Scientific Computing

M. Caliari

a.y. 2017/2018

Contents

Chapter 1

Numerical linear algebra

1.1 BLAS

The BLAS[1] (Basic Linear Algebra Subprograms) are routines that provide standard building blocks for performing basic vector and matrix operations. The Level 1 BLAS perform scalar, vector and vector-vector operations, the Level 2 BLAS perform matrix-vector operations, and the Level 3 BLAS perform matrix-matrix operations. Because the BLAS are efficient, portable, and widely available, they are commonly used in the development of high quality linear algebra software, LAPACK for example. They are written in Fortran77. Fortran77 does not allow dynamic allocation of the memory and arguments are passed by reference. Therefore

```
      PROGRAM main
c     compile with
c     gfortran demo.f -o demo
c     run with
c     ./demo
      integer maxn
      parameter (maxn=100)
      double precision a(maxn,maxn)
      integer i,j,n
      n = 10
      do 10 i = 1,n
         do 20 j = 1,n
            a(i,j) = 1.0d0*i-j
 20      continue
 10   continue
```

[1] http://www.netlib.org/blas

```
      call disp(a,maxn,n)
      stop
      end

      SUBROUTINE disp(a,lda,n)
      integer lda,n
      double precision a(lda,n)
      integer i,j
      do 10 i = 1,n
         write(6,'(10(1x,e8.2))') (a(i,j),j=1,n)
 10   continue
      return
      end
```

The subroutine for general g matrix-matrix multiplication in double precision d is dgemm:

```
 *  =========== DOCUMENTATION ===========
 *
 * Online html documentation available at
 *            http://www.netlib.org/lapack/explore-html/
 *
 *  Definition:
 *  ===========
 *
 *       SUBROUTINE DGEMM(TRANSA,TRANSB,M,N,K,ALPHA,A,LDA,B,LDB,BETA,C,LDC)
 *> \par Purpose:
 *  =============
 *>
 *> \verbatim
 *>
 *> DGEMM  performs one of the matrix-matrix operations
 *>
 *>    C := alpha*op( A )*op( B ) + beta*C,
 *>
 *> where  op( X ) is one of
 *>
 *>    op( X ) = X   or   op( X ) = X**T,
 *>
 *> alpha and beta are scalars, and A, B and C are matrices, with op( A )
 *> an m by k matrix,  op( B )  a  k by n matrix and  C an m by n matrix.
 *> \endverbatim
```

```
*
*  Arguments:
*  ==========
*
*> \param[in] TRANSA
*> \verbatim
*>          TRANSA is CHARACTER*1
*>           On entry, TRANSA specifies the form of op( A ) to be used in
*>           the matrix multiplication as follows:
*>
*>              TRANSA = 'N' or 'n',  op( A ) = A.
*>
*>              TRANSA = 'T' or 't',  op( A ) = A**T.
*>
*>              TRANSA = 'C' or 'c',  op( A ) = A**T.
*> \endverbatim
*>
*> \param[in] TRANSB
*> \verbatim
*>          TRANSB is CHARACTER*1
*>           On entry, TRANSB specifies the form of op( B ) to be used in
*>           the matrix multiplication as follows:
*>
*>              TRANSB = 'N' or 'n',  op( B ) = B.
*>
*>              TRANSB = 'T' or 't',  op( B ) = B**T.
*>
*>              TRANSB = 'C' or 'c',  op( B ) = B**T.
*> \endverbatim
*>
*> \param[in] M
*> \verbatim
*>          M is INTEGER
*>           On entry,  M  specifies  the number  of rows  of the  matrix
*>           op( A )  and of the  matrix  C.  M  must  be at least  zero.
*> \endverbatim
*>
*> \param[in] N
*> \verbatim
*>          N is INTEGER
*>           On entry,  N  specifies the number  of columns of the matrix
*>           op( B ) and the number of columns of the matrix C. N must be
*>           at least zero.
*> \endverbatim
*>
*> \param[in] K
*> \verbatim
*>          K is INTEGER
*>           On entry,  K  specifies  the number of columns of the matrix
*>           op( A ) and the number of rows of the matrix op( B ). K must
*>           be at least  zero.
*> \endverbatim
*>
*> \param[in] ALPHA
*> \verbatim
*>          ALPHA is DOUBLE PRECISION.
*>           On entry, ALPHA specifies the scalar alpha.
*> \endverbatim
*>
*> \param[in] A
*> \verbatim
*>          A is DOUBLE PRECISION array, dimension ( LDA, ka ), where ka is
*>           k  when  TRANSA = 'N' or 'n',  and is  m  otherwise.
*>           Before entry with  TRANSA = 'N' or 'n',  the leading  m by k
*>           part of the array  A  must contain the matrix  A,  otherwise
*>           the leading  k by m  part of the array  A  must contain  the
*>           matrix A.
*> \endverbatim
*>
*> \param[in] LDA
*> \verbatim
*>          LDA is INTEGER
*>           On entry, LDA specifies the first dimension of A as declared
*>           in the calling (sub) program. When  TRANSA = 'N' or 'n' then
*>           LDA must be at least  max( 1, m ), otherwise  LDA must be at
*>           least  max( 1, k ).
*> \endverbatim
*>
*> \param[in] B
*> \verbatim
*>          B is DOUBLE PRECISION array, dimension ( LDB, kb ), where kb is
*>           n  when  TRANSB = 'N' or 'n',  and is  k  otherwise.
*>           Before entry with  TRANSB = ’N’ or ’n’,  the leading  k by n
*>           part of the array  B  must contain the matrix  B,  otherwise
*>           the leading  n by k  part of the array  B  must contain  the
*>           matrix B.
*> \endverbatim
*>
*> \param[in] LDB
*> \verbatim
*>          LDB is INTEGER
*>           On entry, LDB specifies the first dimension of B as declared
*>           in the calling (sub) program. When  TRANSB = ’N’ or ’n’ then
*>           LDB must be at least  max( 1, k ), otherwise  LDB must be at
*>           least  max( 1, n ).
*> \endverbatim
*>
*> \param[in] BETA
*> \verbatim
*>          BETA is DOUBLE PRECISION.
*>           On entry,  BETA  specifies the scalar  beta.  When  BETA  is
*>           supplied as zero then C need not be set on input.
*> \endverbatim
*>
*> \param[in,out] C
*> \verbatim
*>          C is DOUBLE PRECISION array, dimension ( LDC, N )
*>           Before entry, the leading  m by n  part of the array  C must
*>           contain the matrix  C,  except when  beta  is zero, in which
*>           case C need not be set on entry.
*>           On exit, the array  C  is overwritten by the  m by n  matrix
*>           ( alpha*op( A )*op( B ) + beta*C ).
*> \endverbatim
*>
*> \param[in] LDC
*> \verbatim
*>          LDC is INTEGER
*>           On entry, LDC specifies the first dimension of C as declared
*>           in  the  calling  (sub)  program.   LDC  must  be  at  least
*>           max( 1, m ).
*> \endverbatim
*
*  Authors:
```

```
*  ========
*
*> \author Univ. of Tennessee
*> \author Univ. of California Berkeley
*> \author Univ. of Colorado Denver
*> \author NAG Ltd.
*
*> \date December 2016
*
*> \ingroup double_blas_level3
*
*> \par Further Details:
*  =====================
*>
*> \verbatim
*>
*>  Level 3 Blas routine.
*>
*>  -- Written on 8-February-1989.
*>     Jack Dongarra, Argonne National Laboratory.
*>     Iain Duff, AERE Harwell.
*>     Jeremy Du Croz, Numerical Algorithms Group Ltd.
*>     Sven Hammarling, Numerical Algorithms Group Ltd.
*> \endverbatim
*>
*  =====================================================================
      SUBROUTINE dgemm(TRANSA,TRANSB,M,N,K,ALPHA,A,LDA,B,LDB,BETA,C,LDC)
```

The BLAS collection is also known as Reference BLAS. Optimized versions are available: AMD Core Math Library (ACML, dismissed), Intel Math Kernel Library (Intel MKL), Automatically Tuned Linear Algebra Software (ATLAS, free), OpenBLAS (free). They are multithreaded. All high level languages (MATLAB, GNU Octave, Scilab, FreeFem++, ...) use optimized BLAS.

1.2 LAPACK

LAPACK[2] is written in Fortran 90 and provides routines for solving systems of simultaneous linear equations, least-squares solutions of linear systems of

[2] http://www.netlib.org/lapack

equations, eigenvalue problems, and singular value problems. The associated matrix factorizations (LU, Cholesky, QR, SVD, Schur, generalized Schur) are also provided, as are related computations such as reordering of the Schur factorizations and estimating condition numbers. Dense and banded matrices are handled, but not general sparse matrices.

```
*  =========== DOCUMENTATION ===========
*
* Online html documentation available at
*            http://www.netlib.org/lapack/explore-html/
*
*  Definition:
*  ===========
*
*       SUBROUTINE DGETRF( M, N, A, LDA, IPIV, INFO )
*
*       .. Scalar Arguments ..
*       INTEGER            INFO, LDA, M, N
*       ..
*       .. Array Arguments ..
*       INTEGER            IPIV( * )
*       DOUBLE PRECISION   A( LDA, * )
*       ..
*
*
*> \par Purpose:
*  =============
*>
*> \verbatim
*>
*> DGETRF computes an LU factorization of a general M-by-N matrix A
*> using partial pivoting with row interchanges.
*>
*> The factorization has the form
*>    A = P * L * U
*> where P is a permutation matrix, L is lower triangular with unit
*> diagonal elements (lower trapezoidal if m > n), and U is upper
*> triangular (upper trapezoidal if m < n).
*>
*> This is the right-looking Level 3 BLAS version of the algorithm.
*> \endverbatim
```

```
*
*  Arguments:
*  ==========
*
*> \param[in] M
*> \verbatim
*>          M is INTEGER
*>          The number of rows of the matrix A.  M >= 0.
*> \endverbatim
*>
*> \param[in] N
*> \verbatim
*>          N is INTEGER
*>          The number of columns of the matrix A.  N >= 0.
*> \endverbatim
*>
*> \param[in,out] A
*> \verbatim
*>          A is DOUBLE PRECISION array, dimension (LDA,N)
*>          On entry, the M-by-N matrix to be factored.
*>          On exit, the factors L and U from the factorization
*>          A = P*L*U; the unit diagonal elements of L are not stored.
*> \endverbatim
*>
*> \param[in] LDA
*> \verbatim
*>          LDA is INTEGER
*>          The leading dimension of the array A.  LDA >= max(1,M).
*> \endverbatim
*>
*> \param[out] IPIV
*> \verbatim
*>          IPIV is INTEGER array, dimension (min(M,N))
*>          The pivot indices; for 1 <= i <= min(M,N), row i of the
*>          matrix was interchanged with row IPIV(i).
*> \endverbatim
*>
*> \param[out] INFO
*> \verbatim
*>          INFO is INTEGER
*>          = 0:  successful exit
```

```
 *>          < 0:  if INFO = -i, the i-th argument had an illegal value
 *>          > 0:  if INFO = i, U(i,i) is exactly zero. The factorization
 *>                has been completed, but the factor U is exactly
 *>                singular, and division by zero will occur if it is used
 *>                to solve a system of equations.
 *> \endverbatim
 *
 *  Authors:
 *  ========
 *
 *> \author Univ. of Tennessee
 *> \author Univ. of California Berkeley
 *> \author Univ. of Colorado Denver
 *> \author NAG Ltd.
 *
 *> \date December 2016
 *
 *> \ingroup doubleGEcomputational
 *
 *  =====================================================================
       SUBROUTINE dgetrf( M, N, A, LDA, IPIV, INFO )
```

All high level languages use LAPACK.

```
*> \brief \b DPOTRF
 *
 *  =========== DOCUMENTATION ===========
 *
 * Online html documentation available at
 *            http://www.netlib.org/lapack/explore-html/
 *
 *  Definition:
 *  ===========
 *
 *       SUBROUTINE DPOTRF( UPLO, N, A, LDA, INFO )
 *
 *       .. Scalar Arguments ..
 *       CHARACTER          UPLO
 *       INTEGER            INFO, LDA, N
 *       ..
 *       .. Array Arguments ..
 *       DOUBLE PRECISION   A( LDA, * )
 *       ..
 *
 *
 *> \par Purpose:
 *  =============
 *>
 *> \verbatim
 *>
 *> DPOTRF computes the Cholesky factorization of a real symmetric
 *> positive definite matrix A.
 *>
 *> The factorization has the form
 *>    A = U**T * U,  if UPLO = 'U', or
 *>    A = L  * L**T,  if UPLO = 'L',
 *> where U is an upper triangular matrix and L is lower triangular.
 *>
 *> This is the block version of the algorithm, calling Level 3 BLAS.
 *> \endverbatim
 *
 *  Arguments:
 *  ==========
 *
 *> \param[in] UPLO
 *> \verbatim
 *>          UPLO is CHARACTER*1
 *>          = 'U':  Upper triangle of A is stored;
 *>          = 'L':  Lower triangle of A is stored.
 *> \endverbatim
 *>
 *> \param[in] N
 *> \verbatim
 *>          N is INTEGER
 *>          The order of the matrix A.  N >= 0.
 *> \endverbatim
 *>
 *> \param[in,out] A
 *> \verbatim
 *>          A is DOUBLE PRECISION array, dimension (LDA,N)
 *>          On entry, the symmetric matrix A.  If UPLO = 'U', the leading
 *>          N-by-N upper triangular part of A contains the upper
 *>          triangular part of the matrix A, and the strictly lower
 *>          triangular part of A is not referenced.  If UPLO = 'L', the
 *>          leading N-by-N lower triangular part of A contains the lower
 *>          triangular part of the matrix A, and the strictly upper
 *>          triangular part of A is not referenced.
 *>
 *>          On exit, if INFO = 0, the factor U or L from the Cholesky
 *>          factorization A = U**T*U or A = L*L**T.
 *> \endverbatim
 *>
 *> \param[in] LDA
 *> \verbatim
 *>          LDA is INTEGER
 *>          The leading dimension of the array A.  LDA >= max(1,N).
 *> \endverbatim
 *>
 *> \param[out] INFO
 *> \verbatim
 *>          INFO is INTEGER
 *>          = 0:  successful exit
 *>          < 0:  if INFO = -i, the i-th argument had an illegal value
 *>          > 0:  if INFO = i, the leading minor of order i is not
 *>                positive definite, and the factorization could not be
 *>                completed.
 *> \endverbatim
 *
 *  Authors:
 *  ========
 *
 *> \author Univ. of Tennessee
 *> \author Univ. of California Berkeley
 *> \author Univ. of Colorado Denver
 *> \author NAG Ltd.
 *
 *> \date December 2016
 *
 *> \ingroup doublePOcomputational
 *
 *  =====================================================================
       SUBROUTINE dpotrf( UPLO, N, A, LDA, INFO )
```

1.3 SuiteSparse

SuiteSparse[3] is a suite of sparse matrix algorithms, including:

- UMFPACK: multifrontal LU factorization. Appears as LU and `x=A\b` in MATLAB.
- CHOLMOD: supernodal Cholesky. Appears as CHOL and `x=A\b` in MATLAB.
- SPQR: multifrontal QR. Appears as QR and `x=A\b` in MATLAB.
- and many other packages.

```
[L,U,P,Q,R] = lu (S)
```

1.3.1 Sparse matrices

Let A be a square matrix of dimension n with m elements different from zero. We say it is *sparse* if $m = \mathcal{O}(n)$, instead of $m = \mathcal{O}(n^2)$. The Compressed Column Storage (CCS) works in the following way. There are three arrays. The first, `data`, of length m contains all elements different from zero, sorted first by column and then by row. The second, `ridx`, of length m contains the row indexes of the elements of `data`. The third, `cidx`, of length $n+1$, whose i-th element $(i < n+1)$ is the position inside `data` of the first element of column i and the $(n+1)$-th element is the total number of elements different from zero, plus one. For instance, to the matrix

$$A = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 2 & 3 & 0 \\ 4 & 0 & 5 & 6 \\ 0 & 0 & 0 & 7 \end{pmatrix}$$

it corresponds the three vectors

$$\begin{aligned} \texttt{data} &= [1, 4, 2, 3, 5, 6, 7] \\ \texttt{ridx} &= [1, 3, 2, 2, 3, 3, 4] \\ \texttt{cidx} &= [1, 3, 4, 6, 8] \end{aligned}$$

Sometimes, the arrays `ridx` and `cidx` have elements shifted by -1. The number of rows and columns of the matrix has to be given. The number of elements different from zero is given by the last element in `cidx` (minus one). High level languages use this format.

[3] http://faculty.cse.tamu.edu/davis/suitesparse.html

1.4 Errors in solving linear systems

Suppose that a numerical method finds $\tilde{x}$ such that $r = b - A\tilde{x}$, the residual, is small. Then, we have

$$\frac{\|e\|}{\|x\|} = \frac{\|\tilde{x} - x\|}{\|x\|} = \frac{\|A^{-1}((b-r)-b)\|}{\|x\|} = \frac{\|A^{-1}r\|}{\|x\|} \leq \frac{\|A^{-1}\|\,\|r\|}{\|x\|} \leq$$
$$\leq \frac{\|A^{-1}\|\,\|A\|\,\|r\|}{\|b\|} \leq \operatorname{cond}(A)\frac{\|r\|}{\|b\|}$$

1.5 Exercises and projects

1. Implement the function `function w = matvec (data, ridx, cidx, v)` and `w = matvect (data, ridx, cidx, v)` which implement the sparse (transposed) matrix-vector products for square matrices.

2. Project: implement the function `[L, U, P] = lu (data, ridx, cidx)` for LU factorization of a sparse matrix.

Chapter 2

Large sparse linear systems

Besides methods base on the SuiteSparse library (which are direct methods), it is possible to use iterative methods [7]. By iterative methods I mean Krylov methods and not Jacobi, Gauss–Sidel, or SOR methods. It is difficult to say which class is better. The main advantage of iterative methods is that it is possible to specify a tolerance for the error. Moreover, usually the entries of the matrix A are not required, but only the action of A (and possibly of A^T) to a vector.

2.1 SPD systems

The most used method for symmetric (Hermitian) positive definite systems is the Conjugate Gradient method. The idea is that, instead of solving the linear system $Ax = b$, one tries to minimize the quadratic functional

$$J(x) = \frac{1}{2}x^T Ax - x^T b$$

Starting from an initial guess x_0, iterates are given by

$$x_m = x_{m-1} + \alpha_{m-1}p_{m-1}$$

where p_{m-1} is a “descent” direction. Given it, it is easy to determine α_{m-1}: in fact

$$J(x_{m-1}+\alpha p_{m-1}) = \frac{1}{2}x_{m-1}^T Ax_{m-1} - x_{m-1}^T b + \alpha(p_{m-1}^T Ax_{m-1} - p_{m-1}^T b) + \frac{1}{2}\alpha^2 p_{m-1}^T Ap_{m-1}$$

and the minimum, in α, is taken by

$$\alpha_{m-1} = \frac{p_{m-1}^T r_{m-1}}{p_{m-1}^T Ap_{m-1}}$$

In the conjugate gradient method such a formula can be replaced by

$$\alpha_{m-1} = \frac{r_{m-1}^T r_{m-1}}{p_{m-1}^T A p_{m-1}}. \tag{2.1}$$

It is clear that such a quantity has to be positive. If not, the method can fail to converge and, in any case, should issue a warning about the non-positive definitness of the system. Matlab fails with this example

```
N = 10;
A = diag (1:N) + 1i * 1e-04; % clearly not SPD
b = ones (N, 1);
[x,flag] = pcg (A, b, []);
```

The problem seems to be the comparison operator among complex numbers.

We have therefore the following implementation of the method, known as *Hestenes–Stiefel*

```
• $x_0$ given, $p_0 = r_0 = b - Ax_0$
• FOR $m = 1, 2, \ldots$ UNTIL $\|r_{m-1}\|_2 \leq \mathrm{tol} \cdot \|b\|_2$
      $w_{m-1} = Ap_{m-1}$
      $\alpha_{m-1} = \frac{r_{m-1}^T r_{m-1}}{p_{m-1}^T w_{m-1}}$
      $x_m = x_{m-1} + \alpha_{m-1} p_{m-1}$
      $r_m = r_{m-1} - \alpha_{m-1} w_{m-1}$
      $\beta_m = \frac{r_m^T r_m}{r_{m-1}^T r_{m-1}}$
      $p_m = r_m + \beta_m p_{m-1}$
  END
```

2.1.1 NLCG

For more general functionals $J(x)$ we can consider the following *nonlinear conjugate gradient* method.

```
• $x_0$ given, $d_0 = g_0 = -\nabla J(x_0)$
• FOR $m = 1, 2, \ldots$ UNTIL $\|d_{m-1}\|_2 \leq \mathrm{tol} \cdot \|d_0\|_2$
      $\alpha_{m-1} = \arg\min_{\alpha} J(x_{m-1} + \alpha d_{m-1})$
```

```
        $x_m = x_{m-1} + \alpha_{m-1} d_{m-1}$
        $g_m = -\nabla J(x_m)$
        $\beta_m = \frac{g_m^T \nabla J(x_m)}{g_{m-1}^T \nabla J(x_{m-1})}$
        $d_m = g_m + \beta_m d_{m-1}$
    END
```

It is in general not necessary to compute exactly α_{m-1}. In this case we speak about *inexact linesearch*. It can be performed, for instance, by few steps of golden search of $g(\alpha) = J(x_{m-1} + \alpha d_{m-1})$. Or it is possible to look for the zero of the univariate function $d_{m-1}^T \nabla J(x_{m-1} + \alpha d_{m-1})$. The choice of β_m corresponds to Fletcher–Reeves. NLCG is available in FreeFem++.

2.1.2 Preconditioners

The idea is to change

$$Ax = b$$

into

$$P^{-1}Ax = P^{-1}b$$

in such a way that $P^{-1}A$ is better conditioned than A and the conjugate gradient method faster (left preconditioning). The main problem for the CG algorithm is that even if P is SPD, $P^{-1}A$ is not SPD. We can therefore factorize P into $P = R^T R$ and consider the linear system

$$P^{-1}AR^{-1}y = P^{-1}b \Leftrightarrow R^{-T}AR^{-1}y = R^{-T}b, \quad R^{-1}\bar{y} = x$$

Now, $\tilde{A} = R^{-T}AR^{-1}$ is SPD and we can solve the system $\tilde{A}\bar{y} = \tilde{b}$, $\tilde{b} = R^{-T}b$, with the CG method. Setting $\tilde{x}_m = Rx_m$, we have $\tilde{r}_m = \tilde{b} - \tilde{A}\tilde{x}_m = R^{-T}b - R^{-T}Ax_m = R^{-T}r_m$. This is called split preconditioning. It is possible then to arrange the CG algorithm for $\tilde{A}$, $\tilde{x}_0$ and $\tilde{b}$ as

```
• $x_0$ given, $r_0 = b - Ax_0$, $Pz_0 = r_0$, $p_0 = z_0$
• FOR $m = 1, 2, \ldots$ UNTIL $\|r_m\|_2 \leq \text{tol} \cdot \|b\|_2$
        $w_{m-1} = Ap_{m-1}$
        $\alpha_{m-1} = \frac{z_{m-1}^T r_{m-1}}{p_{m-1}^T w_{m-1}}$
        $x_m = x_{m-1} + \alpha_{m-1} p_{m-1}$
        $r_m = r_{m-1} - \alpha_{m-1} w_{m-1}$
        $P z_m = r_m$
        $\beta_m = \frac{z_m^T r_m}{z_{m-1}^T r_{m-1}}$
        $p_m = z_m + \beta_m p_{m-1}$
    END
```

The directions p_m are still A conjugate directions (with $Pp_0 = r_0$). This algorithm requires the solution of the linear system $Pz_m = r_m$ at each iteration. From one side P should be as close as possibile to A, from the other it should be easy to "invert". The simplest choice is $P = \mathrm{diag}(A)$. It is called Jacobi preconditioner. This preconditioner is the default in FreeFem++. If P is not diagonal, usually it is factorized once and for all into $P = R^T R$, R the triangular Cholesky factor, in such a way that z_m can be recovered by two simple triangular linear systems. A possibile choice is the incomplete Cholesky factorization of A. That is, $P = \tilde{R}^T \tilde{R} \approx A$ where

$$\begin{cases} (A - \tilde{R}^T \tilde{R})_{ij} = 0 & \text{if } a_{ij} \neq 0 \\ \tilde{r}_{ij} = 0 & \text{if } a_{ij} = 0 \end{cases}$$

The preconditioned Conjugate Gradient method does not explicitely require the entries of P, but only the action of P^{-1} (which can be $R^{-1}R^{-T}$) to a vector z_m (that is the solution of a linear system with matrix P). The definition of α_{m-1}

$$\alpha_{m-1} = \frac{r_{m-1}^T P r_{m-1}}{p_{m-1}^T A p_{m-1}} \tag{2.2}$$

now requires that both the numerator and the denomination be positive. The usual way to call this method in Matlab is

```
pcg (A, x, tol, maxit, M) % a preconditioner M easy to invert
pcg (A, x, tol, maxit, M1, M2) % a preconditioner M in the form M1 * M2
```

From Matlab documentation, it seems that a left preconditioner is used, but it is not true.

It is possible to apply a preconditioner to the nonlinear conjugate gradient method, too. In fact, suppose that $-\nabla J(x_m)$ is of type $b - A(x_m)$ for some function $A\colon \mathbb{R}^n \to \mathbb{R}^n$. It is then possible to approximate $A(x)$ with $A_m x$ and use the matrix A_m (which is $\nabla(A_m x)$) as preconditioner and compute g_m as $-A_m^{-1}\nabla J(x_m)$. Instead of the matrix A_m, in order to reduce the cost of inverting it, can be replaced by $A_{p\lfloor m/p \rfloor}$, for $p > 1$ integer.

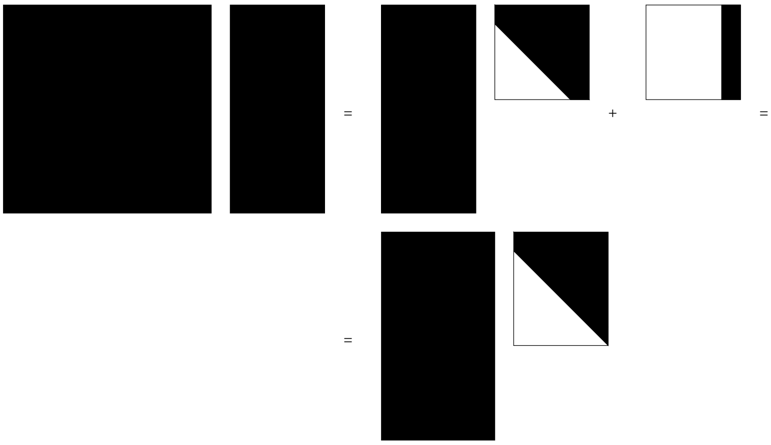
2.2 Unsymmetric systems

2.2.1 Arnoldi factorization

It is possibile to factorize (by Grahm-Schmidt orthogonalization, e.g.) a matrix $A \in \mathbb{R}^{n\times n}$ into

$$\begin{aligned} AV_m &= V_m H_m + w_m e_m^T = \\ &= V_{m+1}\bar{H}_m \end{aligned} \tag{2.3}$$

where the first column of V_m is v_1 given, $V_m \in \mathbb{R}^{n\times m}$ such that $V_m^T V_m = I_m$



and $V_m^T w_m = 0$, H_m is superior Hessenberg and $\bar{H}_m$ is a $(m+1)\times m$ matrix. (see [7, § 6.3]). The cost is $\mathcal{O}(m^2)$. From the relation

$$V_m^T A V_m = H_m$$

we get that if A is symmetric, so is H_m and since it is Hessenberg, it is in fact trigiadonal. Therefore, the Gram–Schmidt procedure is *short* and the cost is $\mathcal{O}(m)$.

2.2.2 GMRES and BiCGStab

When we want to solve a lienar system, we start with an initial guess x_0 and set $v_1 = r_0/\beta$, $\beta = \|r_0\|_2$. Then, we compute the Arnoldi factorization and we can look for an approximate solution of type

$$x = x_0 + V_m y.$$

We define

$$J(y) = \|b - Ax\|_2 = \|b - A(x_0 + V_m y)\|_2.$$

Moreover, we have

$$\begin{aligned} b - Ax = b - A(x_0 + V_m y) = \\ = r_0 - AV_m y = \\ \beta v_1 - V_{m+1}\bar{H}_m y = \\ = V_{m+1}(\beta e_1 - \bar{H}_m y). \end{aligned}$$

Therefore,

$$J(y) = \|\beta e_1 - \bar{H}_m y\|_2$$

The GMRES methods attempts to minimize J and compute

$$y_m = \arg \min_{y \in \mathbb{R}^m} \|\beta e_1 - \bar{H}_m y\|_2, \quad x_m = x_0 + V_m y_m.$$

The linear system $\bar{H}_m y = \beta e_1$ is rectangular (over-determined) and it is possible to find the *least squares* solution using the SVD decomposition. GMRES is available in FreeFem++.

Another popular method for general systems is BiCGStab. The main differences are:

- given the number of iterations m, the cost of GMRES is one matrix-vector product per iteration and m^2 vector operations, while the cost of BiCGStab is two matrix vector products per iteration (one with A and the other with A^T, even if A^T is not required) and m vector operations
- GMRES uses a left preconditioner, while BiCGStab a right preconditioner ($Ax = b \Leftrightarrow Ay = b$, $Px = y$, Matlab documentation is confused).

2.2.3 Preconditioners

Incomplete LU factorization.

2.3 Exercises and projects

1. Understand LSQR http://web.stanford.edu/group/SOL/software/lsqr/

Chapter 3

Eigenvalues for large sparse matrices

The computation of eigenvalues and eigenvectors for “small” full matrices in done in LAPACK with optimized implmentations of the QR algorithm. The QR algorithm is based on the QR factorization of a matrix. Any matrix can be split according to

$$A = QR$$

with Q orthogonal and R upper triangular. The QR algorithm for eigenvalues produces a sequence

$$A_0 = A = Q_0 R_0, \quad A_1 = R_0 Q_0 = Q_1 R_1, \quad A_2 = R_1 Q_1 = \ldots$$

Clearly

$$A_{k+1} = R_k Q_k = Q_k^T Q_k R_k Q_k = Q_k^T A_k Q_k$$

and thus all the matrices in the sequence are similar and share the same eigenvalues. Under some hypothesis, the limit of A_k for $k \to \infty$ is an upper triangular matrix whose eigenvalues are in the diagonal.

For large sparse matrices, several techniques are available.

3.1 ARPACK

3.1.1 Implicit restarted Arnoldi’s algorithm

Let us analyze the method under the popular ARPACK [6] package in Fortran77 for eigenvalue problems. It allows to compute “some” eigenvalues of large sparse matrices (such as the largest in magnitude, the smallest, ...). We start with an Arnoldi factorization ($m \ll n$)

$$V_m^T A V_m = H_m$$

If (θ, s) is an eigenpair for H_m, that is $H_m s = \theta s$, then

$$(v, Ax - \theta x) = 0, \quad \forall v \in \mathcal{K}$$

where $x = V_m s$ and $\mathcal{K}$ is the Krylov space spanned by the columns of V_m. In fact, v can be written as $V_m y$ and therefore

$$(V_m y, Ax - \theta x) = y^T V_m^T A V_m s - y^T V_m^T V_m s \theta = y^T (H_m s - \theta s) = 0$$

The couple (θ, x) is called Ritz pair and it is close to an eigenpair of A. In fact

$$\|Ax - \theta x\|_2 = \|(AV_m - V_m H_m)s\|_2 = \left|\beta_m e_m^T s\right|$$

where $\beta_m = \|w_m\|_2$. "In the Hermitian case, this estimate on the residual can be turned into a precise statement about the accuracy of the Ritz value θ as an approximation to the eigenvalue of A that is nearest to θ. However, an analogous statement in the non-Hermitian case is not possible without further information concerning nonnormality and defectiveness." [6, p. 64]. We can compute the eigenvalues of H_m, for instance by the QR method, and select an "unwanted" eigenvalue μ_m (which is an approximation of an eigenvalue μ of A). Then, we apply one iteration of *shifted* QR algorithm, that is

$$H_m - \mu_m I_m = Q_1 R_1, \quad H_m^+ = R_1 Q_1 + \mu_m I_m$$

Of course, Q_1 is Hessenberg and $Q_1 H_m^+ = H_m Q_1$. Now we right-multiply the Arnoldi factorization, in order to get

$$AV_m Q_1 = V_m H_m Q_1 + w_m e_m^T Q_1 \tag{3.1}$$

With few manipulations

$$\begin{aligned}
&AV_m Q_1 = V_m Q_1 H_m^+ + w_m e_m^T Q_1 \\
&AV_m Q_1 = (V_m Q_1)(R_1 Q_1 + \mu_m I_m) + w_m e_m^T Q_1 \\
&(A - \mu_m I_n) V_m Q_1 = (V_m Q_1)(R_1 Q_1) + w_m e_m^T Q_1 \\
&(A - \mu_m I_n) V_m = V_m Q_1 R_1 + w_m e_m^T
\end{aligned}$$

and by setting $V_m^+ = V_m Q_1$, we have that the first column of the last expression is

$$(A - \mu_m I_n) v_1 = V_m^+ R_1 e_1 = v_1^+ (e_1^T R_1 e_1)$$

that is, the first column of V_m^+ is a multiple of $(A - \mu_m I_n) v_1$. If v_1 was a linear combination of the eigenvectors x_j of A, then

$$v_1^+ \parallel (A - \mu_m I_n) v_1 = \sum_j (\alpha_j \lambda_j x_j - \alpha_j \mu_m x_j)$$

Since μ_m is close to a $\lambda_{\bar{j}}$, v_1^+ lacks the component parallel to $x_{\bar{j}}$. Relation (3.1) can be rewritten as

$$AV_m^+ = V_m^+ H_m^+ + w_m e_m^T Q_1$$

ans if we consider the first column, it is an Arnoldi factorization with a starting vector v_1^+ (which is of unitary norm) lacking the unwanted component. In practice, given the m eigenvalues of H_m, they are split into the k wanted and the $p = m - k$ unwanted and p shifted QR decompositions (with each of the unwanted eigenvalues) are performed. Then, the Arnoldi factorization is right-multiplied by $Q = Q_1 Q_2 \dots Q_p$ and the first k columns kept. This turns out to be an Arnoldi factorization. In fact

$$AV_m^+ I_{m,k} = AV_k^+,$$

where now $V_m^+ = V_m Q$, and

$$V_m^+ H_m^+ I_{m,k} = V_k^+ H_k^+ + (V_m^+ e_{k+1} h_{k+1,k}) e_k^T$$

and, since the Q_j are Hessenberg matrices, the last row of Q, that is $e_m^T Q$, has the first $k-1$ entries which are zero and then a value σ (and then something else). Therefore

$$w_m e_m^T Q I_{m,k} = w_m \sigma e_k^T$$

All together, the first k columns are

$$AV_k^+ = V_k^+ H_k^+ + w_k^+ e_k^T, \quad w_k^+ = (V_m^+ e_{k+1} h_{k+1,k} + w_m \sigma)$$

that is an Arnoldi factorization applied to an initial vector lacking the unwanted components. Then, the the factorization is continued up to m columns.

The easyest to compute eigenvalues with a Krylov methods are the largest in magnitute (as for the power method). Therefore, if some other eigevalues are desired, it is necessary to apply proper transformations. Let us consider the generalized problem

$$Ax = \lambda Mx$$

If we are interested into eigenvalues around σ, first we notice that

$$(A - \sigma M)x = (\lambda - \sigma)Mx \Rightarrow x = (\lambda - \sigma)(A - \sigma M)^{-1}Mx$$

from which

$$(A - \sigma M)^{-1}Mx = \nu x, \quad \nu = \frac{1}{\lambda - \sigma}$$

Therefore, if we apply the Krylov method (or the power method) to the operator $\mathrm{OP}^{-1}B = (A - \sigma M)^{-1}M$ we end up with the eigenvalues closer to σ. In order to do that, we need to be able to solve linear systems with $(A - \sigma M)$ and multiply vectors with M.

High level languages use ARPACK, although Matlab moved to a Krylov–Schur method [8] inside its `eigs` function. Another possible method is described here [2]. A gateway for the Jacobi–Davidson method is here[1]. Skew-symmetric matrices! show Matlab help

3.2 SVD decomposition

Let us start with the SVD decomposition. Given a matrix $A \in \mathbb{R}^{m\times n}$, it is $A = USV^T$, where $U \in \mathbb{R}^{m\times m}$ and $V \in \mathbb{R}^{n\times n}$ are orthogonal matrices and $S \in \mathbb{R}^{m\times n}$ is "diagonal" with elements σ_i, $i = 1, 2, \ldots, \min\{m, n\}$. The elements σ_i satisfy $\sigma_1 \geq \sigma_2 \geq, \ldots, \sigma_{\min\{m,n\}} \geq 0$ and r among them are strictly positive if $r = \text{rank}(A)$. The values σ_i are called "singular values" of A and do coincide with the square roots of the eigenvalues of A^TA. The SVD decomposition in MATLAB is `[U, S, V] = svd (A)` and it is based on LAPACK libraries.

3.2.1 SVD for sparse matrices

"Many problems in scientific computation require knowledge of a few of the largest or smallest singular values of a matrix and associated left and right singular vectors. These problems include the approximation of a large matrix by a matrix of low rank, the computation of the null space of a matrix, total least-squares prolems, as well as tracking of signals" [3]. A simple approach is the following: it is possible to compute few eigenvalues of

$$Z = \begin{bmatrix} 0 & A \\ A^T & 0 \end{bmatrix}$$

In fact, from

$$Z \begin{bmatrix} u \\ v \end{bmatrix} = \lambda \begin{bmatrix} u \\ v \end{bmatrix}$$

we get

$$Av = \lambda u,\ A^Tu = \lambda v \Rightarrow A^TAv = A^T\lambda u = \lambda^2 v.$$

Therefore, $|\lambda|$ is a singular value of A. Notice that Z is a symmetric matrix (all eigenvalues are real) and if $\lambda \in \sigma(Z)$, so is $-\lambda$ (associated to the

[1] `http://www.win.tue.nl/casa/research/scientificcomputing/topics/jd/`

eigenvector $[-u, v]^T$). Therefore, we have

$$\begin{bmatrix} 0 & A \\ A^T & 0 \end{bmatrix} = \begin{bmatrix} U & -U \\ V & V \end{bmatrix} \begin{bmatrix} S & 0 \\ 0 & -S \end{bmatrix} \begin{bmatrix} U^T & V^T \\ -U^T & V^T \end{bmatrix}$$

where if $\sigma \in S$, then $\sigma \geq 0$, from which $A = USV^T$. Since $UU^T + UU^T = I$ and $VV^T + VV^T = I$, $\sqrt{2}U$ and $\sqrt{2}V$ are orthogonal matrices.

Chapter 4

{N}FFT Matlab/GNU Octave friendly

4.1 1-dimensional transform

We consider, for N even,

$$\begin{aligned}\hat{u}(x) &= \sum_{j=-N/2}^{N/2-1} \hat{u}_{j+1+N/2} \frac{\mathrm{e}^{\mathrm{i}j2\pi(x-a)/(b-a)}}{\sqrt{b-a}} = \\ &= \sum_{j=1}^{N} \hat{u}_j \frac{\mathrm{e}^{\mathrm{i}(j-1-N/2)2\pi(x-a)/(b-a)}}{\sqrt{b-a}} = \sum_{j=1}^{N} \hat{u}_j \phi_j(x)\end{aligned} \tag{4.1}$$

where $\hat{u}_j$ is the approximation by trapezoidal quadrature rule of

$$\int_a^b u(x)\overline{\phi_j(x)}\mathrm{d}x$$

(we assume $u(x)$ periodic in $[a,b]$) that is

$$\begin{aligned}u_j = \int_a^b u(x)\overline{\phi_j(x)}\mathrm{d}x = \sqrt{b-a}\int_0^1 u(a+y(b-a))\mathrm{e}^{-\mathrm{i}(j-1)2\pi y}\mathrm{e}^{iN\pi y}\mathrm{d}y \approx \\ \approx \frac{\sqrt{b-a}}{N} \boxed{\sum_{n=1}^{N} \left(u(x_n)\mathrm{e}^{\mathrm{i}N\pi y_n}\right) \mathrm{e}^{-\mathrm{i}(j-1)2\pi y_n}} = \hat{u}_j\end{aligned}$$

where $y_n = (n-1)/N$ and $x_n = a + (b-a)y_n$. We say that the 0 frequency is in the center of spectrum. What is written in the box is the result of `fft`

`([`$u(x_1)\mathrm{e}^{\mathrm{i}N\pi y_1},\ldots,u(x_N)\mathrm{e}^{\mathrm{i}N\pi y_N}$`] )`. We consider now, for $1 \leq j \leq N/2$,

$$\texttt{fft ([}u(x_1),\ldots,u(x_N)\texttt{])}_j = \sum_{n=1}^{N} u(x_n)\mathrm{e}^{-\mathrm{i}(j-1)2\pi y_n} =$$

$$= \sum_{n=1}^{N} \left(u(x_n)\mathrm{e}^{\mathrm{i}N\pi y_n}\right)\mathrm{e}^{-\mathrm{i}(N/2+j-1)2\pi y_n} = \frac{N}{\sqrt{b-a}}\hat{u}_{N/2+j}$$

and, for $N/2 < j \leq N$,

$$\texttt{fft ([}u(x_1),\ldots,u(x_N)\texttt{])}_j = \sum_{n=1}^{N} u(x_n)\mathrm{e}^{-\mathrm{i}(j-1)2\pi y_n} =$$

$$= \sum_{n=1}^{N} \left(u(x_n)\mathrm{e}^{\mathrm{i}N\pi y_n}\right)\mathrm{e}^{-\mathrm{i}(j-N/2-1)2\pi y_n} = \frac{N}{\sqrt{b-a}}\hat{u}_{j-N/2}$$

taking into account that $\mathrm{e}^{iN2\pi y_n} = 1$. Therefore `uhat = fftshift (fft (u)) * sqrt (b - a) / N`. Then we have

$$\hat{\hat{v}}_n = \sum_{k=1}^{N} \hat{v}_k\phi_k(x_n) = \sum_{k=1}^{N} \hat{v}_k \frac{\mathrm{e}^{\mathrm{i}(k-1-N/2)2\pi(x_n-a)/(b-a)}}{\sqrt{b-a}} =$$

$$= \frac{N}{\sqrt{b-a}} \boxed{\frac{1}{N}\left(\sum_{k=1}^{N} \hat{v}_k \mathrm{e}^{\mathrm{i}(k-1)2\pi y_n}\right)} \mathrm{e}^{-\mathrm{i}N\pi y_n}$$

What written in the box is the result of `ifft (vhat)`. We observe that $\mathrm{e}^{-\mathrm{i}N\pi y_n} = (-1)^{n+1}$. We consider now

$$\sum_{k=1}^{N} \texttt{ifftshift ([}\hat{v}_1,\ldots,\hat{v}_N\texttt{])}_k \mathrm{e}^{\mathrm{i}(k-1)2\pi y_n} = \sum_{k=1}^{N/2} \hat{v}_{N/2+k}\mathrm{e}^{\mathrm{i}(k-1)2\pi y_n} + \sum_{k=N/2+1}^{N} \hat{v}_{k-N/2}\mathrm{e}^{\mathrm{i}(k-1)2\pi y_n} =$$

$$= \sum_{k=N/2+1}^{N} \hat{v}_k \mathrm{e}^{\mathrm{i}(k-N/2-1)2\pi y_n} + \sum_{k=1}^{N/2} \hat{v}_k \mathrm{e}^{\mathrm{i}(N/2+k-1)2\pi y_n} =$$

$$= (-1)^{n+1}\sum_{k=1}^{N/2} \hat{v}_k \mathrm{e}^{\mathrm{i}(k-1)2\pi y_n} + (-1)^{n+1}\sum_{k=N/2+1}^{N} \hat{v}_k \mathrm{e}^{\mathrm{i}(k-1)2\pi y_n} =$$

$$= \left(\sum_{k=1}^{N} \hat{v}_k \mathrm{e}^{\mathrm{i}(k-1)2\pi y_n}\right)\mathrm{e}^{-\mathrm{i}N\pi y_n}$$

Therefore, `vhathat=ifft (ifftshift (vhat)) * N / sqrt (b - a)`. It is not difficult to prove that

$$\hat{u}(x_n) = u(x_n)$$

that is, $\hat{u}(x)$ is an approximation of $u(x)$ which interpolates $u(x)$ at x_n, $n = 1, 2, \ldots, N$. The advantage of using the FFT algorithm (over a standard matrix-vector product) is that its cost is $\mathcal{O}(N \log N)$ instead of $\mathcal{O}(N^2)$. The use of the scaling factor `sqrt (b - a) / N` has the following purpose. With the definition (4.1), we have

$$\int_a^b |u(x)|^2 \, \mathrm{d}x = \sum_{j=-\infty}^{\infty} |u_j|^2$$

(Parseval's identity), withouth any scaling factor. When we replace the series by a sum, what happens in practice is

$$\int_a^b |u(x)|^2 \, \mathrm{d}x \approx \frac{b-a}{N} \sum_{n=1}^{N} |u(x_n)|^2 = \sum_{j=1}^{N} |\hat{u}_j|^2 .$$

The two main common mistakes when using FFT in Matlab are

- to forget to remove the last point from `x = linspace (a, b, N + 1)`
- to use the *transpose conjugate* operator `'` (single quote) to simply transpose a vector, while the operator `.'` was probably what needed.

4.1.1 Computation of $u'(x)$ by FFT

We have

$$u'(x) \approx \left(\sum_{j=1}^{N} \hat{u}_j \phi_j(x) \right)' = \sum_{j=1}^{N} \hat{u}_j \phi_j'(x) = \sum_{j=1}^{N} \hat{u}_j \lambda_j \phi_j(x)$$

where $\lambda_j = i(j - 1 - N/2)2\pi/(b - a)$. You can try the following Matlab code

```
a = -2;
b = 2;
N = 32;
x = linspace(a,b,N+1)';
x = x(1:N);
u = 1./(sin(2*pi*(x-a)/(b-a))+2);
```

```
u1 = -cos(2*pi*(x-a)/(b-a))*2*pi/(b-a)./(sin(2*pi*(x-a)/(b-a))+2).^2;
uhat = fftshift(fft(u))*sqrt(b-a)/N;
lambda = 1i*(-N/2:N/2-1)'*2*pi/(b-a);
vhat = uhat.*lambda;
vhathat = ifft(ifftshift(vhat))*N/sqrt(b-a);
norm(vhathat-u1,inf)
```

4.2 2-dimensional and n-dimensional transforms

Given the 2-dimensional array `u` of dimension `N(1)\times \verb`$N(2)+$, we can transform from the rectangle [`a(1)`,`b(1)`] × [`a(2)`,`b(2)`] with the command `fftshift (fft2 (u)) * prod (sqrt (b - a) ./ N)`. For a n-dimensional array, `fftshift (fftn (u)) * prod (sqrt (b - a) ./ N)`.

4.3 NFFT

The Nonuniform Fast Fourier Transform aims at being fast ($\mathcal{O}(N \log N)$) in evaluating a truncated Fourier series at a set of N *general* points. In fact, given the set of points $\{x_n\}_{n=1}^N$, one could compute the matrix

$$\Phi = (\phi_j(x_k))$$

and evaluate

$$\begin{bmatrix} \hat{u}(x_1) \\ \hat{u}(x_2) \\ \vdots \\ \hat{u}(x_N) \end{bmatrix} = \Phi \begin{bmatrix} \hat{u}_1 \\ \hat{u}_2 \\ \vdots \\ \hat{u}_N \end{bmatrix} .$$

This would cost $\mathcal{O}(N^2)$. The NFFT algorithm developed in [5] works in the following way. It evaluates in a fast (and approximate) way a sum of type

$$\sum_{j=-N/2}^{N/2-1} \hat{\psi}_j \mathrm{e}^{-2\pi \mathrm{i} j \xi_n}, \quad \xi_n \in \left[-\frac{1}{2}, \frac{1}{2}\right) .$$

It is clear that we have to manipulate it in order to evaluate a sum of type (4.1), with $x_n \in [a, b)$. In fact, the NFFT algorithm has to be fed with

$$\begin{aligned}\xi_n &= \mod\left(\frac{x_n - a}{b - a}, 1\right) - \frac{1}{2}\\ \hat{\psi}_j &= \hat{u}_j \frac{\mathrm{e}^{\pi \mathrm{i}(j-1-N/2)}}{\sqrt{b-a}} = \hat{u}_j \frac{(-1)^{j-1-N/2}}{\sqrt{b-a}}\end{aligned}$$

Paper [4] contains a library for an easy integration and usage of NFFT (up to dimension 3) in Matlab/GNU Octave.

Chapter 5

Finite Differences

5.1 1-dimensional

On `x = linspace (a, b, n)'`, with `h = (b - a) / (n - 1)` we can construct the discretization matrices of first derivative and second derivative

```
Dx = toeplitz (sparse (1, 2, -1 / (2 * h), 1, n), ...
               sparse (1, 2, 1 / (2 * h), 1, n));
Dxx = toeplitz (sparse ([1, 1], [1, 2], [-2, 1] / h ^ 2, 1, n));
```

5.1.1 Boundary conditions

Dirichlet b.c.

Periodic b.c.

Neumann b.c.

5.2 2-dimensional

Let's check how to construct a grid in Matlab/GNU Octave.

```
x = linspace (a, b, n)';
hx = (b - a) / (n - 1);
Dx = toeplitz (sparse (1, 2, -1 / (2 * hx), 1, n), ...
                sparse (1, 2, 1 / (2 * hx), 1, n));
Dxx = toeplitz (sparse ([1, 1], [1, 2], [-2, 1] / hx ^ 2, 1, n));
y = linspace (c, d, m)';
hy = (d - c) / (m - 1);
Dy = toeplitz (sparse (1, 2, -1 / (2 * hy), 1, m), ...
                sparse (1, 2, 1 / (2 * hy), 1, m));
Dyy = toeplitz (sparse ([1, 1], [1, 2], [-2, 1] / hy ^ 2, 1, m));
[X, Y] = ndgrid (x, y);
```

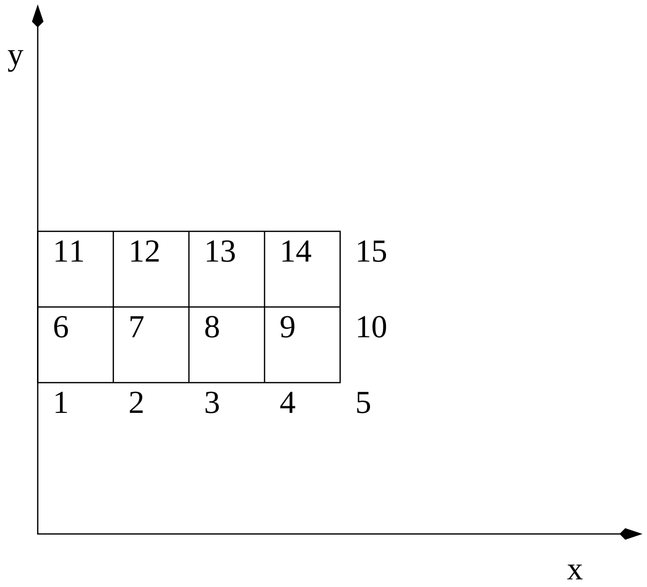


Figure 5.1: 2-dimensional grid.

For instance, for `x = linspace (0, 4, 5)'` and `y = linspace (2, 4, 3)'` it corresponds to the grid in Figure 5.1. Point i has coordinates `X(i)`,`Y(i)`. Matrices of first and second partial derivatives are obtained by

```
DDx = kron (speye (m), Dx);
DDy = kron (Dy, speye (n));
DDxx = kron (speye (m), Dxx);
DDyy = kron (Dyy, speye (n));
```

Given a function `U = f (X, Y)`, it is possible to approximate its partial derivative with respect to x (but pay attention to the boundaries!) by

```
reshape (DDx * U(:), n, m)
```

Chapter 6

Finite Elements

6.1 Poisson problem

We aim at solving

$$\begin{cases} -\Delta u(x,y) = f(x,y) & (x,y) \in \Omega \\ \ u(x,y) = 0 & \qquad\quad \in \partial\Omega \end{cases}$$

We consider the *weak formulation*

$$-\int_\Omega \Delta u v = \int_\Omega f v, \quad \forall v,\ v_{|\partial\Omega} = 0$$

and apply Green's formula on the left hand side

$$\int_\Omega \nabla u \cdot \nabla v - \int_\Omega f v - \int_{\partial\Omega} v \nabla u \cdot \vec{n} = 0 = \int_\Omega \nabla u \cdot \nabla v - \int_\Omega f v, \quad \forall v,\ v_{|\partial\Omega} = 0$$

We introduce a set of discretization points $\{(x_i, y_i)\}_{i=1}^n$ and a triangulation on Ω with triangles T_k and functions $\varphi_i(x,y)$ which are

1. continuous
2. linear in x and y on each triangle
3. with $\varphi_i(x_j, y_j) = \delta_{ij}$

We assume that an approximate solution can be written as

$$\hat{u}(x,y) = \sum_j \hat{u}_j \varphi_j(x,y).$$

We insert such an approximation into the equation and get

$$\int_\Omega \nabla\hat{u}\cdot\nabla\hat{v} - \int_\Omega f\hat{v} = 0, \quad \forall\hat{v},\ \hat{v}_{|\partial\Omega} = 0 \tag{6.1}$$

where by $\nabla\hat{u}$ we mean the piecewise gradient of $\hat{u}$. The first term is a bilinear form on $\hat{u}$ and $\hat{v}$ and the second a linear form on $\hat{v}$ (that is a trivial bilinear form on nothing and $\hat{v}$). If we insert the definitions of $\hat{u}$ and consider all the φ_i instead of all the $\hat{v}$, we get

$$\sum_j \hat{u}_j \left(\sum_k \int_{T_k} \nabla\varphi_j\cdot\nabla\varphi_i\right) - \left(\sum_k \int_{T_k} f\varphi_i\right), \quad \forall\varphi_i.$$

This is a linear system in the unknowns $\hat{u}_j$. Formulation (6.1) can be solved by FreeFem++.

Another possibility it to look for the minimum of the functional

$$J(u) = \frac{1}{2}\int_\Omega |\nabla u|^2 - \int_\Omega fu.$$

In fact, a critical point for J is

$$\delta J(u)v = \int_\Omega \nabla u\cdot\nabla v - \int_\Omega fv = 0, \quad \forall v,\ v_{|\partial\Omega} = 0$$

Of course, we have to restrict the search in the finite dimensional space and look for

$$\hat{u} = \arg\min_{\hat{v},\ \hat{v}_{|\partial\Omega}=0} J(\hat{v})$$

6.2 p-Laplace problem

Now we consider the solution of

$$\begin{cases} -\operatorname{div}(|\nabla u|^{p-2}\nabla u) = f & \Omega \\ u = 0 & \partial\Omega \end{cases}$$

for $p \geq 2$. This problem is the Poisson problem for $p = 2$. The corresponding J functional is

$$J(u) = \frac{1}{p}\int_\Omega |\nabla u|^p - \int_\Omega fu.$$

In FreeFem++ it possible to use the nonlinear conjugate gradient method to find

$$\hat{u} = \arg\min_{\hat{v},\ \hat{v}_{|\partial\Omega}=0} J(\hat{v}).$$

It is necessary to compute

$$\delta J(u)v = \int_\Omega |\nabla u|^{p-2} \nabla u \cdot \nabla v - \int_\Omega fv = \int_\Omega (\nabla u \cdot \nabla u)^{\frac{p-2}{2}} \nabla u \cdot \nabla v - \int_\Omega fv.$$

Since $\hat{u} = \sum_j \hat{u}_j \varphi_j$ and $\hat{v} = \sum_i \hat{v}_i \varphi_i$, it is possible to introduce the functional $\delta J_R(\hat{\boldsymbol{u}})\colon \mathbb{R}^n \to \mathbb{R}$ defined by

$$\begin{aligned}\delta J_R(\hat{\boldsymbol{u}})\hat{\boldsymbol{v}} &= \int_\Omega \left|\sum_j \hat{u}_j \nabla\varphi_j\right|^{p-2} \sum_j \hat{u}_j \nabla\varphi_j \cdot \sum_i \hat{v}_i \nabla\varphi_i - \int_\Omega f \sum_i \hat{v}_i \varphi_i = \\ &= \sum_i \hat{v}_i \left(\int_\Omega \left|\sum_j \hat{u}_j \nabla\varphi_j\right|^{p-2} \sum_j \hat{u}_j \nabla\varphi_j \cdot \nabla\varphi_i - \int_\Omega f\varphi_i \right).\end{aligned}$$

Another way to write it is $\hat{\boldsymbol{v}}^T \delta J_R(\hat{\boldsymbol{u}})$. By the nonlinear conjugate gradient method (`NLCG` in FreeFem++) it is possible to solve

$$\hat{\boldsymbol{u}} = \arg\min_{\hat{\boldsymbol{v}}} J_R(\hat{\boldsymbol{v}}).$$

Only the action of $\delta J_R(\hat{\boldsymbol{u}})$ is required.

6.2.1 Use of a preconditioner

In a minimization method, a sequence $\{\hat{u}^m\}_m$ is produced which tries to minimize J_R defined by

$$J_R(\hat{\boldsymbol{u}}) = \frac{1}{p} \int_\Omega \left|\sum_j \hat{u}_j \nabla\varphi_j\right|^p - \int_\Omega f \sum_j \hat{u}_j \varphi_j$$

From the definition of $\delta J_R(\hat{\boldsymbol{u}})$, we see that it is of type $A(\hat{\boldsymbol{u}}) - \boldsymbol{b} \in \mathbb{R}^n$. At iteration m, its i-th component can be approximated by

$$\begin{aligned}&\int_\Omega \left|\sum_j \hat{u}_j^m \nabla\varphi_j\right|^{p-2} \sum_j \hat{u}_j \nabla\varphi_j \cdot \nabla\varphi_i - \int_\Omega f\varphi_i = \\ &\qquad = \sum_j \hat{u}_j \left(\int_\Omega \left|\sum_j \hat{u}_j^m \nabla\varphi_j\right|^{p-2} \nabla\varphi_j \cdot \nabla\varphi_i \right) - \int_\Omega f\varphi_i\end{aligned}$$

in such a way that the approximate differential can be casted as

$$A^m \hat{\boldsymbol{u}} - \boldsymbol{b},$$

with A^m a symmetric matrix. Well, A^m, which is $\delta(A^m \hat{\boldsymbol{u}})$ (which is an approximation of $\delta(A(\hat{\boldsymbol{u}}) - \boldsymbol{b})$) is a preconditioner.

6.3 Exercises and projects

1. Write a FreeFem++ code which solve the p-Laplace problem for p up to 1000.

Chapter 7

Matrix functions

7.1 Matrix exponential

The matrix exponential for $A \in \mathbb{C}^{n\times n}$ is defined as

$$\exp(A) = \sum_{k=0}^{\infty} \frac{A^k}{k!}$$

In order to numerically evaluate it, we may think to two simple strategies: Taylor's truncated series and eigenvalue decomposition.

7.1.1 Taylor's truncated series

It is a strong temptation to approximate

$$\exp(A) \approx \sum_{k=0}^{m} \frac{A^k}{k!} = T_m(A)$$

Unfortunately, it does not work, even in simple scalar cases (try to compute the relative error $\mathrm{e}^{100}(\mathrm{e}^{-100} - T_m(-100))$ for increasing values of m). The problem is that Taylor's series, although everywhere convergent, is fast enough (that is it converges to machine precision before truncation errors show up) only in a neighborhood of 0. To this aim, the *scaling and squaring* technique may help. That is, taken $s = 2^j$ such that $\|A\| < 1$, $E = \exp(A/s)$ is approximated by $T_m(A/s)$ and later $\exp(A)$ is recovered by

$$E = E^2 \; j \text{ times}$$

Padé approximation

We can change the way $\exp(A/s)$ is approximated. For instance, a very common approximation is the rational *Padé* one. That is

$$e^z \approx \frac{a_p z^p + a_{p-1} z^{p-1} + \ldots + a_1 z + a_0}{b_q z^q + b_{q-1} z^{q-1} + \ldots + b_1 z + 1} = r_{p,q}(z)$$

such that $T_{p+q}(z) = r_{p,q}(z) + \mathcal{O}(z^{p+q+1})$ for $z \to 0$. Let's try for $p = q = 1$: we have

$$e^z \approx \frac{\frac{1}{2}z + 1}{-\frac{1}{2}z + 1} = r_{1,1}(z)$$

By the way, this corresponds to the trapezoidal rule $y_1 = \left(\frac{1}{2}k + 1\right) / \left(-\frac{1}{2}k + 1\right) = r_{1,1}(k)$ for the solution of $y'(t) = y(t)$, $y(0) = 1$ at $t = k$. The extension to the matrix case is trivial

$$\exp(A/s) \approx \left(-\frac{1}{2}A/s + I\right)^{-1} \left(\frac{1}{2}A/s + I\right)$$

Then, the scaling and squaring technique is used. In practice, the degree $p = q$ of common Padé approximations is around 10. Padé approximations for the exponential share the following property

$$r_{p,p}(-z) = \frac{1}{r_{p,p}(z)}$$

7.1.2 Eigenvalue decomposition

If A is diagonalizable, $AV = V\Lambda$, then

$$\exp(A) = V \exp(\Lambda) V^{-1}$$

where $\exp(\Lambda)$ is easily seen to be $\operatorname{diag}(\lambda_1, \ldots, \lambda_n)$. Unfortunately, even in this case it is possible to do very wrong, try with

$$A = \begin{bmatrix} 1 & 1 \\ 0 & 1+\varepsilon \end{bmatrix}$$

and compare in Matlab

```
A=[1,1;0,1+eps];,expm(A),[V,Lambda]=eig(A);,V*diag(exp(diag(Lambda)))/V
```

7.2 Matrix exponential-related functions

We would like to approximate the following functions

$$\varphi_\ell(A) = \sum_{k=0}^{\infty} \frac{A^k}{(k+\ell)!}$$

We have $\varphi_0(z) = \mathrm{e}^z$,

$$\varphi_1(z) = \frac{\mathrm{e}^z - 1}{z}$$

and, in general,

$$\begin{aligned} \varphi_\ell(z) &= z\varphi_{\ell+1}(z) + \frac{1}{\ell!}, & \ell \geq 0 \\ \varphi_\ell(z) &= \frac{1}{(\ell-1)!} \int_0^1 \mathrm{e}^{(1-\theta)z} \theta^{\ell-1} \mathrm{d}\theta, & \ell \geq 1 \end{aligned}$$

Of course it is possible a Taylor or Padé approximation in a neighborhood of 0. The scaling and squaring technique is more involved, however. For instance

$$\varphi_1(z) = \frac{1}{2}(\mathrm{e}^{z/2} + 1)\varphi_1\left(\frac{z}{2}\right)$$

The function ϕ_1 is very important. For instance, the solution of a linear, constant coefficients system of ODEs

$$\begin{cases} y'(t) = Ay(t) + b \\ y(t_0) = y_0 \end{cases}$$

is

$$y(t) = y_0 + (t - t_0)\varphi_1((t - t_0)A)b$$

If one is interested only in $\varphi_\ell(A)w$, $w \in \mathbb{C}^n$, then the following theorem can be used (see [1]).

Theorem 1. *Let $A \in \mathbb{C}^{n\times n}$, $W = [w_1, \ldots, w_p] \in \mathbb{C}^{n,p}$, $\tau \in \mathbb{C}$ and*

$$\tilde{A} = \begin{bmatrix} A & W \\ 0 & J \end{bmatrix} \in \mathbb{C}^{(n+p)\times(n+p)}, \quad J = \begin{bmatrix} 0 & I_{p-1} \\ 0 & 0 \end{bmatrix} \in \mathbb{R}^{p\times p}$$

then for $X = \exp(\tau\tilde{A})$ we have

$$X(1:n, n+j) = \sum_{k=1}^{j} \tau^k \varphi_k(\tau A) w_{j-k+1}, \quad j = 1, 2, \ldots, p$$

Before giving the proof, let us consider a simple example: we want to compute $\varphi_1(\tau A)w$. We consider

$$\tilde{A} = \begin{bmatrix} A & w \\ 0 & 0 \end{bmatrix}$$

and compute $X = \exp(\tau\tilde{A})$, extract the first n rows, last column and divide by τ.

```
n=4;
tau=rand;
A=rand(n);
w=rand(n,1);
Atilde=[A,w;zeros(1,n),0];
X=expm(tau*Atilde);
X(1:n,n+1)/tau
(tau*A)\((expm(tau*A)-eye(n))*w)
```

If interested into $\exp(\tau A)v + \tau\varphi_1(\tau A)w$, we can do

```
n=4;
tau=rand;
A=rand(n);
v=rand(n,1);
w=rand(n,1);
eta=2^(-ceil(log2(norm(w,1))));
Atilde=[A,eta*w;zeros(1,n),0];
X=expm(tau*Atilde)*[v;1/eta];
X(1:n)
expm(tau*A)*v+tau*((tau*A)\((expm(tau*A)-eye(n))*w))
```

The use of the parameter η is for numerical stability (see [1]).

The possibility to compute $\varphi_\ell(A)w$ without computing $\varphi_\ell(A)$ is similar to the possibility to compute the solution of $Ax = w$ without computing A^{-1}.

Proof. We start computing

$$\tilde{A}^2 = \begin{bmatrix} A & W \\ 0 & J \end{bmatrix} \begin{bmatrix} A & W \\ 0 & J \end{bmatrix} = \begin{bmatrix} A^2 & AW + WJ \\ 0 & J^2 \end{bmatrix}$$

and we easily get

$$\tilde{A}^k = \begin{bmatrix} A^k & M_k \\ 0 & J^k \end{bmatrix}$$

with $M_0 = 0$, $M_1 = W$, $M_k = A^{k-1}W + M_{k-1}J$. Then $WJ(:,j) = w_{j-1}$ and $JJ(:,j) = J(:,j-1)$ for $1 \leq j \leq p$ where we define $w_0 = J(:,0) = 0$. Thus

$$\begin{aligned} M_k(:,j) &= A^{k-1}w_j + (A^{k-2}W + M_{k-2}J)J(:,j) = \\ &= A^{k-1}w_j + A^{k-2}w_{j-1} + M_{k-2}J(:,j-1) = \\ &= \sum_{i=1}^{\min\{k,j\}} A^{k-i}w_{j-i+1} \end{aligned}$$

Moreover

$$X(1:n, n+1:n+p) = \sum_{k=0}^{\infty} \frac{\tau^k M_k}{k!} = \sum_{k=1}^{\infty} \frac{\tau^k M_k}{k!}$$

and now we can compute

$$\begin{aligned} X(1:n, n+j) &= \sum_{k=1}^{\infty} \frac{\tau^k M_k(:,j)}{k!} = \sum_{k=1}^{\infty} \frac{1}{k!} \left(\sum_{i=1}^{\min\{j,k\}} \tau^i (\tau A)^{k-i} w_{j-i+1} \right) = \\ &= \sum_{i=1}^{j} \tau^i \left(\sum_{k=i}^{\infty} \frac{(\tau A)^{k-i}}{k!} \right) w_{j-i+1} = \\ &= \sum_{i=1}^{j} \tau^i \left(\sum_{k=0}^{\infty} \frac{(\tau A)^k}{(k+i)!} \right) w_{j-i+1} = \sum_{i=1}^{j} \tau^i \varphi_i(\tau A) w_{j-i+1} \end{aligned}$$

□

Bibliography

[1] A. H. Al-Mohy and N. J. Higham. Computing the action of the matrix exponential with an application to exponential integrators. *SIAM J. Sci. Comput.*, 33(2):488–511, 2011.

[2] J. Baglama. Augmented block Householder Arnoldi method. *Linear Algebra Appl.*, 429:2315–2334, 2008.

[3] J. Baglama and L. Reichel. Augmented implicitly restared Lanczos bidiagonalization methods. *SIAM J. Sci. Comput.*, 27(1):19–42, 2005.

[4] M. Caliari and S. Zuccher. INFFTM: Fast evaluation of 3d Fourier series in MATLAB with an application to quantum vortex reconnection. *Comput. Phys. Commun.*, 213:197–207, 2017.

[5] J. Keiner, S. Kunis, and D. Potts. Using NFFT 3—A Software Library for Various Nonequispaced Fast Fourier Transforms. *ACM Trans. Math. Software*, 36(4):19:1–19:30, 2009.

[6] R. B. Lehoucq, D. C. Sorensen, and C. Yang. *ARPACK User's Guide: Solution of Large Scale Eigenvalue Problems with Implicitly Restarted Arnoldi Methods.* Software Environments Tools. SIAM, 1998.

[7] Y. Saad. *Iterative Methods for Sparse Linear systems.* Other Titles in Applied Mathematics. SIAM, 2nd edition, 2003.

[8] G. W. Stewart. A Krylov–Schur algorithm for large eigenproblems. *SIAM J. Matrix Anal.*, 23(3):601–614, 2001.