Manual for the nearly exact Boundary Element Method (*neBEM*) toolkit

Supratik Mukhopadhyay, Nayana Majumdar

Applied Nuclear Physics Division

Saha Institute of Nuclear Physics

Bidhannagar, Kolkata, WB, India

e-mails

supratik.mukhopadhyay@saha.ac.in

nayana.majumdar@saha.ac.in

1 Introduction

Purpose of the *neBEM* toolkit is to help users solve scientific / technological problems governed by the three-dimensional Laplace's / Poisson's equation. At the end of a successful solution, the user is left with a distribution of singularities that satisfies the boundary conditions of the problem as specified by the user. The procedure that needs to be followed can be described as a three-step one: i) Pre-processing, ii) Solution, iii) Post-processing. The three steps are being described below in brief.

Pre-processing: The geometry of the problem and various material properties (such as the conductor / dielectric nature, potential etc for an electromagnetic problems), amount of discretization or the level of accuracy required etc are specified. These information can be specified using a separate software (e.g., *Garfield*) or a small independent code or can be

directly built into the mandatory interface code. The geometry of the problem is described by wires (1D) and flat surfaces (2D), the latter being rectangular or right triangles at present, arbitrary polygons to be implemented very soon! The wires and flat surfaces used to describe a given physical system are called primitives.

Solution: The real hard work is done here. Possible steps are discretization of the primitives into smaller wire, triangular or rectangular elements on which it is reasonable to assume uniform singularity distribution, setting up of the influence coefficient matrix, its inversion, setting up the right-hand side (RHS) vector and finding the solution. The influence coefficient matrix depends on the geometry, various material properties and remains unchanged for a given device despite a change in the boundary conditions. That is why, we invert the influence coefficient matrix as soon as it is made and keep it stored for use with various possible boundary conditions. The RHS depends on the boundary conditions specified and can be modified while keeping the geometry and other properties of the problem unchanged. The solution (e.g., the charge density in an electrostatic problem) is obtained by multiplying the inverted influence coefficient matrix to the RHS.

Post-processing: The solution is used to estimate necessary properties at any location or to carry out some other estimation, such as finding out the capacitance or force on a component for an electrostatic device. It is quite normal to invoke only the post-processing part of the computation repeatedly for a given device, on which the singularity density distribution has been estimated earlier. This is the reason *neBEM* has the provision to store each solution for a given device model and boundary conditions.

1.1 Prerequisites

The toolkit is strongly dependent on the Inverse Square Law Exact Solutions (ISLES) library. Both 32 and 64 bit versions of this library are packaged within the `neBEM<version>.tgz` file. Recently, ISLES for a Mac running OSX 10.6.4 has been compiled (thanks to Achim). Correct version of the library should be present in the `lib` directory of the distribution, and named

as `libIsles.a`. If necessary, please download this library from the *neBEM* web-page and copy it to the mentioned location.

The toolkit assumes that the *Gnu Scientific Library (GSL)* and the *gnuplot* are installed on the system. The present version also uses some routines from the book *Numerical Recipes in C* by William H Press, Brian P Flannery, Saul A Teukolsky and William T Vetterling, published by the Cambridge University Press. It should be noted here that the *neBEM* toolkit is not intrinsically dependent on any of these external packages / libraries / routines. It is quite likely that in a future version the toolkit will not use any of them except the ISLES library.

2 Installtion

Download `neBEM<Version>.tar.gz` (preferably, the latest version).

1. Unpack the archieve at a convenient location. The necessary directory structure is created as a result. Have a look into the `lib` directory and you'll find that both 32 and 64 bit versions of the libIsles are present in this directory. Based on your system architecture, copy the relevant binary as `lib/libIsles.a` which is the one that is linked during a `make`.

2. The libraries and some example binaries will be there in the `lib` and `bin` directories. However, it is safer to issue `make cleanall` to remove the older object files, libraries and binaries.

3. Create new objects, libraries and binaries by issuing for example,

```
make
```

or,

```
make -f src/Applications/MakeExampleDev
```

or, `./CreateApplications`

In order to remove binaries related to a specific application, a command similar to the

following may be issued

`make -f src/Applications/MakeExampleDev clean`

In order to remove all binaries related to all applications, issue `./CleanApplications`

4. In a similar manner, individual examples or several of them can be tried out.

5. There are some initialization files in the `InitFiles` directory.

Please note the following:

- If you are building neBEM for garfield, do `make all`
- If you are building neBEM as stand-alone, do `./CreateApplications`

Since the former uses *gcc* and the latter *g++*, it is better to clean the directories of old binaries by issuing a `make cleanall` or a `./CleanApplications`.

Please note that when being invoked from *Garfield*, no `init`-file is used and all the parameters are passed to *neBEM* via parameters. For details, please check the *Garfield* help files: (`http://consult.cern.ch/writeup/garfield/help/`).

The `init`-files, when present and relevant, provide certain default values to carry out the computations using neBEM for any given problem. It even allows control over the phase in which the computation is started. For example, if only a new post-processing is necessary for a device that has already been solved for a specific electrostatic configuration and whose charge densities have been saved, it is possible to change the flags related as `NewModel=0, NewMesh=0, NewBC=0, NewPP=1`. It is necessary to maintain the correct counters for these parameters so that the correct model, mesh, boundary conditions are used for the necessary post-processing. One initialization file is usually maintained for each application and they are expected to be stored in `$HOME/.neBEMv<version number>/<ApplicationSpecific>.init` This default behaviour can surely be modified by editing the interface file specific for an application. If an initialization file is not found, the computation is carried out by the parameter-set defined in the `neBEMSetDefaults` function. So, after the files are unpacked and before you

start executing neBEM, it may be useful to create a directory `$HOME/.neBEMv<version-counter>` and copy the initialization files from `InitFiles` to that directory. If necessary, keep a copy of older initialization files.

In the following table, we have explained in brief the different parameters that may be controlled using the init files:

Note that the parameter `NewBC` is of special importance. If this parameter is set to 1, while maintaining `NewModel = NewMesh = 0`, it is assumed that we are delaing with the same geometry and discretization, while having a different boundary condition. The inverted matrix from an earlier calculation (where `NewModel` and `NewMesh` were equal to 1) needs to have been stored for this to proceed. In such an event, the inverted matrix is read in, and the solution for the relevant boundary condition is easily and very quickly obtained. This can save an enormous amount of time, especially for a complex device for which generartion of the influence matrix, as well as its inversion is of significant computational expense. Through Garfield, this is achieved by invoking parameters such as `new-model, reuse-model, keep-inverted-matrix` (please see relevant portion of the Garfield manual).

3 *neBEM* toolkit - a How-To approach

The *neBEM* solver represents any given device by several primitives that can either be one- or two-dimensional. Any thin device component can be modeled by a combination of such straight one-dimensional primitives, while the surface of any three-dimensional device can be modeled by a large number of such two-dimensional flat surfaces. These primitives are then discretized into smaller elements on which the assumption of uniform charge density is considered to be valid. The solver uses elements of three kinds to represent any given device. These are (i) linear, (ii) right triangular and (iii) rectangular. It is quite obvious the linear elements are used to discretize the thin wire-like components, while the other two elements are used to discretize surfaces of two- or three-dimensional components. Among

Parameter	Explanation
`MinNbElementsOnLength`	Minimum number of elements allowed along the length of a primitive.
`MaxNbElementsOnLength`	Maximum number of elements allowed along the length of a primitive.
`ElementLengthRqstd`	Preferred length of the side of an element. neBEM tries to adapt to this but is over-ridden by the minimum and maximum number of elements. A log is maintained in `MeshLog.out` file.
`LengthScale`	Specifies the length scale to be used during a given computation. This can be useful to maintain accuracy of solution for problems very small or very large.
`New or old?`	These parameters, including the accompanying counters have been described below.
`DeviceOutDir`	Output directory. Internal subdirectories are created by `neBEM` automatically within this directory.
`OptDeviceFile`	`1` implies that the user will specify the device by means of input files - useful for *stand-alone* applications.
`DeviceInputFile`	User specified input file. Check stand-alone applications described below.
`OptPrint(s)`	Determines the level of terminal outputs.
`OptGnuplot(s)`	Determines whether files are to be maintained for use with the `gnuplot` routine.
`OptPrimitiveFiles`	Determines whether files describing the primitives are to be stored.
`OptElementFiles`	Determines whether files describing the elements are to be stored.
`OptReuseDir`	Determines whether the output directories are allowed to be overwritten.
`OptInvMatProc`	Procedure to be adopted for inverting the influence matrix (0 or default: LU, 1: SVD).
`OptValidateSolution`	Whether a cross-check on satisfaction of boundary conditions is done (XChk.out).
`OptForceValidation`	Whether cross-check is forced, if needed by recomputing influence matrix (XChk.out).
`OptStorePrimitive`	Determines whether primitive details are stored.
`OptStoreElements`	Determines whether element details are stored.
`OptStoreInflMatrix`	Determines whether the influence matrix is stored.
`OptStoreInvMatrix`	Determines whether the inverted matrix is stored.
`OptFormattedMatrix`	Determines whether files are stored in formatted mode.
`OptUnformattedMatrix`	Determines whether files are stored in unformatted mode.
`OptRepeatLHMatrix`	Determines whether the influence matrix computations are to be repeated in order to validate solution.
`OptSystemChargeZero`	Determines whether the total charge of the system is to be made zero. This usually means addition of a voltage shift throughout the device.

these elements, the linear one consumes minimum amount of computational resources, while the triangular one is, computationally, the most expensive. However, the triangular elements are indispensable since they are the most versatile ones in the task of representing surfaces of arbitrary geometry. Hence, a question of optimization arises that is irrevocably connected to the geometry of the device and the necessary precision of a study.

Since *neBEM* has been developed as a toolkit, some coding is necessary to solve a problem of interest. To begin with, the proverbial main function needs to be supplied. In addition, interface functions and post-processing functions need to be provided. These additional functions can co-exist with the main function in a single source code but we found it a lot more convenient to keep the main source code trimmed to the minimum and having two other source codes, one for the interface functions and another for post-processing. We even preferred to keep these source codes in different sub-directories. All these can be changed according to the user's convenience. However, the changes must also be reflected in the `Makefile` in addition to the source codes.

3.1 The driver code

From `Applications/Plate.c` Several templates have been supplied to illustrate use of the *neBEM* toolkit. The user has to supply a driver main routine invoking *neBEM* functions in a specific sequence, as shown below.

```
int main(void)

{

neBEMInitialize();

neBEMReadGeometry();
```

```
neBEMDiscretize(elementNbs);

neBEMBoundaryConditions();

neBEMSolve();

PostProcess();

neBEMEnd();

}
```

Let us see one of the example codes (named `ExampleDevice`) to check the details. A brief discussion follows the source code which itself is self-explanatory.

```
/* Source code begins */

// Analyze an Example Device

#include <stdio.h>
#include <assert.h>

#include <Interface.h>
#include <Vector.h>
#include <NR.h>
#include <neBEM.h>


int main(void)
```

```
{
int PostProcess(void); // a typical post-processing function

neBEMState = 0;
int fstatus = neBEMInitialize();
assert(fstatus == 0);

fstatus = neBEMReadGeometry();
assert(fstatus == 0);

int **elementNbs;
elementNbs = imatrix(1, NbPrimitives, 1, 2);
for(register int prim = 1; prim <= NbPrimitives; ++prim)
{
elementNbs[prim][1] = tmpNbXSegs[prim];
elementNbs[prim][2] = tmpNbZSegs[prim];
}
fstatus = neBEMDiscretize(elementNbs);
assert(fstatus == 0);

fstatus = neBEMBoundaryConditions();
assert (fstatus == 0);

NewModel = 1; NewMesh = 1; NewBoundaryCondn = 1; NewPostProcess = 1;
ModelCntr = 1; MeshCntr = 1; BoundaryCondnCntr = 1; PostProcessCntr = 1;
TimeStep = 1; EndOfTime = 1;
fstatus = neBEMSolve(); // fresh calculation
assert(fstatus == 0);
```

```
fstatus = PostProcess();
assert(fstatus == 0);

fstatus = neBEMEnd();
assert(fstatus == 0);

return 0;
} // main ends

/* Source code ends */
```

At the beginning of *neBEM* computation, the user needs to assign the state variable `neBEMState`.

```
neBEMState = 0;
int fstatus = neBEMInitialize();
assert(fstatus == 0);
```

Within the `neBEMInitialize` function, default values of global variables used by *neBEM* are set using the

```
neBEMSetDefaults
```

function. If the `OptDeviceFile` variable is set to a non-zero value via `neBEMSetDefaults`, it is decided that the device details are to be read in from a file of name as assigned to a string named `DeviceInputFile`. In such an event `neBEMInitialize` invokes

```
neBEMGetInputFromFile
```

function after setting up the defaults. For an example where the `DeviceInputFile` represents a real input file, check the examples related to `ExampleDevice`, `microMegas` and `Iarocci` tube. On the other hand, if `OptDeviceFile` is set to zero through `neBEMSetDefaults`, it is

assumed that the user will supply all the necessary details related to the device model using hard-coded numbers, evaluation of suitable expressions within the interface source code, or through the execution of other function calls presumably residing in external packages. The `GarfieldInterface`, for example, follows the latter and invokes *Garfield* functions such as `bemnpr`, `bempri` etc to specify necessary details from the functions discussed below. This could be true for any other code being interfaced to the *neBEM* toolkit. For very simple examples that do not use external files for getting in the device details, check the examples related to `Plate`, DielectricInterface (`DI`, `DItri` etc).

The `neBEMReadGeometry` function uses `neBEMGetNbPrimitives`, `neBEMGetPrimitive`, `neBEMVolumeDescription` and `neBEMVolumePoint`, the last one used sparingly, to define the device details.

Function `neBEMDisctrize` sets up a mesh on the primitives used to define the device. It is likely that a routine will be supplied either by *neBEM* or the user to analyze the primitives and to decide the size and shape of elements necessary on each primitive. At present, the corase-ness or fine-ness is dependent on these user-inputs being supplied to `neBEMDiscretize` as `elementNbs` array. For a wire, the first element of this array needs to be 1 or more to be considered for meshing. For a surface, both the elements need to be 1 or more. If the above conditions for the elements of `elementNbs` array fail, the control on meshing goes to the first three variables specified in the `init` files, namely `MinNbElementsOnLength`, `MaxNbElementsOnLength` and `ElementLengthRqstd`, or the NEBEM procedure call in Garfield. We understand that while this is a convenient approach, the resulting mesh is unlikely to have good qualities. While it is true that *neBEM* can produce reasonably good results even on very coarse and uneven mesh, efforts will be made to make the toolkit intelligent enough to take correct decisions on discretization matters.

Since a boundary condition is likely to be associated with each element, the corresponding routine, `neBEMBoundaryConditions` can be called only after the primitives have been discretized.

Finally, the solution is requested after setting up several important flags that define the problem. These flags follow from the assumption that any unique device can have several models, each model can be discretized to different extent, each discretized device can be subjected to different boundary conditions leading to different singularity distributions, and, finally, each solution may be used to estimate various properties of a device configuration of interest. For example, a micromegas mesh in a micromegas TPC under study, can be modeled using surfaces or by using wires resulting into completely different sets of primitves. These primitives can have different meshing in order to attain different levels of accuracy. Till this far, the problem is determined almost entirely by the geometry of the problem and the material properties such as the dielectric permittivity of the components while analyzing an electrostatic problem. The discretized device can be subject to different boundary conditions each of which will result into distinct solutions, i.e., singularity distribution. These singularity distributions can be used to estimate different properties of the configuration under study by carrying out different post-processes. These flags have a hieracrchy, the first mentioned having the highest priority:

Flag	**Explanation**
`NewModel`	`1` implies a fresh calculation.
`NewMesh`	`1` implies a new mesh for a device.
`NewBoundaryCondn`	`1` implies new RHS for the same LHS; skips matrix inversion.
`NewPostProcess`	`1` implies the use of the same solution; skips matrix inversion, as well as the step for computing the solution.

We maintain four counters as well:

Counter	Explanation
`ModelCntr`	keeps track of the model for a given device
`MeshCntr`	keeps track of the mesh for a given model
`BCCntr`	keeps track of the association of the boundary condition and its solution. This has to maintained by the user manually and supplied, for example, while carrying out a post-processing for a solution that was computed before.
`PPCntr`	numbers different post-processes for a given solution resulting from a given set of preceding conditions.

If `NewModel` is set to `1`, all the other flags are automatically ignored and a fresh calculation is initiated. The counters are, however, used only to create a new directory structure or use an existing one.

Please note that for solving a static problem, TimeStep and EndOfTime should both be put equal to one.

Post-processing can be carried out with separate function(s) written within this code, or function(s) residing in a separate source code, or a mix. Our experience indicate that the post-processing functions are quite complex entities themselves and are better off if kept independent of the driver routine. Among many things to be carried out in the post-processing phase, one could be the evaluation of weighting field. This is an important issue for nuclear detectors and can be carried out in two simple steps, as follows:

```
int IdWtField = neBEMPrepareWeightingField(int nprim, int primlist[]);
```

```
int status = neBEMWeightingField(Point3D point, Vector3D *field, int IdWtField);
```

Here, `IdWtField` identifies the weighting field identification tag for which we have `nprim` number of primitives identified by the list `primlist` raised to 1, while all the other primitives

are maintained grounded. At any `point`, the weighting field for the configuration identified by the tag `IdWtField` is obtained through the varaible `field`.

Finally, `neBEMEnd` ends *neBEM* gracefully, writing and closing necessary log files.

3.2 The interface

In addition, the user has to supply an interface between the code and `neBEM` which contains the following and usually resides in the `src/Interface` directory. Few templates exist there for easy reference. If the `DeviceInputFile` is not set to an empty string within the `neBEMSetDefaults` function, the initialization routine looks for the `DeviceInputFile` to read in the details of the problem.

```
int neBEMSetDefaults(void);
int neBEMGetInputsFromFiles(void);
int neBEMGetNbPrimitives(void)
int neBEMGetPrimitive(int prim, int *nvertex,
                      double xvert[], double yvert[], double zvert[],
                      double *xnorm, double *ynorm, double *znorm,
                      int *volref1, int *volref2)
int neBEMVolumeDescription(int volref1, int *shape1,
                           int *material1, double *epsilon1,
                           double *potential1, double *charge1,
                           int *boundarytype1)
int neBEMVolumePoint(double x, double y, double z)
int neBEMGetPeriodicities(int primitive,
int *ix, int *jx, double *sx,
int *iy, int *jy, double *sy,
                          int *iz, int *jz, double *sz)
```

3.3 Creation of a new neBEM application

So, in order to create a new application in the stand-alone mode, the users needs to create (by copying from existing files of example applications) the following five files and make small changes in them (most of the time, as trivial as changing reference to existing names):

- src/Applications/<AppName>.c: Changes may be necessary if time-stepping is necessary. Additional routines are best written here, for example, in order to specify location of known charges (see Gdp.c). Besides these, no other changes are necessary.
- src/Applications/Make<App>: Change SRC1, SRC2, SRC7, OBJ1, OBJ2, OBJ7, and EXE. No other change is necessary if the same directory structure is maintained and no additional source / object code(s) is (are) being processed.
- src/PostProcess/<AppName>PP.c: Depends on the application. Even no change may be necessary if only a typical field map is being sought for.
- src/Interface/<App>2neBEM.c: Make necessary changes in the `DeviceOutDir`, `DeviceInputFile` and `initFileName`
- src/InitFiles/<App>2neBEM.init: Usually modification in lines containing `DeviceOutDir`, `DeviceInputFile` are necessary. Otherwise, it depends on the given execution. Please remember to copy this `init` file to `$HOME/.neBEMv<ersion>` directory if you want these parameters to be set up from values prescribed in this `init` file.

3.4 Device without use of input files

Let us now illustrate setting up of a simple device (a conducting plate) without taking resort to input files. Consider the code `Applications/Plate.c`. The driver routine is similar to the one discussed above. Consider the interface source code as written in `Interface/Plate2neBEM.c`. Note that in `neBEMSetDefaults`, `OptDeviceFile` has been set to zero and `DeviceInputFile` has been set to an empty string, the latter being superfluous, except serving the purpose of

assigning a specific value to the `DeviceInputFile` string a specific value. Since there is one single square conducting plate in this problem as a primitive, the number of primitives is returned to be 1. The same plate (device) could have been represented by several other models. For example, we could have used two right triangles, or several plates joined together. The former model, in fact. has been tried out. The latter could be useful if we wanted to study the concentration of charges in edges / corners of the plate. As is evident from the source code, we have tried out several different configurations changing orientation etc. Depending on the model being developed, it is necessary to change the return value of the `neBEMGetNbPrimitives` function.

`neBEMGetPrimitive` gives the details of the of the primitives as is evident from the name of the function. We pass the primitive number for which we want the details. The details of this interface primitive are returned in terms of number of vertices, coordinates of these vertices, the direction of the out-going normal, volume reference number referring to itself and that referring to the external medium.

Next, the `neBEMVolumeDescription` function is invoked twice for each primitive in order to get various important information such as the shape, material, relative dielectric permittivity, potential, charge density and finally the type of boundary. Based on the type of boundary on both sides of a given interface, an interface type for a given primitive is determined. The allowed boundary and interface types and their respective integer values are shown in the following lists:

BoundaryType	Value
Vacuum	0
Conductor at specified potential	1
Conductor with a specified charge	2
Floating conductor (zero charge, perpendicular E)	3
Dielectric interface (plastic-plastic) without "manual" charge	4
Dielectric with surface charge (plastic-gas, typically)	5
Symmetry boundary, E parallel	6 (may not be necessary)
Symmetry boundary, E perpendicular	7 (may not be necessary)

InterfaceType	Value
To be skipped	0
Conductor-dielectric	1
Conductor with known charge	2
Conductor at floating potential	3
Dielectric-dielectric	4

In the same vein, we list the possible values of other variables: materials from 1 to 10 are conductors and from 11 to 20 are dielectrics keeping in tune with the Garfield code. shape (at present does not have much of an utility) can be 0 for vacuum or a gas, 2 for a wire, 3 for a triangular and 4 for a rectangular interface. In fact, the shape value reflects the shape of the interface rather than that of the volume. For a conductor, the potential is expected to be specified while the relative dielectric permittivity is usually needed for a dielectric material. Known charge distribution is also another possibility for both these types of material. We'll soon have the effect of space charge incorporated into the toolkit. There will be several ways

of incorporating the effect of known charge distributions including the standard Particle-In-Cell (*PIC*) and the Particles-On-Surface (*ParSue*) model, recently proposed by us.

By periodicity of any primitive, we mean having it repeated at some other location in space besides the original one where it is initially defined. The repetition is carried out symmetrically on the positive and negative axes. The repetition is made first on Z, then Y and finally on X. The periodicity can also imply mirror reflections. The type of periodicity is determined from the `i-` parameters, while the number of repetition is obtained from the `j-` parameters. Finally, the distance at which the repeated image is to be constructed is decided by the `s-` parameters. As can be seen from the source code, each primitive can have a different amount of periodicity. On each of these repeated primitives, all the discretization details, boundary conditions on the elements, solved singularity distribution are assumed to be identical. Please note that the handling of repetitive structures (mirrored, or otherwise) is likely to undergo several modifications in the near future. This is especially true for repetitions other than those implying simple copies.

The number of elements into which the primitives are discretized is determined by the hints supplied by the user through the two-dimensional integer array passed as as argument to the `neBEMDiscretize` function. How the user decides on the number of elements depends entirely on the user. The meshing is carried out such that the larger element number is used for the longer side of a primitive. If we have a wire primitive, the first number is used for meshing. Please note that, although planned, no adaptivity has been built into the toolkit as yet.

At the end of this phase, files suitable for viewing the final geometry, primitive, elements is prepared and placed as the `<MeshDir>/GViewDir/gView.gp` file. For viewing, the user needs to issue

```
gnuplot <MeshDir>/GViewDir/gView.gp
```

at the command prompt. It is assumed that the *gnuplot* is installed in the machine on which the code based on the *neBEM* toolkit is being executed.

The `neBEMBoundaryCondition` function takes care of setting up the proper boundary conditions on each of the elements and usually would not require any user intervention. The information related to setting up of boundary condition is extracted from the volume files. As we have noted earlier, each interface is in between two volumes. If one of the volumes is made of conducting material, the interface turns out to be a conducting-dielectric interface. In such an event, the interface is likely to represent a conductor with known potential, or a floating conductor with zero total charge. If both the volumes on either side of an interface is dielectric in nature, the interface is considered to be a dielectric-dielectric interface. The boundary condition in such an event is the continuity of the normal component of the electric displacement vector. For interfaces with known charge distributions, no boundary condition is enforced on the elements and the effect of the known charge distribution is incorporated in a manner similar to the one described for space / volume charge distributions mentioned elsewhere.

Once the boundary conditions are set up, we can proceed to the most important and computationally expensive part of the solution procedure. From the user point of view, the work is, however, as simple as invoking the function `neBEMSolve`, after setting up the proper flags and counters. For the example under discussion, and most of the other examples in the archive at present, the problems are known to be static. This is the reason the invocation is done only once, and the variables related to time (`TimeStep` and `EndOfTime` have both been set to 1. However, for a dynamic problem being solved under the assumption of quasi-statics, there can be several invocation of the `neBEMSolve` function. If the device geometry, discretization and material properties remain unchanged, the matrix inversion step can be avoided that can lead to a huge saving computationally. In fact, this is one of the major advantages of *BEM* techniques. This has been illustrated in a very simple-minded modeling of an idealized plasma device where the positive and negative ions are moved randonly from one time step to another and the voltage on a floating conductor within the device is estimated at each time-step.

Even a static problem with a given geometry can be re-solved with a different boundary

condition. In this case, a similar advantage can be extracted if the inverted matrix is stored. However, storing the inverted matrix may lead to a very large file. Moreover, to ensure that `neBEMSolve` takes advantage of the situation, the user will need to set-up the proper flags and counters if multiple invocations are made for the same device as mentioned above. For example, `NewModel` and `NewMesh` should be made equal to zero, `ModelCntr` and `MeshCntr` being kept at their previous values.

At the end of `neBEMSolve`, we are left with the singularity distribution on the elements. The array of Element structures is updated with the new values of singularity density and thus are completely defined in all respects. The Elements can now be used to evaluate any relevant property at any arbitrary location. In the present study, we have summed up the charges on all the elements to find out the total charge on the plate.

Finally, by using `neBEMEnd`, we end the use of the toolkit gracefully, that allows it to write to certain log files and save them properly.

In `PostProcess/PlatePP.c`, we can see how easily the total charge on the plate is obtained simply by adding the charge on each of the elements present in this simple device.

Finally, we need to create a `Makefile` suitable for this project. The easiest thing is to copy one of the existing ones in the `Applications` directory and change it according requirements. Only a very small number of changes is required usually if the directory structure as provided in the toolkit archive is maintained. The driver routine, the interface routine and the post-processing source and object codes need to be correctly mentioned. In addition, the binary file has to be given a suitable name.

3.5 Device using input files

Next, let us discuss an example where we create a device *ab initio*. Let us consider developing a device that has one rectangle, one triangle and one wire primitive. As you can see from the driver and interface source codes, there is hardly any difference from the earlier one we just

finished discussing. So, we will rather concentrate on the main differences in the interfacing routines.

As expected, the variable `OptDeviceFile` is set to a non-zero value and the `DeviceInputFile` string is assigned a string constant that is, in fact, the name of the file that contains the device details. This prompts the `neBEMIntitialize` function to invoke a routine called (`neBEMGetInputsFromFile`). This routine, in this example, simply uses the supplied file to read all the necessary details as temporary variables (implied by tmp in all the variable names). The *neBEM* global variables necessary for setting up the solution procedure are assigned values from these temporary variables. The rest of the procedure is identical to the one discussed in the earlier sub-section.

One additional question remains. How do we generate the `DeviceInputFile` for a realistic device with several inter-related components? Since this step is in no way related to the *neBEM* toolkit, this depends entirely on the user. However, just to provide a convenient point of departure, we have provided some examples that can be used as templates. Let us consider the problem related to the `Applications/ExampleDevice.c` source code. This hypothetical device has one rectangular flat plate, one triangular flat plate and a wire. The reason for including these shapes is the fact that, at present, these are the primitive shapes that are directly allowed by the *neBEM* toolkit. In addition, since any three dimensional device can be represented by a combination of these three types of primitives, if the user knows how to set up these primitives, she / he should be able to set any device at all. So, let us concentrate on the device generation code related to `Applications/ExampleDevice.c`. This source code can be found in the `Devices` sub-directory and is named `ExampleDevice.c` as well.

Any device is composed of primitives of two kinds: wire (1D) and flat surface (2D). The flat surfaces can be either right triangular and rectangular. In the case of a right triangular primitive, a sequence is maintained such that the second vertex is the right corner. By definition, primitive of the first kind can be used to represent only few very special geometries. On the other hand, the second kind can represent 3D geometries of any kind.

While defining the device, it is also important to define the volumes that define any given primitive. This is so because, the boundary conditions applicable to an interface depends on the properties of the volumes on its either side. So, each primitive has two volumes associated to it. As a result, it is important to identify the volumes that we need to supply in order to completely specify the problem. In addition, please note that primitives should be defined such that they are at the interface of two volumes only.

The input files are defined in the following manner: A device input directory contains all the information for a device under study, one subdirectory for each model representation; the model subdirectories in their turn contain files that store data on each of the primitives, the volume files and the map file. In the present example, `ExampleDevice` is the device directory. The rest of the files are kept in a directory called `Model1` while details of the device is given in the `Model1.inp` file. The device directory is assumed to be already existing.

```
/* ExampleDevice.c */
// An example that creates three primitives:
// a rectangular plane, a right triangular plane and a wire

int main(void)
{
char DeviceInDirName[256], DeviceInFileName[256], ModelName[256], DeviceOutDirName[256];
char *homedir;
homedir = getenv("HOME");
assert(homedir != NULL);
printf("user homedir is %s\n", homedir);
strcpy(DeviceInDirName, homedir);
strcat(DeviceInDirName, "/Inputs/ExampleDevice/");  // existing
strcpy(ModelName, "Model1");  // do NOT put an appending /
strcpy(DeviceOutDirName, homedir);
```

```
strcat(DeviceOutDirName, "/Outputs/ExampleDevice/");
strcat(DeviceOutDirName, ModelName);
strcat(DeviceOutDirName, "/");

char DeviceInDir[256], ModelSubDir[256], ModelInDir[256];
FILE *fModelInp;
strcpy(DeviceInDir, DeviceInDirName); // existing directory
strcpy(ModelSubDir, ModelName); // to be created
fModelInp = PrepareInputFiles(DeviceInDir, ModelSubDir,
                              DeviceInFileName, ModelInDir);

FILE *fData;
fData = PrepareViewingFiles(ModelInDir);

// Create volume reference data
{
void CreateVolume(char ModelInDir[], int VolRef,
                  int VolShape, int VolMaterial, int VolBoundaryType,
                  double VolEpsilon, double VolPotential, double VolCharge);
} // Volume creation ends

// Component 1 - a rectangular plate
{
// a rectangular plate
{
CreatePolygon(fModelInp, fData, ModelInDir,
NbPrimitives,
nvertex, xvert, yvert, zvert,
xnorm, ynorm, znorm,
```

```
VolRef1, VolRef2, NbXSegs, NbYSegs,
PeriodicInX, XPeriod, PeriodicInY, YPeriod, PeriodicInZ, ZPeriod);

} // RectangularPlate ends
} // Component 1 ends

// Component 2 - a triangular plate
{
// a triangular plate
{
CreatePolygon(fModelInp, fData, ModelInDir,
NbPrimitives,
nvertex, xvert, yvert, zvert,
xnorm, ynorm, znorm,
VolRef1, VolRef2, NbXSegs, NbYSegs,
PeriodicInX, XPeriod, PeriodicInY, YPeriod, PeriodicInZ, ZPeriod);
} // TriangularPlate ends
} Component 2 ends


Component 3 - a thin wire
{
// a thin wire
{
CreateLine(fModelInp, fData, ModelInDir,
NbPrimitives,
nvertex, xvert, yvert, zvert,
radius, VolRef1, VolRef2, NbSegs,
PeriodicInX, XPeriod, PeriodicInY, YPeriod, PeriodicInZ, ZPeriod);
```

```
} // wire ends
} Component 3 ends


// data that goes into the Map files
{
CreateMapFile(ModelInDir,
               NbMaps, NbBlock,
               NbXMap, NbYMap, NbZMap,
               MapLX, MapLY, MapLZ,
               MapXMid, MapYMid, MapZMid,
               NbLines, NbSegments,
               XStart, YStart, ZStart, XStop, YStop, ZStop,
               NbPoints, XPoint, YPoint, ZPoint);
} // Map creation ends


// Complete the DeviceInputFile
int NbBCondns = 1;
EndDeviceInputFile(fModelInp, NbBCondns,
DeviceOutDirName, ModelInDir, NbPrimitives);

return 0;
} // ExampleDevice.c main ends
```

The directory structure including the model name are user inputs. These can be read in from an input file or hash-defined, if desired. One point to be noted is that the `ModelName` should not be appended by a trailing despite the fact that this name is eventually used to form the

sub-directory name in which the primitives related to this particular model is stored. Most of the other user-inputs, such as dimensions and locations of the primitives, outward normal on them can be hard-coded or, even better, for a truly elaborate device, a separate input file may be used. Another option could be to have ".def" file which is #included at the beginning of a device code, as has been done in the micromegas device codes. The convenience of using separate def or input files is to avoid scrolling too often. There can, of course, be many such files possibly representing each component of a device.

For this analysis, there are six volumes which is really quite arbitrary. The other examples have a more realistic design as far as volumes are concerned. Here it assumed that the rectangular surface is at the interface of two volumes (1 and 2) The triangular surface is at the interface of two volumes (3 and 4) The wire surface is at the interface of two volumes (5 and 6) Vacuum (VolRef: 1), Conductor 1 (VolRef: 2) Dielectric 1 (VolRef: 3), Dielectric 2 (VolRef: 4) Gas (VolRef: 5), Conductor 2 (VolRef: 6) The surfaces that define the device are interfaces between these volumes. Explanation of the different parameters of the `CreateVolume` function is given below:

InterfaceType	Value
VolRef	n-th volume, to be used as a referral number.
VolShape	does not have any effect now.
VolMaterial	1-10, if conductor; 11-20, if a dielectric.
VolBoundaryType	equivalent to the Interface type described above.
VolEpsilon	relative dieletric.
VolPotential	applied potential on this volume.
VolCharge	applied charge density on the interfaces of this volume.

The generic function calls for creation of volumes and primitives have been shown here while the details are available in the source code. Please note that the second vertex is expected to be the right-corner. In most of these calls, two file handler are being passed - one for the input file and the other for creating a *gnuplot*-friendly datafile. The rest of the arguments are self-explanatory.

After the creation of the volumes and primitives, a map file is created that allows the user to evaluate properties on maps (2D grids), along lines and at given points. There can be as many 2D grids, lines and points as are needed. The maps are created based on their lengths along X, Y and Z and the mid-points of these maps. Points at which the properties are evaluated naturally do not need these information.

Finally, a few more important information is written into the `DeviceInputFile` and it is closed as the program stops execution. One of the important information here is the number of boundary conditions to be used per element. At a later stage, we plan to incorporate ability to tackle over-determined sets of problems through this integer. At present, the number of boundary conditions per element is fixed as one.

At the end of execution, all the files related to the device are created. These may be easily used by `neBEMGetInputsFromFiles` to carry out necessary computations.

3.6 Post-processing

After the singularity distribution is estimated, post-processing is usually carried out to find a number of parameters of specific scientific / technological interest. Each of the examples in the `Applications` directory has a Post-processing code in the `PostProcess` subdirectory and it is a good idea to look into these codes to get an idea of what can be done during this phase for typical electrostatic problems.

3.7 How to build the executable?

Each application needs to provide its own `Makefile` that can be very easily generated by copying the templates. The `Makefile` has to be modified appropriately to incorporate these three source codes (the driver routine, the interface routine and the post-processing routine) while building the executable. The Makefile specific for a new application can be included in the CreateApplications script, if desired.

3.8 Recent developments in neBEM

1) Improvements in computational efficiency:

During the course of this work, the neBEM toolkit has been improved significantly. The major challenge in these developments has been to increase the efficiency of the solver, while maintaining its precision.

Code parallelization: The Open Multi-Processing (OpenMP) is an Application Programming Interface (API) that supports multi-platform shared memory multiprocessor programming in C, C++ and Fortran on most processor architectures and operating systems. It consists of a set of compiler directives, library routines and environment variables that influence run-time behavior. It uses a simple, scalable API for developing parallel applications on platforms ranging from the standard desktop to supercomputers. Recently, we have successfully implemented OpenMP for the neBEM field solver. The parallelization has been implemented in several computation-intensive sub-functions of the toolkit, such as computation of the influence coefficient matrix, matrix inversion and evaluation of field and potential at desired locations. These routines are computation intensive since there can be thousands of elements where charge densities need to evaluated / influence due to all these elements need to be taken care of. The matter is even more complicated through the use of repetition of the basic structure in order to conform to the real geometry of a detector. This has proved to be very important in improving the computational efficiency of the solver. We have tested these implementations on 4, 6, 8 and 16 cores. The observed reduction in computational time has been found to be significant while the precision of the solution has been found to be preserved. In the following table 1, we present a comparison of the time taken to solve charge densities for two typical problems involving 3000 elements and 20 repetitions and 10,000 elements and 10 repetitions of the basic structure. In the next table 2, we present the time taken to generate a three-dimensional-map of the potential and three components of the electric field for the same device. It may be noted here that the user inputs related

to invocation of OpenMP during a specific solution is passed to neBEM via a file (named, neBEMProcess.inp) residing in the directory from where Garfield is being executed.

Table 1: Computational time for calculation of charge density using code parallelization

Problem specification	Thread 1	Thread 2	Thread 4	Thread 6	Thread 8	Thread 16
Element number = 3489 Periodicity = 20	6 m 20 s	4 m 18 s	3 m 29 s	3 m 7 s	3 m 20 s	5 m 2 s
Element number = 10683 Periodicity = 10	64 m 51 s	35 m 43 s	28 m 53 s	34 m 6 s	36 m 49 s	47 m 13 s

Reduced order modeling: Reduced order modeling (ROM) is a concept that is quite commonly used in numerical simulation of complex physical systems such as turbulent fluid flow, plasma dynamics etc. The idea is simple and essentially maintains the details of modeling of physical phenomena to an optimum level. A similar approach, when applied only to spatial discretization of a problem is called adaptive meshing. In the latter, the solution is usually attempted at a given spatial discretization and the solver is expected to increase or decrease the meshing to meet the desired accuracy specifications. For neBEM, we have presently implemented an algorithm which allows us to ignore the finer variations of charge densities on a primitive provided (i) it is not on the base device (as opposed to repetitive virtual devices generated in order to simulate periodic nature of a detector geometry) and (ii) it is at a far enough location so that the influence of the average charge density on the primitive is equivalent to the influence that is estimated preserving the real charge density variation on the primitive. It may be mentioned here that this order reduction in the charge density variation is implemented only at the evaluation stage, and not while actually computing the charge densities on each of the elements. Although the ROM algorithm is implemented only for pe-

Table 2: Computational time for calculation of potential and field map using code parallelization

Problem specification	Thread 1	Thread 2	Thread 4	Thread 6	Thread 8	Thread 16
Element number = 3489 Periodicity = 20	30 m 57 s	30 m 24 s	32 m 1 s	30 m 36 s	31 m 35 s	31 m 42 s
Element number = 10683 Periodicity = 10	26 m 23 s	25 m 53 s	25 m 56 s	27 m 51 s	28 m 19 s	29 m 59 s

riodic geometries, at present, it can be very useful also in non-periodic geometries. Moreover, there is no reason to stop the order reduction at the primitive level. It can continue through merging of original primitives to larger ones and even to lumping of several primitives into a component of the complete device, where the average charge density is assumed to be representative of the component itself.

The user input for controlling the ROM level is done through the same neBEMProcess.inp file that was mentioned above. The parameter primAfter=5 in the input file indicates the solver to ignore the elements in a primitive that is situated on a structure beyond the fifth repetition of the base device. I have presented the effect using different values of primAfter on a typical problem having 2000 number of elements and 60 peridicities of basic structure. In the present case, it can be seen that setting primAfter = 5 has negligible effect on the evaluated potential and field for this device. In the following table 3, I have presented the time taken to generate a map of potential and field using different levels of primAfter and the errors associated.

Fast Volume: As is expected, the time to estimate potential and field for a complex device is significant. This is especially true if the device is composed of hundreds of primitives,

Table 3: Computational time for calculation of charge density, potential and field map and error estimation using ROM

PrimAfter	Time for charge density	Time for potential and field map	Error
0	3 m 25 s	141 m 2 s	
2	4 m 42 s	27 m 59 s	0.5 %
5	4 m 27 s	52 m 56 s	0.3 %
10	3 m 26 s	71 m 28 s	0.1 %

thousands of elements and hundreds of repetitions. Reduction of time taken to estimate the electrostatic properties becomes increasingly important when complex processes such as Avalanche, Monte-Carlo tracking and Micro-Tracking are being modelled. In order to model these phenomena within a reasonable span of time, we have implemented the concept of using pre-computed values of potential and field at large number of nodal points in a set of suitable volumes. These volumes are chosen such that they can be repeated to represent any region of a given device and simple trilinear interpolation is used to find the properties at non-nodal points. The associated volume is named as the Fast Volume and the inputs related to this volume are provided via an input file (FastVol.inp) residing in the directory from which Garfield is being executed. It may be noted here that staggered volumes are allowed (takes care of GEM and other similar structures), it is possible to omit parts of a FastVol from being computed (inside a dielectric, or for other reasons) and to ignore computed FastVol values in certain regions so that the more complete and accurate evaluation is used for points in those regions. In order to preserve accuracy despite the use of trilinear interpolation, it is natural that the nodes should be chosen such that they are sparse in regions where potential and fields are changing slowly and closely packed where these properties are changing fast. Moreover, the singular surfaces and edges should be avoided as much as possible to coincide with the nodes since very sharp gradients are found to occur in these regions which are very unlikely to be correctly modelled under the assumption of linear variations. In the Tavle

4, potential and fields estimated by direct evaluation and those using FastVol have been compared. The maximum difference between the two estimates has been found to be 0.3 % which is very small and its effect on the modelling of avalanche etc has been found to negligible.

Table 4: Effect of FastVol

	Without FastVol	With FastVol
Computation time for charge density	15 s	5 m 16 s (includes calculation of FastVol)
Computation time for field map	6 m 33 s	1 s
Error in electric field		0.3 %
Computation time for 10 drift line	7 m 54 s	2 s
Value of avalache electrons		
Computation time for 100 avalanche		21 s

3d Map for Garfield++: Garfield++ imports map from field solvers such as ANSYS, CST etc. In order to equip neBEM with similar map generation capability, appropriate functions have been written. The functions can compute the map utilizing multiple CPUs in a given compute node using the OpenMP protocol. As a result, the map is computed within a reasonable amount of time. The map file is written in text, resulting in large file sizes. In this version, the generated map has a fixed mesh size throughout the computational volume. Flexibility in this respect is exppected in near future versions which will lead to smaller files and faster computation. Please go through the discussion related to FastVol above, in order to get a more clear idea of the approach adopted for generating the map and the precautions to be taken. The inputs determining the map is given via neBEMMap.inp. The inputs are quite straight-forward. Please note that the present version of the map is 0.1, indicating a

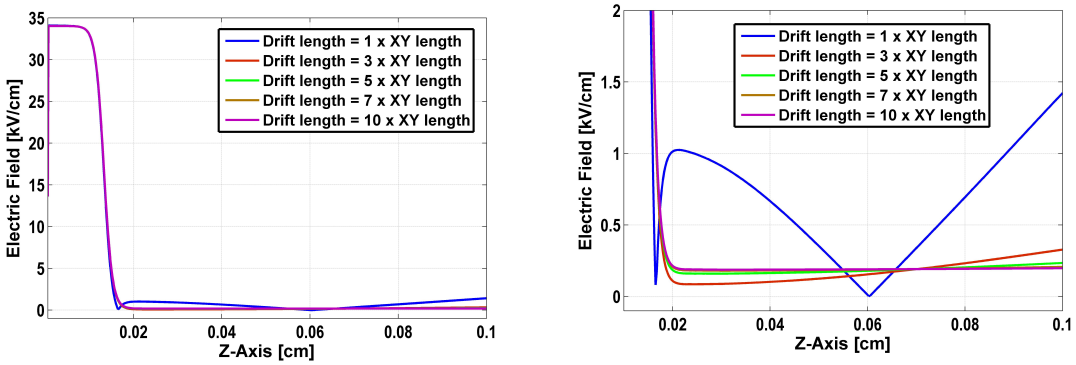
rather early release. There are two output files in the BoundaryCondition (BC) directory (in Outputs/Model/Mesh): MapInfo.out and MapFPR.out. These files need to be read by the Garfield++ script in order to import data related to potential, field and region. A Fortran Garfield script and related files are made available to the user (FieldMapforGEM.tgz). A Garfield++ script that uses the resulting field map is also provided (GEM3dMap.tgz).

2) Optimization of numerical models:

For precise and efficient computation of electrostatic field configuration within a given detector geometry, it is often necessary to optimize the numerical model of the detector. Otherwise, the computation may become unnecessarily detailed on one hand, and on the other, it may lose the accuracy necessary to follow the complicated physics processes occurring inside the detector volume.

Satisfaction of parallel plate condition: In most of the detector geometries considered in this work, the gaps in the detectors are rather small in comparison to the size of the foil or mesh. For example, for a Micromegas detector, the size of the mesh is 10cm by 10cm, while the amplification gap is 128 μm and drift gap is $\sim$1cm. As a result, while modelling the characteristics of the detector, this feature needs to be preserved. However, modelling the full 10cm by 10cm geometry is essentially a waste of computational effort since very little happens beyond the middle of the detector (unless we are interested in the edge effects, in particular). So, we have tried to strike a balance between computational precision and computational effort by optimizing (a) placement of drift, and (b) choice of the number of repetitive structures beyond a base device model. For example, to model the above Micromegas detector, we have considered the length of the basic cell structure to be 63 μm both in X and Y-Axis. Now the choice of the number of repetitive structure would be such that the parallel plate condition is statisfied. As shown in the Figure 1, when the length of X and Y-Axis is 5 times the drift length, the field is smooth throughout the volume. It can be observed that instead

of making the drift gap to be 1.2cm, as in the experiment, it is possible to reduce it to 0.1 cm and still get the same variations for the two fields in question. In the experimental condition, the mesh and drift voltage have been given to -410 and -650 V, respectively. In order to maintain the correct value of the drift field, the potential at the drift plane in the latter case has been adjusted to -427 volts instead of the experimentally applied voltage.

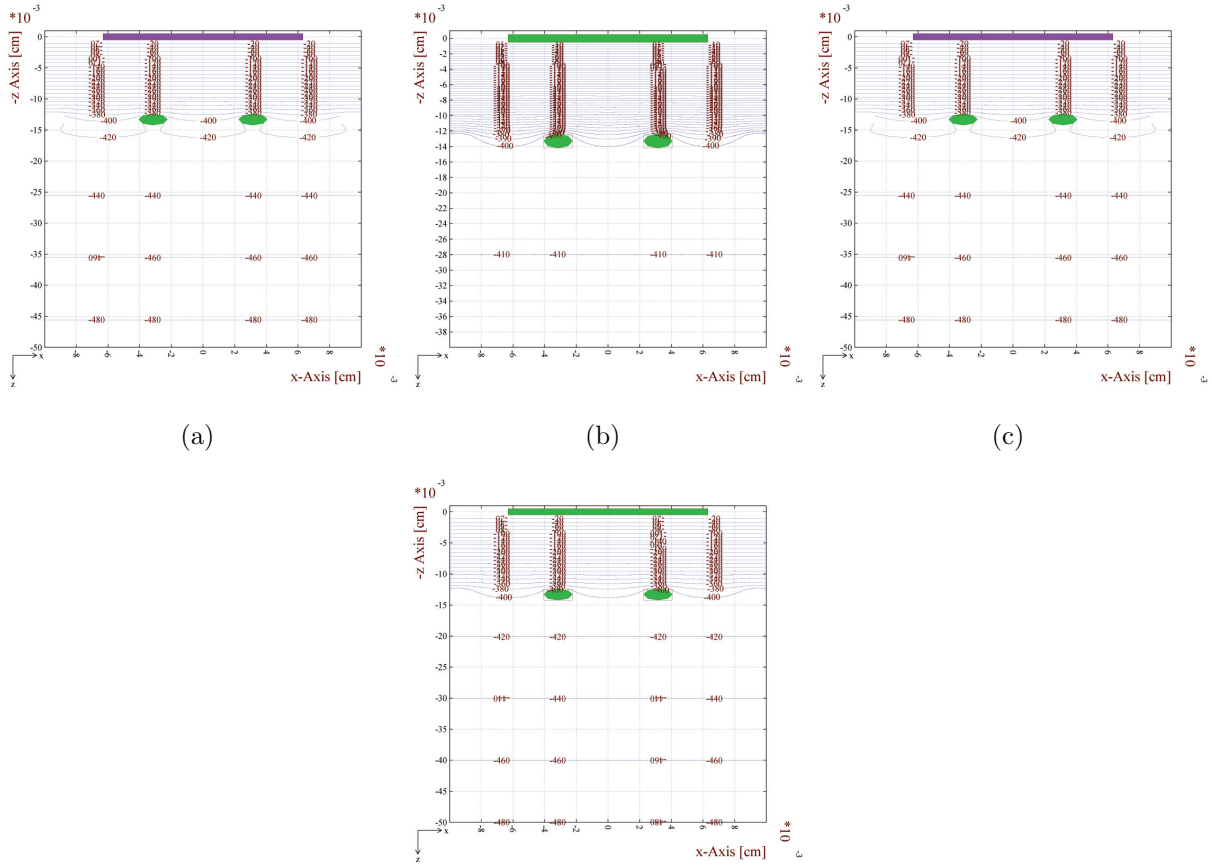


(a) (b)

Figure 1: The axial electric field for different

Wire modelling - thin wire versus polygonal cylinder: Better wire modelling is achieved if the wire is represented as a cylinder whose cross-section is a polygon. The accuracy improves when the polygon matches the cross-section of the real wire. However, computational effort increases substantially as more sides are added to the polygon since each additional side adds one more surface primitive to the problem. A thin wire model is the other extreme of the representation. In this case, it is assumed that the real wire can be replaced by a line charge situated at the axis of the real wire. The potential boundary condition is satisfied at the wire surface, but imposes a cylindrical symmetry at a distance equal to the radius of the real wire. When cylindric elements are used, the voltage boundary condition is applied to each surface panels of the cylinder. Also, the thin-wire approximation neglects the dipole moment, created to ensure an equal potential on both surfaces. The wire element approach, which has the convenience of a much less computational effort, is acceptable only when the asymmetry in the potential distribution around the wire is not very large. In the Figure 2

we have presented the consequences of using the two models in order to represent the same experimental problems. Here, the micromesh of a Micromegas detector has been modelled using wire elements and cylindrical element respectively. The potential contour in the drift and amplification region, using these two elements, are shown in the Figure 2. As illustrated, the calculations using thin-wire approximation affects the potential.



(a) (b) (c)

(d)

Figure 2: The potential contour for wire element at (a)200 V/cm and (c) 2000 V/cm, for cylindrical element at (b) 200 V/cm and (d) 2000 V/cm

4 License

Please note that we want *neBEM* to be used for peaceful and non-profit applications. The license of neBEM is written below:

neBEM Software License Version 1.5, 26 October 2009

neBEM Software Terms and Conditions:

The authors hereby grant permission to use, copy, and distribute this software and its documentation for any peaceful and non-profit purpose, provided that existing copyright notices are retained in all copies and that this notice is included verbatim in any distribution.

Additionally, the authors grant permission to modify this software and its documentation for any peaceful and non-profit purpose, provided that such modifications are not distributed without the explicit consent of the authors and that existing copyright notices are retained in all copies.

User documentation, if any, included with a redistribution, must include the following notice:

"This product includes software neBEM developed by Supratik Mukhopadhyay and Nayana Majumdar, Saha Institute of Nuclear Physics, Kolkata, WB, India"

The name "neBEM" may not be used to endorse or promote software, or products derived therefrom, except with prior written permission from the authors. If this software is redistributed in modified form, the name and reference of the modified version must be clearly distinguishable from that of this software.

You are under no obligation to provide anyone with any modifications of this software that you may develop, including but not limited to bug fixes, patches, upgrades or other enhancements or derivatives of the features, functionality or performance of this software. However, if you

publish or distribute your modifications without contemporaneously requiring users to enter into a separate written license agreement, then you are deemed to have granted the authors of this software a license to your modifications, including modifications protected by any patent owned by you, under the conditions of this license.

You may not include this software in whole or in part in any patent or patent application in respect of any modification of this software developed by you.

Users of the software are requested to feed back problems, benefits, and/or suggestions about the software to the authors:

supratik.mukhopadhyay@saha.ac.in

nayana.majumdar@saha.ac.in

In the event of any documentation / publication through the use of this software, the users are requested to kindly refer to one, or more, of the following references:

[1] Computation of 3D MEMS electrostatics using a nearly exact BEM solver, Supratik Muhkopadhyay, Nayana Majumdar, Engineering Analysis with Boundary Elements 30 (2006) 687–696, doi:10.1016/j.enganabound.2006.03.002

[2] Simulation of three-dimensional electrostatic field configuration in wire chambers: A novel approach, Nayana Majumdar, Supratik Mukhopadhyay, Nuclear Instruments and Methods in Physics Research A 566 (2006) 489–494, doi:10.1016/j.nima.2006.06.035

[3] A study of three-dimensional edge and corner problems using the neBEM solver, Supratik Mukhopadhyay, Nayana Majumdar, Engineering Analysis with Boundary Elements 33 (2009) 105–119, doi:10.1016/j.enganabound.2008.06.003

Support for this software - fixing of bugs, incorporation of new features - is done on a best effort basis. All bug fixes and enhancements will be made available under the same terms and conditions as the original software.

IN NO EVENT SHALL THE AUTHORS OR DISTRIBUTORS BE LIABLE TO ANY

PARTY FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OF THIS SOFTWARE, ITS DOCUMENTATION, OR ANY DERIVATIVES THEREOF, EVEN IF THE AUTHORS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

THE AUTHORS AND DISTRIBUTORS SPECIFICALLY DISCLAIM ANY WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE.

THIS SOFTWARE IS PROVIDED ON AN "AS IS" BASIS, AND THE AUTHORS AND DISTRIBUTORS HAVE NO OBLIGATION TO PROVIDE MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS, OR MODIFICATIONS.

THE AUTHORS MAKE NO REPRESENTATION THAT THE SOFTWARE AND MODIFICATIONS THEREOF, WILL NOT INFRINGE ANY PATENT, COPYRIGHT, TRADE SECRET OR OTHER PROPRIETARY RIGHT.

This license shall terminate with immediate effect and without notice if you fail to comply with any of the terms of this license, or if you institute litigation against any of the authors with regard to this software.