4 Software Concept and Implementation

In this chapter, we describe the implementation of the monotone multigrid method proposed in Chapter 3, i.e., Algorithm (1). Our implementation of Algorithm (1) forms a flexible and powerful tool for solving highly nonlinear elliptic problems. In particular, it can be regarded as an efficient and easy to use tool for turning *linear* solvers into *nonlinear* ones.

The monotone multigrid method (1) has been implemented in the framework of the finite element toolbox UG, see [BBJ+97], for scalar problems as well as for systems of elliptic PDEs in two and three space dimensions, i.e., $d = 2, 3$. Scalar obstacle problems and contact problems with or without friction can be solved numerically using the same code. Due to the object oriented structure of the code, there is no difference between the cases $d = 2$ and $d = 3$. Moreover, only the *algebraic* part of the datastructure of UGis used, making it possible to use discretizations having degrees of freedom associated with different geometrical objects, i.e., nodes, edges, sides or elements. For example, any finite element associated with a Lagrangian basis can be used, e.g., linear or quadratic finite elements. The developed code supports local refinement, in particular at the Signorini boundary. Thus, the nodes $p \in \mathcal{N}^{(J)^\bullet}$ belonging to the coincidence set can be distributed over different levels. The user interface is small and requires only the obstacles. At the moment, the implementation including all related tools takes about 12,000 lines of C–code.

The abstract structure of our implementation is based on the following two structural advantages of the monotone multigrid method:

- *all* nonlinear operations are local and *all* linear transformations are local,
- the method is independent of the used extended splitting.

Our code takes care of these two advantages in an abstract and object oriented way. In the following, we describe in detail the developed data structure.

The core of our implementation is the *obstacle problem class*, consisting of class members and member functions. Any particular obstacle problem is represented by a specific realization of the problem class and suitable implementations of the obstacle problem class' member functions.

In addition to the abstract obstacle problem class, the following tools have been implemented:

- projecting nonlinear symmetric and unsymmetric (block) Gauß–Seidel method,
- different local projections used by the Gauß–Seidel method, (contact, contact and friction, elastic contact)
- different variants of monotone restrictions,
- transformation routines for the linear system,
- local reassembling of the coarse grid stiffness matrices,
- different tools for handling the obstacle, computing boundary stresses and modifying the transformed linear system,

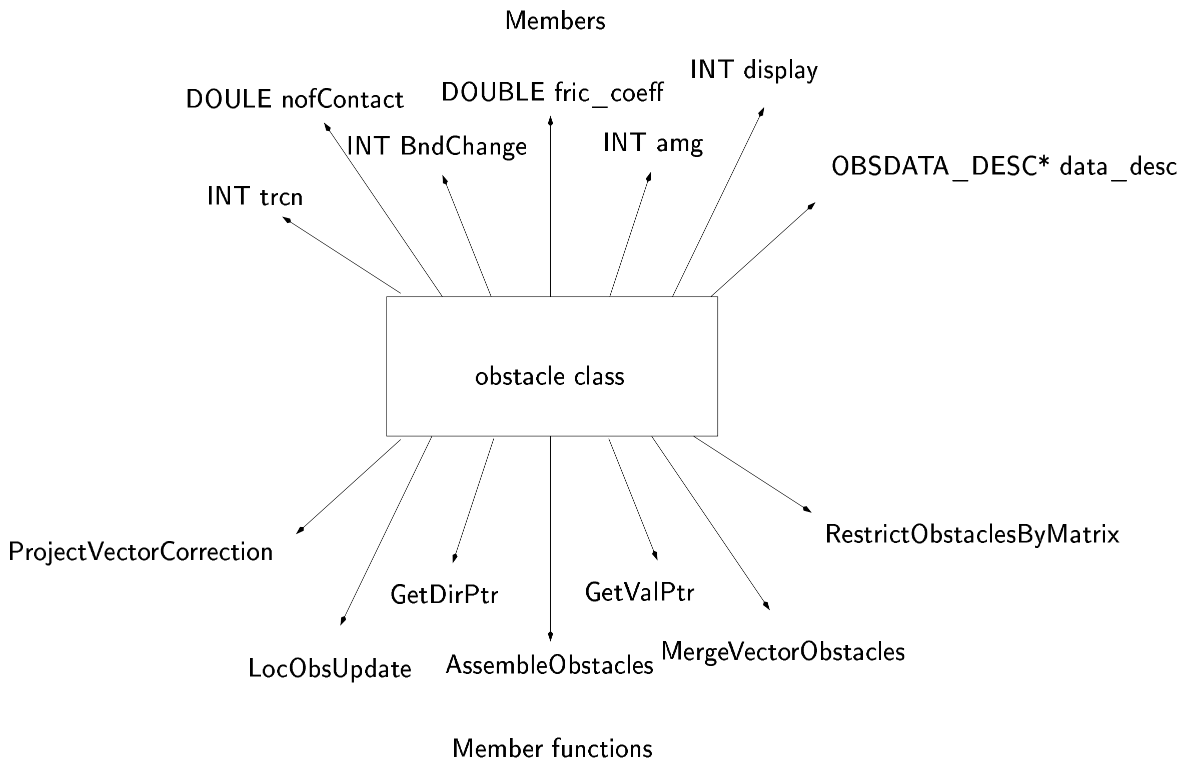

Figure 4.1: Obstacle problem class and member functions

- leading surface Gauß-Seidel,
- local linearization for problems with piecewise smooth nonlinearities,
- Krylov subspace methods working on the constrained space of the asymptotic linear problem,
- routines handling the information transfer for parallel computing,
- different tools for data postprocessing, including visualization,
- hierarchical a posteriori error estimator for obstacle problems, cf. Section 5.1.

An overview of the problem class and its member functions is depicted in Figure 4.1. In what follows, we describe the underlying concept in detail. To this end, in the rest of this chapter we proceed as follows.

The new obstacle problem class and the user interface is described in Section 4.1 and 4.2, respectively. In Section 4.3, we explain the abstract nonlinear Gauß–Seidel and the modified restriction and prolongation operator necessary for resolving the nonlinearity. These modified operators are used for assembling the modified coarse grid matrices. In Section 4.5, we show how the coarse grid matrices can be reassembled locally in a fast and efficient way.

In Section 4.6 of this chapter, we use the code developed to turn an algebraic multigrid method into a nonlinear solver. Here, we benefit from the abstract structure of our code.

Remark 4.1 *In our implementation of monotone multigrid methods, we extensively make use of the concept of* vector classes *described in [BBJ⁺97]. Vector classes are designed for the efficient handling of locally refined grids. For the sake of completeness, here we give a short introduction to this concept and refer the reader for details to [BBJ⁺97]. The* VECTOR *datatype is defined as structure giving access to all degrees of freedom associated with a particular geometrical object. For example, for piecewise linear elements, there is exactly one* VECTOR *per node, for standard quadratic elements we have one* VECTOR *per node and one* VECTOR *per edge. To any vector there is assigned a* vector class, *indicating whether the vector's geometrical object is associated with an element which has been refined or not. In particular, using locally refined grids, all degrees of freedom on the grid's surface might be associated with geometrical objects on different levels of the multigrid hierarchy. This is of importance for our method, since we strongly have to distinguish between those vectors associated with subspaces $V_1 + \ldots + V_{n_\ell}$ and those associated with $V^\nu_{n_\ell+1} + \cdots + V^\nu_m$, i.e., fine or coarse grid nodes. This is done by using vector classes.*

For numerical examples illustrating the robustness and efficiency of the method for different types of problems in two and three space dimensions, we refer to Chapter 5, Section 6.3 and Section 7.2.

4.1 The Obstacle Problem Class

The purpose of the obstacle problem class is to implement an abstract framework for monotone multigrid methods. The implementation of the obstacle problem class has been done within the linear solver class of UG. To be more specific, the obstacle problem class is derived from the class `fetransfer`, which implements the restriction and prolongation operators for linear multigrid methods. Although monotone multigrid methods are constructed to solve nonlinear elliptic partial differential equations, there are several reasons for deriving the nonlinear obstacle class from a linear problem class. The first one can be found in Theorem 3.5: After some search phase where the coincidence set is determined, a linear problem has to be solved. Secondly, the stiffness matrix assembled on the finest grid, i.e. corresponding to the high frequency basis functions, does not have to be changed, since the nonlinearity is taken care of within the nonlinear Gauß–Seidel iteration. Thirdly, the modifications of the coarse grid matrices require only suitable *local* reassembling of the coarse grid matrices. Thus, the linear structure is dominating.

In Figure 4.1, the most important members and member functions of the obstacle class are shown. In the following, we explain the members and member functions in more detail. To characterize the problem class let us first define what we understand

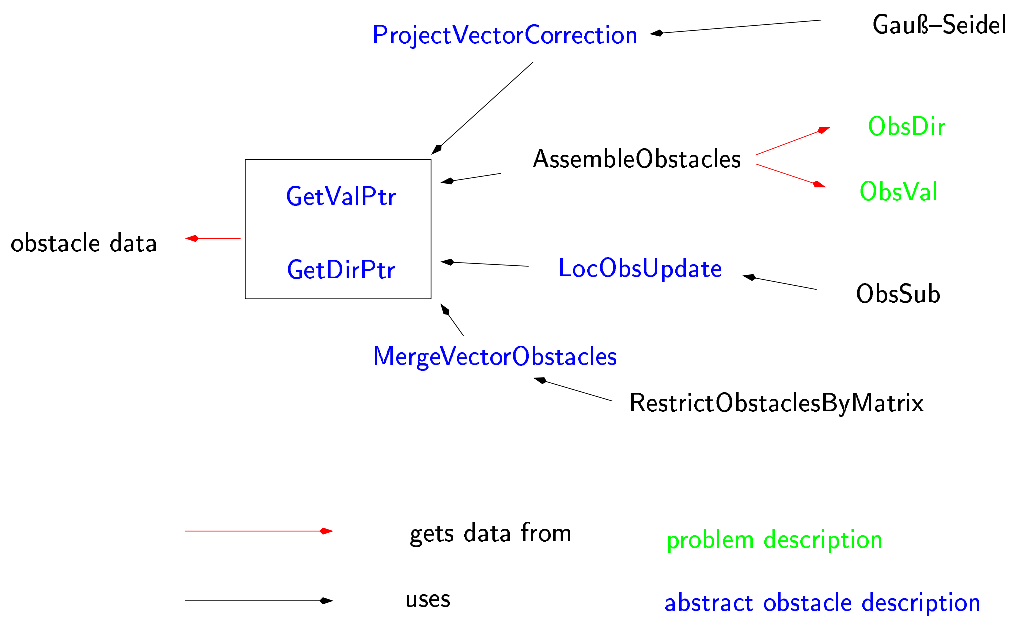

Figure 4.2: Relations between the member functions of the obstacle problem class

```
struct obsdata_desc {
  char     name[NOBSTYPE][MAX_NAME_LEN];
  char     dirname[NDIR][NOBSTYPE][MAX_NAME_LEN];
  SHORT ndir[NOBSTYPE];
  SHORT nval_of_dir[NDIR][NOBSTYPE];
  SHORT* cmpptr_of_val_of_dir[NDIR][NOBSTYPE];
  SHORT* cmpptr_of_dir[NDIR][NOBSTYPE];
};
typedef struct obsdata_desc OBSDATA_DESC;
```

Figure 4.3: Abstract Obstacle Descriptor

to be an *abstract obstacle*. An obstacle consists of a finite set of directions, the *obstacle directions*, with corresponding values, the *obstacle values*. The number of obstacle values associated with every obstacle direction is arbitrary. We define two obstacles to be of the same *obstacle type*, if the number of directions and the corresponding number of values coincide. Different obstacle types might be useful l for coupling different types of obstacle problems, i.e., Stefan problems and contact problems as might occur in the simulation of continuous casting plants or contact problems and plasticity. Within the code, an abstract obstacle is represented by the structure obsdata_desc, see Code 4.3.

In addition, some general information (name, dirname) is stored. The actual number of directions per obstacle type is given by ndir. The number of values associated with a given obstacle direction is stored in nval_of_dir and the number of components of that direction in ncmps_of_dir. Both depend on the obstacle type. For accessing the obstacle data, the component pointer cmpptr_of_val_of_dir and cmpptr_of_dir are designed to be used in combination with the UGmacro VVALUE.

For scalar obstacle problems one direction and one corresponding obstacle value is sufficient, whereas for frictionless contact problems in $d = 2, 3$ space dimensions d directions and $2d$ values are necessary. Here, the first and second obstacle value per direction can be interpreted as lower and upper obstacle, respectively. For contact problems with Coulomb friction, one additional value for the normals stresses is necessary, see also Chapter 6.

In addition to the obstacle descriptor defined above, the obstacle class possesses several member functions used as interface. By implementing the member functions, the user defines the current obstacle problem.

Let us consider the member functions given in the obstacle problem class introduced in Section 4.1 on page 53 in more detail, starting with AssembleObstacle. This function computes the obstacles and writes the values into the obstacle obs_vd for later acces by other member functions. If required, the user can implement his own version of AssembleObstacle.

Remark 4.2 *As it is the case for all member functions, the first argument of the function* AssembleObstacle *is the obstacle class itself, giving access to all members of the obstacle class. This is related to the* this *pointer from* C++*, which is also giving access to class members.*

```
/*assembling of obstacles, depends on coeffcient function number OBS_FCT */
INT (*AssembleObstacles)
     (struct np_obs_transfer *,     /* pointer to obstacle numproc    */
      MULTIGRID*,                   /* pointer to multigrid           */
      INT,                          /* mode: ALL_VECTORS or ON_SURFACE */
      INT,                          /* from level                     */
      INT,                          /* to level                       */
      VECDATA_DESC* );              /* renew critical nodes yes/no    */

/* general restriction routine for the obstacles, should be preferred   */
INT (*RestrictObstaclesByMatrix)
     (struct np_obs_transfer *,     /* pointer to obs. numproc        */
      GRID *,                       /* pointer to finegrid            */
      VECDATA_DESC *,               /* restrict to ...                */
      VECDATA_DESC *);              /* using data from                */

/* merges two obstacles. Used by the obstacle restrictions              */
INT (*MergeVectorObstacles)
     (struct np_obs_transfer *,     /* pointer to obstacle numproc    */
      VECTOR*,                      /* vector for which new obs should
                                       be computed                    */
      VECTOR**,                     /* list of vectors for merging    */
      MATRIX**,                     /* list of corresp. matrices      */
      SHORT);                       /* length of vector-list          */

/* project correction locally into admissible set, userdefined          */
INT (*ProjectVectorCorrection)
     (struct np_obs_transfer *,     /* pointer to obstacle numproc    */
      VECTOR*,                      /* vector                         */
      VECDATA_DESC*,                /* correction to project          */
      VECDATA_DESC*,                /* actual solution                */
      DOUBLE*,                      /* local stiffness matrix         */
      SHORT*,                       /* components of local matrix     */
      DOUBLE*);                     /* right hand side                */

/* Local obstacle update, also userdefined                              */
INT (*LocObsUpdate)(struct np_obs_transfer *, /* obstacle numproc */
                    DOUBLE*,                /* obstacle values      */
                    DOUBLE*,                /* obstacle directions  */
                    SHORT,                  /* actual direction     */
                    SHORT,                  /* obstacle type        */
                    VECTOR*,                /* current vector       */
                    SHORT*,                 /* component pointer    */
                    SHORT*,                 /* component pointer    */
                    INT);                   /* current level        */
```

Figure 4.4: Member functions

```
static INT ContactBallVal(DOUBLE *pos, DOUBLE *value)
{
  SHORT dir = (SHORT) value[0];

  value[0] = −OBS_MAX;
  value[1] = OBS_MAX;

  if(dir == ROT_DIR)
    value[1] = 0.5 + pos[2];

  return(0);
}
```

Figure 4.5: Obstacle definition

Our implementation of AssembleObstacle is the function SetStandardObstacles. For any vector on the surface of the multigrid hierarchy, a user defined function is called. For any other vector, the values of the obstacle are set to ∞ or $-\infty$, respectively. These values are needed only for the extended relaxation steps and are set by the monotone restriction to finite values.

Within SetStandardObstacles, information about the obstacle is needed. This information is provided by an user defined function, giving the distance of the undeformed body $\mathscr{B}$ to the obstacle in normal direction. For an example of this user defined function, see Figure 4.5. The argument list of ContactBallVal consists of the position pos at which the obstacle has to be computed and the obstacle values value. In case of the obstacle direction dir being the normal direction, i.e., dir == ROT_DIR, the distance of the body in it's reference configuration to the obstacle is written into value[0]. All other values are set to ∞ and $-\infty$, respectively.

Here, the obstacle has to be specified by the user. Using the default-settings in combination with the body's geometry and the material settings, this is all additional information which has to be specified.

The monotone restriction proposed in Section 3.3 on page 44 is implemented in terms of the following two member functions, RestrictObstaclesByMatrix and MergeVectorObstacles. The function MergeVectorObstacles is called inside of RestrictObstaclesByMatrix and computes locally the coarse grid obstacle for a given list of local interpolation matrices and corresponding values of fine grid obstacles. These lists are set up in RestrictObstaclesByMatrix. Our implementation of RestrictObstaclesByMatrix additionally takes care of some data transfer needed for the parallel version.

The last member function to be mentioned is LocObsUpdate. This function is called within the function ObsSub, which prepares the obstacles for restriction. In LocObsUpdate, the actual distance of the solution to the obstacle is computed. This depends on the obstacle specification given by the user and is thus implemented as a user function. Our implementation of LocObsUpdate does not only compute the actual distance of the solution to the obstacle, but also checks the critical and non critical flags set by Pro-

```
static INT OBSTransferConstruct (NP_BASE *theNP)
{
  NP_TRANSFER *np = (NP_TRANSFER *) theNP;
  NP_OBS_TRANSFER *ObsNP = (NP_OBS_TRANSFER *) theNP;

  theNP->Init = OBSTransferInit;
  theNP->Display = OBSTransferDisplay;
  theNP->Execute = NULL;
  /* code ommitted */
  ObsNP->AssembleObstacles      = SetStandardObstacles;
  ObsNP->RestrictObstaclesByMatrix = RestrictObstaclesByMatrix;
  ObsNP->ProjectVectorCorrection = ProjectVectorCorrectionDual;
  ObsNP->MergeVectorObstacles = MinMergeVectorObstacles;
  ObsNP->LocObsUpdate           = LocObsUpdate_Ctct;
  ObsNP->GetValPtr              = GetValPtr_Ctct;
  ObsNP->GetDirPtr              = GetDirPtr_Ctct;

  return(0);
}
```

Figure 4.6: Constructor of the obstacle problem class

jectVectorCorrection for consistency.

Summarizing, we have built an abstract class for nonlinear monotone multigrid methods for scalar equations as well as systems of equations. Our current implementation is capable of handling scalar obstacle problems as well as contact problems in linear elasticity. In particular, implementing the necessary tools for problems involving elastic contact with friction, see Chapter [], was possible without any modification of the problem class.

Let us finally take a look at the user specific parts of the obstacle problem class constructor, which can be seen in Figure 4.6. The first five member functions have been discussed already in Section 4.1. The functions GetValPtr_Ctct and GetDirPtr_Ctct provide acces to double values storing the directions vectors and associated values of the abstract obstacle. At the moment, GetValPtr_Ctct, GetValPtr_Fric, GetValPtr_Scalar and GetValPtr_ElasticFric are available. This concept provides flexible acces to the values needed independent of the way they have been stored. It is extremely important in the case where the values of the obstacle are not part of the problem description but depend on the smoothed iterate $\bar{\boldsymbol{u}}_J^\nu$. In Chapter 7, we present a nonlinear Dirichlet–Neumann algorithm, see also [KW00]. Here, the obstacle has to be modified in each outer iteration step.

The following options are available:

4.2 Basic User Interface

As has been shown in the previous section, the obstacle problem class provides a flexible description for different types of obstacle problems. These problems include scalar obstacle

Option	Values	Explanation
x	⟨vecdata descriptor⟩	solution
c	⟨vecdata descriptor⟩	correction
b	⟨vecdata descriptor⟩	rhs
A	⟨matdata descriptor⟩	stiffness matrix
n	⟨vecdata descriptor⟩	normals
Obs	⟨vecdata descriptor⟩	obstacle
sd	⟨vecdata descriptor⟩	For storing the diagonal for frictional problems
L	⟨matdata descriptor⟩	Copy of the stiffness matrix (parallel only)
trcn	1 or 2	Use truncated multigrid (1). Truncate tangential basis functions if contact in normal direction has been found (2)
fric	⟨number⟩	Coefficient of friction
cut	⟨number⟩	Also truncate, if the distance to the obstacle is less than ⟨number⟩
noRotate	none	Do not rotate stiffness and interpolation matrices and right hand side on any level
nozero	none	Do not rotate stiffness matrix and right hand side on level 0.
scalar	none	Use accelerations for scalar obstacle problems.

Table 4.1: Options for use with monotone multigrid method

problems, contact problems in linear elasticity and, more generally, obstacle problems involving piecewise smooth nonlinearities as are frictional contact problems and porous media flow. In this subsection, we describe how to define a new obstacle problem within the obstacle problem class. For defining a new obstacle problem, the user has to

- choose (or define) an abstract obstacle type,
- provide some geometrical information.

At the moment, abstract obstacles for the following problem classes are available:

- scalar obstacle problems,
- contact problems in linear elasticity,
- contact problems in linear elasticity with Coulomb friction,
- elastic contact,
- elastic contact with Coulomb friction.

If one of these problem classes is chosen, the user only has to specify geometry of the obstacle. This is done by implementing two functions. The first one gives the distance of the body $\mathscr{B}$ in it's reference configuration to the obstacle. For an example, see Figure 4.5 on page 57 The second one gives the outer normal at each point $p \in \Gamma_S$. If this function is set to the NULL pointer, by default the outer normal to the current triangulation is taken. Let us point us, that, using the default settings *only the obstacle's geometry* has to be provided. Thus, the only difference between a linear problem in linear elasticity and a nonlinear contact problem is a short function as printed in Figure 4.5. The user interface is extremely small and easy to handle.

In case, one want's to specify an additional abstract obstacle, one has to proceed in the following way. Firstly, the abstract obstacle type has to be defined by creating an abstract obstacle descriptor. This is done by implementing a suitable constructor. Secondly, the functions giving access to the obstacle values have to be implemented, i.e., implementations of GetValPtr and GetDirPtr. Then, an implementation of LocObsUpdate has to be provided. For piecewise smooth nonlinearities, in addition the Jacobian has to be provided.

4.3 Abstract Nonlinear Gauß–Seidel

The member function ProjectVectorCorrection is the "heart" of our nonlinear method. Any of it's realizations implements a local projection onto the admissible set $\mathcal{K}_J$. As is known from, e.g., [Glo84], see also [Kor97a], convergence of the method can only be established for *localizable* nonlinearities. Thus, providing a local projection is sufficient. The actual realization of ProjectVectorCorrection is called within the nonlinear Gauß–Seidel method and returns the local correction which is admissible with respect to the given constraints. For frictionless contact problems, this is the projection onto the admissible set with respect to the energy scalar product. Several local projection routines have been implemented for one sided contact with and without friction and for elastic contact with and without friction, see [KW00]. The local projection implemented for one sided contact problems also implements the scalar case. Taking a look at the argument list, we not only see the values of the solution but also the values of the local residual. These are needed for computing the modified residual, cf. [Kor97a]. The local projection also sets appropriate flags for proper handling of the coincidence set $\mathcal{N}^{(J)^\bullet}(\bar{\boldsymbol{u}}_J^\nu)$ by using the SETCRITICAL and SETNONCRITICAL macros. These flags are used within the modified restriction and prolongation and within the local reassembling of coarse grid matrices, see Section. 4.5

Remark 4.3 *Although for linear problems, it is common practice to update the residual after each iteration step,* repeated updates of the obstacle must not be done. *Updates of the obstacle can lead to instabilities of the projected Gauß–Seidel method. Thus, it is necessary to compute the distance to the obstacle in current obstacle direction each time* ProjectVectorCorrection *is called.*

4.4 Modified Restriction

The source code shown in Figure 4.7 on the following page is the core of our implementation MinMergeVectorObstacles of MergeVectorObstacles. Here, all code related to scalar obstacles and debugging has been omitted for the sake of clarity. In particular, let us consider the lines

```
if(CRITBIT(Fvec, Fdir) && ObsNP->trcn)
  continue;
```

The coarse grid obstacles are not modified, if truncation is used (ObsNP$->$ trcn $==$ 1) and if contact has been found at the fine grid vector Fvec in obstacle direction Fdir. Since the obstacles on the coarse grid are set to $\infty, -\infty$ by the obstacle assembling routine, this leads to unconstrained coarse grid corrections in direction of Fdir.

4.5 Fast Modification of the Coarse Grid Matrices

On the coarser grids, we have to assemble the entries of the stiffness matrices $A^{(j)}$, $j < \ell$, associated with the extended search directions $\boldsymbol{\mu}_p^j$, $p \in \mathcal{N}^{(j)}$. It ist not reasonable, to completely reassemble the coarse grid matrices $A^{(j)}$, $j < \ell$ within each iteration step with respect to the actual coincidence set, since the computational cost of the reassembling would dominate the whole iteration process. In particular, after the discrete contact boundary has been fixed, no reassembling is necessary. Thus, our aim is to reassemble the entries of the stiffness matrices $A_{pq}^{(j)}$, $j < \ell$ if and only if the phase at some node $r \in \operatorname{supp} \boldsymbol{\mu}_p^{j+1} \cup \operatorname{supp} \boldsymbol{\mu}_q^{j+1}$ has changed. Additionally, we have to take special care for the case of locally refined grids. Here, we have to carefully distinguish between degrees of freedom associated with the surface of the grid and degrees of freedom associated with the extended relaxation steps. This is difficult, since the surface in general can only be built up using different levels of the multigrid hierarchy.

```
INT MinMergeVectorObstacles(NP_OBS_TRANSFER* ObsNP, VECTOR* Fvec,
                            VECTOR** CvecList, MATRIX** matrixList, SHORT noOfCvec) {
  OBSDATA_DESC* obs = OBS_DESC(ObsNP);
  SHORT Fvtype   = VTYPE(Fvec);
  SHORT Fotype   = OBSTYPE(Fvec);
  SHORT Fndir    = OD_NDIR(obs, Fotype);
  SHORT i, Cdir;
  for(i = 0; i < noOfCvec; i++) {
    VECTOR* Cvec  = CvecList[i];
    MATRIX* imat  = matrixList[i];
    SHORT Cvtype  = VTYPE(Cvec);
    SHORT Cotype  = OBSTYPE(Cvec);
    SHORT Cndir   = OD_NDIR(obs, Cotype);
    SHORT Fdir;

    for(Fdir = 0; Fdir < Fndir; Fdir++) {
      SHORT Fnvalcmp = OD_NVAL_OF_DIR(obs, Fdir, Fotype);
      SHORT Fndircmp = OD_NCMPS_OF_DIR(obs, Fdir, Fotype);
      SHORT *Fvalcmp = OD_CMPPTR_OF_VAL_OF_DIR(obs, Fdir, Fotype);
      SHORT *Fdircmp = OD_CMPPTR_OF_DIR(obs, Fdir, Fotype);
      DOUBLE *FobsVal = ObsNP->GetValPtr(ObsNP, Fvec, Fdir);
      SHORT Cdir, nzero[Cndir], sum_nzero;

      if(CRITBIT(Fvec, Fdir) && ObsNP->trcn)
        continue;
      sum_nzero = 0;
      for(Cdir = 0; Cdir < Cndir; Cdir++)
        if(ABS(MVALUE(imat, Cdir*DIM+Fdir)) > DBL_EPSILON)
          sum_nzero++;

      for(Cdir = 0; Cdir < Cndir; Cdir++) {
        SHORT Cnvalcmp = OD_NVAL_OF_DIR(obs, Cdir, Cotype);
        SHORT Cndircmp = OD_NCMPS_OF_DIR(obs, Cdir, Cotype);
        SHORT *Cvalcmp = OD_CMPPTR_OF_VAL_OF_DIR(obs, Cdir, Cotype);
        SHORT *Cdircmp = OD_CMPPTR_OF_DIR(obs, Cdir, Cotype);
        DOUBLE *CobsVal = ObsNP->GetValPtr(ObsNP, Cvec, Cdir);
        DOUBLE weight = MVALUE(imat, Cdir*DIM+Fdir) * (DOUBLE) sum_nzero
                                                   * (DOUBLE) noOfCvec;
        if(ABS(weight) <= DBL_EPSILON)
          continue;
        if(weight > 0.0) {
          CobsVal[Cvalcmp[1]] = MIN( FobsVal[Fvalcmp[1]] / weight, CobsVal[Cvalcmp[1]]);
          CobsVal[Cvalcmp[0]] = MAX( FobsVal[Fvalcmp[0]] / weight, CobsVal[Cvalcmp[0]]);
        }
        else if (weight < 0.0) {
          CobsVal[Cvalcmp[1]] = MIN( FobsVal[Fvalcmp[0]] / weight, CobsVal[Cvalcmp[1]]);
          CobsVal[Cvalcmp[0]] = MAX( FobsVal[Fvalcmp[1]] / weight, CobsVal[Cvalcmp[0]]);
        } } } }
  REP_ERR_RETURN(0);
}
```

Figure 4.7: Local monotone restriciton

Thus, our goal is to reassemble *locally* the entries of the stiffness matrices on the coarser grids *if necessary*. This is achieved by associating the flag CHANGEBIT(vec) with any VECTOR vec, where the the phase has changed during the preceding relaxation steps. Using these flags, for any level j the information whether there was a change of phase or not is stored in the member BndChange[j] of the current obstacle. Local reassembling of $A^{(j)}$ is only done, if BndChange[j+1] was true. Using this mechanism, no additional work is required for computing extended search directions after the coincidence is fixed.

Locality of the reassembling is preserved as follows: In a first step, using the function PrepareAssembling, for all coarse grid VECTORS Cvec connected to fine grid vectors Fvec with CHANGEBIT(Fvec) set , CHANGEBIT(Cvec) is also set. All matrix entries $A_{pq}^{(j)}$ with CHANGEBIT(p) or CHANGEBIT(q) true are cleared and then reassembled. Here, we extensively use the macro SKIP_MATRIX_FROM_TO(p, q), which is defined by

```
#define SKIP_MATRIX_FROM_TO(v,d) ( (!CHANGEBIT(v) && !CHANGEBIT(d)) ||
        (VNCLASS(v) < NEWDEF_CLASS && VNCLASS(d) < NEWDEF_CLASS) )
```

This definition is also useful for locally refined grids, since it uses the UG–macros VCLASS and VNCLASS. These macros provide access to the vector class mechanism of UG, see Remark 4.1

If SKIP_MATRIX_FROM_TO(p, q) is true, the $d \times d$–block matrix $\boldsymbol{A}_{pq}$ coupling the two VECTORS p and q has not to be recomputed. In Figure 4.8, the first iteration of the nonlinear search phase for an $\mathcal{V}(4,4)$–cycle is shown. Here, as coarse grid solver the algebraic monotone multigrid method described in Section 4.6 is used, leading to negative level indices. The number of vectors in contact is shown in the second column, the number of vectors with CHANGEBIT set in the third column. The last three columns show the number of contact nodes in each obstacle direction. On level 1, there are eight critical nodes, illustrating the recursive truncation of basis functions.

```
**** statistics of transfer.obstacle.obstacle ****
Level  NVec  Nchg  (time)  Total   0    1    2
  4      1     1  (  1)      1     1    0    0
  3      0     1  (  1)      0     0    0    0
  2      0     1  (  0)      0     0    0    0
  1      8     8  (  0)      8     0    0    8
  0      5    15  (  0)      5     0    0    5
 −1      0     7  (  0)      0     0    0    0
 −2      0     4  (  0)      0     0    0    0
Sum     14    37  (  2)     −−     1    0   13
```

Figure 4.8: Obstacle statistics

Remark 4.4 *In case of locally refined grids, the coarse grid matrices can become singular, if all fine grid functions contributing to a particular coarse grid function are truncated. This is detected within the function* ModifyMatrix*, which is called after reassembling the* *coarse grid matrices. Degrees of freedom with zero entries in the stiffness matrix can be regarded as an artifact from the linear data structure. We overcome this difficulty by setting the corresponding diagonal element of the stiffness matrix to one and the corresponding row and column to zero. Thus, the* VECTOR *is decoupled from all other* VECTOR*s and the stiffness matrix is still regular.*

4.6 An Application of the Concept: Algebraic Monotone Multigrid

As an application of the abstract software concept developed in the previous sections, in this section we consider an algebraic monotone multigrid method. We apply our code to a *linear* algebraic multigrid method and turn it into a *nonlinear* solver. The non-linear algebraic monotone multigrid method described in this chapter is based on the implementation of linear algebraic multigrid methods in the finite element toolbox UG.

Let us recall that on the coarsest grid of our multigrid hierarchy we have to solve nonlinear obstacle problems. In Algorithm (1), this is done by a nonlinear projected Gauß–Seidel method. This is efficient for for scalar obstacle problems and small coarse grids. However, in case of complicated geometries in three space dimensions the coarse grid might consists of several hundreds or thousands unknowns. Here, applying a nonlinear Gauß–Seidel method, in particular for systems of equations, requires many iteration steps on the coarse grid. Thus, the efficiency of the method is lost.

Unfortunately, due to the nonlinearity of the problem, no direct method can be applied. The local corrections have to be admissible with respect to the quasioptimal obstacles introduced in Section (3.3). Therefore, standard linear solvers cannot be used for the coarse grid problems. The remedy is, to use nonlinear monotone algebraic multigrid as a coarse grid solver. The resulting method is a hybrid geometrical and algebraic multigrid methods with only a few unknowns on the coarsest grid. The geometrical multigrid is used for the hierarchy of triangulations resulting from successive refinement. Whenever available, we use these finite element spaces, since for these spaces the approximation property (2.26) is well known. If no hierarchy of nested spaces is a priori available, we use the coarse grid spaces constructed implicitly by applying an algebraic multigrid method.

Algebraic multigrid methods for *linear* problems have been implemented in the framework of the finite element toolbox UG. For a short introduction to algebraic multigrid methods, we refer the reader to [Wag98] and the literature cited therein.

In contrast to standard multigrid methods, for algebraic multigrid methods the interpolation operators are built up matrix dependent, i.e., the transfer operators depend on the actual representation of the bilinear form $a(\cdot,\cdot)$. Thus, no geometrical objects are associated with any degree of freedom on algebraic levels $l < 0$. Due to our abstract concept, this is of only minor significance for the obstacle toolbox describedd above:

- The coarse grid obstacles are constructed in a purely algebraic way using RestrictObstaclesByMatrx,
- obstacles and obstacle directions are associated in an abstract way with VECTORS, but not with nodes, edges or sides.

For using the obstacle toolbox in combination with algebraic multigrid methods, only minor changes to the code have been necessary. These are mostly related to the fact, that the *solution* computed by the algebraic multigrid method on the coarsest level is a *correction* with respect to the obstacle problem given on level $\ell > 0$. These changes are mainly of technical nature. In Table 4.2, all options related to the proper use of the monotone multigrid method in combination with algebraic multigrid method are listed. Note, that these options take only effect for levels $l \leq 0$.

Option	Values	Explanation
amg	none or 1	create obstacle for use with algebraic multigrid method.
amg	2	create obstacle for use with amg and re-assemble stiffness matrices A^l, $l < 0$
amg	3	create obstacle for use with amg and re-assemble stiffness matrices A^l, $l < 0$. Also truncate interpolation matrices if destination vector of none obstacle type
noRotate	none	Do not rotate stiffness and interpolation matrices and right hand side on any level
nozero	none	Do not rotate stiffness matrix and right hand side on level 0.
fatherobstacle	⟨obstacle numproc⟩	Copy all obstacle information from obstacle ⟨obstacle numproc⟩.

Table 4.2: Options for use with algebraic multigrid method

The numerical experiments described in Chapter 5 have partly been carried out using algebraic multigrid methods as basesolver, i.e., as solver on the coarsest grid. For instance, the solution for the example given in Section 5.3 has been computed using nonlinear algebraic multigrid as a basesolver. Here, on the coarsest grid we have about 600 unknowns and the algebraic multigrid method is more than twice as fast than a nonlinear Gauß–Seidel method.

Remark 4.5 *For algebraic multigrid methods, the interpolation operators used depend on the stiffness matrix* $\boldsymbol{A}$. *Since the stiffness matrix does depend on the basis and since our method requires* local *change of basis, the convergence speed of the algebraic multigrid method depends on whether applied to the transformed system or not. There are two possibilities*

- *Local transformation of the stiffness matrix, then assembling of transfer operators and no local transformation of interpolation matrices*
- *Assembling of transfer operators, then local transformation of the stiffness matrix and then local transformation of interpolation matrices*

For both methods, we get different results. Better results are usually achieved, if the interpolation matrices are assembled with respect to the locally transformed stiffness matrix. In other words, the hierarchy of spaces constructed by the algebraic multigrid method is *not invariant with respect to orthogonal transformations as the standard nodal multilevel basis is.*

We have to take into account another difference between geometrical and algebraic multigrid method. The interpolation matrices constructed by the selection schemes of algebraic multigrid methods are usually more dense than those arising from geometrical multigrid method. In particular, it cannot be guaranteed that corrections at the contact boundary are always originating from vectors at the contact boundary. This does not affect the theory presented in Section 3, but is of importance if obstacle values are associated only with VECTORS at the contact boundary. This is preferable for use of our method with standard multigrid method, since any correction at any interior vector is always admissible.

Thus, if using algebraic multigrid method, not admissible coarse grid corrections might occur. This can spoil the convergence of the method during the transient phase. We observed this in numerical experiments. There are two possibilities to overcome this difficulty. The first possibility is to truncate, i.e., set to zero, all entries of the interpolation matrices, which could lead to not admissible corrections. This is done by option `amg 3` listed in Table 4.2. Using this option usually slows down convergence, but we the global convergence of the method is guaranteed. The second possibility is to associate obstacle values with all vectors on all levels, leading to possibly restricted coarse grid corrections in the interior of the domain. For problems with constraints not being as local as for contact problems, however, this difficulty does not arise. Here, all vectors are associated with obstacle values.