) and delete the names.

`void deallocate()`
: release data, shape, names and auxiliary arrays.

`void copyonwrite()`
: detach all members (deep copy if shared).

11.3.1 Conversions

`explicit Tensor(const NRVec<T> &x)`
: rank-1 tensor, offset 0, data shared with `x`.

`explicit Tensor(const NRMat<T> &x, bool flat=false)`
: rank-2 tensor with a copy of the data and offset 0. A square matrix gives one non-symmetric group of 2 indices unless `flat` is true; otherwise two groups of one index. Because of the storage order, `t(i,j)==x(j,i)`: tensor index 0 is the column index.

`explicit Tensor(const NRSMat<T> &x)`
: one group of two indices, data shared with `x`; symmetric (symmetry 1) for real, hermitian (symmetry 2) for complex types, as `NRSMat` itself (chapter 6); equal to `Tensor(NRMat<T>(x))`. If `NRSMat` gets the planned `hermitian` flag (section 6.2.3), this constructor has to respect it.

`NRMat<T> matrix(int n=1) const`
: reinterpret the data as a matrix whose column index runs over the n leftmost index groups (e.g. an index made leftmost by `unwind_index()`) and whose row index runs over the remaining groups. The matrix *shares* the data with the tensor. `Tensor<T>(x).matrix()` reproduces `x`.

`template<fourindexsymtype S, typename T, typename I> Tensor<T> fourindex2tensor(const fourindex_dense<S,T,I> &f)`
: conversion from a dense four-index object (chapter 12), see the table below; indices get offset 1.

`template<fourindexsymtype S, typename T, typename I> void tensor2fourindex(const Tensor<T> &t, fourindex_dense<S,T,I> &f)`
: inverse conversion; shape, ranges, offsets and symmetries are checked.

`fourindex2tensor`/`tensor2fourindex` are implemented (overloaded) for the following symmetry types. Since the leftmost tensor index is the least significant, `f(i,j,k,l)` corresponds to `t(k,l,i,j)` (for the types marked * this is equal to `t(i,j,k,l)` or `t(l,k,j,i)` by symmetry):

`fourindexsymtype`	tensor shape
`nosymmetry`	4 groups of 1 index, range n
`twoelectronrealmullikanAB`*	2 symmetric groups of 2 indices
`twoelectronrealmullikan`*	2 symmetric groups of 2 indices (the pair symmetry is lost, data expanded)
`T2IjAb_aces`	4 groups of 1 index, ranges `nvra, nvrb, noca, nocb`
`T2ijab_aces`	2 antisymmetric groups of 2 indices, ranges `nvrt`, `nocc`
`antisymtwoelectronrealdiracAB`*	4 groups of 1 index (data expanded)
`antisymtwoelectronrealdirac`*	2 antisymmetric groups of 2 indices (data expanded)

In the reverse direction the symmetric matrix of pairs is obtained by symmetrizing the tensor data.

11.3.2 NumPy files

`void numpy_write(const char *name, const char *descr="<f8") const`
: write a flat tensor (`is_flat()`) as a NumPy `.npy` file (format version 1.0, C order, the shape is written in reverse so that the NumPy array index order equals the tensor index order). `descr` must match `T` (`"<f8"` for `double`, `"<c16"` for complex); this is not checked.

`void numpy_read(const char *name)`
: read a `.npy` file (version 1.0, C or Fortran order) into a flat tensor with offsets 0; the type is not checked.

11.4 Element access

`T operator()(LA_index i1...) const`
: element by a variable list of `rank()` indices (of type `int`); returns the value including the symmetry sign/conjugation, 0 for a repeated index in an antisymmetric group.

`T operator()(const FLATINDEX &I) const; T operator()(const SUPERINDEX &I) const`
: the same with the indices in a vector.

`Signedpointer<T> lhs(LA_index i1...); Signedpointer<T> lhs(const FLATINDEX &I); Signedpointer<T> lhs(const SUPERINDEX &I)`
: element as an l-value: the returned proxy supports `=`, `+=`, `-=` (the value is transformed by the sign/conjugation of the permutation before being stored), `*=` and `/=` (for an element stored as a conjugate the stored value is multiplied by the conjugated factor, so that the element itself is multiplied by the factor). Writes to elements with a repeated index of an antisymmetric group are ignored (a DEBUG build prints a warning once). Call `copyonwrite()` first if the tensor may be shared.

`LA_largeindex index(int *sign, const SUPERINDEX &I) const; LA_largeindex index(int *sign, const FLATINDEX &I) const`
: position in `data` of the canonical element and the sign: 1, -1, 2 (conjugate), -2 (negative conjugate), or 0 and position -1 for an element which vanishes by antisymmetry. Index ranges are checked in DEBUG builds.

`LA_largeindex vindex(int *sign, LA_index i1, va_list args) const`
: the same for a `va_list`; ends `args`.

`SUPERINDEX inverse_index(LA_largeindex s) const`
: canonical indices of the element stored at position `s` (not efficient).

11.4.1 Looping over all elements

`void loopover(void (*callback)(const SUPERINDEX &, T *), bool`

skipzeros=false)

call `callback` for every stored element with its canonical indices and a pointer into `data`. With `skipzeros` the index value 0 is skipped in all indices (useful e.g. for indices running $\ldots, -2, -1, 1, 2, \ldots$ with a negative offset); the pointer passed to the callback always points to the element of the passed indices (with `skipzeros` it is computed from the indices).

`void constloopover(void (*callback)(const SUPERINDEX &, const T *), bool skipzeros=false) const`

the same for reading.

`void grouploopover(void (*callback)(const GROUPINDEX &, T *)); void constgrouploopover(void (*callback)(const GROUPINDEX &, const T *)) const`

loop over all stored elements with the position inside each group's compressed storage instead of the individual indices.

`INDEXMATRIX indexmatrix() const`

matrix whose row s is the flat index of `data[s]`; allows a simple `for` loop over all elements instead of a callback.

11.4.2 Index bookkeeping

`int myflatposition(int group, int index) const; int myflatposition(const INDEX &i) const`

position of an index in the flat index list.

`INDEX myindexposition(int flatindex) const`

inverse of `myflatposition`.

`int flatposition(int group, int index, const NRVec<INDEXGROUP> &shape); int flatposition(const INDEX &i, const NRVec<INDEXGROUP> &shape); INDEX indexposition(int flatindex, const NRVec<INDEXGROUP> &shape)`

free versions for a given shape.

`LA_largeindex subindex(int *sign, const INDEXGROUP &g, const NRVec<LA_index> &I)`

position of the index tuple `I` inside the storage of one group, with sign as for `index()`.

`NRVec<LA_index> inverse_subindex(const INDEXGROUP &g, LA_largeindex s)`

canonical index tuple of position `s` in a group.

`FLATINDEX superindex2flat(const SUPERINDEX &I)`

concatenate the groups of a superindex.

`bool zero_in_index(const FLATINDEX &); bool zero_in_index(const SUPERINDEX &); bool zero_in_index(const INDEXMATRIX &, const`

`LA_largeindex row)`
: does any index equal 0 (for index ranges with a negative offset where 0 is excluded)?

`bool shares_index(const FLATINDEX &I, const FLATINDEX &J)`
: do `I` and `J` have a common index value?

11.5 Arithmetic

Binary operations require identical shapes (`laerror` otherwise) and, if both tensors are named, identical names.

`Tensor& operator*=(const T &a); Tensor operator*(const T &a) const; Tensor& operator/=(const T &a); Tensor operator/(const T &a) const`
: scaling.

`Tensor& operator+=(const Tensor &rhs); Tensor& operator-=(const Tensor &rhs); Tensor operator+(const Tensor &rhs) const; Tensor operator-(const Tensor &rhs) const`
: elementwise sum and difference of the stored data.

`Tensor& axpy(const T alpha, const Tensor &rhs)`
: `*this += alpha*rhs`.

`Tensor operator-() const`
: negation; the result has no index names.

`Tensor operator*(const Tensor &rhs) const`
: outer product; the indices of `rhs` become the less significant (leftmost) ones: `(a*b)(J,I)=a(I)*b(J)`. Names are concatenated, so either both or none of the factors must be named.

`T dot(const Tensor &rhs) const`
: full contraction $\sum_I \overline{a_I} b_I$ (complex conjugation of `*this`); antisymmetric groups are accounted for by the factor $m!$, symmetric and (anti)hermitean groups are not supported (`laerror`).

`typename LA_traits<T>::normtype norm() const`
: Frobenius norm of the *stored* data (redundant elements are not counted).

`Tensor& conjugateme(); Tensor conjugate() const`
: complex conjugation.

`void clear()`
: set all elements to zero.

`void randomize(const typename LA_traits<T>::normtype &x)`
: random elements, followed by `enforce_hermiticity()`.

`void enforce_hermiticity()`
: zero the imaginary (hermitean) or real (antihermitean) part of elements with repeated indices in (anti)hermitean groups.

`bool fulfills_hermiticity() const`
: check the above condition.

11.6 Contractions

Contractions are performed by unwinding the contracted indices of both tensors to the leftmost positions (expanding them to their full range, `unwind_index()`), followed by one matrix multiplication (`cblas_dgemm`/`cblas_zgemm`). The index order of the result is: the remaining indices of the second tensor (`rhs`, `rhs2`) as the less significant ones, followed by the remaining indices of the first one (`*this` or `rhs1`); each remaining group keeps its symmetry (a group from which an index was contracted keeps its symmetry with one index less). If both tensors are named, the names of the contracted indices must agree and the result is named accordingly.

In the `add...` versions the result is accumulated into `*this`: `*this` $\leftarrow \beta$ `*this` $+ \alpha$ contraction; with `doresize=true *this` is first resized to the result shape (this requires `beta==0`), otherwise its shape (and names, if named) must match. `conjugate1` conjugates the first tensor, `conjugate`/`conjugate2` the second one.

`void addcontraction(const Tensor &rhs1, int group, int index, const Tensor &rhs2, int rhsgroup, int rhsindex, T alpha=1, T beta=1, bool doresize=false, bool conjugate1=false, bool conjugate=false)`
: contract one index of `rhs1` with one index of `rhs2`.

`void addcontraction(const Tensor &rhs1, const INDEX &I1, const Tensor &rhs2, const INDEX &I2, ...); void addcontraction(const Tensor &rhs1, const Tensor &rhs2, const INDEXNAME &iname, ...)`
: the same with `INDEX` positions or with the common index name (same trailing arguments).

`Tensor contraction(int group, int index, const Tensor &rhs, int rhsgroup, int rhsindex, T alpha=1, bool conjugate1=false, bool conjugate=false) const; Tensor contraction(const INDEX &I, const Tensor &rhs, const INDEX &RHSI, ...) const; Tensor contraction(const Tensor &rhs, const INDEXNAME &iname, ...) const`
: return α `*this`·`rhs` contracted over one index.

`void addcontractions(const Tensor &rhs1, const INDEXLIST &il1, const Tensor &rhs2, const INDEXLIST &il2, T alpha=1, T beta=1, bool doresize=false, bool conjugate1=false, bool conjugate2=false); void addcontractions(const Tensor &rhs1, const Tensor &rhs2,`

`const NRVec<INDEXNAME> &names, ...)`
: contract the indices `il1[i]` with `il2[i]` simultaneously in a single matrix multiplication. Finding an optimal contraction order is the responsibility of the user (contracting one index at a time may be cheaper). For empty lists the outer product is added.

`Tensor contractions(const INDEXLIST &il1, const Tensor &rhs2, const INDEXLIST &il2, T alpha=1, bool conjugate1=false, bool conjugate2=false) const; Tensor contractions(const Tensor &rhs2, const NRVec<INDEXNAME> &names, ...) const`
: returning versions.

`void addgroupcontraction(const Tensor &rhs1, int group, const Tensor &rhs2, int rhsgroup, T alpha=1, T beta=1, bool doresize=false, bool conjugate1=false, bool conjugate=false); Tensor groupcontraction(int group, const Tensor &rhs, int rhsgroup, T alpha=1, bool conjugate1=false, bool conjugate=false) const`
: contract over all indices of a whole group without unwinding them; the groups must have equal shape and be non-symmetric or antisymmetric (for antisymmetric groups the factor $m!$ is included, so the result equals the contraction of the flattened tensors).

`Tensor innercontraction(const INDEXLIST &il1, const INDEXLIST &il2) const; Tensor innercontraction(const INDEX &i1, const INDEX &i2) const; Tensor innercontraction(const NRVec<INDEXNAME> &nl1, const NRVec<INDEXNAME> &nl2) const; Tensor innercontraction(const INDEXNAME &n1, const INDEXNAME &n2) const`
: trace(s) inside one tensor: indices `il1[i]` and `il2[i]` are set equal and summed over; the result is a scalar tensor if no index remains.

`void add_permuted_contractions(const char *antisymmetrizer, const Tensor &rhs1, const Tensor &rhs2, T alpha=1, T beta=1, bool conjugate1=false, bool conjugate2=false)`
: for uniquely named tensors: contract all indices whose names occur in both `rhs1` and `rhs2`, apply the antisymmetrizer given as a string, permute the result to the index order of `*this` and accumulate. The string uses the notation of Kucharski and Bartlett: space-separated factors, each a list of index names where `/` separates classes and `,` separates names within a class; e.g. `"k/l,m"` is $P(k/lm) = 1 - P_{kl} - P_{km}$. Indices not mentioned are not permuted; `""` means just contraction and reordering. See `general_antisymmetrizer` in chapter 13.

11.7 Restructuring

Permutations are `NRPerm<int>` (chapter 13) and act on group numbers $1 \ldots n$. Names are permuted together with the indices unless stated oth-

erwise.

`Tensor permute_index_groups(const NRPerm<int> &p) const`
: reorder the data permuting whole index groups: group i of the result is group p_i of the source (both counted from 1).

`Tensor permute_index_groups(const NRVec<INDEXNAME> &names) const`
: the same with the permutation given by the requested order of the groups' first index names (for a flat tensor: of all names).

`Tensor unwind_index_group(int group) const`
: move a group to the leftmost (least significant) position, the others keep their order.

`Tensor unwind_index(int group, int index) const; Tensor unwind_index(const INDEX &I) const; Tensor unwind_index(const INDEXNAME &N) const`
: take one index out of its group and make it a separate leftmost non-symmetric group, expanded to its full range; the rest of the group keeps its symmetry. Without data reorganization if the index is already first in a non-symmetric group 0.

`Tensor unwind_indices(const INDEXLIST &il) const; Tensor unwind_indices(const NRVec<INDEXNAME> &names) const`
: the same for several indices, which become groups $0, 1, \ldots$ in the listed order.

`Tensor flatten(int group= -1) const`
: expand the given group (all groups for -1) into single non-symmetric indices, keeping the flat index order and names.

`Tensor merge_indices(const INDEXLIST &il, int symmetry=0) const; Tensor merge_indices(const NRVec<INDEXNAME> &nl, int symmetry=0) const`
: opposite of `flatten`: the listed indices (equal range and offset) form a new leftmost group of the given symmetry; for a nonzero symmetry the data are (anti)symmetrized, i.e. averaged over the permutations with the appropriate signs.

`Tensor merge_index_groups(const NRVec<int> &groups) const`
: permute the listed non-symmetric groups to the front (in the given order, the remaining groups follow in their original order) and merge them into one group; group numbers from 0, repeated numbers raise an error.

`void split_index_group(int group)`
: in place, formal (no data movement): split a non-symmetric group into single-index groups.

`void split_index_group1(int group)`
: in place, formal: split off the first index of a non-symmetric group.

`void merge_adjacent_index_groups(int groupfrom, int groupto)`
: in place, formal: merge consecutive non-symmetric groups of equal range and offset into one.

`void permute_inside_group(int group, const NRPerm<int> &p, bool inverse=false)`

for a group with nonzero symmetry: apply the sign/conjugation of the permutation parity to the data and permute the corresponding names.

`Tensor subtensor1(int i) const`

the slice for the value `i` of the last (most significant) index, which must form a group of its own.

11.8 Permutation algebra, decompositions and transformations

`void apply_permutation_algebra(const Tensor &rhs, const PermutationAlgebra<int,T> &pa, bool inverse=false, T alpha=1, T beta=0, bool conjugate_by_parity=false)`

$*$`this` $\leftarrow \beta *$`this`, then for all stored elements I of `*this`: `this`(I) $+= \alpha \sum_P c_P$ `rhs`$(P(I))$, with P acting on the flat index list (inverse permutations if `inverse`); with `conjugate_by_parity` odd permutations take the complex conjugate. Can represent general (anti)symmetrizers, e.g. `general_antisymmetrizer` (chapter 13). `*this` may have a symmetry which `rhs` lacks. The names of `*this` are taken from (or checked against) the names of `rhs` permuted by the first permutation. Element-by-element, not optimally efficient.

`void apply_permutation_algebra(const NRVec<Tensor> &rhsvec, const PermutationAlgebra<int,T> &pa, bool inverse=false, T alpha=1, T beta=0)`

the same with `rhs` being the outer product of the tensors in `rhsvec`, which is not formed explicitly (e.g. Grassmann products building a density matrix from cumulants); rather inefficient.

`NRVec<NRMat<T> > Tucker(typename LA_traits<T>::normtype thr=1e-12, bool inverseorder=false)`

HOSVD/Tucker decomposition: `*this` is replaced by the flat core tensor and the factor matrices (one per index, columns orthonormal) are returned; singular values below `thr` are dropped, which reduces the corresponding range of the core.

`Tensor inverseTucker(const NRVec<NRMat<T> > &x, bool inverseorder=false) const`

rebuild the tensor from the core (`*this`) and the matrices returned by `Tucker` with the same `inverseorder`.

`Tensor linear_transform(const NRVec<NRMat<T> > &x) const`
: transform every index of group g by the matrix `x[g]` (`x[g].ncols()` must equal the range; the new range is `x[g].nrows()`), preserving the group symmetries: $t'_{\ldots a\ldots} = \sum_i x_{ai}\, t_{\ldots i\ldots}$.

`Tensor linear_transform(const int g, const Tensor<T> &x) const;`
`Tensor linear_transform(const int g, const NRMat<T> &x) const`
: transform only group `g`, keeping its position and symmetry; `x` must be a flat rank-2 tensor (useful for raising/lowering indices) or a matrix.

For the `Tensor` argument the contraction runs over its index 0, i.e. $t'_{\ldots a\ldots} = \sum_i x(i,a)\, t_{\ldots i\ldots}$ in tensor indexing, which is the same as the matrix convention above for `x` built by `Tensor<T>(matrix,true)`.

11.9 Names and shape queries

`bool is_named() const`
: are names set? (`laerror` if their number differs from the rank).

`bool is_uniquely_named() const`
: named without repetitions.

`void defaultnames(const char *basename="i")`
: set names `i000`, `i001`, ...

`int findflatindex(const INDEXNAME nam) const`
: flat position of the named index, -1 if not found (`laerror` if unnamed).

`INDEX findindex(const INDEXNAME nam) const; NRVec<INDEX>`
`findindexlist(const NRVec<INDEXNAME> &names) const`
: position(s) of named indices; `laerror` if not found.

`void renameindex(const INDEXNAME namfrom, const INDEXNAME nameto)`
: rename an index (see the copy-on-write remark above).

`int rank() const; LA_largeindex size() const`
: number of indices; number of stored elements.

`int calcrank(); LA_largeindex calcsize()`
: recompute `myrank`, resp. `groupsizes`, `cumsizes` from `shape` and return the rank, resp. size.

`void canonicalize_shape()`
: set symmetry 0 for single-index groups; check the symmetry codes.

`bool is_flat() const; bool is_compressed() const; bool`
`has_symmetry() const; bool has_hermiticity() const`
: all groups have one index; some multi-index group has nonzero symmetry; some group has nonzero symmetry; complex `T` and some group is (anti)hermitean.

11.10 Input and output

`void put(int fd, bool with_names=false) const; void get(int fd, bool with_names=false)`
: binary I/O of shape, auxiliary arrays, data and optionally names (section 3.7); with `with_names` the names are always written (an empty list if the tensor has none), so `put` and `get` must use the same `with_names`.

`template<typename T> std::ostream & operator«(std::ostream &s, const Tensor<T> &x)`
: text output: the shape, the names, then one line per stored element with its flat indices and value.

`template<typename T> std::istream & operator»(std::istream &s, Tensor<T> &x)`
: reads the format written by «.

`std::ostream & operator«(std::ostream &s, const INDEXGROUP &x); std::istream & operator»(std::istream &s, INDEXGROUP &x)`
: `number symmetry offset range`.

`std::ostream & operator«(std::ostream &s, const INDEXNAME &x); std::istream & operator»(std::istream &s, INDEXNAME &x)`
: the name as a word.

`std::ostream & operator«(std::ostream &s, const INDEX &x); std::istream & operator»(std::istream &s, INDEX &x)`
: `group index`.

11.11 Helpers in the header

`template<typename T> T signeddata(const int sgn, const T &data)`
: apply a sign code (±1, ±2 = with conjugation, 0 gives 0) to a value.

`template<typename T> class Signedpointer`
: proxy returned by `lhs()`, see section 11.4.

11.12 Examples

An antisymmetric rank-3 tensor:

```
int n=4;
INDEXGROUP g;
g.number=3; g.symmetry= -1; g.offset=0; g.range=n;
Tensor<double> t(g);       // stores n(n-1)(n-2)/6 elements
t.clear();
t.lhs(0,1,2) = 1.;         // implies t(1,0,2)=-1 etc.
t.lhs(2,1,3) = 5.;         // stored as t(1,2,3) = -5
const Tensor<double> &tc=t;
```

```
cout << tc(1,2,3) << " " << tc(3,2,1) << " " << tc(1,1,2) << endl
    ; // -5 5 0
Tensor<double> tf = t.flatten();   // the same tensor stored in
    full, n^3 elements
```

Contractions of named tensors:

```
INDEXGROUP s; s.number=1; s.symmetry=0; s.offset=0; s.range=n;
NRVec<INDEXGROUP> shape(4); for(int i=0; i<4; ++i) shape[i]=s;
Tensor<double> x(shape), y(shape);
x.randomize(1.); y.randomize(1.);
INDEXNAME xn[4]={"i","j","k","l"}; x.names=xn;
INDEXNAME yn[4]={"m","n","j","i"}; y.names=yn;
INDEXNAME cn[2]={"i","j"};
// z(m,n,k,l) = sum_ij x(i,j,k,l) y(m,n,j,i); remaining indices
    of y come first
Tensor<double> z = x.contractions(y,NRVec<INDEXNAME>(cn));
// result in a prescribed index order, antisymmetrized in k,l:
// w(k,l,m,n) = z(m,n,k,l) - z(m,n,l,k)
Tensor<double> w(shape);
INDEXNAME wn[4]={"k","l","m","n"}; w.names=wn;
w.add_permuted_contractions("k/l",x,y,1.,0.);
```

Tucker decomposition:

```
NRVec<INDEXGROUP> sh(3);
for(int i=0; i<3; ++i) {sh[i].number=1; sh[i].symmetry=0; sh[i].
    offset=0; sh[i].range=n+i;}
Tensor<double> a(sh); a.randomize(1.);
Tensor<double> a0(a); a0.copyonwrite();       // keep the
    original
NRVec<NRMat<double> > u = a.Tucker(1e-12);    // a now holds the
    core tensor
Tensor<double> b = a.inverseTucker(u);        // b == a0
Tensor<double> c = a.linear_transform(u);     // the same
```

Conversion from a dense four-index object (chapter 12):

```
fourindex_dense<nosymmetry,double,int> f(n);
f.randomize(1.);
Tensor<double> t4 = fourindex2tensor(f);     // f(i,j,k,l) == t4(
    k,l,i,j), indices from 1
fourindex_dense<nosymmetry,double,int> ff;
tensor2fourindex(t4,ff);
```

11.13 Pitfalls

- The leftmost index is the fastest; a tensor built from a matrix is its transpose in index notation: `Tensor<T>(x)(i,j)==x(j,i)`.
- `norm()`, `operator+=` etc. act on the stored data only; the norm of a compressed tensor differs from the norm of its flattened form.

- Results of contractions have the remaining indices of the *second* tensor first.
- `lhs()` writes require an unshared tensor: call `copyonwrite()` after copying. There is no `operator|=`.
- Default-constructed `INDEXGROUP`s are uninitialized; set all members.
- Operations based on `loopover()` use static global variables to pass data to the callbacks; they are neither reentrant nor thread safe.
- Index names longer than 7 characters are not NUL-terminated.
- An outer product of a named and an unnamed tensor raises an error (inconsistent number of names).

Chapter 12

Four-index quantities

Header: `fourindex.h, fourindex.cc`

12.1 Purpose and main idea

The header provides containers for four-index quantities with permutational symmetry, typically two-electron integrals, coupled-cluster T_2 amplitudes and two-particle (transition) density matrices. Only permutationally non-equivalent elements are stored. There are three kinds of classes:

- `fourindex<I,T>`: in-core *sparse* storage, a reference-counted linked list of elements `(i,j,k,l,value)` with indices of type `I` (e.g. `unsigned char` or `int`);
- `fourindex_ext<I,T>`: the same list stored in a binary file, read and written in large page-aligned blocks;
- `fourindex_dense<S,T,I>`: *dense* in-core storage for a given symmetry type `S`, derived from `NRMat<T>` or `NRSMat<T>` ("supermatrix" of index pairs).

Iterators run over the stored elements or over all their symmetry-equivalent permutations, so that algorithms like Fock matrix builds can be written without caring about the symmetry. Conversions between `fourindex_dense` and the general `Tensor` class are described in chapter 11.

Indices are opaque to the sparse classes (they are never compared with the dimension), so both 0- and 1-based indexing are possible there; the dense classes use indices from 1.

Only `fourindex<unsigned char,double>`, `fourindex_ext<unsigned char,double>` and `fourindex_dense<nosymmetry,double,unsigned char>` are explicitly instantiated in fourindex.cc; everything else is implemented in the header and instantiated on use, except `fscanf()` and `fprintf()` which exist only for `fourindex<unsigned char,double>`.

12.2 Symmetry types

```
typedef enum {undefined_symmetry=-1, nosymmetry=0, ...}
    fourindexsymtype;
```

Each symmetry type is defined by the list of index permutations (with a sign) generating the equivalent elements, in the static table `fourindex_permutations[fourindex_n_symmetrytypes][max_fourindex_permnumbers+1][5]`; an entry (p_0, p_1, p_2, p_3, s) means that the element with indices $(I_{p_0}, I_{p_1}, I_{p_2}, I_{p_3})$ equals s times the stored element $(I_0, I_1, I_2, I_3) = (i, j, k, l)$. `fourindex_permnumbers[]` gives the number of permutations. Several enumerators are aliases (same value).

enumerator(s)	perms	equivalent elements
`nosymmetry`	1	none
`twoelectronrealmullikan`, `twoelectronrealmullikanAA`	8	$(ij\|kl) = (ji\|kl) = (ij\|lk) = (kl\|ij) = \ldots$ (real integrals, Mulliken notation)
`twoelectronrealdirac`	8	$\langle ij\|kl\rangle = \langle kj\|il\rangle = \langle il\|kj\rangle = \langle kl\|ij\rangle = \langle ji\|lk\rangle = \ldots$ (Dirac notation)
`T2ijab_aces`, `trdm2AA`	4	antisymmetric in i, j and in k, l
`antisymtwoelectronrealdirac`, `rdm2AA`	8	$\langle ij\|\|kl\rangle$: antisymmetric in i, j and in k, l, symmetric in the exchange of the pairs
`T2IjAb_aces`, `trdm2AB`	1	none (different index ranges)
`twoelectronrealmullikanAB`	4	symmetric in i, j and in k, l, no pair exchange (integrals over two different spin orbital sets)
`T2ijab_unitary`	8	antisymmetric in i, j and in k, l, antisymmetric in the pair exchange (antihermitean exponent)
`T2IjAb_unitary`	2	$(kl, ij) = -(ij, kl)$
`antisymtwoelectronrealdiracAB`, `rdm2AB`, `twoelectroncomplexmullikan_without_conjugation`	2	symmetric in the pair exchange $(ij, kl) = (kl, ij)$
`twoelectronrealmullikan reducedsymAA`, `rdm2SS`	4	$(ij\|kl) = (kl\|ij) = (ji\|lk) = (lk\|ji)$ (e.g. unitary downfolded CC effective integrals)
`twoelectronrealmullikan reducedsymAB`	2	$(ij\|kl) = (ji\|lk)$ (e.g. spin-summed 2-RDM)

enumerator(s)	perms	equivalent elements
`twoelectroncomplexmullikan`	4	$(ij\|kl) = (kl\|ij)$, $(ji\|lk) = (lk\|ji) = (ij\|kl)^*$

The last entry of each permutation is applied by `fourindex_permfactor()`: ± 1 is a sign, ± 2 means complex conjugation (with the sign), used for `twoelectroncomplexmullikan`: $(ij|kl) = (kl|ij) = (ji|lk)^* = (lk|ji)^*$.

`template<class I, class T> void symmetry_faktor(const fourindexsymtype symmetry, const union packed_index<I> &index, T &elem)`

multiply `elem` by the factor 1/2 for each coincidence of indices which makes two of the generated permutations identical, so that summing over all permutations generated by a `piterator` counts every distinct element exactly once. Implemented for all symmetry types: the element is scaled by $1/n$, n the number of the type's permutations leaving the index tuple unchanged (e.g. $1/(1+m)$, $m \in \{0,1,3\}$, for `twoelectroncomplexmullikan` and `twoelectronrealmullikanreducedsymAA`); for coincidences making an element vanish by antisymmetry no scaling is applied (the generated contributions cancel).

12.3 Element records

`template<class I> struct indiv_index {I i; I j; I k; I l;}`

the four indices (packed with GNU C).

`template<class I> union packed_index {I packed[4]; struct indiv_index<I> indiv;}`

the indices accessible by name or as an array.

`template<class I, class T> struct matel4 {T elem; matel4 *next; union packed_index<I> index;}`

list element of `fourindex`; also the type returned by the permutation iterators.

`template<class I, class T> struct matel4stored {T elem; union packed_index<I> index;}`

packed record of the binary file format (used by `fourindex::put/get`, `fourindex_ext` and `fourindex_dense::putext`).

The binary format is thus a raw sequence of `matel4stored<I,T>` records (machine dependent, no header unless written by `fourindex::put(fd,true)`).

12.4 In-core sparse storage: `fourindex<I,T>`

12.4.1 Reference counting

`fourindex` is reference counted: the copy constructor and `operator=` share the list, `copyonwrite()` copies it if shared. There is no `operator|=`. `permuteme()`, `operator+=` and `join()` call `copyonwrite()`, `resize()` detaches. The `add()` functions (and therefore `get()` and `operator»`) are element-level access by design: they prepend new elements to this object's list without `copyonwrite()`, so call `copyonwrite()` once before adding to a possibly shared object (otherwise the copies diverge while sharing one reference count). Writing through the non-const `iterator` modifies the shared list; call `copyonwrite()` first. A default-constructed `fourindex` is empty but defined and can be filled by `add()` or `get()`.

12.4.2 Construction and basic operations

`fourindex()`
: undefined (no list, symmetry `undefined_symmetry`, dimension 0).

`fourindex(const I n, const fourindexsymtype symmetry0=undefined_symmetry, const I terminator0= -1, const bool doscaling0=true)`
: empty list for dimension (number of orbitals) `n`, with symmetry, terminator for text I/O and scaling flag for `piterator`.

`fourindex(const fourindex &rhs); fourindex & operator=(const fourindex &rhs)`
: shallow copy (list shared, all attributes including `symmetry` copied).

`~fourindex()`
: deletes the list when the last reference disappears.

`void resize(const I n)`
: discard all elements (detach if shared) and set the dimension; `n` must be positive.

`void dealloc()`
: intended to release the list; calls `resize(0)`, which raises `laerror`.

`void copyonwrite()`
: copy the list if shared (the order of elements is reversed by copying).

`int getcount() const`
: reference count (0 if undefined).

`I size() const`
: the dimension `n`, *not* the number of elements.

`unsigned long length() const`
: number of stored elements.

`matel4<I,T> *getlist() const`
: the head of the list.

`void setsymmetry(fourindexsymtype s); fourindexsymtype getsymmetry() const`
: symmetry type.

`void setterminator(const I terminator0); I getterminator() const`
: index value marking the end of a text list.

`void setscaling(const bool doscaling0)`
: whether `piterator` applies `symmetry_faktor`.

`void add(const I i, const I j, const I k, const I l, const T elem); void add(const union packed_index<I> &index, const T elem); void add(const I (&index)[4], const T elem); void add(const matel4<I,T> &rhs); void add(const matel4stored<I,T> &rhs)`
: prepend one element (no check for duplicates or canonical order). The iteration order is thus the reverse of the insertion order.

`fourindex & operator+=(const fourindex &rhs)`
: append (copy) all elements of `rhs`; dimensions must agree, symmetries are not checked.

`fourindex & join(fourindex &rhs)`
: move the list of an unshared `rhs` to the end of this list; `rhs` is emptied. Cheaper than `+=`.

`void permuteme(const NRPerm<int> &p, const bool inverse=false)`
: relabel all indices $i \to p_i$ (orbital permutation); `p.size()` must equal the dimension and the indices must be 1-based.

`void swap(fourindex &rhs)`
: exchange contents without copying.

`+=` and `join()` require equal symmetries of the operands (`laerror` otherwise); an empty object with undefined symmetry adopts the symmetry of the right-hand side.

12.4.3 Iterators

`class iterator; iterator begin() const; iterator end() const`
: forward iterator over the stored elements; `*it` and `it->` give `matel4<I,T>` (modifiable through a non-const iterator), pre- and postincrement, `==`, `!=`.

`class piterator; piterator pbegin() const; piterator pend() const`
: iterates over all permutations of all stored elements as defined by the symmetry type; `*p`/`p->` give a const `matel4<I,T>` with permuted indices and the element multiplied by the sign and, if scaling is on, by `symmetry_faktor` for repeated indices. Contributions of all generated elements can therefore simply be accumulated. Only preincrement; test the end by `p.notend()` or `p.end()` (comparing with `pend()` is less efficient).

`class qiterator; qiterator qbegin(); qiterator qend() const`
: iterates over the *distinct* permutations of each stored element (duplicates arising from equal indices are skipped, no scaling); sign applied. Only preincrement; use `notend()`/`end()`.

The symmetry must be set (not `undefined_symmetry`) for `piterator` and `qiterator`.

12.4.4 Input and output

`unsigned long put(int fd, bool withattr=true) const`
: write all elements as `matel4stored` records, preceded (if `withattr`) by the symmetry and the dimension, each padded to the size of a union of `fourindexsymtype`, `I` and `T`; returns the number of elements.

`unsigned long get(int fd, bool withattr=true)`
: read the same format until end of file, *adding* the elements to the list; returns their number.

`template <class I, class T> std::ostream& operator«(std::ostream &s, const fourindex<I,T> &x)`
: text output: the dimension, lines `i j k l value`, and a final line with four terminators and 0.

`template <class I, class T> std::istream& operator»(std::istream &s, fourindex<I,T> &x)`
: read the text format; `x` is resized to the read dimension (the symmetry is not set) and reading stops at the first line containing the terminator of `x` as an index.

`void fscanf(FILE *f); void fprintf(FILE *f, char *format) const`
: C-style text I/O, only for `fourindex<unsigned char,double>`; line format `value i j k l`. `fscanf` adds elements until a line in which any index equals the terminator (compared after conversion to the index type, so -1 in the file matches the default terminator 255); `fprintf` applies `format` (which must consume a `double` and four `int`s) to each element and ends with a line `0 t t t t` with the terminator t. With terminator 0 this is the FCIDUMP-like convention (blocks separated by zero lines, read by consecutive `fscanf` calls; a one-electron line $i\,j\,0\,0$ also ends a block).

12.5 Disc storage: `fourindex_ext<I,T>`

`fourindex_ext` accesses a file of `matel4stored<I,T>` records through an already opened file descriptor. It is not copyable (private default constructor, copy constructor and assignment) and is not reference counted. Reading and writing go through one internal page-aligned buffer whose size is a common multiple of the record size, the page size and the file system block size; the class tries to `mlock` the buffer (failure ignored). The buffer is shared by all iterators of an object, therefore only one iteration may be active at a time.

The file may be opened with `O_DIRECT` to avoid filling the page cache when a large file is read once sequentially; this is supported: the buffer is page-aligned and its size is a multiple of the page and filesystem block size, as `O_DIRECT` requires (tested on Linux 6.6 with ext4, also for a file size that is not a multiple of the block size; Linux 2.4 is not supported).

`fourindex_ext(const int file, const fourindexsymtype s=undefined_symmetry, const I n=0, const unsigned int b=1024, const I terminator0= -1, const bool doscaling0= true)`
: attach to the open descriptor `file`; `b` is the approximate buffer size in records.

`~fourindex_ext()`
: frees the buffer; it does *not* flush pending output and does not close the file.

`void setsymmetry(fourindexsymtype s); fourindexsymtype getsymmetry() const; void setterminator(const I terminator0); I getterminator() const; void setscaling(const bool doscaling0)`
: as for `fourindex`.

`void resize(I n); I size() const`
: set/get the dimension (no effect on the file).

`void rewind() const`
: seek to the beginning of the file.

`void put(const matel4stored<I,T> x); void put(I i, I j, I k, I l, const T &elem)`
: buffered write of one record.

`void flush()`
: write the buffered records; must be called after the last `put()`.

`class iterator; iterator begin() const; iterator end() const`
: rewind and iterate over all records of the file; `*it`/`it->` give `const matel4stored<I,T>`; preincrement only.

`class piterator; piterator pbegin(); piterator pend() const`
: all permutations, as for `fourindex`.

`class qiterator; qiterator qbegin(); qiterator qend() const`
: distinct permutations, as for `fourindex`.

`template <class I, class T> std::ostream& operator«(std::ostream &s, const fourindex_ext<I,T> &x)`
: text output of the whole file in the format of `fourindex`.

`template <class I, class T> std::istream& operator»(std::istream &s, fourindex_ext<I,T> &x)`
: read the text format and append the elements to the file (with `flush()`).

12.6 Dense storage: `fourindex_dense<S,T,I>`

```
template<fourindexsymtype S, class T, class I> class
    fourindex_dense;  // only specializations exist
```

The dense classes are derived publicly from `NRMat<T>` or `NRSMat<T>`, so the whole interface of the base class (arithmetic, `norm()`, `put()/get()`, `randomize()`, `|=`, `copyonwrite()`, ..., chapters 5 and 6) is available, and they follow its reference counting and copy-on-demand scheme. The third template parameter `I` is the index type of the sparse classes and matters only for the constructors from `fourindex<I,T>` and `fourindex_ext<I,T>` (named `DUMMY` in some places). `LA_traits<fourindex_dense<S,T,I> >` defines `elementtype` and `normtype`. The four-index `operator()` returning a reference is an element access in the sense of section 3.1: call `copyonwrite()` before writing into a possibly shared object (a DEBUG build raises `laerror` for a shared object). The same holds by design for the element-level members `set()`, `add()` and `add_unique()`: they do not call `copyonwrite()`, the DEBUG count check reports a write into shared data.

All specializations use indices from 1.

12.6.1 twoelectronrealmullikanAB

Base `NRMat<T>` of dimension $n(n+1)/2 \times n(n+1)/2$; row = symmetric pair (ij), column = pair (kl).

`fourindex_dense(); explicit fourindex_dense(const int n); fourindex_dense(const T &a, const int n); fourindex_dense(const T *a, const int n)`
: empty; uninitialized; filled with `a`; copied from an array.

`fourindex_dense(const NRMat<T> &rhs)`
: reinterpret a matrix (shared data); also an implicit conversion from the base class.

`fourindex_dense(const fourindex<I,T> &rhs); fourindex_dense(const fourindex_ext<I,T> &rhs)`
: expand a sparse list of the same symmetry type (`laerror` otherwise); missing elements are zero.

`T& operator()(unsigned int i, unsigned int j, unsigned int k, unsigned int l); const T& operator()(unsigned int i, unsigned int j, unsigned int k, unsigned int l) const`
: element $(ij|kl)$.

`void resize(const int n); int nbas() const`
: resize for n orbitals; the number of orbitals.

`void putext(int f, T thr=1e-15)`
: write the non-equivalent elements with $|x| >$ `thr` as `matel4stored<I,T>` records to the file descriptor `f` (readable by `fourindex_ext` or `fourindex::get(fd,false)`).

12.6.2 twoelectronrealmullikan

Base `NRSMat<T>` of dimension $n(n+1)/2$ (symmetric matrix of symmetric pairs), i.e. full 8-fold symmetry. The interface is identical to `twoelectronrealmullikanAB` with `NRSMat<T>` in place of `NRMat<T>`.

12.6.3 nosymmetry

Base `NRMat<T>` of dimension $n^2 \times n^2$; element (i,j,a,b) is the matrix element $((j-1)n+i-1,\ (b-1)n+a-1)$.

`fourindex_dense(); explicit fourindex_dense(const int nnn); explicit fourindex_dense(const int nnn, const NRMat<T> &mat)`
: empty; uninitialized; reinterpret `mat` (shared).

`T& operator()(unsigned int i, unsigned int j, unsigned int a, unsigned int b); const T& operator()(...) const`
: element access.

`void resize(const int nnn); unsigned int nbas() const`
: dimension.

`void print(std::ostream &out) const`
: print all elements as lines `i j a b value`.

12.6.4 T2IjAb_aces

Spin block t_{Ij}^{Ab} of CC amplitudes in the ACES storage order; base `NRMat<T>` of dimension (`noca`·`nocb`)×(`nvra`·`nvrb`), element (i,j,a,b) at $((j-1)\,\texttt{noca}+i-1,\ (b-1)\,\texttt{nvra}+a-1)$; i,a are α and j,b are β indices.

`unsigned int noca, nocb, nvra, nvrb`
: public dimensions (occupied/virtual, α/β).

`fourindex_dense(); explicit fourindex_dense(const int nocca, const int noccb, const int nvrta, const int nvrtb); explicit`

`fourindex_dense(const int nocca, const int noccb, const int nvrta, const int nvrtb, const NRMat<T> &data)`
: empty; uninitialized; reinterpret `data` (shared, dimensions checked).

`void resize(const int nocca, const int noccb, const int nvrta, const int nvrtb)`
: resize.

`T& operator()(unsigned int i, unsigned int j, unsigned int a, unsigned int b); const T& operator()(...) const`
: element access.

`void print(std::ostream &out) const`
: print all elements.

12.6.5 T2ijab_aces

Same-spin amplitudes t_{ij}^{ab}, antisymmetric in i, j and in a, b; base `NRMat<T>` of dimension $\binom{\texttt{nocc}}{2} \times \binom{\texttt{nvrt}}{2}$ storing the elements with $i > j$, $a > b$.

`unsigned int nocc, nvrt`
: public dimensions.

`fourindex_dense(); explicit fourindex_dense(const int noc, const int nvr); explicit fourindex_dense(const int noc, const int nvr, const NRMat<T> &data)`
: empty; uninitialized; reinterpret `data` (shared, dimensions checked).

`void resize(const int noc, const int nvr)`
: resize.

`T operator()(unsigned int i, unsigned int j, unsigned int a, unsigned int b) const`
: element value with the sign from the index order; 0 for $i = j$ or $a = b$. No l-value access because of the sign.

`void set(unsigned int i, unsigned int j, unsigned int a, unsigned int b, T elem); void add(unsigned int i, unsigned int j, unsigned int a, unsigned int b, T elem)`
: set or increment an element given in any index order (the sign of the permutation is applied); elements with $i = j$ or $a = b$ are identically zero and ignored (a nonzero value raises `laerror` in DEBUG builds).

`void print(std::ostream &out) const`
: print the stored elements.

12.6.6 antisymtwoelectronrealdiracAB

Base `NRSMat<T>` of dimension n^2: symmetric matrix of ordered pairs; element (i, j, a, b) is the matrix element $((j-1)n+i-1, (b-1)n+a-1)$, i.e. symmetric only under exchange of the pairs. Here i, a are α and j, b β indices.

`fourindex_dense(); explicit fourindex_dense(const int n); explicit fourindex_dense(const int n, const NRSMat<T> &data)`
: empty; uninitialized; reinterpret `data` (shared, dimension checked).

`T& operator()(unsigned int i, unsigned int j, unsigned int a, unsigned int b); const T& operator()(...) const; void set(unsigned int i, unsigned int j, unsigned int k, unsigned int l, T elem)`
: element access.

`void resize(const int n); unsigned int nbas() const`
: dimension.

`void print(std::ostream &out) const`
: print the non-equivalent elements.

12.6.7 antisymtwoelectronrealdirac

Antisymmetrized integrals $\langle ij \| kl \rangle$ with the full 8-fold symmetry; base `NRSMat<T>` of dimension $\binom{n}{2}$ (symmetric matrix of antisymmetric pairs, `ASMat_index_1` of smat.h).

`fourindex_dense(); explicit fourindex_dense(const int n); explicit fourindex_dense(const int n, const NRSMat<T> &data); fourindex_dense(const T &a, const int n); fourindex_dense(const T *a, const int n)`
: empty; uninitialized; reinterpret `data` (shared); filled with `a`; copied from an array.

`fourindex_dense(const fourindex<I,T> &rhs); fourindex_dense(const fourindex_ext<I,T> &rhs)`
: build $\langle pq \| rs \rangle = (pr|qs) - (ps|qr)$ from Mulliken integrals: `rhs` must have symmetry `twoelectronrealmullikan`.

`const T operator()(unsigned int i, unsigned int j, unsigned int k, unsigned int l) const`
: element value with sign, 0 for $i = j$ or $k = l$. No l-value access.

`void set(unsigned int i, unsigned int j, unsigned int k, unsigned int l, T elem); void add(unsigned int i, unsigned int j, unsigned int k, unsigned int l, T elem)`
: set/increment with the sign from the index order; elements with $i = j$ or $k = l$ are ignored, or raise `laerror` if the macro `FORBID_NONEXISTENT_ASSIGNMENTS` is defined.

`void add_unique(unsigned int i, unsigned int j, unsigned int k, unsigned int l, T elem)`
: increment only if (i, j, k, l) is the canonical representative ($i > j$, $k > l$, pair $(ij) \geq (kl)$), otherwise do nothing.

`void resize(const int n); int nbas() const`
: dimension.

`void print(std::ostream &out) const`
: print the non-equivalent elements.

12.6.8 Other symmetry types

No dense specializations exist for `twoelectronrealdirac`, the `..._unitary` types, the `...reducedsym...` types and the complex types (source TODO: "implement fourindex_dense for unitary t2 symmetry types").

12.7 Examples

A sparse list of real Mulliken integrals, a contraction over all equivalent elements, and conversion to dense storage via a disc file:

```
int n=3;
fourindex<int,double> f(n,twoelectronrealmullikan);
f.add(2,1,1,1,0.5);          // (21|11)
f.add(2,2,1,1,0.3);          // (22|11)
f.add(2,1,2,1,0.1);          // (21|21)

// Coulomb-like matrix J_ij = sum_kl (ij|kl) D_kl over ALL
    elements
NRSMat<double> D(1.,n);
NRMat<double> J(0.,n,n);
for(fourindex<int,double>::piterator p=f.pbegin(); p.notend(); ++
    p)
    J(p->index.indiv.i-1,p->index.indiv.j-1) +=
        p->elem * D(p->index.indiv.k-1,p->index.indiv.l-1);

// the distinct elements (here 10), unscaled
int cnt=0;
for(fourindex<int,double>::qiterator q=f.qbegin(); q.notend(); ++
    q) ++cnt;

// via a binary file to dense storage
int fd=open("ints.bin",O_CREAT|O_TRUNC|O_RDWR,0644);
f.put(fd,false);                                  // raw records
     only
fourindex_ext<int,double> fe(fd,twoelectronrealmullikan,n);
fourindex_dense<twoelectronrealmullikan,double,int> fdense(fe);
```

```
const fourindex_dense<twoelectronrealmullikan,double,int> &fc=
    fdense;
cout << fc(1,2,2,1) << endl;                          // 0.1 =
    (21|21)
close(fd);
```

Writing a file through `fourindex_ext`:

```
int fd2=open("ints2.bin",O_CREAT|O_TRUNC|O_RDWR,0644);
{
fourindex_ext<int,double> fo(fd2,twoelectronrealmullikan,n);
fo.put(2,1,1,1,0.5);
fo.put(2,2,1,1,0.3);
fo.flush();                                           // not done by
     the destructor
for(fourindex_ext<int,double>::iterator it=fo.begin(); it!=fo.end
    (); ++it)
    cout << it->elem << endl;
}
close(fd2);
```

See also the conversion examples `fourindex2tensor`/`tensor2fourindex` in chapter 11.

12.8 Pitfalls

- `fourindex::size()` is the number of orbitals; the number of elements is `length()`.
- The sparse classes do not enforce canonical index order or uniqueness; adding two equivalent elements counts them twice in the iterators and in the dense constructors the later one wins.
- `piterator` scaling is not implemented for all symmetry types; use `setscaling(false)` if `laerror` reports it.
- `fourindex_ext` must be flushed explicitly after writing; only one iterator per object may be active.
- Constructors of the dense classes from `NRMat`/`NRSMat` share the data.

Chapter 13

Permutations and the symmetric group

Header: `permutation.h, permutation.cc, qsort.h`

This chapter describes the classes for permutations (in the one-line and in the cycle notation), for linear combinations of permutations (elements of the group algebra of S_n), for partitions and Young tableaux, the character table of S_n, and the sorting templates which return the parity of the sorting permutation. All class templates are explicitly instantiated in `permutation.cc` for `T = int` and `T = unsigned int`; `WeightPermutation<T,R>` and `PermutationAlgebra<T,R>` for `T = int` with `R = int`, `double`, `std::complex<double>`. Ranks, counts of generated objects and class sizes have the type

```
typedef unsigned long long PERM_RANK_TYPE;
```

(declared outside of namespace `LA`); $n!$ overflows it for $n > 20$.

13.1 Conventions

- **Permutations are numbered from 1.** `NRPerm<T>`, `CyclePerm<T>`, `Partition<T>`, `CompressedPartition<T>` and `YoungTableaux<T>` are derived from `NRVec_from1` (chapter 4), i.e. `p[1]` ... `p[n]`. When a permutation is applied to a vector or matrix indexed from 0, the offset 1 is subtracted internally (section 13.8).
- The one-line notation stores the images: `p[i]` $= p(i)$. A cycle $(c_1\, c_2 \ldots c_l)$ maps $c_1 \mapsto c_2 \mapsto \cdots \mapsto c_l \mapsto c_1$.
- Product: $(pq)(i) = p(q(i))$, i.e. in `p*q` the right operand `q` is applied first.

- Parity is returned as $+1$ (even) or -1 (odd) by the `parity()` members; the sorting templates of section 13.10 return the parity bit 0 (even) or 1 (odd).
- The generators (`generate_all`, ...) call a user callback for each generated object and return their number. They keep their state in file-static variables, so they are neither reentrant nor thread safe; do not start another generator of the same kind from inside the callback.

Copy on demand. All classes of this chapter except `WeightPermutation`, `Sn_characters`, `CycleIndex` (which are plain aggregates of such objects) are derived from `NRVec/NRVec_from1` and are reference counted as described in section 3.1: copy construction and `=` share the data, `|=` copies; before writing through `operator[]` the user has to call `copyonwrite()`. Nested types (`CyclePerm`, `YoungTableaux`, `PermutationAlgebra`) are vectors of vectors; their `copyonwrite()` is deep (it also detaches the inner vectors). Most modifying members call `copyonwrite()`; the exceptions found are listed in the pitfalls (section 13.12).

13.2 NRPerm: permutations in one-line notation

`template <typename T> class NRPerm : public NRVec_from1<T>` stores a permutation of $\{1, \ldots, n\}$ as the vector of images. Relational operators `== != < >` (lexicographic) and the I/O operators `«`, `»` (the size followed by the elements) are inherited from `NRVec` (section 3.7). Many members check `is_valid()` of their operands only in DEBUG builds.

13.2.1 Construction and conversion

`NRPerm(); NRPerm(const int n)`
: empty; size n, elements uninitialized (call `identity()`).

`template<int SIZE> explicit NRPerm(const T (&a)[SIZE])`
: from an array or brace list, e.g. `NRPerm<int> p({1,4,2,3})`.

`NRPerm(const NRVec_from1<T> &rhs); NRPerm(const T *a, const int n)`
: from a vector (shares data) or from n elements at `a` (copy).

`explicit NRPerm(const CyclePerm<T> &rhs, const int n=0)`
: from the cycle notation; size `n`, or `rhs.max()` if `n==0` (fixed points beyond the largest moved element are then lost).

`explicit NRPerm(const int type, const NRVec_from1<T> &inversions)`
: reconstruct from an inversion table of the given `type` (see `inversions()`).

`explicit NRPerm(const int n, const PERM_RANK_TYPE rank)`
: unranking: the permutation of size n with the lexicographic rank `rank` $\in [0, n! - 1]$ (checked in DEBUG).

13.2.2 Basic operations

`void resize(int n); int size() const`
: as for `NRVec`.

`void identity(); void identity(int n)`
: set to the identity (of size n); calls copyonwrite.

`bool is_valid() const`
: true if the elements are a permutation of $1 \dots n$.

`bool is_identity() const`
: true for the identity (also for size 0).

`int parity() const`
: $+1/-1$; returns 0 for an invalid permutation. Counts inversions for $n < 10$, sorts a copy otherwise.

`NRPerm inverse() const`
: p^{-1}.

`void inverseme()`
: in-place inversion, $O(n^2)$ cycle-leader algorithm (cf. arXiv:1901.01926); detaches shared data first.

`void reverse_cycle(int i); int cycle_leader(int i) const`
: auxiliary for `inverseme()`: reverse in place the cycle containing i (no `copyonwrite()`, the caller detaches first); return the smallest element of that cycle.

`NRPerm reverse() const`
: the sequence in backward order: $r(i) = p(n - i + 1)$.

`NRPerm operator*(const NRPerm &q) const`
: product pq, $(pq)(i) = p(q(i))$; sizes must agree.

`NRPerm multiply(const NRPerm<T> &q, bool inverse) const`
: pq, or pq^{-1} if `inverse` (without forming q^{-1}).

`NRPerm operator*(const CyclePerm<T> &r) const`
: product with a permutation in cycle notation, converted to size `size()`.

`template<typename R> PermutationAlgebra<T,R> operator*(const PermutationAlgebra<T,R> &pa) const`
: left multiplication of each term of `pa` by p.

`NRPerm conjugated_by(const NRPerm &q, bool reverse=false) const`
: $q^{-1}pq$, or qpq^{-1} if `reverse`.

`NRPerm commutator(const NRPerm &q, bool inverse=false) const`
: $p^{-1}q^{-1}pq$, or $q^{-1}p^{-1}qp$ if `inverse`.

`NRPerm pow(const int n) const`
: p^n, also for $n \le 0$ (by repeated squaring, `power()` of `nonclass.h`).

`CompressedPartition<T> cycles() const`
: cycle type: number of cycles of length $1, 2, \ldots, n$ (fixed points included).

`NRPerm operator&(const NRPerm &rhs) const`
: direct sum: $r = (p(1), \ldots, p(n), n+q(1), \ldots, n+q(m))$ with `q=rhs`, $m =$ `rhs.size()`.

`NRPerm operator|(const NRPerm &rhs) const`
: $r = (n + q(1), \ldots, n + q(m), p(1), \ldots, p(n))$, i.e. the renumbered `rhs` comes first.

`NRPerm cutleft(int n) const`
: inverse of `&` with an identity on the left: requires $p(i) = i$ for $i \le n$ (laerror otherwise), returns the remaining part renumbered to $1 \ldots$`size()`$-n$.

`NRPerm cutright(int n) const`
: requires the last n elements to be fixed points and removes them.

`void randomize(void)`
: uniformly random permutation (Fisher–Yates shuffle with `RANDINT32()`); the size must be set.

13.2.3 Ranking and inversion tables

`PERM_RANK_TYPE rank() const`
: lexicographic rank, 0 for the identity and $n! - 1$ for $(n, \ldots, 1)$.

`NRVec_from1<T> inversions(const int type, PERM_RANK_TYPE *prank=NULL) const`
: inversion table I of size n: `type` 0: $I_j =$ number of elements greater than j standing left of the value j (Knuth's inversion table); 1: number of elements smaller than j standing right of the value j; 2: $I_j = |\{k < j : p(k) > p(j)\}|$; 3: $I_j = |\{k > j : p(k) < p(j)\}|$ (Lehmer code; rank $= \sum_j I_j\,(n-j)!$). Inverse: `NRPerm(type,I)`. If `prank` is given (only for type 3, otherwise `laerror`), `*prank` is set to the rank computed from the Lehmer code, the same value as `rank()`.

13.2.4 Generating all permutations

The generators overwrite `*this`, which is passed to the callback `void callback(const NRPerm<T>&)`; a `NULL` callback only counts.

`bool next()`
: replace by the lexicographically next permutation (works also for repeated elements); returns false (and leaves `*this` unchanged) for the last one. Calls copyonwrite.

`PERM_RANK_TYPE generate_all_lex(void (*callback)(const NRPerm<T>&))`
: all $n!$ permutations in lexicographic order, starting from the identity, using `next()`.

`PERM_RANK_TYPE generate_all(void (*callback)(const NRPerm<T>&), int parity_select=0)`

all permutations by adjacent transpositions (plain changes, Steinhaus–Johnson–Trotter order [36]), so the parity alternates; fastest but not lexicographic. This is Knuth's Algorithm P. `parity_select`: 0 all, +1 (positive) only even, −1 (negative) only odd permutations; then $n!/2$ is returned.

`PERM_RANK_TYPE generate_all2(void (*callback)(const NRPerm<T>&))`

recursive generation, not lexicographic.

`PERM_RANK_TYPE generate_all_multi(void (*callback)(const NRPerm<T>&))`

all distinct permutations of a multiset: `*this` must be filled beforehand with the (possibly repeated) elements in any order; they are sorted and permuted in lexicographic order (Knuth's algorithm for multisets). The result is not a permutation in the strict sense.

`PERM_RANK_TYPE generate_restricted(void (*callback)(const NRPerm<T>&), const NRVec_from1<T> &classes, int restriction_type=0)`

permutations restricted with respect to a partitioning of $1 \ldots n$ into classes (`classes[i]` is the class label of i, size n): `restriction_type` 0: all; 1: only p with `classes[p(i)]==classes[i]` (permutations within classes); −1: only p with `classes[p(i)]!=classes[i]` for all i; −2: only permutations keeping the elements of each class in increasing order ("shuffles", the $n!/\prod_c n_c!$ coset representatives used in antisymmetrizers like $P(ij/kl)$).

`PermutationAlgebra<T,T> list_all(int parity_select=0)`

all permutations (order of `generate_all`) as a `PermutationAlgebra` with weights ± 1 = parity, i.e. the unnormalized antisymmetrizer; with `parity_select` ± 1 only the even/odd permutations, all with weight `parity_select`.

`PermutationAlgebra<T,T> list_all_lex()`

all permutations in lexicographic order (term k has rank k), all with weight 1, i.e. the unnormalized symmetrizer.

`PermutationAlgebra<T,T> list_restricted(const NRVec_from1<T> &classes, int restriction_type=0, bool inverted=false)`

the permutations of `generate_restricted`, weight = parity; with `inverted` the inverse permutations are stored.

13.3 CyclePerm: cycle notation

`template <typename T> class CyclePerm`, derived from `NRVec_from1< NRVec_from1<T> >`, is a list of cycles. The representation is not unique

(order of cycles, starting element, presence of cycles of length 1); `CyclePerm(const NRPerm&)` omits fixed points. The size of the permuted set is not stored: wherever it matters it is `max()` unless given explicitly.

`CyclePerm(); template<int SIZE> explicit CyclePerm(const NRVec_from1<T> (&a)[SIZE])`
: empty (the identity); from an array of cycles, e.g. `CyclePerm<int> c({NRVec_from1<int>({1,2}),NRVec_from1<int>({3,4,5})})`.

`explicit CyclePerm(const NRPerm<T> &rhs)`
: cycle decomposition, nontrivial cycles only.

`bool is_valid() const`
: no empty cycle, positive elements, no element repeated.

`bool is_identity() const; void identity()`
: test for no cycle longer than 1; set to the identity (no cycles).

`CyclePerm inverse() const`
: reverse all cycles.

`int parity() const`
: -1 for an odd number of even-length cycles.

`T max() const`
: largest element occurring.

`CompressedPartition<T> cycles(T n = 0) const`
: cycle type for size n (default `max()`), fixed points counted as 1-cycles.

`PERM_RANK_TYPE order() const`
: order in the group, the lcm of the cycle lengths.

`CyclePerm operator*(const CyclePerm &q) const`
: product (`q` applied first), computed through `NRPerm` of the size of the larger maximum of the two operands.

`NRPerm<T> operator*(const NRPerm<T> &r) const`
: mixed product, result of size `r.size()`.

`CyclePerm conjugated_by(const CyclePerm &q) const`
: $q^{-1}pq$.

`bool operator==(const CyclePerm &rhs) const`
: compares via conversion to `NRPerm` (so independent of the cycle representation).

`void simplify(bool keep1=false)`
: remove cycles of length 0 and 1 (with `keep1` only empty ones); detaches shared data first.

`CyclePerm pow(const int n, const bool keep1=false) const`
: p^n (also $n < 0$) computed cycle by cycle (a cycle of length l splits into $\gcd(n, l)$ cycles); with `keep1` the resulting fixed points are kept as 1-cycles.

`CyclePerm pow_simple(const int n) const`
: p^n through `NRPerm::pow`.

`void readfrom(const std::string &line)`
: parse cycles like `(1 2 3)(4,5)`; elements separated by blanks or commas, empty cycles skipped.

`std::istream & operator»(std::istream &s, CyclePerm<T> &x)`
: reads one line and calls `readfrom`.

`std::ostream & operator«(std::ostream &s, const CyclePerm<T> &x)`
: writes `(1 2 3)(4 5)` (no newline).

13.4 Weighted permutations and the group algebra

13.4.1 WeightPermutation

`template <typename T, typename R> class WeightPermutation` is a pair of public members `R weight; NRPerm<T> perm;`, i.e. the term $w\,p$ of the group algebra $\mathbb{R}S_n$ (or with coefficients of type `R`). The `LA_traits` specialization (section 3.3) declares it non-plain with `coefficienttype` and `elementtype R`, so that `NRVec` sorting and `NRVec_simplify` work on vectors of it.

`WeightPermutation(); explicit WeightPermutation(const NRPerm<T> &p, const R w=1)`
: weight 0 and empty permutation; $w\,p$.

`int size() const`
: size of the permutation.

`bool is_zero() const; bool is_scaledidentity() const; bool is_identity() const`
: $w = 0$; $p =$ identity; $w = 1$ and $p =$ identity.

`bool is_plaindata() const; void copyonwrite()`
: returns false; `perm.copyonwrite()`.

`void identity(); void identity(int n)`
: $w = 1$, $p =$ identity (of size n).

`WeightPermutation operator&(const WeightPermutation &rhs) const; ... operator|(...) const`
: direct sums of the permutations (as for `NRPerm`), weights multiplied.

`WeightPermutation operator*(const WeightPermutation &rhs) const; WeightPermutation operator*(const R &x) const`
: product of terms; scaling of the weight.

`bool operator==(const WeightPermutation &rhs) const; != < > <= >=`
: compare *only the permutations*, ignoring the weights (needed for sorting and simplification).

`WeightPermutation & operator=(const WeightPermutation &rhs)`
: memberwise (the permutation is shared).

`std::ostream & operator«(std::ostream &s, const WeightPermutation<T,R> &x); std::istream & operator»(...)`
: weight followed by the permutation.

13.4.2 PermutationAlgebra

`template <typename T, typename R> class PermutationAlgebra : public NRVec<WeightPermutation<T,R> >` represents $\sum_k w_k p_k$ as a vector of terms indexed *from 0*. Products and sums simplify the result; a canonical form (sorted by permutation, equal permutations merged, zero terms removed) is obtained by `simplify()`. Such objects are used e.g. by `Tensor::apply_permutation_algebra` (chapter 11) and by the `NRMat` constructor of section 13.8.

`PermutationAlgebra(); PermutationAlgebra(int n); PermutationAlgebra(const NRVec<WeightPermutation<T,R> > &x)`
: empty; n uninitialized terms; from a vector of terms.

`template <typename U> PermutationAlgebra(const PermutationAlgebra<T,U> &rhs)`
: conversion of the weight type (cast `(R)`).

`int size() const; void resize(int n); void copyonwrite()`
: as for `NRVec`.

`int sort(int direction = 0, int from = 0, int to = -1, int *permut = NULL, bool stable=false)`
: `NRVec::sort` of the terms by their permutations.

`void simplify(const typename LA_traits<R>::normtype thr=0)`
: sort, add weights of equal permutations, drop zero terms (`NRVec_simplify`; terms are dropped if $w = 0$ or if `abs2(w)<thr`, i.e. `thr` is compared with $|w|^2$).

`PermutationAlgebra operator*(const PermutationAlgebra &rhs) const`
: algebra product (all pairs of terms), simplified; permutation sizes must agree.

`PermutationAlgebra operator+(const PermutationAlgebra &rhs) const`
: sum (concatenation), simplified.

`PermutationAlgebra operator&(const PermutationAlgebra &rhs) const; ... operator|(...) const`
: all pairs of terms combined by `&` or `|` (tensor product of algebras acting on disjoint sets); not simplified.

`PermutationAlgebra operator*(const NRPerm<T> &rhs) const;` `operator&(const NRPerm<T> &rhs) const; operator|(const NRPerm<T> &rhs) const`
: applied to all terms: $\sum_k w_k\, p_k q$, $p_k \,\&\, q$, $p_k \,|\, q$.

`PermutationAlgebra cutleft(int n) const; PermutationAlgebra cutright(int n) const`
: `NRPerm::cutleft/cutright` of all terms.

`PermutationAlgebra conjugated_by(const NRPerm<T> &q, bool reverse=false) const; PermutationAlgebra commutator(const NRPerm<T> &q, bool inverse=false) const`
: applied to all terms.

`PermutationAlgebra &operator*=(const R &x); PermutationAlgebra operator*(const R &x) const`
: scale all weights.

`bool operator==(PermutationAlgebra &rhs)`
: simplifies *both* operands (hence non-const) and compares terms including weights.

`bool is_zero() const; bool is_scaled_identity() const; bool is_identity() const`
: tests valid for simplified objects only: no term; single term with identity permutation (and weight 1).

`void identity(int n)`
: the single term $1 \cdot \mathrm{id}_n$.

Inherited `NRVec` I/O prints the number of terms followed by the terms.

Related free functions.

`template <typename T> PermutationAlgebra<T,T> symmetrizer(int n);` `... antisymmetrizer(int n)`
: $\sum_{p\in S_n} p$ and $\sum_{p\in S_n} \mathrm{sgn}(p)\, p$ (unnormalized), built as Young operators of a single row/column.

`template<typename T> PermutationAlgebra<T,T> general_antisymmetrizer(const NRVec<NRVec_from1<T> > &classgroups, int restriction_type=0, bool inverted=false)`
: direct product (by `&`) of `list_restricted(classgroups[i], restriction_type, inverted)` for all groups; group i acts on the next `classgroups[i].size()` elements. Used for antisymmetrizers in coupled-cluster-like expressions (see the tensor chapter).

`template <typename T, typename R, typename U> void cast_permutation_algebra(PermutationAlgebra<T,R> &lhs, const PermutationAlgebra<T,U> &rhs)`
: conversion of the weight type into an existing object.

`template<typename T> NRMat<PERM_RANK_TYPE> Multable(T n)`
: multiplication table of S_n ($n! \times n!$) in the basis of permutations ordered by lexicographic rank: `r(i,j)` = rank of $p_i p_j$. Consistency (Latin square) is checked in DEBUG.

`template<typename T, typename R> NRMat<R> RegularRepresentation(const PermutationAlgebra<T,R> &a, const NRMat<PERM_RANK_TYPE> &mtable)`
: matrix of left multiplication by `a` in the regular representation: $R_{\mathrm{rank}(p_k p_j),\, j}$ += w_k; `mtable` from `Multable(n)`.

13.5 Partitions

Two representations of a partition $\lambda \vdash n$ are used, both as vectors of size n (indexed from 1): `Partition<T>` stores the parts in non-increasing order, padded with zeros ($[3,1]$ of 4 is `3 1 0 0`); `CompressedPartition<T>` stores the multiplicities m_i of parts of size i ($[3,1]$ is `1 0 1 0`), printed as `3,1`; $[2,1^2]$ is printed `2,1^2`. Partitions label the conjugacy classes of S_n (cycle types) and its irreducible representations.

13.5.1 CompressedPartition

`CompressedPartition(); CompressedPartition(const int n); template<int SIZE> explicit CompressedPartition(const T (&a)[SIZE])`
: empty; size n uninitialized; from an array.

`explicit CompressedPartition(const Partition<T> &rhs)`
: conversion.

`T sum() const; T nparts() const; T nclasses() const`
: $\sum_i i\, m_i$; number of parts $\sum_i m_i$; number of distinct part sizes.

`bool is_valid() const`
: `size()==sum()`.

`PERM_RANK_TYPE Sn_class_size() const`
: size of the conjugacy class of S_n with this cycle type, $n!/\prod_i i^{m_i} m_i!$.

`int parity() const`
: parity of permutations with this cycle type.

`std::ostream & operator«(std::ostream &s, const CompressedPartition<T> &x)`
: e.g. `3,2^2,1`.

13.5.2 Partition

`Partition(); Partition(const int n); template<int SIZE> explicit`

`Partition(const T (&a)[SIZE])`
: empty; size n uninitialized; from an array (zero-padded to n by the user).

`explicit Partition(const CompressedPartition<T> &rhs); explicit Partition(const YoungTableaux<T> &x)`
: conversion; the shape of a Young tableau.

`T nparts() const; bool is_valid() const`
: number of nonzero parts; `size()==sum()` and non-increasing.

`Partition adjoint() const`
: conjugate (transposed) partition λ'.

`PERM_RANK_TYPE Sn_irrep_dim() const`
: dimension of the irrep $[\lambda]$ of S_n (hook length formula).

`PERM_RANK_TYPE Un_irrep_dim(const int n) const`
: dimension of the irrep of $U(n)$ with the highest weight λ [37, eq. 10-25]; 0 if λ has more than n parts.

`int parity() const`
: parity of permutations with the cycle type λ.

`PERM_RANK_TYPE generate_all(void (*callback)(const Partition<T>&), int nparts=0)`
: all partitions of `size()` in reverse lexicographic order ($[n]$ first, $[1^n]$ last); `nparts`>0: exactly `nparts` parts, <0: at most $-$`nparts` parts, 0: all. Overwrites `*this`; returns the count.

`PERM_RANK_TYPE partitions(int n, int k= -1)`
: number of partitions of n into exactly k parts, or the total number $p(n)$ for $k=-1$ [38]; tabulated recursively up to $n=260$ (larger n only for $k \geq n/2$).

13.6 Young tableaux

`template <typename T> class YoungTableaux`, derived from `NRVec_from1< NRVec_from1<T> >`, stores the rows of a Young diagram (row i has length λ_i), each box holding a number (0 = empty).

`YoungTableaux(); explicit YoungTableaux(const Partition<T> &frame)`
: empty; empty (zero-filled) diagram of the shape `frame`.

`template<int SIZE> explicit YoungTableaux(const NRVec_from1<T> (&a)[SIZE])`
: from an array of rows.

`bool is_valid() const`
: nonempty rows of non-increasing length.

`int nrows() const; int ncols() const`
: number of rows; length of the first row.

`bool is_standard() const`
: numbers non-decreasing along rows and columns (repeated numbers allowed).

`T sum() const; T max() const`
: number of boxes n; largest number filled in.

`NRVec_from1<T> yamanouchi() const`
: Yamanouchi symbol of a standard tableau: element $n-k+1$ is the row containing the number k.

`PERM_RANK_TYPE generate_all_standard(void (*callback)(const YoungTableaux<T>&))`
: fill `*this` in all standard ways with $1 \dots n$, call the callback for each; returns their number (= `Sn_irrep_dim()` of the shape).

`PermutationAlgebra<T,T> young_operator() const`
: Young operator of a standard tableau, $Y = \sum_{a \in C} \sum_{s \in R} \mathrm{sgn}(a)\, a\, s$ (R, C: row and column groups), unnormalized, $\prod_i \lambda_i! \prod_j \lambda'_j!$ terms.

`T character_contribution(int ncyc=0) const`
: auxiliary for `Sn_character`: sign contribution of a tableau filled by a regular application of `ncyc` cycles (default `max()`).

`std::ostream & operator<<(std::ostream &s, const YoungTableaux<T> &x)`
: one row per line.

13.7 Characters of the symmetric group

`template <typename T> T Sn_character(const Partition<T> &irrep, const Partition<T> &cclass)`
: character $\chi^{[\lambda]}(\mu)$ of the irrep `irrep` on the class of cycle type `cclass`, computed by enumerating the regular applications of the cycles to the Young diagram (Murnaghan–Nakayama type rule, [37]).

`T Sn_character(const CompressedPartition<T> &irrep, const CompressedPartition<T> &cclass)`
: the same for compressed partitions.

`template <typename T> class Sn_characters` is the whole character table of S_n with public members

```
T n;
NRVec_from1<CompressedPartition<T> > classes;  //cycle types
NRVec_from1<CompressedPartition<T> > irreps;   //irrep labels
NRVec_from1<PERM_RANK_TYPE> classsizes;
NRMat_from1<T> chi;                            //chi(irrep,class)
```

The irreps are ordered as in `Partition::generate_all` ($[n]$ first, $[1^n]$ last) and `classes[i]` is the conjugate of `irreps[i]`, so class 1 is the identity and `chi(i,1)` the dimension.

`Sn_characters(const int n0)`
: compute the table (cost grows quickly with n).

`bool is_valid() const`
: consistency checks: dimensions vs. hook formula, $\sum d^2 = n!$, class sizes, trivial and sign irreps, orthogonality.

`T irrepdim(T i) const; T sumirrepdims() const`
: dimension of irrep i; $\sum_i d_i$ (number of standard tableaux of size n).

`std::ostream & operator«(std::ostream &s, const Sn_characters<T> &c)`
: prints class sizes, class labels `(..)`, and rows `[irrep] chi`.

`template <typename T> class CycleIndex` (public `classes`, `classsizes`) is the cycle index of S_n for Pólya counting:

`CycleIndex(const Sn_characters<T> &rhs); bool is_valid() const`
: take the classes from a character table; check that all classes are partitions of the same n.

`Polynomial<T> substitute(const Polynomial<T> &p, PERM_RANK_TYPE *denom) const`
: $\sum_\mu |C_\mu| \prod_j p(x^j)^{m_j(\mu)}$ and `*denom` $= \sum_\mu |C_\mu| = n!$; the generating function of inequivalent colorings is the result divided by `*denom`. See chapter 14 for `Polynomial`.

13.8 Applying permutations to vectors and matrices

The following members are declared in the vector and matrix headers (chapters 4, 5, 6, 7) and take `NRPerm<int>` or `CyclePerm<int>`; the permutation is numbered from 1 while `NRVec` and `NRMat` are indexed from 0. With $p =$ `p`, v the source and r the result (indices from 1):

call	result
`r=v.permuted(p)`	$r_{p(i)} = v_i$
`r=v.permuted(p,true)`	$r_i = v_{p(i)}$
`v.permuteme(p,inverse)`	in place, as `permuted`
`v.permuteme(c)` (`CyclePerm`)	in place, $v_{c(i)} \leftarrow v_i$
`applypermutation(p,v)` (`NRVec_from1`)	$r_{p(i)} = v_i$, as `permuted`
`applypermutation(p,v,true)`	$r_i = v_{p(i)}$
`A.permuted_rows(p)`,	$r_{p(i),j} = A_{ij}$ etc.; with `true`:
`permuted_cols`, `permuted_both(p,q)`	$r_{ij} = A_{p(i),j}$
`A.permuteme_rows(c)`,	in place, as
`permuteme_cols(c)` (`CyclePerm`)	`permuted_rows(NRPerm(c))`
`S.permuted(p)` (`NRSMat`)	$r_{p(i)p(j)} = S_{ij}$; with `true`: $r_{ij} = S_{p(i)p(j)}$
`NRMat<T>(p, direction, parity=false)`	permutation matrix: $A_{i,p(i)} = 1$ if `direction`, else $A_{p(i),i} = 1$ (times `p.parity()` if `parity`)
`NRMat<T>(wp,direction)`, `NRMat<T>(pa,direction,nforce=0)`	$\sum_k w_k P_k$ for a `WeightPermutation` / `PermutationAlgebra`
`A.axpy(alpha,p,direction)`	$A \mathrel{+}= \alpha P$

```
template <typename T, typename X> NRVec_from1<X>
    applypermutation(const NRPerm<T> &p, const NRVec_from1<X> &set,
    bool inverse=false)
```
a free function (element type independent of `T`), same convention as `NRVec::permuted`.

```
int NRVec<T>::sort(int direction, NRPerm<int> &perm, bool
    stable=false)
```
sort the vector (0 ascending, 1 descending), return the parity bit; afterwards `sorted == original.permuted(perm,true)`.

For the matrix representation of S_n by permutation matrices note that it is reducible ($[n] \oplus [n-1,1]$), so e.g. Young projectors are not faithfully represented; use `RegularRepresentation` instead.

13.9 Small number-theoretic helpers

```
template <typename T> T gcd(T big, T small)
```
greatest common divisor by Euclid's algorithm; laerror if both are 0. For negative arguments the sign of the result is not normalized.

```
template <typename T> inline T lcm(T a, T b)
```
$(a/\gcd(a,b))\,b$.

13.10 Sorting

Header: `qsort.h`

The sorting templates are used by `NRVec::sort` and by other classes and return the parity bit (0 even, 1 odd) of the permutation performed, which is needed e.g. for signs of determinants or of fermionic operator strings. The quicksorts use median-of-3 pivoting and recurse first into the shorter part; they are not stable (except `memqsortstable`).

`template<typename INDEX, typename COMPAR> int genqsort(INDEX l, INDEX r, COMPAR (*cmp)(const INDEX, const INDEX), void (*swap)(const INDEX,const INDEX))`
: sort the abstract range $[l, r]$ (inclusive) in ascending order; `cmp(i,j)` returns negative/zero/positive like `strcmp`, `swap(i,j)` exchanges the items.

`template<int type, typename SORTABLE, typename INDEX, typename PERMINDEX> int memqsort(SORTABLE &object, PERMINDEX *perm, INDEX l, INDEX r)`
: sort elements $l \ldots r$ of `object`, ascending for `type=0`, descending for `type=1`; the comparisons go through `LA_sort_traits<SORTABLE,INDEX,type>`, which call `object.bigger(i,j)` / `object.smaller(i,j)`; `object.swap(i,j)` exchanges. If `perm!=NULL`, `perm[l..r]` is permuted in the same way.

`template<int type, typename SORTABLE, typename INDEX, typename PERMINDEX> int memqsortstable(SORTABLE &object, PERMINDEX *perm, INDEX l, INDEX r)`
: stable variant (original position as secondary key; requires also `object[i]==object[j]`), uses `memqsortaux`.

`template<int type, typename SORTABLE, typename INDEX, typename PERMINDEX> int memqsortaux(SORTABLE &object, PERMINDEX *perm, INDEX l, INDEX r, INDEX *aux)`
: worker of `memqsortstable` with the auxiliary key array `aux[l..r]`.

`template<typename S, typename PERMINDEX> int ptrqsortup(S *l, S *r, PERMINDEX *perm=NULL); ... ptrqsortdown(...)`
: sort the array $[l, r]$ (pointers, inclusive) ascending/descending by `operator<`, `operator>`; `perm[0..r-l]` is permuted along.

`template<typename INDEX, typename DATA> void myheapsort(INDEX n, DATA *ra, int typ=1)`
: heapsort of `ra[0..n-1]`, ascending for `typ>0`, descending otherwise; no parity.

`template<typename T> int netsort(const int n, T *data)`
: ascending sort of a short array by optimal sorting networks for $n \leq 8$ (`netsort_0` ... `netsort_8`), `ptrqsortup` for longer; returns the parity bit.

To make a class sortable by `memqsort` it has to provide `bool bigger(int i, int j) const`, `bool smaller(int i, int j) const`, `void swap(int`

`i, int j)` (and `operator[]` for the stable variant), as `NRVec` does.

13.11 Examples

Basic operations, ranking and conversion to cycles:

```
NRPerm<int> p({2,3,1});
NRPerm<int> q=p*p;                          // (3 1 2)
std::cout << p.rank() << ' ' << p.parity() << std::endl;   // 3 1
NRPerm<int> r(3,p.rank());                  // unranking, r==p
CyclePerm<int> c(p);                        // (1 2 3)
std::cout << c << ' ' << c.cycles() << std::endl;
NRVec_from1<double> v({10.,20.,30.});
std::cout << v.permuted(p) << applypermutation(p,v);  // 30 10 20
     / 30 10 20
```

Generating permutations with a callback (store copies with `|=`, cf. the pitfalls):

```
static void printme(const NRPerm<int> &p) {std::cout << p.parity
    () << ' ' << p;}
...
NRPerm<int> p(4);
PERM_RANK_TYPE n=p.generate_all_lex(printme);    // 24
NRVec_from1<int> classes({1,1,2,2});
PermutationAlgebra<int,int> a=p.list_restricted(classes,-2);  //
    6 shuffles with signs
```

Character table, Young operators and their regular representation (cf. `t.cc`):

```
static NRMat<PERM_RANK_TYPE> mtable;

static void ycallback(const YoungTableaux<int> &y)
{
PermutationAlgebra<int,int> Y = y.young_operator();
NRMat<int> R = RegularRepresentation(Y,mtable);
std::cout << y << "Y = " << Y << "trace of R = " << R.trace() <<
    std::endl;
}

int main()
{
Sn_characters<int> Sn(4);
std::cout << Sn;
if(!Sn.is_valid()) laerror("internal error");
mtable = Multable(4);
Partition<int> lambda({2,1,1,0});
YoungTableaux<int> y(lambda);
PERM_RANK_TYPE dim = y.generate_all_standard(ycallback);  // 3 ==
     lambda.Sn_irrep_dim()
}
```

13.12 Pitfalls

- Permutations are numbered from 1, `PermutationAlgebra` terms from 0; vectors and matrices from 0 (the library subtracts the offset).
- The two conventions of applying a permutation to a vector (table in section 13.8) differ; check with a small example.
- `CyclePerm` does not know the size of the permuted set; pass `n` to `NRPerm(const CyclePerm&,n)` and `cycles(n)`.
- The `operator==` of `WeightPermutation` ignores the weights; that of `PermutationAlgebra` simplifies its operands.
- In the callbacks of the generators the argument refers to the working object of the generator. All generators detach it after the callback, so a shallow copy stored by the callback stays intact.
- Copy on demand: the auxiliary `NRPerm::reverse_cycle()` writes through `operator[]` without `copyonwrite()`; call `copyonwrite()` before using it directly (`inverseme()` does so).
- Generators and `Sn_character` use static variables: not reentrant, not thread safe.
- $n!$ and class sizes are `unsigned long long`: overflow for $n > 20$.

Chapter 14

Polynomials and continued fractions

This chapter describes the polynomial class `Polynomial<T>` and the exact rational and continued-fraction arithmetics (`Rational<T>`, `ContFrac<T>` with Gosper's algorithms).

14.1 Polynomial

Header: `polynomial.h, polynomial.cc`

14.1.1 Purpose and storage

`template <typename T> class Polynomial : public NRVec<T>` represents $p(x) = \sum_{i=0}^{d} c_i x^i$; the coefficient c_i is the vector element `p[i]` (indexed from 0, i.e. by the power) and the vector has $d+1$ elements. The member `size()` is made private to prevent confusion; use `degree()`. The leading coefficient may be zero (e.g. after a subtraction); `simplify()` removes such terms. The class is explicitly instantiated for `int`, `unsigned int`, `double`, `std::complex<double>`; root finding and SVD-based gcd require a floating point type.

All inherited `NRVec` functionality (norms, `randomize`, `clear`, I/O operators printing the number of coefficients followed by the coefficients, ...) is available. `Polynomial` follows the copy-on-demand scheme of section 3.1: its modifying members call `copyonwrite()` or `resize()`; writing coefficients through `operator[]` requires an explicit `copyonwrite()` as for `NRVec`. The operations are CPU only (`NOT_GPU`, section 3.5).

14.1.2 Construction

`Polynomial()`
: empty (no coefficients).

`Polynomial(const int n)`
: degree n, coefficients uninitialized. Note that `Polynomial<int>(3)` is *not* the constant 3.

`Polynomial(const T &a, const int n)`
: degree n with $c_0 = a$ and all other coefficients zero.

`template<int SIZE> Polynomial(const T (&a)[SIZE])`
: from coefficients $c_0, c_1, \ldots$, e.g. `Polynomial<double> p({1,2,3})` is $1 + 2x + 3x^2$.

`Polynomial(const NRVec<T> &v)`
: implicit conversion from a vector of coefficients (shares data).

`void binomial(const int n)`
: set to $(1 + x)^n$.

14.1.3 Arithmetic

`int degree() const`
: the degree d (number of coefficients minus 1).

`void resize(const int n, const bool preserve=true)`
: set the degree to n; added coefficients are zero when preserving.

`Polynomial& operator+=(const T &a); Polynomial& operator-=(const T &a); Polynomial operator+(const T &a) const; Polynomial operator-(const T &a) const`
: add/subtract a constant.

`Polynomial operator-() const`
: negation.

`Polynomial operator*(const T &a) const; Polynomial operator/(const T &a) const; Polynomial& operator*=(const T &a); Polynomial& operator/=(const T &a)`
: scaling.

`Polynomial& operator+=(const Polynomial &rhs); Polynomial& operator-=(const Polynomial &rhs); Polynomial operator+(const Polynomial &rhs) const; Polynomial operator-(const Polynomial &rhs) const`
: sum and difference; the degree is the maximum of both (no simplification).

`Polynomial operator*(const Polynomial &rhs) const; Polynomial& operator*=(const Polynomial &rhs)`
: product, naive $O(d_1 d_2)$ algorithm (no FFT). It hides the inherited `NRVec` scalar product.

`void polydiv(const Polynomial &rhs, Polynomial &q, Polynomial &r) const`
: division with remainder, $p = q \cdot \mathrm{rhs} + r$, $\deg r < \deg \mathrm{rhs}$ (the remainder has the degree $\deg \mathrm{rhs} - 1$ formally); laerror if the leading coefficient of `rhs` is zero.

`Polynomial operator/(const Polynomial &rhs) const`
: quotient of `polydiv`; `operator%` with the same signature returns the remainder.

`Polynomial pow(const int i) const`
: p^i, $i \geq 0$ (repeated squaring).

`Polynomial shifted(const int shift) const`
: multiplication by x^{shift}; for negative `shift` division by $x^{-\mathrm{shift}}$, dropping the low-order terms.

`void simplify(const typename LA_traits<T>::normtype thr=0)`
: remove leading coefficients with $|c_i| \leq$ `thr` (the degree stays ≥ 0).

`void normalize()`
: divide by the leading coefficient (monic polynomial); laerror if it is zero.

Free operators.

`template <typename T> Polynomial<T> operator+(const T &a, const Polynomial<T> &rhs)`
: $a + p$.

`template <typename T> Polynomial<T> operator-(const T &a, const Polynomial<T> &rhs) = delete`
: Deliberately not available (compile error); write `-p+a`.

14.1.4 Calculus and transformations

`Polynomial derivative(const int d=1) const`
: d-th derivative ($d < 0$: integral); the zero polynomial of degree 0 if $d > \deg p$.

`Polynomial integral(const int d=1) const`
: d-fold integral with zero integration constants ($d < 0$: derivative).

`Polynomial composition(const Polynomial &rhs) const`
: $p(\mathrm{rhs}(x))$, the same as `value(p,rhs)`.

`Polynomial powx(const int i) const`
: $p(x^i)$; for $i = 0$ the constant $p(1)$.

`Polynomial even_powers() const; Polynomial odd_powers() const`
: E and O such that $p(x) = E(x^2) + x\,O(x^2)$.

14.1.5 Evaluation

`template <typename T, typename C> C value(const Polynomial<T> &p, const C &x)`
: Horner scheme; `C` may be a scalar, a square matrix (`NRMat`, `NRSMat`, ...; constants are added to the diagonal), or another polynomial (composition). `C` needs a copy constructor, `=0`, `+=` with `T` and `operator*`.

`template <typename T, typename C> C even_value(const Polynomial<T> &p, const C &x); C odd_value(const Polynomial<T> &p, const C &x)`
: sums of the even resp. odd terms only, $E(x^2)$ and $x\,O(x^2)$; `C` may also be a square matrix.

`template <typename T, typename C> NRVec<C> values(const Polynomial<T> &p, const NRVec<C> &x)`
: `value` at each element of `x`.

For polynomials of matrices with fewer matrix multiplications see `polynom()` in `matexp.h`, which accepts the coefficient vector, hence also a `Polynomial` (section 8.6).

14.1.6 Roots, interpolation, gcd

`NRMat<T> companion() const`
: companion matrix ($d \times d$, ones on the superdiagonal, last row $-c_i/c_d$) whose characteristic polynomial is p/c_d.

`NRVec<typename LA_traits<T>::complextype> roots() const`
: all roots as eigenvalues of the companion matrix (`gdiagonalize`, chapter 8); for `double` and `std::complex<double>` only (laerror for integer types).

`NRVec<T> realroots(const typename LA_traits<T>::normtype thr) const`
: roots with $|\mathrm{Im}| <$ `thr`, real parts sorted ascending (`T = double`).

`T newton(const T x0, const typename LA_traits<T>::normtype thr=1e-14, const int maxit=1000) const`
: Newton iteration from `x0` until $|p(x)| <$ `thr`; no convergence report.

`template <typename T> Polynomial<T> polyfromroots(const NRVec<T> &roots, const int skip= -1)`
: monic $\prod_{i \neq \mathrm{skip}} (x - r_i)$.

`template <typename T> Polynomial<T> lagrange_interpolation(const NRVec<T> &x, const NRVec<T> &y)`
: interpolating polynomial of degree $n-1$ through the n points; laerror for repeated x.

`template <typename T> NRMat<T> Sylvester(const Polynomial<T> &p, const Polynomial<T> &q)`
: Sylvester matrix of dimension $\deg p + \deg q$ (its determinant is the resultant).

`template <typename T> Polynomial<T> poly_gcd(const Polynomial<T> &p, const Polynomial<T> &q, const typename LA_traits<T>::normtype thr=0, const int d= -1)`

Euclid's algorithm with `simplify(thr)` of the remainders, result not normalized. It stops at a zero remainder (degree 0 with $|c_0| \leq$`thr`); with `d>=0` also when the remainder degree drops below `d`. For coprime polynomials the result is a nonzero constant; $\gcd(p, 0) = p$.

`template <typename T> Polynomial<T> svd_gcd(const Polynomial<T> &p, const Polynomial<T> &q, const typename LA_traits<T>::normtype thr=0)`

numerically more robust gcd: the degree $\deg p + \deg q - \operatorname{rank} S$ is determined from the singular values of the Sylvester matrix S (threshold `thr` scaled by $\sqrt{\deg p \deg q}$), then `poly_gcd` with this degree. Floating point types only.

`template <typename T> Polynomial<T> poly_lcm(const Polynomial<T> &p, const Polynomial<T> &q, const typename LA_traits<T>::normtype thr=0)`

$pq/\gcd(p, q)$.

14.1.7 Orthogonal polynomials

`template <typename T> Polynomial<T> hermite_polynomial(int n)`

Hermite polynomial H_n in the physicists' normalization (leading coefficient 2^n).

No other families are implemented.

14.1.8 Example

```
Polynomial<double> p=polyfromroots(NRVec<double>({1.,2.,3.}));
    // (x-1)(x-2)(x-3)
Polynomial<double> q({1.,1.});
    // 1+x
Polynomial<double> a,b;
p.polydiv(q,a,b);                       // p = a*q + b
std::cout << (a*q+b-p).norm() << std::endl;
NRVec<double> r = p.realroots(1e-10);   // 1 2 3
Polynomial<double> pp = p.integral(2).derivative(2);  // == p
NRMat<double> m(3,3); m.randomize(1.);
NRMat<double> pm = value(p,m);          // p(m)
Polynomial<double> h = hermite_polynomial<double>(4);
std::cout << value(h,0.5) << ' ' << even_value(h,0.5) << std::
    endl;
```

14.2 Rational numbers and continued fractions

Header: `contfrac.h, contfrac.cc`

The classes are instantiated for `T = int`, `long`, `long long`. Integer overflow is not detected.

14.2.1 Rational

`template <typename T> class Rational` is a small value class with the public members `T num, den`; it is not reference counted. Results of `+` and `-` of two rationals are reduced and have a positive denominator; other operations cancel common factors only partially.

`Rational(); Rational(const T p, const T q); explicit Rational(const T (&a)[2])`
: uninitialized; p/q (not reduced); from `{p,q}`.

`explicit Rational(const ContFrac<T> &cf)`
: value of the (full) continued fraction.

`void simplify()`
: make the denominator positive and divide by the gcd. All operations keep the denominator positive.

`Rational operator-() const`
: negation.

`Rational & operator+=(const T &rhs); -=, *=, /= (const T&); Rational operator+(const T &rhs) const; -, *, /`
: arithmetic with an integer.

`Rational & operator*=(const Rational &rhs); /=; Rational operator+(const Rational &rhs) const; -, *, /; +=, -=`
: rational arithmetic (`+`/`-` via the lcm of the denominators).

`bool operator==(const Rational &rhs) const; != >= <= > <`
: via the sign of the numerator of the difference.

`ContFrac<T> operator+(const ContFrac<T> &rhs) const; -, *, /`
: mixed arithmetic, the result is a continued fraction.

`std::ostream & operator«(std::ostream &s, const Rational<T> &x); std::istream & operator»(...)`
: format `p/q` (input: any single separator character).

14.2.2 ContFrac

`template <typename T> class ContFrac : public NRVec<T>` stores a finite simple continued fraction

$$[a_0; a_1, \ldots, a_n] = a_0 + \cfrac{1}{a_1 + \cfrac{1}{\ddots + \cfrac{1}{a_n}}}$$

in vector elements $0 \ldots n$; `length()` returns n (`size()` is private). A zero term $a_i = 0$ at a position $i > 0$ means ∞, i.e. the fraction terminates before it. The canonical form has $a_i \geq 1$ for $i > 0$ and $a_n > 1$ for $n > 0$; comparisons are guaranteed only for canonical forms. Arithmetic with continued fractions uses Gosper's algorithms [39] (cf. also `https://perl.plover.com/classes/cftalk/TALK`), which produce the terms of the result while consuming the terms of the operands, without forming the rational value. `ContFrac` is reference counted (section 3.1); its modifying members call `copyonwrite()` or `resize()`.

`ContFrac(); ContFrac(const int n)`
: empty; length n uninitialized (not the value n!).

`template<int SIZE> ContFrac(const T (&a)[SIZE]); ContFrac(const NRVec<T> &v); explicit ContFrac(const std::list<T> &x)`
: from the terms $a_0, a_1, \ldots$

`ContFrac(T p, T q); explicit ContFrac(const Rational<T> &r)`
: exact expansion of p/q in canonical form (also for negative values: $a_0 = \lfloor p/q \rfloor$).

`explicit ContFrac(double x, const int n, const T thres=0)`
: expansion of a real number to at most n terms; stops earlier when the remainder is zero or (if `thres!=0`) when the next term would exceed `thres` in absolute value. The result may be non-canonical.

`int length() const; void resize(const int n, const bool preserve=true)`
: n; set the length, new terms are 0 (= termination) when preserving.

`void canonicalize()`
: bring to the canonical form (absorbs a trailing 1, truncates at the first zero term, handles negative terms). It works directly on the terms without forming p/q (no overflow for fixed-size integer `T`), and therefore requires that the terms $a_1, \ldots, a_n$ all have the same sign; with mixed signs the result is wrong (use `ContFrac(p,q)` from `convergent()` if the values fit into `T`).

`void convergent(T *p, T *q, const int trunc= -1) const`
: numerator and denominator of the convergent $[a_0; a_1, \ldots, a_{\text{trunc}}]$ (all terms for -1; `trunc` must not exceed `length()`).

`Rational<T> rational(const int trunc= -1) const; double value(const int trunc= -1) const`
: the convergent as a `Rational`; as `double`.

`ContFrac reciprocal() const`
: $1/x$ (removes or prepends $a_0 = 0$).

`ContFrac operator-() const`
: $-x$ in canonical form, $[-a_0 - 1; 1, a_1 - 1, a_2, \ldots]$.

`ContFrac & operator+=(const T &rhs); -=; ContFrac operator+(const T &rhs) const; -`
: shift of a_0.

`ContFrac operator*(const T &rhs) const; /; ContFrac operator+(const Rational<T> &rhs) const; -, *, /`
: via a homographic function (`Homographic::value`).

`ContFrac operator+(const ContFrac &rhs) const; -, *, /`
: via a bihomographic function (`BiHomographic::value`).

`T compare(const ContFrac &rhs) const`
: negative, zero or positive as $x <, =, >$ `rhs` (canonical forms).

`bool operator==(const ContFrac &rhs) const; != > < >= <=`
: via `compare`.

`iterator begin() const; iterator end() const; iterator beyondend() const`
: forward iterator over the terms; `end()` points past a_n and `beyondend()` one further (used by Gosper's algorithm to feed the final ∞).

Output by the inherited `NRVec` operator « prints the number of terms and the terms.

14.2.3 Gosper's arithmetics

`template <typename T> class Homographic` holds `T v[2][2]` $= \{\{a, b\}, \{c, d\}\}$ representing $z = (a + bx)/(c + dx)$ (first index: numerator/denominator, second: power of x). `template <typename T> class BiHomographic` holds `T v[2][2][2]` $= \{\{\{a, b\}, \{c, d\}\}, \{\{e, f\}, \{g, h\}\}\}$ representing

$$z = \frac{a + bx + cy + dxy}{e + fx + gy + hxy}$$

(indices: numerator/denominator, power of y, power of x); e.g. $x + y$ is `{{{0,1},{1,0}},{{1,0},{0,0}}}`.

`Homographic(); explicit Homographic(const T (&a)[2][2])`
: uninitialized; from the coefficients.

`ContFrac<T> value(const ContFrac<T> &z) const`
: the continued fraction of $h(z)$.

`BiHomographic(); explicit BiHomographic(const T (&a)[2][2][2])`
: uninitialized; from the coefficients.

`ContFrac<T> value(const ContFrac<T> &x, const ContFrac<T> &y) const`
: the continued fraction of $z(x, y)$.

`input(const T &x, const bool inf) const; inputx, inputy; output(const T &z) const; bool outputready(T &x, bool first) const; int inputselect() const; bool terminate() const`
: the elementary steps of Gosper's algorithm (consume a term or ∞, emit a term, test whether the next term is determined, choose the operand to read next, test for the end); used by `value`.

14.2.4 Example

```
ContFrac<int> x(11,101), v(3,7);
std::cout << Rational<int>(x+v) << ' ' << Rational<int>(x*v) <<
    std::endl;   // 380/707 33/707
BiHomographic<int> h({{{12,4},{3,1}},{{0,1},{-1,0}}});   // (x+3)
    (y+4)/(x-y)
std::cout << Rational<int>(h.value(x,v)) << std::endl;      //
    -4867/113
ContFrac<long long> pi(3.14159265358979,10,1000000);
Rational<long long> r = pi.rational(3);   // 355/113
```

14.2.5 Pitfalls

- `Polynomial<T>(n)` and `ContFrac<T>(n)` with an integer argument allocate a polynomial of degree n / a fraction of length n; they do not represent the number n.
- The leading coefficient of a `Polynomial` can be zero after arithmetic; call `simplify()` before `polydiv`, `companion`, `normalize`.
- `Rational` and `ContFrac` arithmetics silently overflow for large terms; use `long long`.
- Relational operators of `ContFrac` are reliable only for canonical forms (e.g. after `canonicalize()` or from `ContFrac(p,q)`).

Chapter 15

3D vectors and matrices: Vec3, Mat3

Header: `vecmat3.h, vecmat3.cc`

Fixed-size 3D vectors and matrices; plain data, no reference counting (unlike the classes of chapter 4 and 5).

15.1 Purpose

`Vec3<T>` and `Mat3<T>` are small fixed-size classes for real 3-dimensional vectors and 3×3 matrices, intended for geometry, rotations and Euler angles. They are compatible with the quaternion functions of chapter 16 (rotation matrices, SO(3) parametrization). They contain only a C array and all operations are hand-unrolled; no BLAS/LAPACK is used.

The files are written so that they can be compiled separately from the rest of LA (e.g. for microcontrollers). The following macros control this:

- `AVOID_STDSTREAM` – do not include `<iostream>`, no stream operators;
- `NO_NUMERIC_LIMITS` – do not use `std::numeric_limits`; `DBL_EPSILON` is then defined as the single-precision epsilon;
- `QUAT_NO_RANDOM` – omit `randomize()` (which needs `la_random.h`);
- `QUAT_NO_DOUBLE` – instantiate only `float` (otherwise `float` and `double`).

The templates are explicitly instantiated in `vecmat3.cc` for `float` and `double` only.

The classes live in namespace `LA_Vecmat3`; the header ends with a global `using namespace LA_Vecmat3;`, so the names are visible without qualification after including `vecmat3.h` (or `la.h`, which includes it). `LA_traits`

specializations declare both classes plain data with a no-op `copyonwrite`, so they can be used as elements of `NRVec`, `NRMat` etc.

15.2 Storage

`Vec3<T>` holds the public array `T q[3]`, `Mat3<T>` the public array `T q[3][3]` in row-major (C) order, i.e. `q[i][j]` is row i, column j. There is no reference counting; the compiler-generated copy constructor and assignment copy the data. `elements()` returns a reference to the C array, conversion to `T*` gives a pointer to the first element.

15.3 Vec3 reference

`Vec3(); Vec3(const T x, const T u=0, const T v=0)`
: Uninitialized vector; vector (x, u, v).

`Vec3(const T *x); Vec3(const T (&a)[3])`
: Copy three elements from memory or from a C array (also allows brace initialization `Vec3<double> x({1,2,3})`).

`T (&elements())[3]; const T (&elements() const)[3]`
: Reference to the underlying array.

`operator T*(); operator const T*() const`
: Pointer to the data.

`T &operator[](const int i)`
: Element access (no range check), const version available.

`void clear()`
: Set to zero (memset, `T` must be plain data).

`Vec3 &operator*=(const T); Vec3 &operator/=(const T); Vec3 operator*(const T) const; Vec3 operator/(const T) const`
: Scaling by a scalar.

`T sum() const; T asum() const; T sumsqr() const; T prod() const`
: $\sum_i q_i$, $\sum_i |q_i|$, $\sum_i q_i^2$, $\prod_i q_i$.

`Vec3 operator-() const; Vec3 &operator+=(const Vec3 &); Vec3 &operator-=(const Vec3 &); Vec3 operator+(const Vec3 &) const; Vec3 operator-(const Vec3 &) const`
: Vector algebra.

`Vec3 operator*(const Vec3 &rhs) const`
: *Vector (cross) product* $\mathbf{q} \times \mathbf{rhs}$.

`T dot(const Vec3 &rhs) const`
: Scalar product.

`Vec3 elementwise_product(const Vec3 &rhs) const`
: Componentwise product.

`T normsqr() const; T norm() const`
: Squared and Euclidean norm.

`Vec3 &normalize()`
: Divide by the norm (no check for zero).

`Vec3 &fast_normalize()`
: For `float` uses the "fast inverse square root" bit trick `fast_sqrtinv(float)` (approximate); for other types identical to `normalize()`.

`Vec3 operator*(const Mat3<T> &rhs) const`
: Row vector times matrix, $\mathbf{q}^\mathsf{T} M$.

`Vec3 timesT(const Mat3<T> &rhs) const`
: $\mathbf{q}^\mathsf{T} M^\mathsf{T}$, i.e. $M\mathbf{q}$.

`Mat3<T> outer(const Vec3 &rhs) const`
: Outer (tensor) product $\mathbf{q}\,\mathbf{rhs}^\mathsf{T}$.

`void addouter(Mat3<T> &m, const Vec3 &rhs, const T weight) const`
: $m \mathrel{+}= w\,\mathbf{q}\,\mathbf{rhs}^\mathsf{T}$.

`void inertia(Mat3<T> &itensor, const T weight) const`
: Adds the contribution of a point mass `weight` at position $\mathbf{q}$ to the inertia tensor: $I \mathrel{+}= w\,(r^2\mathbf{1} - \mathbf{q}\mathbf{q}^\mathsf{T})$.

`void randomize(const T x)`
: Elements uniformly random in $[-x, x)$ via `RANDDOUBLESIGNED` (chapter 19); for `float` and `double`.

`int fprintf(FILE *f, const char *format) const; int sprintf(char *f, const char *format) const`
: C-style output, `format` must contain three conversions.

`int fscanf(FILE *f, const char *format); int sscanf(const char *f, const char *format)`
: C-style input, `format` contains three conversions (e.g. `"\%lf \%lf \%lf"`); returns the number of items read.

Free function: `T hypot3(const Vec3<T> &c, const Vec3<T> &d)` – Euclidean distance $|\mathbf{c} - \mathbf{d}|$.

15.4 Mat3 reference

`Mat3(); Mat3(const T (&a)[3][3]); Mat3(const T *x)`
: Uninitialized; from a C array (also brace initialization `Mat3<double> y({{1,2,3},{4,5,6},{7,8,9}})`); from 9 elements in row-major order.

`Mat3(const T x00, const T x01, const T x02, const T x10, const T x11, const T x12, const T x20, const T x21, const T x22)`
: From nine elements, row by row.

`Mat3(const T x); Mat3 &operator=(const T &x)`
: Scalar matrix $x\mathbf{1}$. A single *integer* constructor argument is forbidden (deleted), since through the conversion `NRMat(const Mat3&)` it would silently turn e.g. `NRMat<double> a(5)` into a 3×3 matrix; write `Mat3<double>(5.)`.

`void indentity()`
: Set to the unit matrix (sic, the name is misspelled in the code).

`T (&elements())[3][3]; operator T*()`
: Reference to the array, pointer to the data (const versions exist).

`T *operator[](const int i); T &operator()(const int i, const int j)`
: Row pointer (so `m[i][j]` works) and element access, const versions exist.

`void clear()`
: Zero matrix.

`Mat3 &operator+=(const T); Mat3 &operator-=(const T); Mat3 operator+(const T) const; Mat3 operator-(const T) const`
: Add/subtract a multiple of the unit matrix (diagonal only).

`Mat3 &operator*=(const T); Mat3 &operator/=(const T); Mat3 operator*(const T) const; Mat3 operator/(const T) const`
: Scaling.

`void randomize(const T x, const bool symmetric=false)`
: Random elements in $[-x, x)$, optionally symmetric; for `float` and `double`.

`Mat3 operator-() const; Mat3 &operator+=(const Mat3 &); Mat3 &operator-=(const Mat3 &); Mat3 operator+(const Mat3 &) const; Mat3 operator-(const Mat3 &) const`
: Matrix algebra.

`Mat3 operator*(const Mat3 &rhs) const; Mat3 timesT(const Mat3 &rhs) const; Mat3 Ttimes(const Mat3 &rhs) const; Mat3 TtimesT(const Mat3 &rhs) const`
: AB, AB^T, $A^\mathsf{T}B$, $A^\mathsf{T}B^\mathsf{T}$.

`Vec3<T> operator*(const Vec3<T> &rhs) const; Vec3<T> Ttimes(const Vec3<T> &rhs) const`
: $A\mathbf{v}$ and $A^\mathsf{T}\mathbf{v}$.

`T trace() const; T determinant() const`
: Trace and determinant.

`void transposeme(); Mat3 transpose() const`
: Transpose in place / return the transpose.

`Mat3 inverse(T *det = NULL) const`
: Inverse by the adjugate formula; optionally returns the determinant. No check for singularity (division by zero determinant).

`Vec3<T> linear_solve(const Vec3<T> &rhs, T *det = NULL) const`
: Solves $A\mathbf{x} = \mathbf{rhs}$ by Cramer's rule (alternative to `simple_gaussj`, section 19.4); no singularity check.

`int fprintf(FILE *f, const char *format) const; int fscanf(FILE *f, const char *format)`
: C-style I/O row by row, `format` contains three conversions.

`void symmetrize()`
: Replace off-diagonal pairs by their average.

`bool eivec_sym(Vec3<T> &w, Mat3 &v, const bool sortdown=false) const`
: Eigenvalues `w` and eigenvectors (*columns* of `v`) of a real symmetric matrix by the cyclic Jacobi method (at most 50 sweeps); symmetry is not checked. Eigenvalues sorted ascending, or descending if `sortdown`. Returns `false` on success, `true` if not converged. (An analytic Cardano-based solver is present in the source in a disabled `#if 0` block, abandoned for numerical instability.)

`T norm(const T scalar = 0) const`
: Frobenius norm of $A - \text{scalar} \cdot \mathbf{1}$.

`void qrd(Mat3 &q, Mat3 &r)`
: QR decomposition $A = QR$ by three Givens rotations; *destroys* `*this` (non-const).

`void svd(Mat3 &u, Vec3<T> &w, Mat3 &v, bool proper_rotations=false) const`
: Singular value decomposition $A = U\,\mathrm{diag}(w)\,V^{\mathsf{T}}$ following [40] (eigenvectors of $A^{\mathsf{T}}A$ by `eivec_sym`, then QR of AV). Singular values in descending order. If `proper_rotations` is false, a negative w_2 is made positive by changing the sign of the last column of U; if true, U and V are proper rotations ($\det U = \det V = +1$, the sign is moved into w_2, which may then be negative), as needed e.g. for the Kabsch alignment without mirroring.

`void diagmultl(const Vec3<T> &rhs); void diagmultr(const Vec3<T> &rhs)`
: $A \leftarrow \mathrm{diag}(\mathbf{rhs})A$ and $A \leftarrow A\,\mathrm{diag}(\mathbf{rhs})$.

`Mat3 svdinverse(const T thr=1000*DBL_EPSILON) const`
: Pseudoinverse via SVD; singular values below the *absolute* threshold `thr` are treated as zero.

15.5 Free functions: I/O, Euler angles, projection

`std::ostream &operator«(std::ostream &s, const Vec3<T> &x); std::istream &operator»(std::istream &s, Vec3<T> &x)`
: Three numbers separated by spaces (no dimensions, no newline).

`std::ostream &operator«(std::ostream &s, const Mat3<T> &x);`

`std::istream &operator»(std::istream &s, Mat3<T> &x)`
: Nine numbers, one row per line on output.

`void euler2rotmat(const T *eul, Mat3<T> &a, const char *type, bool transpose=0, bool direction=0, bool reverse=0)`
: Rotation matrix from three Euler angles `eul[0..2]`; see below.

`void rotmat2euler(T *eul, const Mat3<T> &a, const char *type, bool transpose=0, bool direction=0, bool reverse=0)`
: Inverse of `euler2rotmat` with the same options.

`void perspective(T *proj_xy, const Vec3<T> &point, const Mat3<T> &rot_angle, const Vec3<T> &camera, const Vec3<T> &plane_to_camera)`
: Perspective projection of a 3D point onto a 2D plane: $\mathbf{d} = R\,\mathbf{point} - \mathbf{camera}$, $\mathrm{proj}_i = (e_z/d_z)\,d_i + e_i$, $i = x, y$, where **e**=`plane_to_camera`.

Euler angle conventions. `type` is a three-character string naming the rotation axes: one of the six proper Euler sequences `"xzx"` `"xyx"` `"yxy"` `"yzy"` `"zyz"` `"zxz"` or the six Tait–Bryan sequences `"xzy"` `"xyz"` `"yxz"` `"yzx"` `"zyx"` `"zxy"`. With default options the result is the product of active elementary rotations

$$a = R_{\mathrm{type}[0]}(\mathrm{eul}_0)\,R_{\mathrm{type}[1]}(\mathrm{eul}_1)\,R_{\mathrm{type}[2]}(\mathrm{eul}_2),$$

following the table in the Wikipedia article “Euler angles”. The options modify this: `transpose` returns a^{T}; `direction` changes the sign of all angles; `reverse` swaps the roles of `eul[0]` and `eul[2]`. In `rotmat2euler`, the middle angle is returned in $[0, \pi]$ for proper Euler sequences and in $[-\pi/2, \pi/2]$ for Tait–Bryan ones; the degenerate (gimbal lock) case is not treated specially. An unknown `type` leaves the output untouched without an error. The macro `Euler_case(a,b,c)` maps the three axis characters to an integer used in the internal switch.

Chapter 16

Quaternions

Header: `quaternion.h, quaternion.cc`

Quaternions; plain data, no reference counting.

16.1 Purpose

`Quaternion<T>` implements the quaternion algebra $\mathbb{H}$, $q = q_0 + q_1\mathrm{i} + q_2\mathrm{j} + q_3\mathrm{k}$, and in particular unit quaternions as a parametrization of SO(3) (and SU(2)): conversions to and from rotation matrices, Euler angles and axis–angle, rotation of vectors, derivatives of the rotation matrix with respect to the quaternion components, and the functions exp, log, power. Like `vecmat3.h` it can be compiled without the rest of LA: define `AVOID_LA` (do not include `la.h`, no `LA_traits`), and optionally `AVOID_STDSTREAM`, `QUAT_NO_RANDOM`, `QUAT_NO_DOUBLE`. The class and functions are in namespace `LA_Quaternion` (*not* imported globally, use `using namespace LA_Quaternion;`). The member functions and `exp`, `log`, `pow` are instantiated for `float` and `double` only; the conversion functions taking a generic matrix type `M` are templates defined in the header.

16.2 Storage

Plain data `T q[4]`, q_0 being the real (scalar) part; no reference counting. `LA_traits<Quaternion<T>>` declares it plain data. The stream operators are templates in the header, so that e.g. `Quaternion<NRMat<double>>` (a quadruple of matrices, used for rotation matrix derivatives) can be printed.

16.3 Reference

`Quaternion(); Quaternion(const T (&a)[4])`
: Uninitialized; from a C array (brace initialization `Quaternion<double> q({1,2,3,4})`).

`Quaternion(const T x, const T u=0, const T v=0, const T w=0)`
: $x + u\mathrm{i} + v\mathrm{j} + w\mathrm{k}$; with one argument a real quaternion.

`Quaternion(const std::complex<T> &rhs)`
: $\mathrm{Re} + \mathrm{Im}\,\mathrm{i}$.

`explicit Quaternion(const T *x, const int shift=1)`
: Sets $q_0 = 0$ and copies $4 - \mathrm{shift}$ values to `q+shift`; with the default, a pure quaternion from a 3D vector (also works with a `Vec3` through its pointer conversion).

`void identity(); void clear()`
: Set to 1 / to 0.

`T operator[](const int i) const; T &operator[](const int i); operator T*()`
: Element access and pointer to data.

`Quaternion &operator=(const T x)`
: Real quaternion x.

`Quaternion &operator+=(const T); Quaternion &operator-=(const T)`
: Add/subtract to the real part only (non-mutating `+`, `-` versions exist).

`Quaternion &operator*=(const T); Quaternion &operator/=(const T)`
: Scaling (non-mutating `*`, `/` versions exist).

`Quaternion operator-() const; operator+=, operator-=, operator+, operator-`
: Algebra with quaternions.

`Quaternion operator*(const Quaternion &rhs) const`
: Hamilton product.

`Quaternion times_vec3(const T *rhs) const; Quaternion vec3_times(const T *rhs) const`
: Product $q\,(0, \mathbf{r})$ resp. $(0, \mathbf{r})\,q$ with a pure quaternion given as a 3-vector (saves flops).

`Quaternion &conjugateme(); Quaternion conjugate() const`
: $\bar{q} = q_0 - q_1\mathrm{i} - q_2\mathrm{j} - q_3\mathrm{k}$.

`T dot(const Quaternion &rhs) const; T normsqr() const; T norm() const`
: Euclidean scalar product in $\mathbb{R}^4$, $|q|^2$, $|q|$.

`Quaternion &normalize(T *getnorm=NULL, bool unique_sign=false)`
: Normalize to unit norm; optionally return the original norm; with `unique_sign` the sign is chosen so that $q_0 \geq 0$ (fixes the double cover).

`Quaternion &fast_normalize(bool unique_sign=false)`
: As `normalize`, using `fast_sqrtinv` for `float`.

`Quaternion inverse() const; Quaternion operator/(const Quaternion &rhs) const`
: $q^{-1} = \bar{q}/|q|^2$; $q\,\mathrm{rhs}^{-1}$.

`void rotate(T *to, const T *from, Quaternion<T> *grad=NULL) const`
: Rotates the 3-vector `from` by the *normalized* `*this`: $(0, \mathbf{to}) = \bar{q}\,(0, \mathbf{from})\,q$, i.e. $\mathbf{to} = A\,\mathbf{from}$ with A from `normquat2rotmat(q,A)`. If `grad` is given (array of 3 quaternions), `grad[i][k]` receives $\partial\,\mathrm{to}_i/\partial q_k$.

`Quaternion rotateby(const Quaternion &rhs)`
: Returns $\bar{r}\,q\,r$ for normalized r=`rhs`; *the real part of* `*this` *is ignored* (taken as 0), so this is meant for pure quaternions representing vectors.

`Quaternion rotate_match(T *to, const T *from, const T *match) const`
: Computes `to` as in `rotate` and returns the gradient of $\frac{1}{2}|\mathbf{to} - \mathbf{match}|^2$ with respect to q (for gradient-descent fitting of a rotation).

`Quaternion commutator(const Quaternion &rhs) const; Quaternion anticommutator(const Quaternion &rhs) const`
: $qr - rq$, $qr + rq$.

`T geodesic_distance(const Quaternion &rhs) const`
: $\arccos(2(q\cdot r)^2 - 1)$, the rotation angle between the two (normalized) rotations.

`void normquat2euler(T *eul, const char *type) const`
: Euler angles of the rotation, implemented as `rotmat2euler(eul, normquat2rotmat(q), type)`.

`void euler2normquat(const T *eul, const char *type)`
: Unit quaternion from Euler angles by explicit formulas, consistent with `euler2rotmat` (see the conventions below).

`void normquat2eulerzyx(T *eul) const; void eulerzyx2normquat(const T *eul)`
: Special pair for yaw–pitch–roll; equivalent to `euler2rotmat(...,"xyz",true,true,true)`; `eulerzyx2normquat` calls `euler2normquat(eul,"zyx")`.

`void axis2normquat(const T *axis, const T &angle); void normquat2axis(T *axis, T &angle) const`
: $q = (\cos\frac{\phi}{2}, \mathbf{n}\sin\frac{\phi}{2})$ for a unit axis $\mathbf{n}$, and back (angle in $[0, 2\pi]$; undefined for the identity).

`void random_rotation()`

Uniformly distributed random unit quaternion (Marsaglia's method [41]).

`int fprintf(FILE *f, const char *format) const; int sprintf(char *f, const char *format) const; int fscanf(FILE *f, const char *format); int sscanf(const char *f, const char *format)`

C-style I/O with four conversions.

Free functions (namespace `LA_Quaternion`; `M` is any type with `a[i][j]` indexing: C array, `Mat3`, `NRMat`):

`std::ostream &operator«(std::ostream &, const Quaternion<T> &); std::istream &operator»(std::istream &, Quaternion<T> &)`

Four components separated by spaces.

`void normquat2su2mat(const Quaternion<T> &q, M &a)`

SU(2) matrix $\begin{pmatrix} q_0 + \mathrm{i}q_1 & q_2 + \mathrm{i}q_3 \\ -q_2 + \mathrm{i}q_3 & q_0 - \mathrm{i}q_1 \end{pmatrix}$ (elements `std::complex<T>`); $\pm q$ give different elements.

`void normquat2rotmat(const Quaternion<T> &q, M &a, bool transpose=false)`

Rotation matrix of a unit quaternion, $a_{00} = q_0^2 + q_1^2 - q_2^2 - q_3^2$, $a_{01} = 2(q_1 q_2 + q_0 q_3)$, $a_{10} = 2(q_1 q_2 - q_0 q_3)$, ... ($\pm q$ give the same matrix). `transpose` should give a^T ("to match the NASA paper").

`void quat2rotmat(Quaternion<T> q, M &a, bool transpose=false, const bool already_normalized=false)`

Normalizes a copy of q first unless `already_normalized`.

`void normquat2rotmatder(const Quaternion<T> &q, Quaternion<M> &a, bool transpose=false)`

`a[k]` $= \partial A / \partial q_k$, $k = 0..3$, for A from `normquat2rotmat` (the 3×3 matrices must be preallocated if `M` is `NRMat`; `M` must have `transposeme()` if `transpose` is used).

`void rotmat2normquat(const M &a, Quaternion<T> &q, bool transpose=false)`

Unit quaternion from a rotation matrix (method of Sarabandi and Thomas, branch on the sign of the trace); inverse of `normquat2rotmat`, convention compatible with Madgwick's MEMS paper.

`Quaternion<T> exp(const Quaternion<T> &x)`

$e^{x_0}(\cos|\mathbf{v}|, \hat{\mathbf{v}} \sin|\mathbf{v}|)$, $\mathbf{v} = (x_1, x_2, x_3)$.

`Quaternion<T> log(const Quaternion<T> &x)`

$(\ln|x|, \hat{\mathbf{v}} \arccos(x_0/|x|))$ – principal branch; $\log(\exp(x)) \neq x$ in general and $\log(xy) \neq \log x + \log y$.

`Quaternion<T> pow(const Quaternion<T> &x, const T &y)`

$x^y = |x|^y(\cos y\varphi, \hat{\mathbf{v}}\sin y\varphi)$, $\varphi = \arccos(x_0/|x|)$. For real quaternions ($\mathbf{v} = 0$) the limits are used; for a negative real x_0 (where $\hat{\mathbf{v}}$ is undefined) `log` and `pow` choose the i axis as for complex numbers: $\log x = (\ln|x_0|, \pi, 0, 0)$, $x^y = |x_0|^y(\cos\pi y, \sin\pi y, 0, 0)$.

Conventions. Note the passive convention: A=`normquat2rotmat(q)` satisfies $(0, A\mathbf{v}) = \bar{q}\,(0, \mathbf{v})\,q$; it is the transpose of the "textbook" active matrix $q\,\mathbf{v}\,\bar{q}$. For all 12 Euler sequences `euler2normquat(e,type)` is consistent with `euler2rotmat`: `normquat2rotmat(q)` = `euler2rotmat(e,type)`, and `normquat2euler` (defined via `rotmat2euler(normquat2rotmat(q))`) is its inverse, i.e. `euler2normquat(normquat2euler(q,type),type)` $= \pm q$ (the same rotation). `normquat2eulerzyx`/`eulerzyx2normquat` are the yaw–pitch–roll pair for `"zyx"`.

16.4 Example

```
#include "la.h"
using namespace LA;
using namespace LA_Quaternion;

Quaternion<double> q(1.,2.,3.,4.);
q.normalize(NULL,true);                  //unit quaternion with q
    [0]>=0
Mat3<double> a;
normquat2rotmat(q,a);
Vec3<double> v(1.,-2.,.5), w;
q.rotate(w,v);                           //w == a*v
Quaternion<double> back;
rotmat2normquat(a,back);                 //back == q
Vec3<double> axis={0,1/sqrt(2),1/sqrt(2)};
Quaternion<double> r;
r.axis2normquat(axis,0.5);
Quaternion<double> rv = Quaternion<double>(v).rotateby(r);
```

16.5 Pitfalls

Caution: Most functions assume a *normalized* quaternion (names `normquat...`) and do not check it. Use `quat2rotmat` or call `normalize()` first.

Caution: `Vec3::operator*(const Vec3&)` is the cross product, not the dot product; `Vec3*Mat3` is a row vector times a matrix.

Chapter 17

Bit vectors and polynomials over GF(2)

Header: `bitvector.h, bitvector.cc`

Compressed bit vectors, which also represent polynomials over GF(2) and elements of GF(2^n). `bitvector` is a reference-counted class derived from `NRVec` (see section 3.1).

17.1 Purpose and storage

`bitvector` is a compressed vector of bits stored in 64-bit blocks (`typedef uint64_t bitvector_block`, macro `blockbits`=64). It is publicly derived from `NRVec<bitvector_block>`, so it shares its reference counting and copy-on-demand behaviour; the number of blocks is `NRVec::size()` of the base, the number of bits is the (hiding) `bitvector::size()`. Bit i is bit i mod 64 of block $\lfloor i/64 \rfloor$ (least significant first). The private member `modulo` $= n$ mod 64 remembers the bit length; the unused high bits of the last block ("padding") may contain garbage and are cleared by `zero_padding()` where needed. `bitvector_rounded(n)` rounds n up to a multiple of 64.

Besides a plain bit set, a `bitvector` is interpreted as a polynomial over GF(2): bit i is the coefficient of x^i. Addition and subtraction are XOR, multiplication and division are polynomial operations, and with an irreducible polynomial the field GF(2^n) is available. A comment in the header plans a `monic_bitvector` class (not storing the leading 1, so that field operations need no resizing); it is *not implemented*.

17.2 Copy on demand

Copy constructor and `operator=` share the data (`operator=` is compiler-generated, calling `NRVec::operator=`), `|=` of the base class makes a

deep copy (but beware it is overloaded as bitwise OR here, see Pitfalls). The following members call `copyonwrite()`: `clear`, `fill`, `randomize`, `operator++`, `operator-`, `&=`, `|=`, `^=` (with a bitvector argument), `>>=`, `leftshift`/`<<=`; `resize` detaches. The single-bit and block setters do *not*: `set`, `reset`, `flip`, `assign`, `setblock`, and `&=`, `|=`, `^=` with a `bitvector_block` argument (also the `bitmatrix` and `bitvector_from1` wrappers of `set`/`reset`/`flip`/`assign`). They are element-level access by design (used in inner loops) and behave like writes through `operator[]`: call `x.copyonwrite()` once before using them on a possibly shared vector. In DEBUG builds they raise `laerror` when the data are shared (count check as in `NRVec::operator[]`).

17.3 Reference: construction, access

`bitvector(); explicit bitvector(const unsigned int n)`
: Empty; n bits, cleared.

`bitvector(const bitvector_block a, const unsigned int n)`
: n bits, every *block* set to `a`.

`bitvector(const bitvector &rhs)`
: Shallow copy.

`explicit bitvector(const uint8_t *data, const unsigned int n); void getdata(uint8_t *data)`
: Import/export n bits from/to a byte array (`(n+7)/8` bytes, bit i in byte `i/8`); little-endian machines only (laerror otherwise).

`void resize(const unsigned int n, bool preserve=false)`
: New bit length; with `preserve` keep the old bits and zero the new ones, otherwise clear everything.

`unsigned int size() const`
: Number of bits.

`const bool operator[](const unsigned int i) const; const bool get(const unsigned int i) const`
: Read bit i (no assignable non-const `[]` exists).

`bitvector_block getblock(const unsigned int i) const; void setblock(const unsigned int i, const bitvector_block b); int getblocksize() const`
: Raw block access, block size in bits (64).

`void set(const unsigned int i); void reset(const unsigned int i); void flip(const unsigned int i); const bool assign(const unsigned int i, const bool r)`
: Set to 1, to 0, negate, assign bit i (range checked in DEBUG; no copyonwrite, see above).

`void clear(); void fill()`
: All bits 0 / 1.

`void zero_padding() const`
: Clear the unused bits of the last block (logically const).

`bool is_zero() const; bool iszero() const; bool is_one() const`
: Test for the zero vector / the polynomial 1.

`void randomize()`
: Random bits via `RANDINT64` (chapter 19).

`void read(int fd, bool dimensions=1, bool transp=0); void write(int fd, bool dimensions=1, bool transp=0)`
: Binary I/O on a file descriptor: `modulo` followed by `NRVec::get/put` (section 3.7).

17.4 Reference: bitwise and integer operations

`bitvector &operator++(); bitvector &operator--()`
: Increment/decrement as a multi-word binary integer (bit 0 least significant), wrapping around.

`bool operator==(const bitvector &) const; bool operator!=(const bitvector &) const`
: Equality; vectors of different length are equal iff the extra bits of the longer one are zero.

`bool operator<(const bitvector &) const; operator>, operator<=, operator>=`
: Comparison as unsigned integers (bit i has the weight 2^i, the same reading as `getblock` and `++`), only for equal lengths (laerror otherwise).

`bitvector operator~() const`
: Bitwise negation.

`bitvector &operator&=(const bitvector &); operator|=, operator^=; operator&, operator|, operator^`
: Bitwise operations; `*this` is enlarged (zero-filled) if `rhs` is longer.

`bitvector &operator&=(const bitvector_block rhs); operator|=, operator^=`
: Bitwise operation with the *first block* only.

`bitvector &operator»=(unsigned int i); bitvector operator»(unsigned int i) const`
: Logical right shift (towards bit 0), length kept.

`bitvector &leftshift(unsigned int i, bool autoresize=false); bitvector &operator«=(unsigned int i); bitvector`

`operator«(unsigned int i) const`
: Logical left shift; `«` and `«=` enlarge the vector by i bits (autoresize), `leftshift` by default keeps the length and drops the top bits.

`unsigned int population(const unsigned int before=0) const`
: Hamming weight (number of ones); `before`$\neq 0$ is not implemented (laerror).

`unsigned int bitdiff(const bitvector &y) const`
: Hamming distance (same number of blocks required).

`unsigned int nlz() const; unsigned int ntz() const`
: Number of leading (from the top bit) and trailing zeros.

17.5 Reference: polynomials over GF(2) and GF(2^n)

`bitvector operator+(const bitvector &) const; operator-, operator+=, operator-=`
: Addition = subtraction = XOR.

`bitvector multiply(const bitvector &rhs, bool autoresize=true) const; bitvector operator*(const bitvector &rhs) const; bitvector &operator*=(const bitvector &rhs)`
: Polynomial product (naive shift-and-add, no carry-less multiply instructions; for short vectors). With `autoresize` the result has `size()+rhs.size()` bits, otherwise `max` of the sizes (the caller guarantees no overflow).

`bitvector pow(unsigned int n) const`
: p^n by repeated squaring.

`bitvector division(const bitvector &rhs, bitvector &remainder) const; bitvector operator/(const bitvector &) const`
: Polynomial division with remainder (`operator%` returns the remainder); the remainder is resized to `rhs.size()`. Division by zero: laerror.

`bitvector gcd(const bitvector &rhs) const; bitvector lcm(const bitvector &rhs) const`
: Greatest common divisor (Euclid) and least common multiple.

`bitvector composition(const bitvector &rhs) const`
: $p(r(x))$ by the Horner scheme.

`unsigned int degree() const`
: Polynomial degree (0 for the zero polynomial).

`void truncate(int t=0)`
: Resize to `max(degree()+1, t)` bits, preserving data.

`bool is_irreducible() const`

Irreducibility test over GF(2) (Rabin-type test: $x^{2^d} \equiv x$ and $\gcd(x^{2^{d/p}} - x, f) = 1$ for prime divisors p of the degree d, cf. Brent–Zimmermann; uses `factorization` of section 19.8).

`bitvector field_mult(const bitvector &rhs, const bitvector &irpolynom) const`

Product in GF(2^n) = GF(2)[x]/(irpolynom), n = deg(irpolynom); the operands must have at least n bits.

`bitvector field_inv(const bitvector &irpolynom) const; bitvector field_div(const bitvector &rhs, const bitvector &irpolynom) const`

Multiplicative inverse (extended Euclid; laerror if the polynomial is reducible or the input is 0 modulo it) and division in GF(2^n).

`bitvector field_pow(unsigned int n, const bitvector &irpolynom) const; bitvector field_sqrt(const bitvector &irpolynom) const`

Power and square root ($\sqrt{a} = a^{2^{n-1}}$) in GF(2^n).

`bitvector field_composition(const bitvector &rhs, const bitvector &irpolynom) const`

$p(r)$ evaluated in GF(2^n).

`bitvector find_irreducible(int deg, int pop=-1, int nth=1)`

Free function: an irreducible polynomial of degree `deg`. With `pop`$=-1$ random trials; otherwise the `nth` one (in the order of a recursive search) with Hamming weight `pop` (odd, ≥ 3); returns a zero vector if none was found.

17.6 bitmatrix, bitvector_from1 and helper functions

`bitmatrix` (derived from `bitvector`) stores an $n \times m$ bit matrix row-wise (bit `i*m+j`). `bitvector_from1` is a `bitvector` indexed from 1.

`bitmatrix(); bitmatrix(unsigned int nn0, unsigned int mm0); void resize(const unsigned int n, const unsigned int m)`

Construct/resize (data cleared).

`unsigned int nrows() const; unsigned int ncols() const`

Dimensions.

`const bool operator()(const unsigned int i, const unsigned int j) const`

Read element.

`void set(i,j); void reset(i,j); void flip(i,j); const bool assign(i,j,const bool r)`

Write element (no copyonwrite).

`bitvector_from1(); bitvector_from1(const bitvector &rhs); explicit bitvector_from1(const unsigned int n)`
: Constructors.

`operator[](i); get(i); set(i); reset(i); assign(i,r); population(before)`
: As in `bitvector`, with indices shifted by one. (Note `operator[]` and `get` are non-const here.)

`void bitvector_expand(const bitvector &v, NRVec<T> &r); void bitvector_compress(bitvector &r, const NRVec<T> &v)`
: Convert to/from a vector of 0/1 values of type `T`.

`bitvector mantissa(T x, int nbits, int shift=0)`
: Binary expansion of $x\,2^{\text{shift}} \in [0,1)$ (laerror otherwise): bit i ($1 \le i <$ nbits) is the coefficient of 2^{-i}.

`void bitvector_decimal(T &x, const bitvector &b, int shift=0)`
: Inverse of `mantissa`: $x = 2^{\text{shift}} \sum_i b_i 2^{-i}$.

`std::ostream &operator«(std::ostream &, const bitvector &); std::istream &operator»(std::istream &, bitvector &)`
: Text I/O as a string of 0/1 characters, bit 0 first, no length (input: one whitespace-delimited token, any character other than `'0'` is 1).

`std::ostream &operator«(std::ostream &, const bitmatrix &); std::istream &operator»(std::istream &, bitmatrix &)`
: Dimensions, then rows of 0/1 (input: whitespace-separated integers).

`LA_traits<bitvector>` declares `elementtype` and `normtype bool`.

17.7 Example

```
int d=13;
bitvector ir=find_irreducible(d,3);     //trinomial, if any
if(ir.is_zero()) ir=find_irreducible(d,5);
bitvector a(d);
do a.randomize(); while(a.is_zero());
bitvector ai = a.field_inv(ir);
if(!a.field_mult(ai,ir).is_one()) laerror("GF(2^n) inversion
    failed");
bitvector s = a.field_sqrt(ir);         //s*s == a in GF(2^13)
bitvector u(100),v(100);
u.randomize(); v.randomize();
bitvector w=u*v;                        //polynomial product, 200
     bits
if(!(w%u).is_zero() || !(w/u-v).is_zero()) laerror("division");
```

17.8 Pitfalls

Caution: `|=` is bitwise OR here, not the deep copy of the base class `NRVec`; to get an independent copy use `bitvector b(a); b.copyonwrite();`.

Caution: Bits beyond `size()` in the last block may be nonzero after `fill()`, `setblock()` or the block constructor; most predicates call `zero_padding()`, but use it explicitly before raw block access.

Chapter 18

Regression

This chapter describes the kernel regression module: the class `REG` for kernel regression of a scalar function of points in $\mathbb{R}^d$, and the class `REGSURF` for regression of (potential energy and other) surfaces of molecular geometries, including the computation of absorption spectra by the machine-learning nuclear ensemble approach (ML-NEA).

Authorship and status. The files `reg.h`, `reg.cc`, `regsurf.h`, `regsurf.cc` and the test programs `test_reg.cc`, `test_regsurf.cc` state "Kernel ridge regression module Copyright (C) 2024 Pavel Florian `<florian43@seznam.cz>` and Jiri Pittner", under the same GNU GPL version 3 (or later) licence as the rest of LA. The module is written in a different style from the rest of the library: it works mostly with raw pointers into internally allocated `NRVec`/`NRMat` objects, returns `unsigned int` status codes (0 = success) in addition to calling `laerror`, and reads its input from text files. The headers are *not* included by `la.h`; include `reg.h` or `regsurf.h` explicitly. Note that both headers include `<bits/stdc++.h>` (GCC specific) and contain `using namespace std;` at global scope. The classes are instantiated for `T=double` only.

The descriptions below say what the code does.

Caution: The regression module is experimental and has not yet been tested much in practice: a number of errors were found and fixed only in October 2026. The comments in the code mention an elastic-net loss, which the fitting algorithms do not implement; the descriptions below document what the code actually does.

18.1 Kernel functions

Header: `reg.h, reg.cc`

All kernels take a difference vector `x` of length `dimensions` and a width σ.

`T Gaussian_kernel(const T *x, const unsigned int dimensions, const T sigma)`
: $\exp(-|\mathbf{x}|^2/2\sigma^2)$.

`T Laplacian_kernel(const T *x, const unsigned int dimensions, const T sigma)`
: $\exp(-\sum_i |x_i|/\sigma)$ (1-norm).

`T Exponential_kernel(const T *x, const unsigned int dimensions, const T sigma)`
: $\exp(-|\mathbf{x}|/\sigma)$ (2-norm).

`T Matern_kernel(const T *x, const unsigned int dimensions, const T sigma, const unsigned int n, const T *Matern_comb_number)`
: Matérn kernel of half-integer order $\nu = n + \frac{1}{2}$: $e^{-r/\sigma}\sum_{k=0}^{n} c_k\,(2r/\sigma)^{n-k}$, $r = |\mathbf{x}|$, i.e. the usual form with length scale $l = \sigma\sqrt{2\nu}$ ($n = 0$ exponential, $n = 1$: $(1 + r/\sigma)e^{-r/\sigma}$); evaluated by the Horner scheme, $k(0) = 1$. `Matern_comb_number` must have $n + 1$ elements.

`unsigned int Matern_coefficient(const unsigned int n, T *comb_number)`
: Fills $c_k = \frac{(n+k)!}{(2n)!}\binom{n}{k}$, $k = 0..n$ ($n + 1$ elements), by the recurrence $c_n = 1$, $c_{k-1} = c_k\,k/((n + k)(n - k + 1))$ (no factorials, no overflow for large n).

The kernel is selected by name: `"Gaussian"`, `"Laplacian"`, `"Exponential"`, `"Matern"`; for Matérn the order n is an additional parameter (rounded to an integer).

18.2 REG: kernel regression in $\mathbb{R}^d$

Header: `reg.h, reg.cc`

18.2.1 Model and parameters

The training set consists of N points $\mathbf{x}_i \in \mathbb{R}^d$ (array of $N \cdot d$ values, point after point) with values y_i. The model is a weighted sum of kernels centred at the training points. The parameter array is $(\lambda, \alpha, \sigma[, n])$: regularization strength λ, mixing $\alpha \in [0, 1]$ (clamped) between the "kernel ridge" part $\kappa = \lambda(1 - \alpha)/2$ and the "lasso" part $\lambda\alpha$, kernel width σ and, for the Matérn kernel, the order n. As implemented, the prediction for a point $\mathbf{x}$ is

$$\hat{y}(\mathbf{x}) = \Big(\sum_j w_j k(\mathbf{x} - \mathbf{x}_j)\Big)\,\frac{S + \kappa}{S} + \lambda\alpha, \qquad S = \sum_j k(\mathbf{x} - \mathbf{x}_j),$$

(the factor is omitted if $\kappa = 0$ or $S = 0$).

18.2.2 Construction

`REG(const T *x, const T *y, const T *params, const unsigned int data_size, const unsigned int x_dimensions, const string kernel)`
: Model from memory (data are copied).

`REG(const string x_file, const string y_file, const string param_file)`
: Model from text files, see below.

`void Init(const T *x, const T *y, const T *params, const unsigned int data_size, const unsigned int x_dimensions, const string kernel)`
: Called by the constructors; allocates the working storage. With an unknown kernel name it returns silently and the object stays uninitialized (later calls fail with laerror).

`~REG()`
: Frees the internal buffers.

File formats (one number per line): `param_file` contains N, d, the kernel name, λ, α, σ and for `"Matern"` n; `x_file` contains $N{\cdot}d$ coordinates, `y_file` N values. If a file cannot be read the constructor returns silently without initializing the object.

18.2.3 Fitting and prediction

`unsigned int REG_Fit(const T max_Loss, const unsigned int max_learning_cycles)`
: Iterative fit. Builds the kernel matrix $K_{ij} = k(\mathbf{x}_i - \mathbf{x}_j)$ (unit diagonal) and its row sums s_i on first use, starts from $w = y$ and repeats at most `max_learning_cycles` times the multiplicative update $p = Kw$, $p_i \leftarrow p_i(s_i + \kappa)/s_i + \lambda\alpha$, $w_i \leftarrow w_i y_i/p_i$. If `max_Loss`$> 0$, stops when `Loss_Function()`$<$`max_Loss`.

`unsigned int REG_Fit_grad(const T max_Loss, const unsigned int max_learning_cycles, const T learning_rate)`
: Gradient-type variant: with p as above (same κ), $g_j = \sum_k (p_k - y_k)K_{jk}/\sqrt{N}$ and $w_j \leftarrow w_j/(1 - \eta g_j)$, η=`learning_rate`.

`T REG_RMSE()`
: Root mean square deviation of the predictions stored during the last fitting cycle from the training values.

`T Loss_Function()`
: Currently identical to `REG_RMSE()` (the regularization terms mentioned in the comment are not added).

`unsigned int REG_Get_predictions(const T *x, T *y_preds, const unsigned int data_size)`
: Predictions for `data_size` new points (array of `data_size`$\cdot d$ values).

`unsigned int REG_Get_Score(const T *x, T *y_preds, const unsigned int data_size, const T *y, T *score)`
: Predictions and `score[0]`$= \sum_i (y_i - \hat{y}_i)^2$ (sum of squares, not RMSE).

`unsigned int REG_Save_fitted(const string model_file); unsigned int REG_Load_fitted(const string model_file)`
: Write/read the N weights, one per line (the model itself must be constructed with the same data and parameters).

`unsigned int REG_Set_params(const T *params, const string kernel); unsigned int REG_Get_params(T *params, string *kernel)`
: Set/get $(\lambda, \alpha, \sigma[, n])$ and the kernel name. If the kernel, σ or n changes, the kernel matrix is rebuilt in the next fit (a change of λ, α only needs no rebuild).

`unsigned int REG_Get_weight_coeff(T *weight)`
: Copy the N weights.

`unsigned int Build_REG_distances_mat(); unsigned int Build_REG_kernels_mat(); unsigned int Build_REG_kernels_mat_sums(); T Kernel(const T *x, const unsigned int dimensions, const unsigned int kernel_id)`
: Internal steps, public in the class: difference vectors, kernel matrix, its row sums, kernel by numeric id (0 Gaussian, 1 Laplacian, 2 Exponential, 3 Matérn).

18.2.4 Example

Abridged from `test_reg.cc`:

```
#include "reg.h"
using namespace LA;

REG<double> reg("reg_test_x.csv", "reg_test_y.csv", "reg_test_p.
    csv");
reg.REG_Fit(0.0, 20);          //20 cycles, no loss threshold
double x=0.5, y;
reg.REG_Get_predictions(&x, &y, 1);
reg.REG_Save_fitted("reg_test_m.csv");
```

18.3 REGSURF: regression of molecular surfaces and ML-NEA spectra

Header: `regsurf.h, regsurf.cc`

18.3.1 Model

A `REGSURF` model describes `count_levels` surfaces (e.g. the ground and excited state energies) as functions of the geometry of a system of `count_kernels` particles (atoms) in `count_dimensions` dimensions, trained on `count_geometries` geometries. For each geometry the matrix of interparticle distances r_{jk} is converted to a *descriptor* matrix D_{jk}:

- `"RE"`: $D_{jk} = r^{(0)}_{jk}/r_{jk}$, with the first training geometry as reference, zero diagonal;
- `"Coulomb_matrix"`: $D_{jk} = Z_j Z_k / r_{jk}$, $D_{jj} = Z_j^{2.4}$ (needs the nuclear charges Z);
- `"Inverse_pairs"`: $D_{jk} = 1/r_{jk}$, zero diagonal;
- a user function set by `s_Set_extern_descriptor`.

For geometry g the kernel matrix is $K^{(g)}_{jk} = k(D^{(g)}_{jk} - D^{(0)}_{jk})$ (one-dimensional kernel of the descriptor difference with respect to the first geometry) and its row sums $s^{(g)}_j$ enter the prediction for level ℓ, which is essentially $\sum_j w^{(\ell)}_j s^{(g)}_j$ with one weight per particle and level, corrected by a kernel ridge factor and the additive $\lambda_\ell \alpha_\ell$ as in `REG`. The parameter array is $(\lambda_0..\lambda_{L-1}, \alpha_0..\alpha_{L-1}, \sigma[, n])$, L=`count_levels`.

18.3.2 Construction and data files

```
REGSURF(const unsigned int count_dimensions, const unsigned int
    count_kernels, const unsigned int count_geometries, const
    unsigned int count_levels, const string descriptor, const string
    kernel, const T *params, const T *geometries, const T *values,
    const unsigned int *Z = nullptr)
```
Model from memory. `geometries`: $G \cdot K \cdot d$ coordinates (geometry, particle, coordinate); `values`: $L \cdot G$ values, level-major (all geometries of level 0 first); `Z`: K charges (needed for `"Coulomb_matrix"`, otherwise `laerror`; may be omitted for the other descriptors).

```
REGSURF(const string geometries_file, const string energies_file,
    const string parameters_file, const string Z_file = "")
```
Model from text files, see below.

```
void Init(...)
```
Same arguments as the memory constructor; called by both constructors. Unknown kernel: laerror.

```
~REGSURF()
```
Frees the internal buffers.

File formats: the parameters file contains, one per line, d, K, G, L, the descriptor name, the kernel name, $\lambda_0..\lambda_{L-1}$, $\alpha_0..\alpha_{L-1}$, σ and for `"Matern"` n. The geometries file is free format: all numbers containing a decimal point (or comma) are read in order, other tokens (atom counts, element symbols, comments) are skipped, so concatenated XYZ files can be used. The energies file has one value per line in level-major order, the Z file one integer per line.

18.3.3 Fitting, prediction and parameters

`unsigned int s_Fit(const T max_Loss, const unsigned int max_learning_cycles, const T learning_rate, const bool element_invariant=true, unsigned int *kernel_classes=nullptr)`

Fits the weights of all levels: builds distances, descriptors, kernel matrices and row sums on first use, initializes the weights of level i to the constant $(y_{i,0} - \lambda_i\alpha_i)/(K^2 + \kappa_i)$ ($y_{i,0}$ value at the base geometry 0, $\kappa_i = \lambda_i(1 - \alpha_i)K/2$) and performs at most `max_learning_cycles` sweeps of gradient-like updates over geometries $1..G-1$ with step `learning_rate`; stops a level early if `max_Loss`> 0 and its loss is below `max_Loss`. With `element_invariant`, the update of each weight is replaced by the mean over the particles of the same class (`kernel_classes`, default the charges Z; without both no averaging is done).

`T s_RMSE(unsigned int level); T Loss_Function(unsigned int level)`

RMSE of the predictions on the training geometries for one level; `Loss_Function` currently returns the same.

`unsigned int s_Get_predictions(const unsigned int count_geometries, const unsigned int count_levels, const T *geom, const unsigned int *surfaces, T *predictions)`

Predictions for new geometries `geom` and the levels listed in `surfaces[0..count_levels-1]`; `predictions[i*count_levels+j]` is geometry i, level `surfaces[j]`.

`unsigned int s_Score(const unsigned int count_geometries, const unsigned int count_levels, const T *geom, const unsigned int *surfaces, T *predictions, const T *y_values, T *score)`

Predictions and their RMSE against `y_values` in `score[0]`.

`unsigned int s_Save_fitted(const string weights); unsigned int s_Load_fitted(const string weights)`

Write/read the $L \cdot K$ weights, one per line.

`unsigned int s_Set_params(const string descriptor, const string kernel, const T *params); unsigned int s_Get_params(string`

`*descriptor, string *kernel, T *params)`

Change/query descriptor, kernel and parameter array; a change of descriptor, kernel, σ or n invalidates the precomputed matrices (rebuilt in the next fit; λ, α need no rebuild). Unknown descriptor or kernel names raise `laerror` (also in the constructors). The descriptor is always set from its name, so a descriptor installed by `s_Set_extern_descriptor` must be installed again after `s_Set_params`.

`unsigned int s_Set_extern_descriptor(unsigned int (*extern_func)(const unsigned int, const unsigned int, const T*, T*))`

Use a user descriptor function `f(count_particles, count_geometries, distances, desc)` which maps the $K \times K$ distance matrices of all geometries (consecutive) to descriptor matrices of the same layout.

`unsigned int s_Get_weight_coeff(const unsigned int s_number, T *weights)`

Copy the K weights of level `s_number`.

`T Kernel(const T *x, const unsigned int dimensions, const unsigned int kernel_id); unsigned int Build_s_distances_mat(...); unsigned int Build_s_descriptor_mat(...); unsigned int Build_s_kernels_mat(...); unsigned int Build_s_kernels_mat_sums()`

Internal steps, public in the class.

Free functions `Build_s_RE_descriptor_mat`, `Build_s_coulomb_matrix_descriptor_mat`, `Build_s_inverse_pairs_descriptor_mat` (`count_particles, count_geometries, distances, desc` and the reference distances resp. Z) implement the built-in descriptors.

18.3.4 ML-NEA absorption spectra

`unsigned int s_Load_ML_NEA_geometries(const unsigned int count_geometries, const T *geometries); unsigned int s_Load_ML_NEA_geometries(const unsigned int count_geometries, const string geometries_file)`

Load an ensemble of geometries (e.g. sampled from a ground-state distribution) from memory or from a free-format geometry file as above.

`unsigned int s_Predict_ML_NEA_geometries()`

Predict all levels for all ensemble geometries.

`unsigned int s_Compute_ML_NEA_spectra(const T E_min, const T step,`

`const unsigned int count_steps, const T delta)`

Spectrum on the grid $E_k = E_{\min} + k \cdot \text{step}$, $k <$ count_steps, as a sum over geometries and levels of Gaussians of width δ centred at the predicted excitation energies ΔE (energies assumed in atomic units), weighted by $\Delta E\, f/E$. The Gaussians are normalized, $\sqrt{2/\pi}\,\delta^{-1}\exp(-2(E-\Delta E)^2/\delta^2)$ (standard deviation $\delta/2$), and only grid points within 20δ of ΔE are evaluated (beyond that the exponential underflows to zero). The oscillator strength $f = \Delta E_{\min}/\Delta E$ ($\Delta E_{\min}$ the minimum of the level over the ensemble) is only a placeholder for the true oscillator strength, which the model does not predict.

`unsigned int s_Save_ML_NEA_spectra(const string ML_NEA_spectra_file)`

Write the spectrum normalized to unit sum, one value per line.

18.3.5 Example

Abridged from `test_regsurf.cc` (10 levels):

```
#include "regsurf.h"
using namespace LA;

REGSURF<double> regsurf("xyz.dat", "E_sum.dat", "conf_surf.dat",
    "Z.dat");
regsurf.s_Fit(0.0, 100, 0.5);
for(unsigned int i=0; i<10; ++i) cout << regsurf.s_RMSE(i) <<
    endl;
regsurf.s_Load_ML_NEA_geometries(50000, "nea_geoms.xyz");
regsurf.s_Predict_ML_NEA_geometries();
regsurf.s_Compute_ML_NEA_spectra(0.00, 0.0025, 4000, 0.2);
regsurf.s_Save_ML_NEA_spectra("spectra.dat");
```

18.4 Pitfalls

Caution: `REG` and `REGSURF` hold raw pointers to their own buffers and have no user-defined copy constructor or assignment: copying an object leads to double deletion. Pass them by reference.

Caution: Errors in the input files are partly reported by `laerror` and partly lead to a silently uninitialized object (`REG`); check the return codes and the results.

Chapter 19

Utilities

This chapter documents the auxiliary parts of LA which are not linear algebra classes: a scratch-file storage of fixed-size records used by the out-of-core variants of the iterative solvers, the random number generators used by all `randomize()` functions, a few special functions and combinatorial numbers, small self-contained algorithms for embedded use, the reference of the error handling functions, the single include file `la.h` and the version string, elementary number theory and graph algorithms. The small classes Vec3/Mat3, Quaternion and bitvector are described in chapters 15, 16 and 17.

19.1 Scratch-file storage: AuxStorage

Header: `auxstorage.h`

19.1.1 Purpose and storage

`AuxStorage<T>` stores numbered records of equal length in a binary file, using unformatted `read`/`write` of the raw element array. It is used by `davidson`, `lanczos`, `gmres` and `DIIS` (chapter 9) to keep the subspace vectors on disk when the in-core option is not selected, and to write eigenvectors to a named file. The record length (in bytes) is fixed by the first `put()`; all later records must have the same size. Record `pos` starts at byte offset `pos*recl`, so records may be written in any order; reading a never written record lying before the end of the file gives zeros (a hole of the file), reading past the end of the file calls `laerror`.

The class itself holds only a file descriptor and the file name; it is *not* reference counted. The objects stored are ordinary LA objects: `get()` calls `x.copyonwrite()` before reading into `x`, `put()` only reads its argument.

The comment in the header gives the limits of the design:

- `T` must be plain data (`double`, `std::complex<double>`, ...); it will not work if `T` is a class with dynamically allocated components (e.g. `NRVec<NRVec<double> >` or `NRVec<NRMat<double> >`), since only the bytes of the top-level array are written;
- it cannot be implemented for `SparseMat`, which lacks a fixed record length;
- for more complex I/O the `put()` and `get()` methods of the individual classes are to be used (section 3.7).

19.1.2 Reference

`template<typename T> class AuxStorage`
: Scratch file of records whose elements are of type `T`.

`AuxStorage(void)`
: Creates and opens a new unique temporary file `AUX_XXXXXX` in `\$TMPDIR` if it is set, otherwise in the current working directory, by `mkostemp()` (atomic creation, with `O_LARGEFILE`), and unlinks it at once: the open descriptor keeps it usable, and its disk space is released when it is closed, also after a crash. `laerror` if the file cannot be created.

`AuxStorage(const char *name)`
: Opens the file `name` for reading and writing; an existing file of this name is first deleted (`unlink`), so the file always starts empty. The file is *not* deleted by the destructor (used to keep eigenvectors). `laerror` if it cannot be opened or if the name has 32 or more characters.

`~AuxStorage(void)`
: Closes the file; deletes it if it was created by the default constructor.

`AuxStorage(const AuxStorage &) = delete; AuxStorage &operator=(const AuxStorage &) = delete`
: Not copyable (a copy would close the file twice); use pointers, as the iterative solvers do.

`void put(const NRVec<T> &x, const int pos)`
: Writes the elements of `x` as record `pos` (counted from 0). The first `put` defines the record length; `laerror` if a later object has a different size in bytes.

`void get(NRVec<T> &x, const int pos) const`
: Reads record `pos` into `x`, which must be already allocated with the right size. Calls `x.copyonwrite()`. `laerror` if nothing has been written yet, if the size of `x` in bytes differs from the record length, or if the record cannot be read in full.

`void put(const NRMat<T> &x, const int pos); void get(NRMat<T> &x,`

`const int pos) const`
: The same for a dense matrix (record of `nrows()*ncols()` elements in the storage order of `NRMat`).

`void put(const NRSMat<T> &x, const int pos); void get(NRSMat<T> &x, const int pos) const`
: The same for a packed symmetric matrix (record of $n(n+1)/2$ elements).

Only the byte size is compared, so a record written as a vector can be read into a matrix of the same number of elements.

19.1.3 Example

```
AuxStorage<double> st;                  //temporary file
    AUX_xxxxxx in $TMPDIR or the working directory
NRVec<double> v(1.,100), w(100);
st.put(v,3);                            //record 3, record length
    100 doubles
st.get(w,3);                            //w must be allocated with
     100 elements
```

19.1.4 Pitfalls

- The target of `get()` must be preallocated with exactly the record size (otherwise `laerror`).
- The temporary file is placed in `\$TMPDIR`, or in the current working directory if `TMPDIR` is not set; it is not visible in the directory listing (unlinked), but occupies disk space there while the object exists.
- The file position is shared: `get()` and `put()` use `lseek64` followed by `read`/`write`, so one object must not be used concurrently from several threads.
- `lseek64`, `off64_t`, `O_LARGEFILE` and `mkostemp` are Linux (glibc) specific.
- A short write is not detected (only a negative return value of `write` is checked).

19.2 Random numbers

Header: `la_random.h, la_random.cc`

All random-number consumers of the library (`randomize()` of `NRVec`, `NRMat`, `NRSMat`, `Vec3`, `Mat3`, `Quaternion`, `bitvector`, random permutations, ...) obtain random numbers through four macros, so that the generator

can be replaced without changing the library. `la_random.h` is included by `la_traits.h`, i.e. by every LA header.

`double randdouble()`
: Uniform in $[0, 1)$: `random()/(1.+RAND_MAX)`.

`double randdoublesigned()`
: Uniform in $[-1, 1)$: `2.*random()/(1.+RAND_MAX)-1.`.

`unsigned int randint32()`
: 32 random bits from two calls of `random()` (which gives 31 bits with glibc).

`uint64_t randint64()`
: 64 random bits from three calls of `random()` ($31 + 31 + 2$ bits).

`RANDDOUBLE, RANDDOUBLESIGNED, RANDINT32, RANDINT64`
: Macros used at all call sites; unless defined before the inclusion of `la_random.h` they expand to `LA::randdouble`, `LA::randdoublesigned`, `LA::randint32`, `LA::randint64`.

`WEAK_SYMBOL`
: Expands to `__attribute__((weak))` with GNU compilers, empty otherwise.

19.2.1 Replacing the generator

The four functions are defined in `la_random.cc` as *weak symbols*. A program can replace any of them by its own (strong) definition with the same signature in namespace `LA`; the linker then uses the user's function also in the calls made from inside the precompiled library, and the remaining default functions stay available:

```
#include "la.h"
namespace LA {
double randdoublesigned() {return my_generator.uniform(-1.,1.);}
      //replaces the library's weak definition
}
```

Defining one of the macros (e.g. `-DRANDDOUBLE=myrand`) affects only code compiled by the user (inline and template code in the headers), not the calls in the compiled library; the weak-symbol replacement is therefore the mechanism to use for a consistent change.

19.2.2 Seeding

The default functions use the C library generator `random()`, so they are seeded by `srandom(seed)`; with glibc `srand()` seeds the same generator (as used in `testoptimize.cc` and `t.cc`). Without seeding the sequence is the same in every run. A seed from the system can be taken e.g. from `/dev/random`, as in `t.cc`:

```
int seed;
int f=open("/dev/random",O_RDONLY);
if(sizeof(int)!=read(f,&seed,sizeof(int))) laerror("cannot read /
    dev/random");
close(f);
srand(seed);
```

The default generators are not thread safe in the sense of reproducibility (they share the global state of `random()`); a thread-safe or higher-quality generator is to be supplied by replacement.

19.3 Special functions and combinatorial numbers

Header: `miscfunc.h, miscfunc.cc`

Plain functions in namespace `LA` for `double` and 64-bit unsigned integers. They keep internal static tables (filled on demand), so they are not thread safe on the first calls.

`const double c_euler`
: Euler–Mascheroni constant $\gamma = 0.5772156649\ldots$

`unsigned long long factorial(const int n)`
: $n!$ from a table, $0 \le n \le 20$; `laerror` for $n < 0$ and for $n > 20$ (overflow of 64 bits). Exact; used where integer arithmetic is needed (permutation ranks and counts, `binom`).

`double double_factorial(const int n)`
: $n!$ (not the double factorial $n!!$) in `double` precision, $0 \le n \le 170$: `factorial(n)` for $n \le 20$, beyond from a table extended on demand. `laerror` for $n < 0$ and $n > 170$. Used by `bernoulli_number`, `zeta` and `gamma`.

`double factorialln(const int n)`
: $\ln n!$ for any $n \ge 0$: tabulated up to $n = 30$, accumulated (and cached) up to $n = 180$, Stirling series with Bernoulli-number corrections beyond. `laerror` for $n < 0$.

`unsigned long long binom(int n, int k)`
: Binomial coefficient $\binom{n}{k}$; 0 for $k < 0$ or $k > n$. With k replaced by $\min(k, n-k)$, computed exactly by the recurrence $\binom{n-k+i}{i} = \binom{n-k+i-1}{i-1}(n-k+i)/i$, dividing by the common factor before each multiplication, so that no intermediate exceeds the result; cached for $n \le 100$. `laerror` if the result does not fit into 64 bits (all $\binom{n}{k}$ fit for $n \le 67$).

`unsigned long long longpow(unsigned long long x, int i)`
: x^i by binary powering (modulo 2^{64}, overflow not detected); returns 0 for $i < 0$.

`double zeta(double x)`

Riemann zeta function for real $x \neq 1$. Integer arguments: exact values from Bernoulli numbers for $x \leq 0$ and $x = 2, 4, 6, 8$; otherwise (also for other integers): the defining sum for $x \geq 9.1$, the Euler–Maclaurin summation formula with $N = 10$ and ten Bernoulli terms for $-1.02 \leq x < 9.1$ (including the neighbourhood of the pole, whose term $1/(x-1)$ is exact), and the reflection formula $\zeta(x) = 2(2\pi)^{x-1} \sin(\pi x/2)\Gamma(1-x)\zeta(1-x)$ for $x < -1.02$. Relative accuracy about 10^{-13} or better (except close to the trivial zeros). `laerror` for $x = 1$.

`double gamma(double x)`

$\Gamma(x)$ for $x \leq 171$: $(x-1)!$ by `double_factorial` for integers, otherwise upward recursion to $x \geq 8$ and the Stirling series. `laerror` for $x > 171$ and for non-positive integers.

`double gammln(double x)`

$\ln|\Gamma(x)|$, for integers via `factorialln`; `laerror` for non-positive integers.

`double bernoulli_number(int n)`

Bernoulli number B_n with $B_1 = -\frac{1}{2}$, $B_{2k+1} = 0$ for $k \geq 1$; B_{2k} from a table up to B_{10}, otherwise from $\zeta(2k)$ by $B_{2k} = (-1)^{k+1}\, 2\,(2k)!\,\zeta(2k)/(2\pi)^{2k}$ (cached). `laerror` for $n < 0$.

`unsigned long long simplicial(int d, int n)`

d-dimensional simplicial (figurate) number $s(d,n) = \binom{n+d-1}{d}$, i.e. the number of non-decreasing d-tuples of indices from n values (used for the indexing of symmetric tensors, chapter 11). $s(d,n) = 0$ for $n \leq 0$, 1 for $d = 0$ or $n = 1$, n for $d = 1$. Computed by the recurrence $s(d,n) = s(d,n-1) + s(d-1,n)$ into a table growing on demand; `laerror` for $d \geq 16$ or $n \geq 1024$.

`int inverse_simplicial(int d, unsigned long long s)`

Largest n with `simplicial(d,n)` $\leq s$ (Newton iteration on the polynomial $s(d,n)$ followed by an integer correction); 0 for $s = 0$. `laerror` for $d \geq 16$.

19.4 Simple algorithms for embedded use

Header: `simple.h, simple.cc`

`simple.h` contains small algorithms on C arrays of compile-time size, independent of BLAS, LAPACK and of the rest of LA. It is intended for embedded computers and can be compiled separately; it is installed with the library but *not* included by `la.h`. Everything is in namespace `LA_Simple`. With `AVOID_STDSTREAM` defined, `iostream` is not included and the stream operator is omitted. `simple.cc` is currently empty. The element type `T` is a real floating-point type (`abs` and comparisons are used for pivoting).

`template<typename T, int n, int m> T simple_gaussj(T (&a)[n][n], T (&b)[m][n])`

Gauss–Jordan elimination with full pivoting. On input `a` is the $n \times n$ matrix A and `b` contains m right-hand sides as its *rows* `b[l][0..n-1]`. On return `a` contains A^{-1}, the rows of `b` the solutions, and the function returns $\det A$. For a singular matrix it returns 0 and `a`, `b` are left partially transformed. `T (&a)[n][n]` binds e.g. to `Mat3<double>::elements()`.

`template<typename T, int n> class simple_linfit`

Linear least-squares fit $y \approx \sum_{i=0}^{n-1} c_i f_i(x)$ by accumulating the normal equations; the data points are not stored. Public members: `T fitmat[n][n]`, `T rhsmat[1][n]` (accumulated $\sum f_i f_j$, $\sum f_i y$), `T fitcoef[n]` (result c_i), `int npoints`.

`simple_linfit()`

Empty fit, coefficients zero.

`void clear(bool keepresults=false)`

Discard the accumulated data (and the coefficients unless `keepresults`).

`void input(const T (&funcs)[n], const T y)`

Add a data point given by the values $f_i(x)$ and y.

`T solve(const T preserve=0)`

Solves the normal equations by `simple_gaussj`, stores the coefficients in `fitcoef` and returns the determinant of the normal matrix (0 if singular). If fewer than n points have been input it returns 0 and leaves `fitcoef` unchanged. With `preserve==0` the accumulated data are cleared afterwards (the next fit starts from scratch); with `preserve!=0` they are kept and, if `preserve!=1`, multiplied by `preserve`, which gives an exponentially decaying weight of old points in a running fit (`npoints` is not reset then).

`const T &operator[](int i) const`

Coefficient c_i.

`std::ostream& operator«(std::ostream &o, const simple_linfit<T,n> &f)`

Prints the coefficients separated by spaces.

Example (from `t.cc`), fit of a parabola:

```
simple_linfit<double,3> fit;
double funcs[3];
for(int i=0; i<100; ++i)
        {
        double x = 10*RANDDOUBLESIGNED();
        funcs[0]=1; funcs[1]=x; funcs[2]=x*x;
        fit.input(funcs, 2*x*x-3*x+1);
        }
```

```
double det = fit.solve();
cout <<"det= "<<det<<" fit "<<fit<<endl;    //1 -3 2
```

Caution: `simple.h` defines and afterwards undefines a macro `SWAP`; a user macro of this name defined before the inclusion is lost.

19.5 Error handling reference

Header: `laerror.h, laerror.cc`

The concept (errors are reported by `laerror`, which throws an exception after printing diagnostics; checks enabled by `DEBUG`) is described in section 3.6. All entities are in namespace `LA`.

`laerror(const char *message)`
: Macro expanding to `LA::laerror2(message, __PRETTY_FUNCTION__)` with GNU compilers and `LA::laerror2(message, __func__)` otherwise. Usable in user code, also outside namespace `LA`.

`void laerror2(const char *message, const char *function)`
: Flushes and prints a newline on `std::cout` and `std::cerr`, then on both streams `LA:ERROR - message` and `Called from function` ("unspecified" if `message` is `NULL`); the system error message by `perror` if `errno` is nonzero; the command line (from `/proc/self/cmdline`); user, host name and working directory (via `system()`); a listing of the working directory if `LA_error_list_files` is true. Finally it throws `LAerror(message)`.

`class LAerror : public std::exception`
: Exception thrown by `laerror2`. Public member `std::string msg`, a copy of the message (so it stays valid also for messages built in local buffers, e.g. by the BLAS error handlers); constructor `LAerror(const char *s)` (`NULL` gives "unspecified"); `what()` returns the message, so an uncaught error or a handler catching `std::exception` shows it.

`std::ostream & operator«(std::ostream &s, const LAerror &x)`
: Prints `x.msg`.

`bool LA_error_list_files`
: Default `false`; if set, `laerror2` also runs `ls -l -t ..`

`bool _LA_count_check`
: Default `true`. In `DEBUG` builds the non-const `operator[]` and `operator()` of `NRVec`, `NRMat` and `NRSMat` call `laerror` if the reference count is larger than 1 (section 3.1) only while this variable is true; setting it to false switches this check off (the check is in inline header code, so it affects code compiled with `DEBUG`).

`const bool _LA_warn_empty_copyonwrite`
: Compile-time `false` in the library; if it were true, `copyonwrite()` of an empty object would print a warning. Not user-adjustable.

`int traceback(int flags)`
: Prints a stack trace of the running process by forking `/usr/bin/gdb` in batch mode attached to the process (allowed by `prctl(PR_SET_PTRACER)`); output goes to `stderr`. `flags` is currently unused. Linux specific; requires gdb. C linkage.

`void sigtraceback(int sig, int flags)`
: Installs a one-shot handler for signal `sig` which flushes `stdout`, calls `traceback(flags)` and re-raises the signal (default action, e.g. core dump); if the handler is entered again it calls `_exit(1)`. C linkage.

`int endianity()`
: Index of the byte holding the least significant byte of an `int`: 0 on little-endian machines, nonzero (`sizeof(int)-1`) on big-endian ones. Used e.g. by `bitvector` to refuse big-endian platforms.

Errors detected by BLAS and LAPACK (`xerbla_`), ATLAS (`ATL_xerbla`) and CBLAS (`cblas_xerbla`, `cblas_errprn`) are redirected to `laerror` by replacement handlers defined in `laerror.cc`. The CBLAS ones are omitted when the library is configured without CBLAS (`NONCBLAS`), `cblas_xerbla` also when `AVOID_DUPLICATE_CBLAS_XERBLA` is defined (for CBLAS implementations providing their own).

Typical use at the start of `main()` (`t.cc`):

```
sigtraceback(SIGSEGV,1);
sigtraceback(SIGABRT,1);
sigtraceback(SIGBUS,1);
sigtraceback(SIGFPE,1);
```

19.6 The single include file la.h

Header: `la.h`

`la.h` is meant as the single include file for the end user. It includes the headers of all classes and modules (traits, error handling, `AuxStorage`, vectors, dense, symmetric and sparse matrices, the non-class functions, iterative solvers, `DIIS`, matrix exponential, four-index transformation, permutations, polynomials, continued fractions, tensors, optimization, numbers, bitvector, Vec3/Mat3, quaternions, `miscfunc.h` and the version declaration), and `la_random.h` indirectly. It does *not* include `simple.h`, `graph.h`, `reg.h` and `regsurf.h`, which have to be included explicitly when needed.

`la.h` puts `using namespace LA;` into the global namespace and defines the abbreviations

```
typedef NRMat<int> NRIMat;      typedef NRVec<int> NRIVec;
typedef NRMat<double> NRDMat;   typedef NRVec<double> NRDVec;
typedef NRMat<std::complex<double> > NRCMat;
typedef NRVec<std::complex<double> > NRCVec;
```

Code which wants to avoid the global `using` directive can include the individual headers instead.

19.7 Version information

The library contains the string `extern const char LA::version[]` (declared in `LA_version.h`, included by `la.h`). Its definition `version.cc` is regenerated at every `make` by the script `get_git_version` and contains the output of `git describe`, the HEAD commit hash and date, the compilation date, host and user name. A program can print it to record which library build produced its results:

```
cout <<"LA version = "<<LA::version<<endl;
```

19.8 Number-theoretic utilities

Header: `numbers.h, numbers.cc`

Elementary number theory with naive algorithms, suitable for small numbers (trial division). All functions are templates in the integer type `N` but are instantiated only for `uint64_t` (the inline ones `isprime`, `isoddprime`, `eulerphi(const N&)` therefore also link only for `uint64_t`).

`N primefactor(const N &x, const N &last=0)`
: Smallest prime factor of x which is $\geq$`last` (assuming x has no smaller odd factors); returns x for $x \leq 2$ and for primes.

`bool isprime(const N &x); bool isoddprime(const N &x)`
: x prime; x odd prime.

`class FACTORIZATION<N> : public std::list<std::pair<N,N>`
: List of (prime, exponent) pairs in increasing order.

`FACTORIZATION<N> factorization(const N &x)`
: Prime factorization (empty for $x \leq 1$).

`N nextprime(N x)`
: Smallest prime $> x$ (2 for $x < 2$).

`std::ostream &operator<<(std::ostream &s, const FACTORIZATION<N> &x)`
: Prints `(p,e) (p,e) ...`.

`N eulerphi(const FACTORIZATION<N> &f); N eulerphi(const N &x)`
: Euler's totient function.

`N pow(const N &x, N i)`
: x^i by repeated squaring (overflow not checked).

`N multmod(N x, N y, const N &m); N powmod(const N &x, N i, const N &m)`
: $xy \bmod m$ by shift-and-add (avoids overflow of xy up to about $m < 2^{63}$), $x^i \bmod m$.

Example: `uint64_t n=360; cout « factorization(n) « eulerphi(n);` prints `(2,3) (3,2) (5,1) 96`.

19.9 Graph algorithms

Header: `graph.h, graph.cc`

A small collection of graph algorithms. There is no graph class: the functions are templates in the adjacency matrix type `G`, which must provide `nrows()`, an element access `operator()(i,j)` convertible to `bool`, a copy constructor/assignment and `submatrix(const NRVec<int>&)` returning a `G`. They are instantiated only for `G = NRSMat<char>` (undirected graphs, chapter 6). Vertices are numbered from 0; diagonal elements are ignored.

`NRVec<int> neighbors(const G &adjacency, const int v); NRVec<int> nonneighbors(const G &adjacency, const int v)`
: Sorted lists of vertices adjacent / not adjacent to v (excluding v).

`NRVec<int> findclique(const G &adjacency, const int v=0)`
: A large clique found by the recursive Ramsey heuristic of Boppana and Halldórsson [42] started at vertex v; returns the sorted vertex list (empty for an empty graph). Exponential worst case in recursion; intended for small graphs.

`NRVec<int> complement(const G &adjacency, const NRVec<int> &vertexlist)`
: Vertices not contained in `vertexlist`.

`NRVec<int> cliquecover(const G &adjacency, int *ncliques=NULL)`
: Greedy clique cover by repeated `findclique` on the remaining subgraph; returns for each vertex the number of its clique (counted from 1) and optionally the number of cliques.

```
NRSMat<char> adj;
cin >> adj;
NRVec<int> clique = findclique(adj);
NRVec<int> cover = cliquecover(adj);
NRPerm<int> p(cover.size());
cover.sort(0,p,true);                    //permutation grouping
    the cliques
NRSMat<char> adjperm = adj.permuted(p); //block structure
```

Bibliography

[1] E. R. Davidson, J. Comput. Phys. **17**, 87 (1975).

[2] P. Pulay, Chem. Phys. Lett. **73**, 393 (1980).

[3] P. Pulay, J. Comput. Chem. **3**, 556 (1982).

[4] J. Nocedal, S. J. Wright, *Numerical Optimization*, 2nd ed., Springer, New York 2006.

[5] A. R. Conn, N. I. M. Gould, Ph. L. Toint, *Trust-Region Methods*, SIAM, Philadelphia 2000.

[6] R. P. Brent, *Algorithms for Minimization without Derivatives*, Prentice-Hall, Englewood Cliffs 1973.

[7] P. Császár, P. Pulay, J. Mol. Struct. **114**, 31 (1984).

[8] Ö. Farkas, H. B. Schlegel, Phys. Chem. Chem. Phys. **4**, 11 (2002).

[9] W. W. Hager, H. Zhang, SIAM J. Optim. **16**, 170 (2005).

[10] J. M. Bofill, J. Comput. Chem. **15**, 1 (1994).

[11] J. J. Moré, D. C. Sorensen, SIAM J. Sci. Stat. Comput. **4**, 553 (1983).

[12] T. Steihaug, SIAM J. Numer. Anal. **20**, 626 (1983).

[13] A. Banerjee, N. Adams, J. Simons, R. Shepard, J. Phys. Chem. **89**, 52 (1985).

[14] J. Baker, J. Comput. Chem. **7**, 385 (1986).

[15] K. Madsen, H. B. Nielsen, O. Tingleff, *Methods for Non-Linear Least Squares Problems*, 2nd ed., IMM, Technical University of Denmark 2004.

[16] J. C. Lagarias, J. A. Reeds, M. H. Wright, P. E. Wright, SIAM J. Optim. **9**, 112 (1998).

[17] F. Gao, L. Han, Comput. Optim. Appl. **51**, 259 (2012).

[18] M. J. D. Powell, Comput. J. **7**, 155 (1964).

[19] M. J. D. Powell, in *Advances in Optimization and Numerical Analysis*, eds. S. Gomez, J.-P. Hennart, Kluwer, Dordrecht 1994, p. 51.

[20] M. J. D. Powell, Acta Numerica **7**, 287 (1998).

[21] M. J. D. Powell, *The NEWUOA software for unconstrained optimization without derivatives*, Report DAMTP 2004/NA08, University of Cambridge 2004.

[22] M. Ostaszewski, E. Grant, M. Benedetti, Quantum **5**, 391 (2021).

[23] K. M. Nakanishi, K. Fujii, S. Todo, Phys. Rev. Research **2**, 043158 (2020).

[24] D. Wierichs, J. Izaac, C. Wang, C. Y.-Y. Lin, Quantum **6**, 677 (2022).

[25] K. Mitarai, M. Negoro, M. Kitagawa, K. Fujii, Phys. Rev. A **98**, 032309 (2018).

[26] M. Schuld, V. Bergholm, C. Gogolin, J. Izaac, N. Killoran, Phys. Rev. A **99**, 032331 (2019).

[27] J. Stokes, J. Izaac, N. Killoran, G. Carleo, Quantum **4**, 269 (2020).

[28] S. Amari, Neural Comput. **10**, 251 (1998).

[29] I. Sutskever, J. Martens, G. Dahl, G. Hinton, in *Proceedings of the 30th International Conference on Machine Learning (ICML)*, 2013.

[30] D. P. Kingma, J. Ba, in *International Conference on Learning Representations (ICLR)*, 2015.

[31] S. J. Reddi, S. Kale, S. Kumar, in *International Conference on Learning Representations (ICLR)*, 2018.

[32] I. Loshchilov, F. Hutter, in *International Conference on Learning Representations (ICLR)*, 2019.

[33] J. C. Spall, IEEE Trans. Autom. Control **37**, 332 (1992).

[34] J. C. Spall, IEEE Trans. Aerosp. Electron. Syst. **34**, 817 (1998).

[35] K. Müller, L. D. Brown, Theor. Chim. Acta **53**, 75 (1979).

[36] D. E. Knuth, *The Art of Computer Programming, Vol. 4A: Combinatorial Algorithms, Part 1*, Addison-Wesley, 2011 (section 7.2.1.2, Generating all permutations).

[37] M. Hamermesh, *Group Theory and Its Application to Physical Problems*, Addison-Wesley, 1962.

[38] M. Aigner, *Combinatorial Theory*, Springer, 1979.

[39] M. Beeler, R. W. Gosper, R. Schroeppel, *HAKMEM*, MIT AI Memo 239, 1972 (item 101, continued fraction arithmetic).

[40] A. McAdams, A. Selle, R. Tamstorf, J. Teran, E. Sifakis, Computing the singular value decomposition of 3×3 matrices with minimal branching and elementary floating point operations, Technical Report 1690, University of Wisconsin–Madison, 2011.

[41] G. Marsaglia, Choosing a point from the surface of a sphere, Ann. Math. Stat. 43, 645 (1972).

[42] R. Boppana, M. M. Halldórsson, Approximating maximum independent sets by excluding subgraphs, BIT 32, 180 (1992).