Technical Manual of the program system *maiprogs*

Matthias Maischak[*]
Brunel University, Uxbridge, UK
Institut für Angewandte Mathematik, Universität Hannover

April 25, 2015

0 Overview

This manual contains the parameter lists of subroutines in the libraries and other source files of the program system *maiprogs* which are shared by the different programs and which can also be used for own programs. The interfaces are mostly stable and probably will not change in the future. But its recommended that you try to use the newest version if possible.

For most internal data structures access functions are provided, which will only enhanced in their functionality, but which will remain compatible on the language level. It is strongly recommended that you use data structures only via their access functions, to keep the amount of effort limited, which is necessary to update to a new *maiprogs* version.

For simple changes to *maiprogs* the config script allows to define additional directories in depmod.conf, which are used to search for main programs and additional modules and whose contents also override the versions in the main directories if you use the same name for a file like in the main directory. By using this mechanism there is no need to change the original file at all. Only make a copy to your own directory and change the copy. Be aware, you should never use the same name for a module in your own source file as in a standard source file, when the source files are differently named. This will confuse the build mechanism.

Remark: There is no warranty that these routines will work correctly as specified. The use of these routines is on your own risk. Using these routines in commercial programs is prohibited as long as not explicitly allowed by the author.

[*]email: matthias.maischak@brunel.ac.uk

Contents

1 Introduction

maiprogs is a fully fledged f90 program and makes use of most modern f90 features. Especially the whole package is structured by the use of modules, encapsulation by using generic interfaces and declaring the module procedures private and also by the definition of data types dedicated to the description of meshes and splines. Global constants are always declared in the module header. Especially we have to note the definition of the kind parameter `dp=kind(0.0d0)` which is used to define the precision of real variables, which is part of the module `const`.

It is strongly recommended that you never access the internal data structures directly, but use the access functions provided. If for some reason, you see no possibility to avoid the direct use of internal data structures, either for reasons of speed or because some functionality is missing, please contact the author of this package, for providing the needed functionality in a future release.

2 Components of a FEM/BEM programme

In this section we want to outline the general structure of a FEM/BEM programme, i.e. we want to present the sequence of steps which are to be taken to achieve an efficient FEM/BEM code and give the reasons for the most crucial choices. We do not want discuss here the general bookkeeping, but we want to concentrate on the way the whole programme is broken down in smaller units which are mostly independent of each other and communicate only by some data structures.

There are the following components: Geometry, Mesh, Splines, Matrix, Block Matrix, Right-hand Side, Solver.

3 Internal data structures for geometry definition

There is only one general data format for geometry definitions in *maiprogs*. This general container format is used in all 3 dimensions.

The components ending on 'g' contain a geometry definition which plainly defines the geometry itself. The components ending on 's' contain an alternative definition of the same geometry with additional information on the location of singularities which is used for the generation of structured meshes.

In the following we will omitt the trailing g or s.

ng contains the number of macro elements in the geometry definition, rnode the node numbers for each macro element, type the kind of macro element, rn an additional normal direction, dn is a descriptor for every macro element which can take any value (currently alway 0), ori in case of singularities describes the location. node contains the coordinates of all nodes.

bmode(0:1) as usual describes the dimension of the geometry.

The one-dimensional rnode structure can only be read consecutively. For every element the first entry is the number of nodes (nr) belonging to this element, followed by the node numbers.

type and number of nodes (nr) for every element define the kind of macro element.

```
 type geom
  integer, dimension(:), pointer :: rnodeg,typeg,dng
  real(kind=dp), dimension(:), pointer :: rng
  integer, dimension(:), pointer :: rnodes,types,ori,dns
  real(kind=dp), dimension(:), pointer :: rns
  real(kind=dp), dimension(:), pointer :: nodes

  real(kind=dp), dimension(0:2) :: g0

  integer :: bmode(0:1)
  integer :: nnodes,nrnodeg,ngg
  integer ::        nrnodes,ngs
  character(len=64) :: name
 end type geom
```

bmode=(1,1)

(type,nr)	
(0,2)	line segment

bmode=(1,2)

(type,nr)	
(0,2)	line segment
(1,3)	arc segment, described by three nodes located at the beginning, the middle and the end of the segment
(10,n)	Curvi-linear element represented by Lagrange based uniform-interpolation polynomial with n nodes

bmode=(2,2)

(type,nr)	
(0,3)	triangle
(0,4)	quadrilateral element
(1,4)	triangle with one curved (arc) boundary segment, the first three nodes define the arc segment, whereas the last node defines a corner
(1,5)	quadrilateral element with one curved (arc) boundary segment, the first three nodes define the arc segment, whereas the two nodes define corners. Effectively nodes 1,3,4,5 define the corners of the curved quadrilateral element, whereas node 2 is used to define the curvature
(1,6)	quadrilateral element with two curved (arc) boundary segments, opposite to each other. Nodes 1,3,4,6 define the corners of the curved quadrilateral element. Node 2 defines the curvature of the first arc segment and node 5 defines the curvature of the second arc segment
(10,n^2)	Curvi-linear element (square) represented by Lagrange based uniform-interpolation polynomial with n^2 nodes

bmode=(2,3)

(type,nr)	
(0,3)	triangle
(0,4)	quadrilateral element
(1,4)	triangle with one curved (arc) boundary segment, the first three nodes define the arc segment, whereas the last node defines a corner
(2,4)	curved triangular element (sphere-shell). The first three nodes define the corners of the curved triangle, the last node is a node in the curved triangle which allows to determine the center of the sphere
(2,6)	defines a curved quadrilateral element (cylinder-shell). The first three nodes define the first arc segment, the last three nodes the second arg segment
(10,n^2)	Curvi-linear element (square) represented by Lagrange based uniform-interpolation polynomial with n^2 nodes

bmode=(3,3)

(type,nr)	
(0,4)	tetrahedral element
(0,5)	pyramidal element (the first four nodes give the base of the pyramid)
(0,6)	prismatic element (the first three nodes define the bottom)
(0,8)	hexahedral element (the first four nodes define the bottom)
(1,8)	cylinder with one curved surface. The first 4 nodes define one end of the cylinder, the next 4 nodes the other end
(1,10)	cylinder, this time based on a quadrilateral element
(1,12)	hollow cylinder shell (not fully working yet). The first 6 nodes define one end of the cylinder, the second 6 nodes the other end. From each set of 6 nodes the first 3 nodes describe the outer arc and the other ones the second arc (cf. bmode=(/2,2/),typ=1,nr=6).
(2,4)	sphere. The first node is the center, the last three nodes define the surface segment
(2,8)	spherical segment with two curved surfaces. The first three nodes define the corners of the first curved triangle, the fourth node is an additional node on the surface. The next three nodes define the corners of the second curved triangle, the last node is an additional node on the second surface.

3.1 Geometry database

The search function to access the geometry database is `qgm`, i.e. with `gm=>qgm(str)` the pointer `gm` contains the result of our database query for a geometry with the name given by the variable `str`.

Don't confuse the database of defined geometries with the database of pre-defined geometries (ads/geometries).

4 Internal data structures for Meshes

There are three data types created for the description of meshes in defined on a manifold with 1, 2 or 3 dimensions. Note that the dimension of the manifold can be different from the dimension of the space. Depending on this dimension the definitions are contained in the modules spline1, spline2 or spline3 located in the file splines1.f90, splines2.f90 or splines3.f90 (mesh and spline definitions are located in the same files).

The three types mesh1,mesh2 and mesh3 have the following components in common:

- The component `ng` denotes the number of elements of the mesh.
- The component `name` contains the full-name of the mesh.
- The component `bmode(0:1)` contains in `bmode(0)` the dimension of the manifold and in `bmode(1)` the dimension of the space.

4.1 One dimensional manifolds

```
type mesh1
 real(kind=dp), dimension(:,:), pointer :: rx,rh,rn
 integer, dimension(:,:), pointer :: rc,rct
 integer :: ng,bmode(0:1)
 character(len=32) :: name
end type mesh1
```

The mesh consists of `ng` boundary elements given by start points `rx(:,i)` and distance vectors `rh(:,i) (i=0,ng-1)`. For every boundary element, the normal direction is given by `rn(:,i)`. `rc(0,i)` gives the element number of the left neighbor (connected to the point `rx(:,i)`) and `rc(2,i)` gives the element number of the right neighbor (connected to the point `rx(:,i)+rx(:,i)`). In most cases we don't need to deal with the internals of mesh representation. The access subroutine

```
subroutine mh2mdn(mh,i,mx,dx,nx)
type(mesh1), intent(in) :: mh
integer, intent(in) :: i
real(kind=dp), intent(out) :: mx(0:1),dx(0:1),nx(0:1)
```

extracts the midpoint `mx(0:1)`, the distance vector `dx(0:1)` (pointing from the midpoint to the right endpoint) and the normal direction `nx(0:1)` of the i-th element from the mesh `mh`, independent of the internal representation.

4.2 Two dimensional manifolds

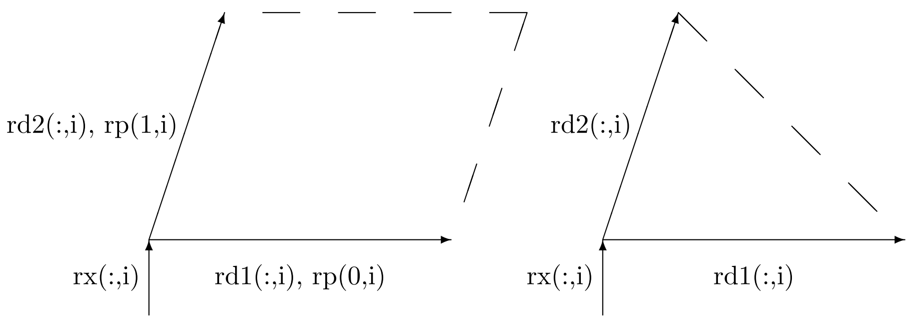
```
type elem2
 integer :: rnodes(0:4) ! rnodes(0)   : 3=triangle, 4=quadrilateral
                        ! rnodes(1:4) : node numbers
 integer :: rc(0:3)     ! neighbours
 integer :: rct(0:3)    ! sides of neighbours
 integer :: father
 integer :: sons(1:4)
 integer :: ref
 real(kind=dp) :: rn(0:2) ! Normal for 3d-mesh
end type elem2

type mesh2
 real(kind=dp), dimension(:,:), pointer :: rx,rd1,rd2,rn
 integer, dimension(:), pointer :: rt
 integer, dimension(:,:), pointer :: rc,rct

 real(kind=dp), dimension(:,:), pointer :: nodes
 integer :: nnodes

 integer, dimension(:,:), pointer :: rnodes
 integer :: ng,bmode(0:1)
 character(len=32) :: name

 type(elem2), dimension(:), pointer :: elem  ! Array of mesh elements
                                             ! allowing hanging nodes
 integer :: nelem
end type mesh2
```



parallelogram, rt(i)=4   triangle, rt(i)=3

4.3 Three dimensional spaces

```
type elem3
 integer :: rnodes(0:8) ! rnodes(0)   : 8=hexaeder, 4=tetraeder
                        ! rnodes(1:8) : node numbers
 integer :: rc(0:5)     ! neighbours
 integer :: rct(0:5)    ! sides of neighbours
 integer :: father
 integer :: sons(1:8)
 integer :: ref
end type elem3

type mesh3
 real(kind=dp), dimension(:,:), pointer :: nodes
 integer :: nnodes
 integer, dimension(:,:), pointer :: rnodes
 integer :: ng,bmode(0:1)
 integer, dimension(:,:), pointer :: rc,rct
 character(len=32) :: name

 type(elem3), dimension(:), pointer :: elem  ! Array of mesh elements
                                             ! allowing hanging nodes
 integer :: nelem
```

```
end type mesh3
```

4.4 Mesh databases

5 Internal data structures for Splines

There are three data types created for the description of splines defined on a manifold with 1, 2 or 3 dimensions. Note that the dimension of the manifold can be different from the dimension of the space. Depending on this dimension the definitions are contained in the modules spline1, spline2 or spline3 located in the file splines1.f90, splines2.f90 or splines3.f90 (mesh and spline definitions are located in the same files).

The three types spline1, spline2 and spline3 have the following components in common:

- The component `ng` denotes the number of elements of the mesh to which the spline is subordinated.
- The component `name` contains the full-name of the spline-variable.
- The component `bmode(0:1)` contains in `bmode(0)` the dimension of the manifold and in `bmode(1)` the dimension of the space.

Initializing the internal coefficients necessary for a spline defined in the module polydat can be done by calling the generic subroutine

```
subroutine polyinit(sp)
use spline{1,2,3}
type(spline{1,2,3}), intent(in) :: sp
```

The individual representations are defined in spline1, spline2 and spline3.

This section deals with the most important data structures of the whole program system, i.e. the structures for meshes and solutions. We use a consistent scheme for all cases (2D-BEM,3D-BEM,2D-FEM), but note that our scheme is most appropriate for the implementation of a general hp-mesh, i.e. an arbitrary distribution of mesh elements and polynomial degrees.

Lets first make a few restricting assumptions on the kind of splines which we want to deal with. Later we can generalize everything.

Our fundamental assumptions are the following:

- Our (global) basis functions are based on a mesh.
- On every mesh element we have a given set of local basis functions.
- The local basis functions are generated by mapping a reference element to the mesh element.
- Basis functions are a linear combination of local basis functions.
- Every local basis functions belongs only to one and only one global basis function.

Using this fundamental assumptions we have the following objects to deal with:

- The mesh consisting of mesh elements: ω_i.
- The mappings $F_i : Q \mapsto \omega_i$ from the reference element to the mesh elements.
- The set of basis function on the reference element: $\phi_k^{\text{ref}} : Q \mapsto \mathbb{R}$ (or $\mathbb{R}^d$).
- The local basis functions on every element: $\phi_{i,k}^{\text{loc}}(x) = \phi_k^{\text{ref}}(F_i^{-1}(x)) : \omega_i \mapsto \mathbb{R}$, or if ϕ_k^{ref} is vector valued: $\phi_{i,k}^{\text{loc}}(x) = B_i\phi_k^{\text{ref}}(F_i^{-1}(x)) : \omega_i \mapsto \mathbb{R}^d$ with a transformation matrix $B_i \in \mathbb{R}^{d\times d}$.

Then every global basis function can be represented in the following way:

$$\phi_p(x) = \sum_{p=r_{i,k}} w_{i,k}\phi_{i,k}^{\text{loc}}(x) \tag{1}$$

Here $w_{i,k}$ are weights belonging to every local basis functions and $r_{i,k}$ denotes the global basis functions to which every local basis function belongs to.

First we start with the elements which are in common for all cases.

```
type spline{1,2,3}
integer :: ng,ktyp,vtypc,vtypf,dof
integer, dimension(:), pointer :: ri=>null()  ! ri(0:ng)
integer, dimension(:), pointer :: rci=>null() ! rci(0:)
real(kind=dp), dimension(:), pointer :: rcw=>null() ! rcw(0:)
```

`ktyp`=1 means we deal with real numbers, and `ktyp`=2 means we are dealing with complex valued numbers. `vtypc` denotes the number of coefficients belonging to a basis function. `vtypf`=1 means we deal with a solution with scalar components, whereas `vtypf`=2,3 means we are dealing with a vector-valued solution with two or three components. Note, `vtypc` and `vtypf` can be different.

`ng` denotes the actual number of mesh elements. `ri(i)`, (i=0,...,ng-1) gives the beginning of the local degrees of freedom in a global setting, therefore `ri(i+1)-ri(i)`, (i=0,...,ng-1) gives the number of local degrees of freedom defined on the i-th mesh element. `rci` maps the local degrees of freedom to the global degrees of freedom, i.e.

```
do i=0,ng-1
 do j=0,ri(i+1)-ri(i)-1
  write(*,*) i,j,(x(rci(ri(i)+j)*ktyp+r),r=0,ktyp-1)
 end do
end do
```

prints the values of the vector `x` for all local degrees of freedom. `rci` is also used to eliminate degrees of freedom to incorporate homogeneous boundary conditions. The corresponding local degrees of freedom are just mapped to the first element behind the global degrees of freedom. The number of existing global degrees of freedom is obtained by `dof`. In previous versions the global degrees of freedom could be obtained by `n=rci(ri(ng))`, but this is not longer permissable.

5.1 One-dimensional splines

Compute the value of solution ckom at point x in element i (1D)

```
subroutine fpsi1(x,i,sp,ckom,fw)
use poly, only: ptab1
integer, intent(in) :: i
real(kind=dp), intent(in) :: ckom(0:),x
type(spline1), intent(in) :: sp
real(kind=dp), intent(out) :: fw(0:*)
```

Compute the gradient of solution ckom at point x in element i (1D)

```
subroutine grdpsi1(x,i,sp,ckom,fwx)
use poly, only: pstab1
integer, intent(in) :: i
real(kind=dp), intent(in) :: ckom(0:),x
type(spline1), intent(in) :: sp
real(kind=dp), intent(out) :: fwx(0:*)
```

5.2 Two-dimensional splines

Compute the value of solution ckom at point x in element i (2D)

```
subroutine fpsi2(x,i,typ,sp,ckom,fw,a1,a2,a12)
use poly, only: ptab2n,basenum2
integer, intent(in) :: typ,i
real(kind=dp), intent(in) :: ckom(0:),x(0:1)
type(spline2), intent(in) :: sp

real(kind=dp), dimension(0:sp%bmode(1)-1), intent(in), optional :: a1,a2,a12
real(kind=dp), intent(out) :: fw(0:*)
```

Compute the gradient of solution ckom at point x in element i (2D)

```
subroutine grdpsi2(x,i,typ,sp,ckom,fwx,fwy,a1,a2,a12)
use poly, only: pstab2n,basenum2
integer, intent(in) :: typ,i
real(kind=dp), intent(in) :: ckom(0:),x(0:1)
type(spline2), intent(in) :: sp

real(kind=dp), dimension(0:sp%bmode(1)-1), optional :: a1,a2,a12
real(kind=dp), intent(out) :: fwx(0:*),fwy(0:*)
```

5.3 Three-dimensional splines

Compute the value of solution ckom at point x in element i (3D)

```
subroutine fpsi3(x,i,typ,sp,ckom,fw,a)
```

```
 use poly, only: ptab3, basenum3
 integer, intent(in) :: typ,i
 real(kind=dp), intent(in) :: ckom(0:),x(0:2)
 type(spline3), intent(in) :: sp

 real(kind=dp), intent(in), optional :: a(0:2,0:7)
 real(kind=dp), intent(out) :: fw(0:*)
```

Compute the gradient of solution ckom at point x in element i (3D)

```
 subroutine grdpsi3(x,i,typ,sp,ckom,fwx,fwy,fwz,a)
 use poly, only: pstab3, basenum3
 integer, intent(in) :: typ,i
 real(kind=dp), intent(in) :: ckom(0:),x(0:2)
 type(spline3), intent(in) :: sp

 real(kind=dp), intent(in), optional :: a(0:2,0:7)
 real(kind=dp), intent(out) :: fwx(0:*),fwy(0:*),fwz(0:*)
```

5.4 Spline databases

There exist three databases for spline-definitions, which contain all managed spline-definitions for 1-d,2-d and 3-d. The databases can be queried by the following pointer-valued functions `qspl1`, `qspl2`, `qspl3`, e.g.

```
 type(spline1), pointer :: sp1

 sp1=>qspl1('Name of Spline-definition')
 if (.not.associated(sp1)) then
  write(msg,*) 'Spline-definition not found'
  call logmsg
 end if
```

The coefficients of representations of this spline-definitions can be queried by `qspl1x`, `qspl2x`,`qspl3x`. This results to pointers to a data-structure which contains a pointer to the spline-definition and a pointer to the vector of coefficients. Note that there can be more than one spline using a single spline-definition.

```
 type(spline1x), pointer :: sp1x
 type(spline1), pointer  :: sp1
 real(kind=dp), dimension(:), pointer :: x

 sp1x=>qspl1x('Name of Spline')
 if (associated(sp1x)) then
! sp1x%sp is pointer to spline-definition
! sp1x%x is pointer to coefficient vector
  sp1=>sp1x%sp
  x=>sp1x%x
 end if
```

5.5 bcdat database

Every spline-class has its own local database for additional information, which will not be stored in its own component of the Fortran data structure. It is not necessary to change fundamental data structures if you simply want to introduce new information.

Entries in the database are created by the generic subroutines crbcdat and addbcdat. crbcdat simply creates the entry in the database using the fullname and nickname and reserves the amount of memory given by ni (integer), nr (real) or nc (character). addbcdat uses directly the given data i,r,c.

```
subroutine crbcdat(sp,fullname,nickname,ni,nr,nc)
type(spline?), intent(inout) :: sp
character(len=*), intent(in) :: fullname,nickname
integer, intent(in), optional :: ni,nr,nc

subroutine addbcdat(sp,fullname,nickname,i,r,c)
type(spline?), intent(inout) :: sp
character(len=*), intent(in) :: fullname,nickname
integer, intent(in), optional :: i(:)
real(kind=dp), intent(in), optional :: r(:)
character(len=*), intent(in), optional :: c
```

Using the name (fullname or nickname) you can query the bcdat-database in the usual way. Namely by qbci for integer-data, by qbcr for real data and by qbcc for strings.

```
function qbci(sp,name)
integer, dimension(:), pointer :: qbci
type(spline?), intent(in) :: sp
character(len=*), intent(in) :: name

function qbcr(sp,name)
real(kind=dp), dimension(:), pointer :: qbcr
type(spline?), intent(in) :: sp
character(len=*), intent(in) :: name

function qbcc(sp,name)
type(string), pointer :: qbcc
type(spline?), intent(in) :: sp
character(len=*), intent(in) :: name
```

6 Problem data structure

6.1 matstr

Using `matstr` we describe the construction of the (block-)system-matrix. Entries are separated by ':' or '|'. All matrices after a '|' are still in the matrix database and will be managed and computed automatically, but are not part of the system-matrix. The entries have the format 'test-spline.Matrix-name(arguments).trial-spline', e.g. 'u.A(v,t).u' for a matrix depending on the splines v and t, where test- and trial-splines are the same. `matstr` is analyzed by the subroutine setmatrix, the Matrix-name is evaluated by the subroutine matcomp2c, matcomp3c etc. With `defmatrix` we assign a bilinear form to the matrix. A list of available bilinear forms is given in Section 8.1.

6.2 datstr

Using `datstr` we describe the construction of the right hand side. Entries are separated by ':'. The entries have the form 'test-spline=expression', 'expression' can be any linear combination of given functions 'f,u0,t0', given splines, operators applied to functions or splines, and matrices multiplied with splines.

`datstr` is analyzed by the subroutine setlft. A list of available right hand side operations is given in Section 8.2.

6.3 genstr

The order in genstr determines the order in which the coefficients are ordered in the coefficient vectors 'ckom' and 'lkom'. Entries are separated by ':' or '|'. Only one '|' is allowed. Entries before '|' belong to the unknowns in a linear equation system, whereas entries after '|' are used to denote auxilliary splines. The following functionality is provided (here `u`, `p` denote splines in the domain and `D`, `D1` splines on the boundary, but the names are fully arbitrary, but have to be chosen globally unique):

`u`	The Spline space 'u' will be initalized and the memory for a Spline called 'u' will be allocated.
`u=p`	The Spline space 'u' will be initalized based on the mesh used by 'p' and using the same polynomial degrees as 'p'.
`u=p+1`	The Spline space 'u' will be initalized based on the mesh used by 'p' and using the polynomial degrees of 'p' plus '1'.
`u=1`	The Spline space 'u' will be initalized using the previously given mesh and the polynomial degree 1.
`u=Spline(p)`	The memory for a spline called 'u' will be allocated using the Spline space 'p'.
`D=Dir(u)`	'D' will be the trace space of 'u', the coefficients of 'u' will be reordered, such that 'D' will have consecutive coefficients
`D<Dir(u)+1`	'D' will use the trace mesh of 'u' and polynomial degrees plus 1, but will not share the coefficients with 'u'.
`D1<Restrict(D2,G1)+1`	'D1' will use the mesh and polynomial degrees of 'D2' restricted to the geometry 'G1', but will not share the coefficients with 'D2'
`D1=Reord(D,D1g)`	
`D1<Reord(D,D1g)+1`	

7 Matrices

7.1 Sparse Matrices

Sparse matrices are represented by the data type `matrix_sparse`. The search function to access the sparse matrix database is `qsmat`, i.e. with `smat=>qsmat(str)` the pointer `smat` contains the result of our database query for a sparse matrix with the name given by the variable `str`.

7.2 Dense Matrices

Dense matrices are represented by the data type `matrix_dense`. The search function to access the dense matrix database is `qdmat`, i.e. with `dmat=>qdmat(str)` the pointer `dmat` contains the result of our database query for a dense matrix with the name given by the variable `str`.

7.3 H-Matrices

H-matrices are represented by the data type `matrix_hmat`. The search function to access the H-matrix database is `qhmat`, i.e. with `hmat=>qhmat(str)` the pointer `hmat` contains the result of our database query for a H-matrix with the name given by the variable `str`.

7.4 Block Matrices

There exists a container datatype `matrix_block`, which collects all the individual matrix formats. Using `bmat=qblock(str)` we search all the individual databases and obtain a container variable with pointers to any matrix found. Note, that bmat is not a pointer in itself.

8 Bilinear forms and right hand sides

The choice of global basis functions (1) has far reaching consequences.

Right hand sides of the form $b_p = \int_\Omega f\phi_p, \quad p = 0, \ldots, \texttt{dof} - 1$ can be assembled in the following way:

```
1: for all i do
2:     for all k do
3:         $b^{\text{elem}}_{i,k} \leftarrow \int_\Omega f\phi^{\text{loc}}_{i,k}$
4:     end for
5:     for all k do
6:         $p \leftarrow r_{i,k}$
7:         $b_p \leftarrow b_p + w_{i,k} b^{\text{elem}}_{i,k}$
8:     end for
9: end for
```

Sparse matrices of the form $a_{pq} = \int_\Omega \nabla\phi_p \nabla\phi_q, \quad p, q = 0, \ldots, \texttt{dof} - 1$ can be assembled in the following way:

```
1: for all i do
2:     for all k,l do
3:         $a^{\text{elem}}_{i,k,l} \leftarrow \int_\Omega \nabla\phi^{\text{loc}}_{i,k} \nabla\phi^{\text{loc}}_{i,l}$
4:     end for
5:     for all k do
6:         $p \leftarrow r_{i,k}$
7:         for all l do
8:             $q \leftarrow r_{i,l}$
9:             $a_{p,q} \leftarrow a_{p,q} + w_{i,k} w_{i,l} a^{\text{elem}}_{i,k,l}$
10:        end for
11:    end for
```

```
12: end for
```

8.1 Bilinear forms

We have to distinguish bilinear forms which lead to sparse matrices from bilinear forms which lead to dense matrices. They differ in the way they are assembled.

Laplace We have the following implemented Bilinear forms for FEM and BEM:

A	$a_{ij} = \int_\Omega \nabla\phi_i \nabla\phi_j \, dx$
M	$a_{ij} = \int_\Omega \phi_i \phi_j \, dx$
B	$a_{ij} = \int_\Omega \operatorname{div} p_j * u_i \, dx$
A0	
A1	$a_{ij} = \int_\Omega \nabla\phi_j * \theta_i \, dx$
HDIV	$a_{ij} = \int_\Omega \phi_i\phi_j + \operatorname{div}\phi_i \operatorname{div}\phi_j \, dx$
PenaltyO	
DGE/DGF	
DGC	
V	Single layer potential $V_{ij} = 2\int_\Gamma \phi_i(x) \int_\Gamma G(x,y)\phi_j(y)\, ds_y \, ds_x$
K,K-I,K+I,I+K	Double layer potential $K_{ij} = 2\int_\Gamma \phi_i(x) \int_\Gamma \frac{\partial}{\partial n_y} G(x,y)\psi_j(y)\, ds_y \, ds_x$
W,W+PP	Hypersingular integral operator
S	Exterior Poincare-Steklov Operator $S = \frac{1}{2}(W + (K' + I)V^{-1}(K+I))$
Sin	Interior Poincare-Steklov Operator $S = \frac{1}{2}(W + (K' - I)V^{-1}(K-I))$
R	Inverse Exterior Poincare Steklov Operator

Lame

A	
M	$a_{ij} = \int_\Omega \phi_i \phi_j \, dx$

Helmholtz

A	$a_{ij} = \int_\Omega \nabla\phi_i \nabla\phi_j \, dx - k^2 \int_\Omega \phi_i\phi_j \, dx$
M	$a_{ij} = \int_\Omega \phi_i \phi_j \, dx$

Maxwell

Convection

CONV	Galerkin-Matrix
SUPG	SUPG-Matrix
CMDir	

Stokes

M
VDiv
STr
V,V+NN
K,K-I,K+I,I+K
W,W+PP
NN

$$\begin{aligned} NN(\phi_i e_k, \phi_j e_l) &= \int_\Gamma \mathbf{n}(x)\phi_i e_k \, ds_x \int_\Gamma \mathbf{n}(y)\phi_j e_l \, ds_y \\ &= \int_\Gamma n_k(x)\phi_i \, ds_x \int_\Gamma n_k(y)\phi_j \, ds_y \end{aligned}$$

Bilaplace

A
M $a_{ij} = \int_\Omega \phi_i \phi_j \, dx$

8.2 Right hand sides

f	Volume data
Neu	
Dir	
0	
u0	boundary data (displacement)
t0	boundary data (traction)
NV, NK	Newton-Potenial
SUPG	
'V','W','K','Ks','(K-I)','(K+I)','(Ks+I)','(Ks-I)'	
S	Poincare-Steklov operator (exterior domain)
Sin	Poincare-Steklov operator (interior domain)
R	inverse Poincare-Steklov operator (exterior domain)
NLin	Non-linear term
Matrix()*Spline	Matrix-vector multiplication
Spline	Spline will be tested

9 Adaptive schemes

9.1 Storage for indicators and refinements

The error indicators will be computed by the commands adap,resh, etc. The error indicators itself will be stored in an real-array with the name ekom in the bcdat-database of the corresponding spline. Then the refinement information (later used by `refine`) will be computed and stored in the integer-array with the name ref in the bcdat-database of the same spline. There can be different error indicators for independent splines, which will be later refined independently.

9.2 Indicator components

Indicator	Variable	
RES2CLAPUF(u)	ETALAPUF	$\eta^2 = \sum_{T\in\mathcal{T}_h} h_T^2\|\Delta u_h + f\|^2_{L^2(T)}$
RES2CNEU(u)	ETANEU	$\eta^2 = \sum_{e\in(\partial\mathcal{T}_h\cap\Gamma_N} h_e\|t_0 - \frac{\partial u_h}{\partial n}\|^2_{L^2(e)}$
RES2CJUMP(u)	ETAJUMP	$\eta^2 = \sum_{e\in\partial\mathcal{T}_h} h_e\|[\frac{\partial u_h}{\partial n}]\|^2_{L^2(e)}$
RES2CTRANS(u,D,N)	TUWKS	$\eta^2 = \sum_e h_e\|t_0 - (\varrho\nabla u_h)\mathbf{n} + \frac{1}{2}W(u_0 - u_h) - \frac{1}{2}(K' - I)(\phi_h - t_0)\|^2_{L^2(e)}$
	IKVS	$\eta^2 = \sum_e h_e\|\frac{d}{ds}\{(I-K)(u_0 - u_h) - V(\phi_h - t_0)\}\|^2_{L^2(e)}$
RES2CDIVPF(p)	DIVPF	$\eta^2_{\mathrm{div}} = \sum_{T\in\mathcal{T}_h}\|\operatorname{div} p_h + f\|^2_{L^2(T)}$
RES2CCURLP(p)	CURLP	$\eta^2_{\mathrm{curl}} = \sum_{T\in\mathcal{T}_h}\|h_T \operatorname{curl} p_h\|^2_{L^2(T)}$
RES2CNORMP(p)	NORMP	$\eta^2_p = \sum_{T\in\mathcal{T}_h}\|h_T p_h\|^2_{L^2(T)}$
RES2CPTJUMP(p)	PTJUMP	$\eta^2_{[p\cdot t]} = \sum_e \|h_e^{1/2}[p_h\cdot t]\|^2_{L^2(e)}$
RES2CPTFREE(p)	PTFREE	$\eta^2_{pt} = \sum_{e\cap\Gamma_D\neq}\|h_e^{1/2}p_h\cdot t\|_{L^2(e)^2}$
RES2CGMPN(u,S,N)	GMPN	$\eta_g = (\log[1 + C_{\tilde h}(\Gamma_N)])^{1/2}\|h_e^{1/2}(g - p_h\cdot n)\|_{L^2(\Gamma_N)}$ $\eta^2_\nabla = \sum_{T\in\mathcal{T}_h}\|p_h - \nabla\varphi_h\|^2_{[L^2(T)]^2}$ $\eta^2_u = \sum_{T\in\mathcal{T}_h}\|u_h - \varphi_h\|^2_{L^2(T)}$ $\eta_{\varphi+\xi} = \|\varphi_h + \xi_{\tilde h}\|_{L^2(\Gamma_N)}$
DW2F(u)	ETAF	
DW2JSIGMA(u)	EJSIG	
RES2CXIMTXI(u,D,N)	XIMTXI	$\xi = (I+K)N - V(p\cdot\mathbf{n})$ $\eta_\xi = \|h_e^{1/2}(\xi_{\tilde h} - \xi_h\|_{L^2(\Gamma_N)}$
RES2CPTXIS(p,D,N)	PTXIS	$\xi = (I+K)N - V(p\cdot\mathbf{n})$ $\eta_{pt+\xi'} = \|h_e^{1/2}(p_h\cdot t + \xi'_{\tilde h}\|_{L^2(\Gamma_N)}$
RES2CXIMTXIF(p,S)		
RES2CPTXISF(p,S)		
RES2CPTFREEF(p,S)		
RES2CZETA(u,D,N)	ZETA	$\zeta = (I+K')p\cdot\mathbf{n} + WD$ $\eta^2_\zeta = \sum_e h_e\|\zeta\|^2_{L^2(e)}$
LSQ2CDIVPF(p)	DIVPF	$\|\operatorname{div} p_h + f\|^2_{H^{-1}(T)}$
LSQ2CNABLA(u,p)	NABLA	$\|p_h - \nabla u_h\|^2_{L^2(T)}$
LSQ2CVARPHI(u,D,N)		
LSQ2CDIFXI(u,D,N)	DIFXI	$\|W(u_h - u_0) + 2p_h\cdot\mathbf{n} - 2t_0 - (I - K')(\sigma_h - t_0)\|^2_{\tilde H^{-1/2}(e)}$
PLAP2CETAGR(u)		
PLAP2CETAF(u)		
RES2CTRANSX(u,D,N)		

10 Local basis functions

10.1 Scalar basis functions

The routines for manipulating polynomials are described in their own section. Here we deal with the local properties of continuous splines and their representation. Mainly we use antiderivatives of Legendre polynomials to ensure the continuity.

In one dimension this is not a problem at all, we just use the definition of the antiderivatives of Legendre polynomials at a reference element, e.g.

$$\mathcal{L}_0(x) = (1-x)/2,\ \mathcal{L}_1(x) = (1+x)/2,\ \mathcal{L}_n(x) = (L_n(x) - L_{n-2})/(2n-1) = \int_{-1}^{x} L_{n-1}(y)\,dy$$

where $L_n(x)$ is a Legendre-polynomial defined by

$$L_0(x) = 1,\ L_1(x) = x,\ (n+1)L_{n+1}(x) = (2n+1)xL_n(x) - nL_{n-1}(x)$$

The antiderivatives of Legendre polynomials have the important and simplifying property

$$\mathcal{L}_n(\pm 1) = 0, \quad n \geq 2$$

In two dimensions we have to distinguish two cases, i.e. two reference elements. First, we have the rectangle $Q = [-1,1]^2$. Here we can use tensor products of antiderivatives of Legendre polynomials as base functions, i.e.

$$\phi_{k,l}(x,y) = \mathcal{L}_k(x)\mathcal{L}_l(y), \quad 0 \leq k \leq p_x,\ 0 \leq l \leq p_y$$

If we take at look at this in matrix form we obtain

$(0p_y)$ $\uparrow$ $(0l)$		
01	11	
00	10	$(k0) \rightarrow (p_x0)$

We can read this as follows:

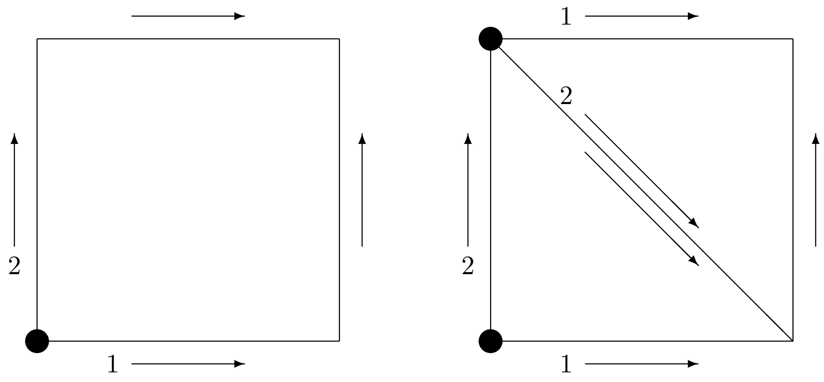
ϕ_{00}		lower left corner
ϕ_{10}		lower right corner
ϕ_{01}		upper left corner
ϕ_{10}		upper right corner
ϕ_{k0}	$(k \geq 2)$	lower edge functions
ϕ_{k1}	$(k \geq 2)$	upper edge functions
ϕ_{0l}	$(l \geq 2)$	left edge functions
ϕ_{1l}	$(l \geq 2)$	right edge functions
ϕ_{kl}		inner (bubble) functions

Second, we have the triangle $T = \{(x,y) : 0 \leq y \leq 1-x, 0 \leq x \leq 1\}$. Here we have the difficulties that we don't use the interval $[0,1]$ instead of the standard interval $[-1,1]$ as before, therefore we have to use the transformed polynomials $\tilde{\mathcal{L}}_k(x) = \mathcal{L}_k(2x-1), x \in [0,1]$, and the main difficulty: We can not use tensor products anymore to construct continuous splines. But we have to ensure that our splines at triangles can be connected easily with splines defined at rectangles. We achieve this goal by using the polynomials $\bar{\mathcal{L}}_k(x) = \tilde{\mathcal{L}}_k(x)/(1-x)$, $k \geq 2$.

This leads to the definitions

$\phi_{00}(x,y) = 1 - x - y$	lower left corner
$\phi_{10}(x,y) = x$	lower right corner
$\phi_{01}(x,y) = y$	upper left corner
$\phi_{k0}(x,y) = \tilde{\mathcal{L}}_k(x) - y\bar{\mathcal{L}}_k(x)$, $2 \le k \le p$	lower edge
$\phi_{0l}(x,y) = \tilde{\mathcal{L}}_l(y) - x\bar{\mathcal{L}}_l(y)$, $2 \le l \le p$	left edge
$\phi_{k1}(x,y) = \bar{\mathcal{L}}_{k+1}(x)y$, $1 \le k \le p-1$	right edge
$\phi_{k,l}(x,y) = \bar{\mathcal{L}}_{k+1}(x)\bar{\mathcal{L}}_l(y)(1-x-y)$, $1 \le k$, $2 \le l$, $k+l \le p$	inner (bubble) functions

For the construction of continuous splines we have to take into account the orientation of the edge functions



10.2 Vector-valued basis functions

10.2.1 Raviart-Thomas elements RT_k on squares

$Q_{l,m}$ all polynomials of degree $\le l$ in x-direction and of degree $\le m$ in y-direction. $p_k(e)$ all polynomials of degree $\le k$ on edge e. Raviart-Thomas space on squares

$$RT_k(K) := Q_{k+1,k} \times Q_{k,k+1}, \quad \dim RT_k(K) = 2(k+1)(k+2)$$

First we can construct a basis for $RT_k(K)$, i.e., we have $RT_k(K) = \text{span}\{\vec{\psi}_i, i = 1, \ldots, 2(k+1)(k+2)\}$ with

$$\vec{\psi}_i(x,y) := \begin{cases} \binom{1}{0} x^r y^s (0 \le r \le k+1, 0 \le s \le k) & \text{if } i = 1, \ldots, (k+1)(k+2) \\ \binom{0}{1} x^r y^s (0 \le r \le k, 0 \le s \le k+1) & \text{if } i = (k+1)(k+2)+1, \ldots, 2(k+1)(k+2) \end{cases} \tag{2}$$

For the construction of a $H(\text{div}, \Omega)$ conforming space we have to ensure continuity of the normal component across element edges. This can be achieved by defining moments m_i, and basis functions $\vec{\phi}_i$ which are orthonormal with respect to this moments. Let

$$m_i(\vec{\phi}) := \begin{cases} \int_{e_0} \vec{\phi} \cdot n_0 p_r \, ds (p_r \in p_k(e_0), \quad 0 \le r \le k) & \text{if } i = 1, \ldots, k+1 \\ \int_{e_1} \vec{\phi} \cdot n_1 p_r \, ds (p_r \in p_k(e_1), \quad 0 \le r \le k) & \text{if } i = k+2, \ldots, 2k+2 \\ \int_{e_2} \vec{\phi} \cdot n_2 p_r \, ds (p_r \in p_k(e_2), \quad 0 \le r \le k) & \text{if } i = 2k+3, \ldots, 3k+3 \\ \int_{e_3} \vec{\phi} \cdot n_3 p_r \, ds (p_r \in p_k(e_3), \quad 0 \le r \le k) & \text{if } i = 3k+4, \ldots, 4k+4 \\ \int_K \vec{\phi} \binom{1}{0} x^r y^s (0 \le r \le k-1, 0 \le s \le k) & \text{if } i = 4k+5, \ldots, (k+1)(k+4) \\ \int_K \vec{\phi} \binom{0}{1} x^r y^s (0 \le r \le k, 0 \le s \le k-1) & \text{if } i = (k+1)(k+4)+1, \ldots, 2(k+1)(k+2) \end{cases} \tag{3}$$

Then, the basis functions $\vec{\phi}_i$ have to satisfy the condition

$$m_j(\vec{\phi}_i) = \delta_{ij}$$

We can represent $\vec{\phi}_i$ with respect to $\vec{\psi}_l$, i.e.

$$\vec{\phi}_i = \sum_{l=1}^{2(k+1)(k+2)} a_{il}\vec{\psi}_l.$$

This leads to the following linear system

$$m_j(\vec{\phi}_i) = \sum_{l=1}^{2(k+1)(k+2)} a_{il} m_j(\vec{\psi}_l) = \delta_{ij}$$

If we define $b_{jl} := m_j(\vec{\psi}_l)$, then we have

$$(a_{il})_{il} = ((b_{jl})_{jl})^{-1}$$

For computational efficiency we later extend the coefficients a_{il} with respect to ψ_l to a basis of $(Q_{k+1,k+2}(K))^2$, which eliminates the need to distinguish several special cases in the evaluation of ϕ_i.

10.2.2 Raviart-Thomas elements RT_k on triangles

$P_k(K)$ all polynomials of total degree $\leq k$ on triangle K. $\tilde{P}(K)$ all polynomials of total degree $= k$ on triangle K. $p_k(e)$ all polynomials of degree $\leq k$ on edge e. Raviart-Thomas space on triangles

$$RT_k(K) := \{P_k(K)^2 + \vec{x}\tilde{P}_k(K)\}, \quad k \geq 0, \quad \dim RT_k = (k+1)(k+3)$$

First we can construct a basis for $RT_k(K)$, i.e., we have $RT_k(K) = \text{span}\{\vec{\psi}_i, i = 1, \ldots, (k+1)(k+3)\}$ with

$$\vec{\psi}_i(x,y) := \begin{cases} \binom{1}{0} x^r y^s (0 \leq r+s \leq k) & \text{if } i = 1, \ldots, (k+1)(k+2)/2 \\ \binom{0}{1} x^r y^s (0 \leq r+s \leq k) & \text{if } i = (k+1)(k+2)/2+1, \ldots, (k+1)(k+2) \\ \begin{pmatrix} x \\ y \end{pmatrix} x^r y^{k-r} (0 \leq r \leq k) & \text{if } i = (k+1)(k+2)+1, \ldots, (k+1)(k+3) \end{cases} \tag{4}$$

For the construction of a $H(\text{div}, \Omega)$ conforming space we have to ensure continuity of the normal component across element edges. This can be achieved by defining moments m_i, and basis functions $\vec{\phi}_i$ which are orthonormal with respect to this moments. Let

$$m_i(\vec{\phi}) := \begin{cases} \int_{e_0} \vec{\phi} \cdot n_0 p_r \, ds (p_r \in p_k(e_0), \quad 0 \leq r \leq k) & \text{if } i = 1, \ldots, k+1 \\ \int_{e_1} \vec{\phi} \cdot n_1 p_r \, ds (p_r \in p_k(e_1), \quad 0 \leq r \leq k) & \text{if } i = k+2, \ldots, 2k+2 \\ \int_{e_2} \vec{\phi} \cdot n_2 p_r \, ds (p_r \in p_k(e_2), \quad 0 \leq r \leq k) & \text{if } i = 2k+3, \ldots, 3k+3 \\ \int_K \vec{\phi} \binom{1}{0} x^r y^s (0 \leq r+s \leq k-1) & \text{if } i = 3k+4, \ldots, 3k+3+k(k+1)/2 \\ \int_K \vec{\phi} \binom{0}{1} x^r y^s (0 \leq r+s \leq k-1) & \text{if } i = 3k+4+k(k+1)/2, \ldots, (k+1)(k+3) \end{cases} \tag{5}$$

Then, the basis functions $\vec{\phi}_i$ have to satisfy the condition

$$m_j(\vec{\phi}_i) = \delta_{ij}$$

We can represent $\vec{\phi}_i$ with respect to $\vec{\psi}_l$, i.e.

$$\vec{\phi}_i = \sum_{l=1}^{(k+1)(k+3)} a_{il}\vec{\psi}_l.$$

This leads to the following linear system

$$m_j(\vec{\phi}_i) = \sum_{l=1}^{(k+1)(k+3)} a_{il} m_j(\vec{\psi}_l) = \delta_{ij}$$

If we define $b_{jl} := m_j(\vec{\psi}_l)$, then we have

$$(a_{il})_{il} = ((b_{jl})_{jl})^{-1}$$

For computational efficiency we later extend the coefficients a_{il} with respect to ψ_l to a basis of $(P_{k+1}(K))^2$, which eliminates the need to distinguish several special cases in the evaluation of ϕ_i.

10.2.3 Nedelec elements ND_k on cubes

$Q_{l,m,n}$ all polynomials of degree $\leq l$ in x-direction, of degree $\leq m$ in y-direction and of degree $\leq n$ in z-direction. Nedelec space on cubes

$$ND_k(Q) := \{u \,:\, u_1 \in Q_{k-1,k,k}, u_2 \in Q_{k,k-1,k}, u_2 \in Q_{k,k,k-1}\}, \dim ND_k = 3k(k+1)^2$$

First we have to construct a basis for $ND_k(Q)$, i.e., we have $ND_k(Q) = \text{span}\{\vec{\psi}_i, i = 1, \ldots, 3k(k+1)^2\}$ with

$$\vec{\psi}_i(x,y,z) := \begin{cases} \begin{pmatrix}1\\0\\0\end{pmatrix} x^r y^s z^t (0 \leq r \leq k-1, 0 \leq s \leq k, 0 \leq t \leq k) & \text{if } i = 1, \ldots, k(k+1)^2 \\ \begin{pmatrix}0\\1\\0\end{pmatrix} x^r y^s z^t (0 \leq r \leq k, 0 \leq s \leq k-1, 0 \leq t \leq k) & \text{if } i = k(k+1)^2+1, \ldots, 2k(k+1)^2 \\ \begin{pmatrix}0\\0\\1\end{pmatrix} x^r y^s z^t (0 \leq r \leq k, 0 \leq s \leq k, 0 \leq t \leq k-1) & \text{if } i = 2k(k+1)^2+1, \ldots, 3k(k+1)^2 \end{cases} \tag{6}$$

For the construction of a $H(\text{curl}, \Omega)$ conforming space we have to ensure continuity of the tangential components across element faces. This can be achieved by defining moments m_i, and basis functions $\vec{\phi}_i$ which are orthonormal with respect to this moments. Let

$$m_i(\vec{\phi}) := \begin{cases} \int_e \vec{\phi} \cdot \vec{t} q \, ds & \text{if } q \in p_{k-1}(e) \\ \int_f \vec{\phi} \times \vec{n} q \, df & \text{if } q \in Q_{k-2,k-1} \times Q_{k-1,k-2} \\ \int_Q \vec{\phi} \cdot q \, dx & \text{if } q \in Q_{k-1,k-2,k-2} \times Q_{k-2,k-1,k-2} \times Q_{k-2,k-2,k-1} \end{cases} \tag{7}$$

Then, the basis functions $\vec{\phi}_i$ have to satisfy the condition

$$m_j(\vec{\phi}_i) = \delta_{ij}$$

We can represent $\vec{\phi}_i$ with respect to $\vec{\psi}_l$, i.e.

$$\vec{\phi}_i = \sum_{l=1}^{3k(k+1)^2} a_{il} \vec{\psi}_l.$$

This leads to the following linear system

$$m_j(\vec{\phi}_i) = \sum_{l=1}^{3k(k+1)^2} a_{il} m_j(\vec{\psi}_l) = \delta_{ij}$$

If we define $b_{jl} := m_j(\vec{\psi}_l)$, then we have

$$(a_{il})_{il} = ((b_{jl})_{jl})^{-1}$$

10.2.4 Nedelec elements ND_k on tetrahedrals

11 Polynomials

Most of the subroutines for use and manipulation of polynomials are contained in the module poly located in the file fox/poly.f90. The module polydat contains some global data structures, created to avoid the recomputation of polynomial coefficients.

Most important is the definition of the polynomial basis functions. They are declared by the integer parameter `ptyp`. In the moment the following values for `ptyp` are allowed

ptyp	basis
0	Monomials
1	Legendre polynomials
2	Tschebyscheff polynomials
3	Antiderivatives of Legendre polynomials
4	Orthonormal Legendre polynomials
5	Lagrange polynomials (uniform nodes)
6	Lagrange polynomials (arbitrary nodes)
10	Raviart-Thomas (RT) polynomials
13	Nedelec (ND) polynomials
14	Traces of Nedelec (TND) polynomials
15	PEERS polynomials

The type of polynomials for a given `ptyp` can be printed in clear text to standard output by `printpoly` where `fin` is an error code if a wrong/non-existent type is chosen.

```
subroutine printpoly(ptyp,fin)
integer :: ptyp,fin
```

Other subroutines recommended for public use are the following:

Print the internal one-dimensional polynomial coefficients of order i to channel out

```
subroutine printpmi(i,out)
use polydat, only: npmi,pmi
integer, intent(in) :: i,out
```

Evaluate base functions up to degree p at point x

```
subroutine ptab1(x,p,ptyp,tab)
integer, intent(in) :: p,ptyp
real(kind=dp), intent(in)  :: x
real(kind=dp), intent(out) :: tab(0:max(1,p))
```

Evaluate derivatives of base functions up to degree p at point x

```
subroutine pstab1(x,p,ptyp,tab)
use polydat, only: pmi,npmi
integer, intent(in) :: p,ptyp
real(kind=dp), intent(in)  :: x
real(kind=dp), intent(out) :: tab(0:max(1,p))
```

Evaluate second derivatives of base functions up to degree p at point x

```
subroutine psstab1(x,p,ptyp,tab)
use polydat, only: npmi,pmi
integer, intent(in) :: p,ptyp
real(kind=dp), intent(in)  :: x
real(kind=dp), intent(out) :: tab(0:max(1,p))
```

Evaluate base functions up to degree (px,py) at point (x,y)

```
subroutine ptab2n(x,y,px,py,typ,ptyp,tab)
use polydat, only: tslegs
real(kind=dp), intent(in) :: x,y
integer, intent(in) :: px,py,typ,ptyp
real(kind=dp), intent(out) :: tab(0:*)
```

Evaluate derivatives of base functions up to degree (px,py) at point (x,y), tab1 contains the x-derivative and tab2 the y-derivative

```
subroutine pstab2n(x,y,px,py,typ,ptyp,tab1,tab2)
real(kind=dp), intent(in) :: x,y
integer, intent(in) :: px,py,typ,ptyp
real(kind=dp), dimension(0:*), intent(out) :: tab1,tab2
```

Evaluate base functions up to degree (px,py,pz) at point (x,y,z)

```
subroutine ptab3(x,y,z,px,py,pz,typ,ptyp,tab)
real(kind=dp), intent(in) :: x,y,z
integer, intent(in) :: px,py,pz,typ,ptyp
real(kind=dp), intent(out) :: tab(0:*)
```

Evaluate derivatives of base functions up to degree (px,py,pz) at point (x,y,z), tab1 contains the x-derivative, tab2 the y-derivative and tab3 the z-derivative

```
subroutine pstab3(x,y,z,px,py,pz,typ,ptyp,tab1,tab2,tab3)
real(kind=dp), intent(in) :: x,y,z
integer, intent(in) :: px,py,pz,typ,ptyp
real(kind=dp), dimension(0:*), intent(out) :: tab1,tab2,tab3
```

Interpolates (fi,xi) with n+1 nodes using monomials

```
subroutine ipol1(fi,xi,n,coeff)
integer, intent(in) :: n
real(kind=dp), intent(in) :: fi(0:n),xi(0:n)
real(kind=dp), intent(out) :: coeff(0:n)
```

Compute all Lagrange base functions with polynomial degrees px on a rectangle or triangle using a tensor product node distribution

```
subroutine ipol2(fi,xi1,xi2,typ,px,coeff)
integer, intent(in) :: px(0:1),typ
real(kind=dp), dimension(0:*), intent(in) :: xi1,xi2
real(kind=dp), dimension(0:mo,0:mo), intent(in) :: fi
real(kind=dp), dimension(0:mo,0:mo), intent(out) :: coeff
```

Compute the coefficients of the i-th Lagrange polynomial with n+1 nodes

```
subroutine lagrangei(xi,n,i,lagpol)
integer, intent(in) :: n,i
real(kind=dp), intent(in) :: xi(0:n)
real(kind=dp), intent(out) :: lagpol(0:n)
```

Compute uniform nodes for continuous (ctyp=1) or discontinuous (ctyp=0) interpolation on an interval [a,b].

```
subroutine ipnodes(n,ctyp,a,b,xi)
integer, intent(in) :: n,ctyp
real(kind=dp), intent(in) :: a,b
real(kind=dp), intent(out) :: xi(0:n)
```

Transformation from monomial base to base 'ptyp' on an interval.

```
subroutine ctpol1(coeff,p,ptyp)
integer, intent(in) :: p,ptyp
real(kind=dp), intent(inout) :: coeff(0:max(1,p))
```

Transformation from monomial base to base 'ptyp' on a rectangle or a triangle.

```
subroutine ctpol2(coeff,px,typ,ptyp)
integer, intent(in) :: px(0:1),typ,ptyp
integer, parameter :: fm=10
real(kind=dp), intent(inout) :: coeff(0:fm,0:fm)
```

Transformation of a polynomial with monomial base functions to a polynomial with Legendre polynomials as base functions.

```
subroutine ctleg1(coeff,p)
use polydat
integer, intent(in) :: p
real(kind=dp), intent(inout) :: coeff(0:p)
```

Transformation of Legendre-coefficients to Antiderivatives of Legendre-polynomials

```
subroutine cbfleg1(coeff,p)
integer, intent(in) :: p
real(kind=dp), intent(inout) :: coeff(0:p)
```

$$n_i = \prod_{\substack{j=0\\j\neq i}}^{p} (x_i - x_j)$$

$$L_i(x) = n_i^{-1} \prod_{\substack{j=0\\j\neq i}}^{p} (x - x_j)$$

$$L_i'(x) = n_i^{-1} \sum_{\substack{j=0\\j\neq i}}^{p} \prod_{\substack{k=0\\i\neq k, j\neq k}}^{p} (x - x_k)$$

$$L_i''(x) = n_i^{-1} \sum_{\substack{j=0\\j\neq i}}^{p} \sum_{\substack{k=0\\i\neq k, j\neq k}}^{p} \prod_{\substack{l=0\\i\neq l, j\neq l, k\neq l}}^{p} (x - x_l)$$

12 Interfaces to the integral libraries

12.1 2D-BEM

The source code files of the integral libraries are called liblap2.f90, liblame2.f90, libhelm2.f90, libint2.f90 and libgint2.f90. The file liblap2.f90 contains the implementation of the Galerkin elements for the Laplacian, the file liblame2.f90 contains the implementation of the Galerkin elements for the Lamé-operator and the file libhelm2.f90 contains the implementation of the Galerkin elements for the Helmholtz-operator. Each of these files can be used independently of the others, but all files need the file libint2.f90 which contains the elementary integrals and the file libgint2.f90 which contains the generic numerical integration subroutines used by the other files.

Before using any subroutines in which numerical integrations involved we have to call the subroutine `initgauss12` with a subroutine as argument which calculates for a given number of nodes the values of nodes and weights. `initgauss12` will generate internally a table for the number of nodes from 1 to 32 for further use.

```
subroutine initgauss12(gauss)
external gauss
```

An example for such a gauss-routine is given in the following by using NAG-routines.

```
subroutine gauss(gqn,gqk,gqw)
integer gqn,itype,ifail
real gqk(1:gqn),gqw(1:gqn)
real a,b
external d01baz,d01bbf
a=-1.0
b= 1.0
itype=0
ifail=1
call d01bbf(d01baz,a,b,itype,gqn,gqw,gqk,ifail)
if (ifail.ne.0) then
```

```
  ifail=0
  call d01bcf(itype,a,b,a,b,gqn,gqw,gqk,ifail)
 end if
 end
```

Internal coefficients controlling the analytical or numerical integrations are set by

```
subroutine setc12(mode,ivalue,rvalue)
integer mode,ivalue
real rvalue
```

where `ivalue` is an integer parameter and `rvalue` a real parameter. `mode`=0 sets the kind of integration, i.e. ivalue=0 means analytical integration, ivalue=1 means doing the outer integration numerically and the inner integration analytically and ivalue=2 means all integrations numerically. The number of Gauss nodes `gqna` for the outer integration is set by mode=1 and the number of Gauss nodes `gqnb` for the inner integration by mode=2. The number of Gauss nodes `gqnc` for the potential integration is set by mode=3. `sigma` and `ijn` give the grading parameter and number of levels for the geometrical refined quadrature. `mu` describes the increasing of the number of quadrature points. `farreg` gives the boundary of the farfield where the usual quadrature rule is used without grading. By using mode=-1 the standard values are set; this subroutine is automatically called by `initgauss12`.

mode	ivalue	rvalue
0	mtyp	
1	gqna	
2	gqnb	
3	gqnc	
4	ijn	sigma
5		farreg
6		mu

We can use the subroutine `getc12` to obtain the current values of the parameters.

```
subroutine getc12(mode,ivalue,rvalue)
integer mode,ivalue
real rvalue
```

In the following the interfaces and data formats of all user usable subroutines are presented. The analogous subroutines of liblap2.f90, liblame2.f90 and libhelm2.f90 share a common interface. Therefore we need only to state a generic interface. The actual names of the subroutines are build by using the prefix lap-, lame- or helm-.

We usually don't state the whole boundary, but we compute all integrals for two boundary elements, from which all other terms can be inferred. There are two ways of describing a boundary element Γ_i. First we can state the start point a_i and the end point e_i of Γ_i, or we can state the mid point $m_i = (e_i + a_i)/2$ and the difference vector from end point and mid point $d_i = (e_i - a_i)/2$.

Most operators demand the definition of some constants.

In the case of the Lamé-equation the Lamé-coefficients are defined in the program by using

```
subroutine initlame(lambda,mu)
real(kind=dp), intent(in) :: lambda,mu
```

before any other subroutine is called. All internal constants which are needed by 'liblame2.f90' are defined by this subroutine. As long as the coefficients don't change it is sufficient to call the subroutine only once.

In the case of the Helmholtz-equation the wavenumber 'kw' and the number of terms of the series expansion 'kn' are defined by

```
 subroutine inithelm(kw,kn)
 real(kind=dp), intent(in) :: kw
 integer, intent(in) :: kn
```

The test and ansatz functions are given by

$$\varphi_{i,k}(\vec{x}) = \begin{cases} \left(\frac{d_i(\vec{x}-m_i)}{d_i^2}\right)^k & ,\vec{x} \in \Gamma_i \\ 0 & ,\text{otherwise} \end{cases} \tag{8}$$

The Galerkin elements are usually given by

$$\begin{aligned} F_{kl} &= \int_{\Gamma_0} \varphi_{0,k}(\vec{x}) \int_{\Gamma_1} \varphi_{1,l}(\vec{y}) F(\vec{x},\vec{y})\, ds_y\, ds_x \\ &= |d_0|\,|d_1| \int_{-1}^{1}\int_{-1}^{1} t_x^k t_y^l F(d_0 * t_x + m_0, d_1 * t_y + m_1)\, dt_y\, dt_x \end{aligned}$$

The values of the Galerkin elements can be real numbers, complex numbers or matrices. They are always stored in the same data structure.

```
 integer, parameter :: fm=40
 real(kind=dp) :: f(0:fm,0:fm,0:nt-1)
```

'fm' denotes the highest allowed polynomial degree which can be used. 'nt' stands for the number of components of the Galerkin element. For example for the Lamé equation we have four components, i.e. nt=4. We can access the Galerkin element $F_{kl}^{(r,s)}$, $r, s = 0, 1$ by $f(k, l, 2 * r + s)$

The Galerkin elements are computed by `-integmd`, which uses midpoint, difference vector and normal vector for the boundary elements 0 and 1. The kind of integration to be performed is defined in advance by `setc12(0,ivalue,rvalue)`.

The highest polynomial degrees on boundary elements 0 and 1 must be given for the single layer potential by pv0 and pv1, for the double layer potential by pk0 and pk1 and for the hypersingular operator by pw0 and pw1. Note that not the hypersingular operator itself is computed, but a partial integrated form, which can be used completely analogous.

```
 subroutine -integmd(m0,d0,n0,m1,d1,n1,pv0,pv1,pk0,pk1,pw0,pw1,fv,fk,fw)
! midpoint, diffvec and normal of boundary element 0
 real(kind=dp), intent(in) :: m0(0:1),d0(0:1),n0(0:1)
! midpoint, diffvec and normal of boundary element 1
 real(kind=dp) :: m1(0:1),d1(0:1),n1(0:1)

 integer, intent(in) :: pv0,pv1,pk0,pk1,pw0,pw1
 integer, parameter :: fm=40
 real(kind=dp), dimension(0:fm,0:fm,0:nt-1), intent(inout) :: fv,fk,fw
```

We can compute the Galerkin elements for the adjoint double layer potential, which are usually computed by taking the transposed matrix of the double layer potential in the same way .

```
 subroutine -integmdks(m0,d0,n0,m1,d1,n1,pks0,pks1,fks)
! midpoint, diffvec and normal of boundary element 0
 real(kind=dp), intent(in) :: m0(0:1),d0(0:1),n0(0:1)
! midpoint, diffvec and normal of boundary element 1
 real(kind=dp), intent(in) :: m1(0:1),d1(0:1),n1(0:1)

 integer, intent(in) :: pks0,pks1
 integer, parameter :: fm=40
 real(kind=dp), intent(inout) :: fks(0:fm,0:fm,0:nt-1)
```

For the computation of the right hand side we need the integrals over a general function multiplied with the test-functions.

$$I_k = \int_{\Gamma_0} f(\vec{x})\varphi_{0,k}(\vec{x})\, ds_x$$

```
      subroutine -id(a0,e0,n0,f,n,gqn,gqk,gqw,iid)
c startpoint,endpoint and normal of the 0-th boundary element
      real a0(0:1),e0(0:1),n0(0:1)
c external subroutine with parameters x,n,f
c x(0:1): source point
c n(0:1): normal direction of the boundary in x
c f(0:1): value of the function in x
      external f
c maximal polynomial degree
      integer n
c quadrature
      integer gqn
      real gqk(1:gqn),gqw(1:gqn)
c table of the identity
      integer fm
      parameter (fm=20)
      real iid(0:fm,0:nt-1)
```

For the computation of the right hand side we also need

$$F_k = \int_{\Gamma_0} f(\vec{x}) \int_{\Gamma_1} F(\vec{x},\vec{y})\varphi_{1,k}(\vec{y})\, ds_y\, ds_x$$

which can be computed by using the following subroutines which use the same parameter list, where 'op' stands for v (weakly singular), k (double layer), ks (adjoint double layer) and w (hypersingular), e.g. lamevpot.

```
      subroutine -op-pot(a0,e0,n0,a1,e1,n1,f,n,op-kr)
c startpoint, endpoint and normal of the 0-boundary element
      real a0(0:1),e0(0:1),n0(0:1)
c startpoint, endpoint and normal of the 1-boundary element
      real a1(0:1),e1(0:1),n1(0:1)
```

```
c external subroutine with parameters x,n,f
c x(0:1): source point
c n(0:1): normal direction of the boundary in x
c f(0:1): value of the function in x
      external f
c highest polynomial degree
      integer n

c table of the values
      integer fm
      parameter (fm=20)
      real op-kr(0:fm,0:nt-1)
```

These subroutines also exist using midpoint and difference vector

```
      subroutine -op-potmd(m0,d0,n0,m1,d1,n1,f,n,op-kr)
```

The values of the potential at a given point

$$F_k(\vec{x}) = \int_{\Gamma_1} F(\vec{x},\vec{y})\varphi_{1,k}(\vec{y})\,ds_y$$

can be computed by using the following subroutines which use the same parameter list.

```
      subroutine -op-potll(x,n0,m1,d1,n1,p1,op-kr)
c source point
      real x(0:1)
c midpoint, diff-vector and normal direction of the boundary element 1
      real m1(0:1),d1(0:1),n1(0:1)
c normal direction in the source point x
      real n0(0:1)
c highest polynomial degree
      integer p1
c table of the values
      integer fm
      parameter (fm=20)
      real op-kr(0:fm,0:nt-1)
```

The value of the potential can be calculated by summing up the results of '-op-potll' over the whole boundary.

The kernel functions $F(x,y)$ of the integral operators are given by

```
      subroutine  -op-kern(d,n0,n1,kern)
      real d(0:1),n0(0:1),n1(0:1),kern(0:nt-1)
```

d(0:1) is the difference of x and y. n0 denotes the normal direction of the boundary in x and n1 in y. kern(0:nt-1) is the value of the kernel function with 'nt' components.

For the implementation of integral equations of the second kind we have to test a base function with another base function

$$I_{kl} = \langle I\varphi_{0,k},\varphi_{1,l}\rangle = \int_{\Gamma_0} \varphi_{0,k}(\vec{x}) \int_{\Gamma_1} \varphi_{1,l}(\vec{y})\,ds_y\,ds_x$$

This is given by the subroutine 'genidgal'.

```
      subroutine genidgal(m0,d0,n0,m1,d1,n1,p0,p1,idkl)
c midpoint, diffvec and normal of boundary element 0
      real m0(0:1),d0(0:1),n0(0:1)
c midpoint, diffvec and normal of boundary element 1
      real m1(0:1),d1(0:1),n1(0:1)
      integer p0,p1
      integer fm
      parameter (fm=20)
      real   idkl(0:fm,0:fm)
```

Various subroutines located in 'libgint2.f90' perform the numerical quadratures.

Numerical computation of the potential for monomials, given only the kernel function

```
      subroutine genpotln(x,n0,m1,d1,n1,py,potkr,kernf,nt,opt)
c source point of the potential
      real x(0:1)
c normal of boundary element 0
      real n0(0:1)
c midpoint, diffvec and normal of boundary element 1
      real m1(0:1),d1(0:1),n1(0:1)
c maximal polynomial degree
      integer py,nt
      external kernf

c tabler of the values of the potential
      integer fm
      parameter (fm=20)
      real potkr(0:fm,0:nt-1)
```

Numerical computation of the potential for arbitrary functions, given only the kernel function

```
      subroutine genfpotln(x,n0,m1,d1,n1,
     *                    potw,kernf,fkt,emult,nt,opt)
c source point of the potential
      real x(0:1)
c normal of boundary element 0
      real n0(0:1)
c midpoint, diffvec and normal of boundary element 1
      real m1(0:1),d1(0:1),n1(0:1)
      integer nt
c kernel function, arbitray function, elementary multiplication
      external kernf,fkt,emult

c table of the potential values
      real potw(0:nt-1)
```

Numerical computation of the Galerkin elements, given only the kernel function

```
      subroutine gengalln(m0,d0,n0,m1,d1,n1,p0,p1,f,kernf,nt,opt)
c midpoint, diffvec and normal of boundary element 0
```

```
      real m0(0:1),d0(0:1),n0(0:1)
c midpoint, diffvec and normal of boundary element 1
      real m1(0:1),d1(0:1),n1(0:1)
      integer p0,p1,nt,opt
      integer fm
      parameter (fm=20)
      real   f(0:fm,0:fm,0:nt-1)
```

Numerical computation of the outer integration, given the analytical computed potential

```
      subroutine gengall(m0,d0,n0,m1,d1,n1,p0,p1,f,potll,nt,opt)
c midpoint, diffvec and normal of boundary element 0
      real m0(0:1),d0(0:1),n0(0:1)
c midpoint, diffvec and normal of boundary element 1
      real m1(0:1),d1(0:1),n1(0:1)
      integer p0,p1
      integer fm
      parameter (fm=20)
      real   f(0:fm,0:fm,0:nt-1)
      external potll
```

Numerical quadrature of a potential with a function

```
      subroutine genpot(m0,d0,n0,m1,d1,n1,f,potll,mult,
     *                  p1,potkr,nt,opt)
c midpoint, diffvec and normal of boundary element 0
      real m0(0:1),d0(0:1),n0(0:1)
c midpoint, diffvec and normal of boundary element 1
      real m1(0:1),d1(0:1),n1(0:1)
c external subroutine with parameters x,n,f
c x(0:1): source point
c n(0:1): normal direction in x
c f(0:1): value in x
      external f,potll,mult
c maximal polynomial degree
      integer p1
      integer nt,opt

c table of the integrated potential
      integer fm
      parameter (fm=20)
      real potkr(0:fm,0:nt-1)
```

Numerical quadrature of a numerical potential with a function

```
      subroutine genpotfln(m0,d0,n0,m1,d1,n1,f,kernf,mult,
     *                  p1,potkr,nt,opt)
c midpoint, diffvec and normal of boundary element 0
      real m0(0:1),d0(0:1),n0(0:1)
c midpoint, diffvec and normal of boundary element 1
      real m1(0:1),d1(0:1),n1(0:1)
c external subroutine with parameters x,n,f
```

```
c x(0:1): source point
c n(0:1): normal direction in x
c f(0:1): values in x
      external f,kernf,mult
c maximal polynomial degree
      integer p1
      integer nt,opt

c table of the integrated potential
      integer fm
      parameter (fm=20)
      real potkr(0:fm,0:nt-1)
```

Numerical quadrature of a double integral consisting of a singular kernel and two functions

```
      subroutine gengalfln(m0,d0,n0,m1,d1,n1,f0,f1,kernf,emult,
     *                 galr,nt,opt)
c midpoint, diffvec and normal of boundary element 0
      real m0(0:1),d0(0:1),n0(0:1)
c midpoint, diffvec and normal of boundary element 1
      real m1(0:1),d1(0:1),n1(0:1)
c external subroutine with parameters x,n,f
c x(0:1): source point
c n(0:1): normal direction in x
c f(0:1): values in x
      external f0,f1,kernf,emult
c maximal polynomial degree
      integer p1
      integer nt,opt

c table of the integral
      real galr(0:nt-1)
```

Computation of the identity where f is allowed to have a singular point

$$I_{kr} = \int_{a0}^{e0} f_r(x) \left(\frac{s_x - (e0 + a0)/2}{(e0 - a0)/2} \right)^k dx$$

```
      subroutine genid12(m0,d0,n0,p0,gqn,gqk,gqw,kid,fkt,nt,opt,sx)
c midpoint, diffvec and normal of boundary element 0
      real m0(0:1),d0(0:1),n0(0:1)
c function
      external fkt
c maximal polynomial degree
      integer p0
c quadrature
      integer gqn
      real gqk(1:gqn),gqw(1:gqn)
      integer nt,opt
c singular point
      real sx(0:1)
c table of the identity
      integer fm
      parameter (fm=20)
      real kid(0:fm,0:nt-1)
```

12.2 3D-BEM

The source code files of the integral libraries in the 3D-case are called liblap3.f90, liblame3.f90, libhelm3.f90, libint3.f90 and libgint3.f90. The file liblap3.f90 contains the implementation of the Galerkin elements for the Laplacian, the file liblame3.f90 contains the implementation of the Galerkin elements for the Lamé-operator and the file libhelm3.f90 contains the implementation of the Galerkin elements for the Helmholtz-operator. Each of this files can be used independently of the others, but all files need the file libint3.f90 which contains the elementary integrals and the file libgint3.f90 which contains generic numerical integration subroutines used by the other files.

Before using any subroutines in which numerical integrations are involved we have to call the subroutine `initgauss23` with a subroutine as argument, which calculates for a given number of nodes the values of the nodes and weights. The subroutines `initgauss23`, `setc23` and `getc23` correspond to the subroutines `initgauss12`, `setc12` and `getc12` in the 2D-BEM case and share the same interface.

```
 subroutine -integ3(bx,dx1,dx2,nx,tx,by,dy1,dy2,ny,ty, &
 &                  pvx,pvy,pkx,pky,pwx,pwy, &
 &                  vklmn,kklmn,wklmn)
! Element type: 3=triangle, 4=rectangle
 integer, intent(in) :: tx,ty
! vector of the origin, boundaries, normal direction
 real(kind=dp), dimension(0:2), intent(in) :: bx,dx1,dx2,nx
 real(kind=dp), dimension(0:2), intent(in) :: by,dy1,dy2,ny
! polynomial degree
 integer, dimension(0:1), intent(in) :: px,pkx,pwx
 integer, dimension(0:1), intent(in) :: py,pky,pwy
! Galerkin matrices
 real(kind=dp),dimension(0:fm,0:fm,0:fm,0:fm,0:nt-1), intent(inout) :: vklmn,kklmn,wklmn
```

In the same way we can compute the Galerkin elements for the adjoint double layer potential, which are usually computed by taking the transposed matrix of the double layer potential.

```
 subroutine -integ3ks(bx,dx1,dx2,nx,tx,by,dy1,dy2,ny,ty, &
 &                    pkx,pky,ksklmn)
! Element type: 3=triangle, 4=rectangle
 integer, intent(in) :: tx,ty
! vector of the origin, boundaries, normal direction
 real(kind=dp), dimension(0:2), intent(in) :: bx,dx1,dx2,nx
 real(kind=dp), dimension(0:2), intent(in) :: by,dy1,dy2,ny
! polynomial degree
 integer, intent(in) :: pkx(0:1),pky(0:1)
! Galerkin matrix
 real(kind=dp), intent(inout) :: ksklmn(0:fm,0:fm,0:fm,0:fm,0:nt-1)
```

For the computation of the right hand side we need the integrals over a general function multiplied with the test-functions.

$$I_{kl} = \int_{\Gamma_0} f(\vec{x})\varphi_{0,kl}(\vec{x})\, ds_x$$

```
 subroutine genid23(bx,dx1,dx2,nx,tx,px,gqn,gqk,gqw,kid,fkt,nt)
! Element typ: 3=triangle, 4=rectangle
 integer, intent(in) tx
! vector of the origin, boundaries, normal direction
 real(kind=dp), dimension(0:2), intent(in) :: bx,dx1,dx2,nx
! polynomial degree
 integer, intent(in) :: px(0:1)
! quadrature
 integer, intent(in) :: gqn
 real(kind=dp), intent(in) :: gqk(1:gqn),gqw(1:gqn)
 integer, intent(in) :: nt
! table of the identity
 real(kind=dp), intent(inout) :: kid(0:fm,0:fm,0:nt-1)
```

For the computation of the right hand side we also need

$$F_{kl} = \int_{\Gamma_0} f(\vec{x}) \int_{\Gamma_1} F(\vec{x},\vec{y})\varphi_{1,kl}(\vec{y})\, ds_y\, ds_x$$

which can be computed by using the following subroutines which use the same parameter list, where 'op' stands for v (weakly singular), k (double layer), ks (adjoint double layer) and w (hypersingular), e.g. lamevpot.

```
 subroutine -op-pot(bx,dx1,dx2,tx,nx,by,dy1,dy2,ty,ny,f,py,op-kr)
! Element typ: 3=triangle, 4=rectangle
 integer, intent(in) :: tx,ty
! vector of the origin, boundaries, normal direction
 real(kind=dp), dimension(0:2), intent(in) :: bx,dx1,dx2,nx
 real(kind=dp), dimension(0:2), intent(in) :: by,dy1,dy2,ny
! polynomial degree
 integer, intent(in) :: py(0:1)
! table of the potential
 real(kind=dp), intent(inout) :: op-kr(0:fm,0:fm,0:*)
```

The values of the potential at a given point

$$F_{kl}(\vec{x}) = \int_{\Gamma_1} F(\vec{x},\vec{y})\varphi_{1,kl}(\vec{y})\, ds_y$$

can be computed by using the following subroutines which use the same parameter list.

```
 subroutine -op-potll(x,nx,by,dy1,dy2,ny,ty,py,op-kl)
!source point
 real(kind=dp), intent(in) :: x(0:2),nx(0:2)
! vector of the origin, boundaries, normal direction
 real(kind=dp), dimension(0:2), intent(in) :: by,dy1,dy2,ny
! Element typ: 3=triangle, 4=rectangle
 integer, intent(in) :: ty
```

```
! polynomial degree
 integer, intent(in) :: py(0:1)
! table of the potential
 real(kind=dp), intent(inout) :: op-kl(0:fm,0:fm,0:*)
```

The value of the potential can be calculated by summing up the results of '-op-potll' over the whole boundary.

The kernel functions $F(x, y)$ of the integral operators are given by

```
 subroutine  -op-kern(d,n0,n1,kern)
 real(kind=dp), intent(in)  :: d(0:2),n0(0:2),n1(0:2)
 real(kind=dp), intent(out) :: kern(0:nt-1)
```

d(0:2) is the difference of x and y. n0 denotes the normal direction of the boundary in x and n1 in y. kern(0:nt-1) is the value of the kernel function with 'nt' components.

12.3 2D-FEM

In case of the 2D-FEM Galerkin elements, we have to distinguish between Galerkin matrices with test and ansatz spaces which are of the same kind

```
 subroutine type-2-op(gm,p1,p2,p3,p4,px,py,typ,ptyp)
 use poly
 real(kind=dp), intent(inout) :: gm(0:*)
 real(kind=dp), dimension(0:1), intent(in) :: p1,p2,p3,p4
 integer, intent(in) :: px,py,typ,ptyp
```

and with different test and ansatz spaces.

```
 subroutine type-2-op(gm,p1,p2,p3,p4,px1,py1,px2,py2,typ,ptyp1,ptyp2)
 real(kind=dp), intent(inout) :: gm(0:*)
 real(kind=dp), dimension(0:1), intent(in) :: p1,p2,p3,p4
 integer, intent(in) :: px1,py1,px2,py2,typ,ptyp1,ptyp2
```

12.4 3D-FEM

```
 subroutine type-3-op(gm,nodes,px,py,pz,typ,ptyp)
 real(kind=dp), intent(inout) :: gm(0:*)
 real(kind=dp), intent(in) :: nodes(0:2,0:7)
 integer, intent(in) :: px,py,pz,typ,ptyp

 subroutine type-3-op(gm,nodes,px1,py1,pz1,px2,py2,pz2,typ,ptyp1,ptyp2)
 real(kind=dp), intent(inout) :: gm(0:*)
 real(kind=dp), intent(in) :: nodes(0:2,0:7)
 integer, intent(in) :: px1,py1,pz1,px2,py2,pz2,typ,ptyp1,ptyp2
```

13 Interface of the batch control language

All programs of *maiprogs* read the commands from a file. The implemented commands are described in the user manual [3]. The subroutines and functions which are needed for implementing the interpretation of the commands are described in the following. We have to distinguish between the routines which open and close the files and the routines for reading the parameters of commands.

The first subroutine is `getopen` which reads the arguments of the command line of the program itself (–o, –o2, –f, –d, –i, –r) and then opens the command file and also initializes most of the data structures.

```
 subroutine getopen(eingabe0,ausgabe0,fin)
! only one call at program start for initialization
 integer, intent(inout) :: fin ! error code, emergency stop
 character(len=*), intent(in) :: eingabe0,ausgabe0
```

Here `eingabe0` is the name of the default input file, whereas `ausgabe0` is the name of the default output file. `fin` is a flag for emergency exit.

The second subroutine is `getline` which reads the next line from the input and tries to interpret the line as a standard command (#in, #out, #out2, #e, set, print, write2, do, continue, history, debug, #id., #ti, #time). If it recognizes no command the input line is returned to the calling program for further interpreting. It also controls the history buffer, do-loop control etc.

```
 subroutine getline(fin)
 integer, intent(inout) :: fin
```

The last subroutine is `getclose` which just closes input and output files.

```
 subroutine getclose()
```

Now the subroutines and functions for interpretation of the input lines are given. All commands which read from the input line use the pointer `inpp` internally.

`outset` looks whether a channel (nr.1 upto 10) has been opened for output, and if not, it opens the corresponding file.

```
 subroutine outset(chno,aus)
 integer, intent(in) :: akno
 integer, intent(out) :: aus
! create a new file for channel akno, allocated unit will be in aus
```

`rdint` reads an integer from the input line, an argument has to be present, otherwise an error message will be given.

```
 function rdint()
 integer :: rdint
```

`rdreal` reads a real number from the input line, an argument has to be present, otherwise an error message will be given.

```
function rdreal
real(kind=dp) :: rdreal
```

`xrdint` reads an integer from the input line and uses a default value `wert` if there is no more input or the symbol '-' is used.

```
function xrdint(wert)
integer :: xrdint
integer, intent(in) :: wert
```

`xrdreal` reads a real number from the input line and uses a default value `wert` if there is no more input or the symbol '-' is used.

```
function xrdreal(wert)
real(kind=dp) :: xrdreal
real(kind=dp), intent(in) :: wert
```

The most modern and comfortable way to access arguments for bcl-commands is to use the following set of commands `yparse`, `yrdint`, `yrdreal` and `yrdstr`. The subroutine `yparse` reads all arguments at once. This makes it possible to enclose the arguments in brackets and to separate them with colons, which was not possible before. Additionally, the arguments can be named and therefore given in arbitrary order.

```
subroutine yparse
```

The name of the optional argument is given in `str` and its default value in `value`.

```
function yrdint(str,value)
integer :: yrdint
integer, intent(in) :: value
character(len=*), intent(in) :: str
```

```
function yrdreal(str,value)
real(kind=dp) :: yrdreal
real(kind=dp), intent(in) :: value
character(len=*), intent(in) :: str
```

```
function yrdstr(str,value)
character(len=80) :: yrdstr
character(len=*), intent(in) :: str,value
```

The subroutine `setglobvar` allows to set global variables which can be used by bcl scripts. Here vt=0 means the variable is of integer type with value 'vi' and vt=1 means the variable is of real type with value 'vr'.

```
subroutine setglobvar(name,vt,vi,vr)
integer, intent(in) :: vt,vi
real(kind=dp), intent(in) :: vr
character(len=*), intent(in) :: name
```

The subroutine `getglobvar` allows to get global variables which are used by bcl scripts. Here vt=0 means the variable is of integer type with value 'vi' and vt=1 means the variable is of real type with value 'vr'. If 'vt=-1' the variable 'name' is not defined.

```
 subroutine getglobvar(name,vt,vi,vr)
 integer, intent(out) :: vt,vi
 real(kind=dp), intent(out) :: vr
 character(len=*) :: name
```

The following subroutines and functions are designed for internal use.

`ausset` looks whether a file has been opened for output, and if not, it opens the file.

```
 subroutine ausset(ausgabe,ausflag,aus)
 integer, intent(inout) :: ausflag
 integer, intent(in) :: aus
 character(len=*), intent(in) :: ausgabe
```

`iscmd` is a logical function tests for the next command in the input line and compares with `str`. The return value of `iscmd` is true after a command is recognized. Then the string `currentcmd` will be set to the value of `str` and the subroutine timing will be called.

```
 function iscmd(str)
 logical :: iscmd
 character(len=*), intent(in) :: str
```

Example 13.1 *This example shows the use of* `getopen`*,* `getline` *and* `getclose`*.*

```
! 'recin' is the name of the default input file
! the main loop starts here
! note: common batch commands are contained in the file bcl.f
!       and executed by the command getline

 call getopen('recin','recout',fin)

   10 call getline(fin)

          if (fin.eq.1) then
! done, don't do any more
 else if (iscmd('#px.')) then
! write mesh to output file
  ...
 else if (iscmd('mesh')) then
! initialisize the mesh
  ...

  call timelog ! write computing time to screen and/or logfile
 else
! error message
  call errcmd()
 end if
 if (fin .ne. 1) goto 10
```

```
call getclose()
```

14 Logging facilities

Messages (informational, warnings, errors) are written to the screen and/or to a log file. In the module 'error' there is the subroutine logmsg and the character variable msg containing the message.

```
subroutine logmsg(type,name,only)
character(len=*), intent(in), optional :: type,name,only
```

Supplying the optional argument type, we can define the type of the message, e.g. Error, Warning, Info. Supplying the optional argument name, we can use the name of the reporting procedure or command in the message. This has the advantage, that the reporting procedure can call first another subroutine for further processing, supplying the name, which finally leads to the call to logmsg. Using the optional argument, we can supercede the general configuration, writing only to 'Screen' or 'File'.

It is strongly advised to use the logging facilities for all kinds of structured output, i.e. for anything besides debugging outputs, so that the output can also be organized in a multi-core/multi-computer environment.

We have to note that the integral libraries are not covered by the logging facilities. The integral libraries are intended to be used also externally, therefore a close connection would be harmful for this purpose. In the next Fortran revision there will also be some kind of exception handling incorporated into the language. Then logmsg will evaluate the type argument for an extended action and error reporting.

The following subroutine `timelog` writes the consumed cpu time since the last command (or timer reset) to the screen and/or log file. The measured time will be marked by `str`. If `str` is missing the current bcl-command name will be used.

```
subroutine timelog(str)
character(len=*), optional, intent(in) :: str
```

15 Parallelization

maiprogs currently supports three kinds of parallelization. First, parallelization on shared-memory systems with OpenMP is fully supported and activated by default ('config -openmp=yes').

Second, parallelization on clusters (multi-computer systems) with MPI is supported (partly still in development) and has to be explicitly activated using 'config -mpi=yes'.

Third, parallelization using graphic cards with CUDA is in the early stages of support and has to be explicitly activated using 'config -cuda=yes'.

16 Internal organization of solvers

The generic and reusable solvers are contained in the files 'fox/lgsrev.f90' for real numbers and 'fox/xlgsrev.f90' for complex numbers. These solvers are used by the subroutines in the files 'fox/lgs.f90', which extend the solvers with the matrix-vector multiplication for real dense matrices, 'fox/xlgs.f90' with matrix-vector multiplication for complex dense matrices, 'fox/femlgs.f90' with matrix-vector multiplication for real and complex sparse matrices and 'fox/couplgs.f90' with dense and sparse matrices for coupled systems.

17 Internal organization of preconditioners

Most preconditioners on matrix level are contained in the file 'fox/mlbas.f90' for dense systems and in 'fox/mlfbas.f90' for sparse systems.

All implemented preconditioners are of Schwarz type. These preconditioners are distinguished by the construction of their subspace decomposition. In general we can write that a space V with base $\{\phi_i\}$ is decomposed in subspaces

$$V = V_0 + V_1 + \ldots + V_N, \quad V_l = \operatorname{span}\{\phi_i^l; i = 1, \ldots, \dim V_l\}$$

For the implementation we have to describe the construction of ϕ_i^l in terms of ϕ_i.

1. The simplest subspace decomposition is given by a partition of the base functions. Restriction and prolongation operators are simply given by the identity acting on a subspace.
2. Another possibility is a subspace decomposition given by a partition of the base functions with an additional coarse grid on which base functions are given as linear combinations of an arbitrary number of fine grid base functions. These are the so called Hh-methods.
3. A famous possibility is nested subspaces where we have a sequence of even coarser subspaces/ grids which are constructed by the linear combination of few fine grid base functions to a coarse grid base function. In the multiplicative Schwarz case these are the multigrid methods and in the additive Schwarz case the BPX-like methods.
4. A further case is a complete transformation of the standard base functions to a base with special properties, where usually a one-dimensional Schwarz preconditioner, the diagonal scaling, is used. Known examples are the hierarchical base and prewavelet or wavelet bases.

In the following we describe the interface to the implementations of these preconditioners.

1. The partition of the base functions is given by the arrays

   ```
   integer nxm,ixm(0:*),xm(0:nxm),am(0:nxm)
   ```

   An additional array for the local inverse is given by

   ```
   integer ipvt(0:*)
   ```

   Using these definitions we implement the additive Schwarz preconditioner

```
      subroutine asmp(ktyp,b,ixm,xm,am,nxm,ipvt,n,z,r)
      integer ktyp,n
      real b(0:*),z(0:n-1),r(0:n-1)
```

the multiplicative Schwarz preconditioner

```
      subroutine msmp(ktyp,b,a,ix,ixm,xm,am,nxm,ipvt,flag,n,z,r)
      integer ktyp,flag,n,ix(0:*)
      real a(0:*),b(0:*),z(0:n-1),r(0:n-1)
```

and the symmetric multiplicative Schwarz preconditioner

```
      subroutine msmp2(ktyp,b,a,ix,ixm,xm,am,nxm,ipvt,flag,n,z,r)
      integer ktyp,flag,n,ix(0:*)

      real a(0:*),b(0:*),z(0:n-1),r(0:n-1)
```

2. The coarse grid base is given by the arrays

```
      integer ncxm,icxm(0:*),cxm(0:ncxm)
      real    wcxm(0:*)
```

An additional array for the coarse grid inverse is given by

```
      integer cipvt(0:*)
```

Now the additive Schwarz preconditioner with coarse grid is given by

```
      subroutine varasmp(ktyp,b,ixm,xm,am,nxm,ipvt,
     *                   icxm,wcxm,cxm,ncxm,cipvt,n,z,r)
      integer ktyp,n
      real b(0:*),z(0:n-1),r(0:n-1)
```

3. The memory distribution of the nested subspaces is given by

```
      integer mtop,am(0:mtop),xfm(0:mtop),xfn(0:mtop)
```

The projections (restriction and prolongation) from level to level are given by the following arrays

```
      integer prom
      parameter (prom=9)
      integer pro(0:prom,0:*), hpro(0:prom,0:*)
      real    prow(1:prom,0:*),hprow(1:prom,0:*)
      integer difsi(0:5,0:*)
      real    difsw(1:5,0:*)
```

Using these definitions we implement the Multilevel additive Schwarz method with exact solution at the lowest level

```
      subroutine bmasv2(ktyp,mtop,mdc,am,a,xfm,xfn,x,f,pro,prow,ix,
     *                 difsi,difsw,rp,ri,ngm,ngn,
     *                 hpro,hprow,haard,ipvt)
c computes one mas step at level mtop
c exact solution at lowest level
```

```
      integer ktyp,mdc
      real a(0:*),f(0:*),x(0:*)
      real haard(0:*)
      integer ix(0:*),ipvt(0:*)

      integer rp(0:*),ri(0:1,0:*)
c Position und Anzahl der Netze
      integer ngm(0:mtop),ngn(0:mtop)
```

the multigrid algorithm

```
      subroutine mg(mtop,mu,nu1,nu2,mds,mdc,mgr,omega,
     *              am,a,xfm,xfn,x,f,pro,prow,ix,
     *              difsi,difsw,drh,rp,ri,rci,ngm,ngn,ncm,ncn,
     *              dntyp,ktyp,nt,ipvt)
c fuehrt einen Multigridschritt auf level mtop durch
      integer mu,dntyp,ktyp,nt,mds,mdc,mgr
      integer nu1(0:mtop),nu2(0:mtop)
      real    omega(0:mtop)
      real a(0:*),f(0:*),x(0:*)
      integer ix(0:*),rci(0:*),ipvt(0:*)

      integer rp(0:nt-1,0:*),ri(0:1,0:*)
      real    drh(0:2,0:*)
c Position und Anzahl der Netze
      integer ngm(0:mtop),ngn(0:mtop)
c Position und Anzahl der logischen Freiheitsgrade
      integer ncm(0:mtop),ncn(0:mtop)
```

and the BPX-algorithm

```
      subroutine bpx(mtop,mdc,xfm,xfn,x,f,pro,prow,
     *               difsi,difsw,drh,rp,ri,ngm,ngn,nt,ktyp)
c computes one bpx step at level mtop
      integer nt,ktyp,mdc
      real f(0:*),x(0:*)

      real    drh(0:2,0:*)
      integer rp(0:nt-1,0:*),ri(0:1,0:*)
c Position und Anzahl der Netze
      integer ngm(0:mtop),ngn(0:mtop)
```

4.
```
      subroutine hiera(ktyp,mtop,xfm,xfn,x,f,hpro,hprow,hdiag)
c Projeziere auf die hierarchische Basis
c Teile durch die Diagonale
c Projeziere zurueck
      integer mtop,ktyp
      integer xfm(0:mtop),xfn(0:mtop)
      real f(0:*),x(0:*),hdiag(0:*)

      integer prom
      parameter (prom=9)
      integer hpro(0:prom,0:*)
      real    hprow(1:prom,0:*)
```

18 Configuration

'makefile' is the topmost Makefile whereas every folder contains its own Makefile which is called by the topmost Makefile. 'config' is a shell script which determines the operating system, the used FORTRAN compiler and which libraries are available. Using this informations 'config' sets the Compiler options and link paths which are necessary to compile the whole program system. It decides also whether it is necessary or possible to add some code to replace the BLAS and LAPACK-libraries, which are primarily used, by an optimized version, e.g. Intel's MKL. (this code is located in the directory ads/). It generates the files 'makefile.mai' and 'makefile.gen'.

19 Functions and boundary data

Functions describing volume data, boundary data, nonlinear behavior and exact solutions and their derivatives are given in the modules 'funct??' located in the files 'funct??.f90', where ?? denotes the short-cut of the configuration. Here we have to distinguish mainly between Dirichlet and Neumann boundary data and volume data with the prefixes d-, n- and v-, respectively. The operators are denoted by the prefixes lap-, lame-, helm- and maxw-. The data function is denoted by the postfix '-f' and the corresponding exact solution is denoted by '-fe'. An additional derivative is indicated by a '-s-'. E.g. the derivative of Dirichlet data for the Laplacian is denoted by 'dslapf' (d-s-lap-f) and the exact solution of the Dirichlet problem by 'dlapfe'. Additionally there is the module 'ltyp??' defined, containing the integer variable 'ltyp(0:1)' steering the choice of the function which has to be evaluated. The subroutines implementing the boundary data functions have the following structure

```
subroutine pre-op-f(x,nx,fx)
use ltyp??
real(kind=dp) :: x(0:*),nx(0:*),fx(0:*)

select case (ltyp(0))
case (0)
case (1)
end select

end subroutine pre-op-f
```

Volume data functions are given by

```
subroutine v-op-f(x,fx)
use ltyp??
real(kind=dp) :: x(0:*),fx(0:*)

select case (ltyp(0))
case (0)
case (1)
end select

end subroutine v-op-f
```

The actual components of the arguments 'x' and 'nx' depends on the configuration. 'nx' describes the direction of a derivative, usually the normal direction. 'fx' contains the result of the evaluation of the function, which is denoted by 'ltyp(0)' given by the common block 'ltypc'. There are printing functions for boundary data and volume data which describe the function in clear-text on standard output.

```
subroutine pr-op-f(ltyp)
integer, intent(in) :: ltyp

select case (ltyp)
case (0)
case (1)
case default
end select

end subroutine pr-op-f

subroutine prv-op-f(ltyp)
integer, intent(in) :: ltyp

select case (ltyp)
case (0)
case (1)
case default
end select
end subroutine prv-op-f
```

References

[1] M. Maischak, The analytical computation of the Galerkin elements for the Laplace, Lamé and Helmholtz equation in 2D-BEM. Preprint 95-15, DFG-Schwerpunkt Randelementmethoden.

[2] M. Maischak, The analytical computation of the Galerkin elements for the Laplace, Lamé and Helmholtz equation in 3D-BEM. Preprint 95-16, DFG-Schwerpunkt Randelementmethoden.

[3] M. Maischak, Manual of the program system maiprogs. Preprint, Institut für Angewandte Mathematik, Universität Hannover.