Grid Importing for CFD Simulations of Turbomachines

Author: S. Lamboo
Student number: s1069446
Organisation: University of Queensland
Supervisor: Dr P. Jacobs
Brisbane, Queensland

University of Twente
Faculty: CTW
Group: Engineering Fluid Dynamics
Supervisor: Prof.dr.ir. H.W.M. Hoeijmakers

Monday 10th November, 2014 - Tuesday 17th February, 2015

Preface

This report is the documentation of my Occupational Traineeship at the School of Mechanical Engineering and Mining at the University of Queensland. The traineeship was done as part of the curriculum of my masters degree Sustainable Energy Technology at the University of Twente. The assignment revolved around the importing of grids for the use in CFD simulations by the Turbomachinery department. The Turbomachinery department has grids in CGNS format, but the CGNS format is not compatible with Eilmer3; the UQ principal simulation code for gas dynamics simulations.

The theses of Wright [1] and Salter [2] were of a similar topic. Wright and Salter studied the possibility to convert CGNS files for use with the open source CFD package OpenFOAM.

Abstract

The turbomachinery department of the University of Queensland Mechanical and Mining Engineering group has CGNS format files that are not compatible with the in house gas dynamics simulation code Eilmer3. For this a conversion software has been designed to read the CGNS files and write the grid to an ASCII format VTK file. The conversion software also writes a text file with the boundary condtions stored in the CGNS file. The software has been proven to succesfully convert the CGNS files into Eilmer3 friendly format files. Next the converted files were read using Python and made ready for simulations with Eilmer3. During preprocessing it became evident that the CGNS file used as a reference throughout the assignment was corrupted. The grid was constructed, but because it was corrupted no flow simulations were possible.

Despite the fact that no full simulations could be run with the example CGNS file a number of things were learned throughout the project. The most important being that working with CGNS files can be time intensive and frustrating. It is therefore recommended that either time is dedicated to CGNS and to work *consistently* when writing CGNS files or to explore possible alternatives like GridPro, Pointwise, or OpenFOAM to work with grids in the future.

Contents

Chapter 1

Introduction

1.1 Background and aim

Computational Fluid Dynamics (CFD) is widely used for the design of turbomachinery components [1, 2]. There are many commercial and in house CFD codes that are used for the turbomachinery design process. The main in house CFD code used for gas dynamics simulations by the turbomachinery department at the University of Queensland is the Eilmer3 code. Eilmer3 is the result of years of development at the UQ of what started as a Multi-Block Compressible Navier-Stokes solver. The code has been improved, expanded, and enhanced for over 20 years. Even so the code does not currently have capabilities to work with CGNS format files. In Chapter 2 CGNS will be discussed in further detail. For now it is enough to mention that the turbomachinery department at the UQ sometimes has to deal with CGNS files and has no means to use these files for CFD simulations with Eilmer3.

The goal of this assignment is to develop a conversion software which can convert CGNS format files to formats which are more suited for the Eilmer3 code. The software will be tested by means of an example case, a stator blade mesh built for CFD simulations for the design of a radial inflow turbine from the PhD thesis of Carlos de Miranda Ventura [3]. The conversion software will be written as general as possible, but will be specifically be adjusted to the example case. The mesh is shown in Figure 1.1.

1.2 Structure

In Chapter 2 CGNS is discussed and the structure of the CGNS example case is studied. In Chapter 3 the CGNS conversion software is discussed. In Chapter 4 the Eilmer3 test simulation setup and results will be discussed. Finally the report is concluded with a conclusion and recommendations section.

Figure 1.1: Mesh for CGNS example stator [3].

Chapter 2

CGNS

In this chapter CGNS and the structure of CGNS format files will sbe discussed shortly. Then it will be demonstrated how the content of CGNS files can be visualized using the CGNS package *cgnstools*.

2.1 Introduction to CGNS

The CFD General Notation System, better known as CGNS, is a notation system developed in the mid 1990's by NASA, Boeing, and McDonnell Douglas. The purpose of CGNS is to create a standard file format for the storage and retrieval of CFD analysis data. The main CGNS software is used to read, write, and modify the files written in this format. A very useful package to add to the standard CGNS library is the *cgnstools* package.

CGNS files have a hierarchical structure, with nodes stemming from a single root node. There is quite some freedom in the construction of the tree of nodes, which is something I will come back to in Chapter 3. In the next setion two tools from the *cgnstools* package will be used to study the example case *stator8.cgns*.

2.2 Stator8.cgns

The example Stator8 CGNS file has been visualized in Figure 2.1 using the *cgnsview* tool. *Cgnsview* is a very useful tool for studying the hierarchical structure of a CGNS file, as can be seen in Figure 2.1. The root node in Figure 2.1 has one daughter node: the Base node. The Base node in turn has 30 daughter nodes known as Zone nodes and named domain.000xx. All the grid information such as grid structure, connectivity, and boundary conditions are stored under the Zone nodes.

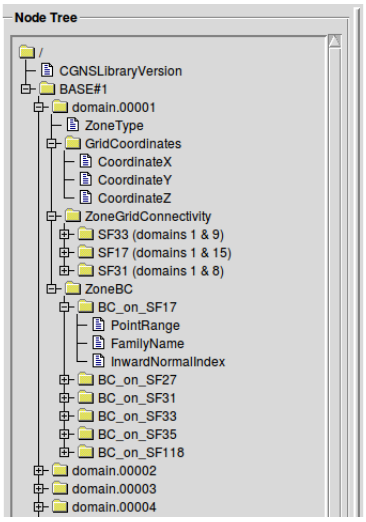

Figure 2.1: CGNS node tree for Stator8.

It is important to note that in this case the information stored under the BC_on_SF## nodes is *UserDefined* for all boundary conditions. The actual boundary condition name (HUB, BLADE, etc.) is stored under the FamilyName node.

The Stator8 example is a fairly simple CGNS file, storing just the grid and boundary conditions. In general much more information can be stored in a CGNS file, such as the flow solution or a reference state.

Whilst it is useful to know what information is stored in a CGNS file and under which nodes the information is stored, the amount of information showed by *cgnsview* can make it difficult to visualize the grid itself. The tool *cgnsplot* more appropriate to visualize the grid. The tool is also useful to see how the blocks fit together and what boundary conditions apply on the surfaces. The resulting visualization of the Stator8 grid is shown in Figure 2.2. The grid in Figure 2.2 seems in accordance with the geometry that was shown in Figure 1.1.

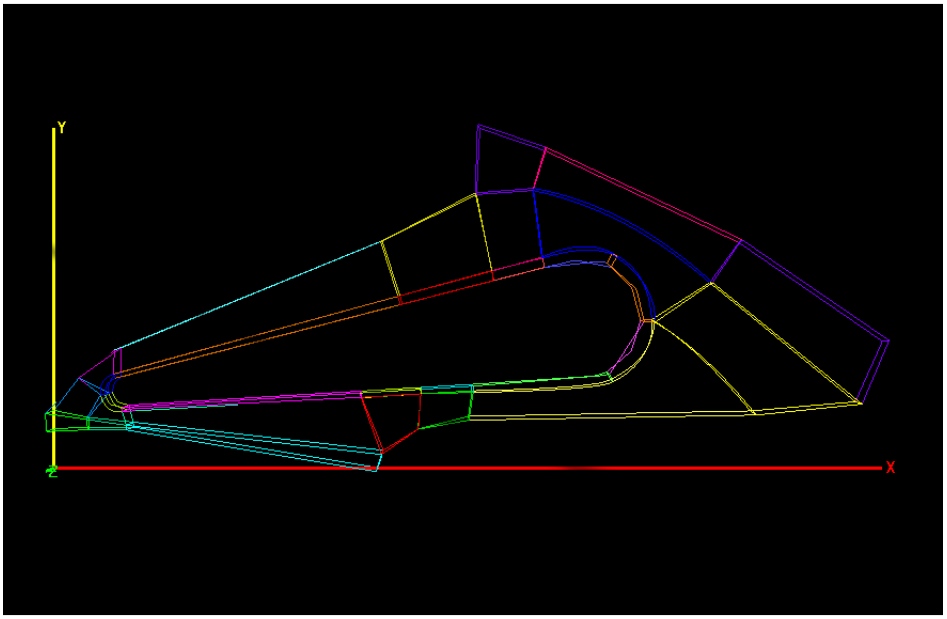

Figure 2.2: CGNS plot for Stator8.

Chapter 3

CGNS to Eilmer3 conversion

Unfortunately, the CGNS format is not compatible with the principal gas dynamics symulation code used at the UQ: Eilmer3. Therefore, conversion software was developed to convert the CGNS file to a more suitable format for simulations with Eilmer3. More acurately, the existing *cgns_to_vtk.c* converter code, part of *cgnstools* and also available at the CGNS website [4], was altered to write the grid to VTK files and a text file with the corresponding boundary conditions from a CGNS file.

The original *cgns_to_vtk.c* writes binary format VTK files by default, without writing the boundary conditions. For use with Eilmer3 ASCII format VTK files are preferred and therefore the code was altered to write ASCII format VTK files. Furthermore, code was added to read and write the boundary conditions. The boundary conditions are written in a simple text format, which can then easily be read with Python for simulations with Eilmer3. In the text file the zone (block) number, surface number, BC type and i-, j-, and k- ranges of the surface are printed for each of the 6 block surfaces.

As mentioned in Chapter 2, the boundary conditions in *stator8.cgns* are defined in a somewhat irregular fashion. The boundary conditions are initially read from the Zone_BC node, where it is common to expect the boundary condition information to be stored in CGNS files. The software then checks whether the BC type read is *UserDefined*, as is the case with the Stator8 example case. If the BC type is not *UserDefined* the BC type is printed to the outfile. If the BC type is *UserDefined* the CGNS function *cgns_goto* is used to navigate to the directory where the correct BCs can be read in the Stator8 case. The BCs are subsequently read from the FamilyName node and written to the outfile. If the BCs are not found under the FamilyName node the code returns an error and exits.

The approach used here works to read boundary conditions from CGNS files where the boundary conditions are stored under the Zone_BC node or files with exactly the same node structure as the *stator8.cgns* file is written. For files written with another CGNS node tree structure, the software will need to be adapted to read the boundary conditions. From this a drawback of the freedom of writing CGNS files becomes evident, for it makes developing conversion software applicable to all CGNS files difficult.

Lastly, large chunks of the *cgns_to_vtk.c* code that were not relevant for the *cgns2eilmer.c* code were deleted. Even withouth these parts the code is quite long, but a copy has been included in Appendix A.

The following commands can be used to compile and write the output files, respectively:

```
$ cc cgns2eilmer.c -o cgns2eilmer -lm -lcgns -lhdf5
$ ./cgns2eilmer file.cgns [dir]
```

The output files will be written to the specified directory *[dir]* or to the current directory if none is specified.

Chapter 4

Eilmer3 simulation

Now that the CGNS file has been converted to a more Eilmer3 friendly format, simulations can be set up. Python is used to read the output files from the CGNS converter, both because of its great text reading capabilities and because Python is used to set up Eilmer3 simulations. In this chapter the Python code will be discussed shortly and the simulation results are discussed. A copy of the Python code can be found in Appendix B.

4.1 Python code for Eilmer3 simulation

Per block the *cgns2eilmer* code writes two text based files that can subsequently be used to set up the Eilmer3 simulation. In order to do this the text files have to be read and the grid has to be prepared for simulations. For this three functions are necessary:

1. Read grid from the VTK file and create blocks.
2. Read the BC conditions and translate to Eilmer3 equivalent.
3. Assign BC conditions to correct surface.

Reading the grid is quite straightforward as this has been done before for previous Eilmer3 simulations and some code exists to facilitate this. Reading and converting the boundary conditions in itself does not pose a great problem, but the specific way the boundary conditions have been assigned in the Stator8 CGNS file does lead to some difficulties. Most of the surfaces of the Stator8 blocks have the boundary condition "ORPHAN". After some investigation these seem to be the surfaces which are adjacent to a surface from another block. These boundary conditions are irrelevant after the use of the *identify_block_connections* function which connects all adjacent blocks to each other. However, besides the surfaces that are adjacent to one another there are some other surfaces on which the boundary condition

is "ORPHAN". These surfaces turn out to be the surfaces where a symmetry boundary would be applied for simulations of a complete turbine. The symmetry boundary condition surfaces surround almost the entire contour of the geometry (with the exception of the inlet and outlet) and without any other information it would take a lot of work to map out surfaces which would connect in order for the symmetry to be of effect. For the initial test the remaining "ORPHAN" boundary conditions were assigned as the default Eilmer3 boundary condition "SLIP_WALL" which is effectively a solid but inviscid wall.

Next the boundary conditions have to be assigned to the correct surfaces of the constructed blocks. Eilmer3 uses a NORTH-SOUTH-EAST-WEST-TOP-BOTTOM definition to assign boundary conditions to block surfaces. A function was hence introduced to check which i, j, and k values a surface has. The surface is then identified as one of the NSEWTB surfaces by checking the minimum and maximum i, j, and k values and the BC type is applied on the surface.

The rest of the simulation is set up for a fairly simple case with inviscid, supersonic flow and an ideal gas model. The goal of the simulation is simply to check whether the generated files can be used for Eilmer3 simulations.

4.2 Simulation results

An Eilmer3 simulation can be broken down into 3 steps: preprocessing, running the simulation, and postprocessing. It is common to run the preprocessing and postprocessing once before simulating to check the grid. During the first preprocessing run an error was returned saying that some faces could not be matched because of coinciding points on the blocks. There were two of such cases, which were subsequently excluded from the *identify_block_connections* function. Excluding these connections a grid was succesfully generated, the result is shown in Figure 4.1.

The generated grid seemed to have two collapsed blocks, which have been circled in Figure 4.1. The grid in *stator8.cgns* was checked to see if it looked collapsed already, using different visualization software. These checks confirmed that the grid was collapsed already in the *stator8.cgns* file and that it was not a result of the conversion to Eilmer3 suitable formats that caused the grid to collapse. With the collapsed grid however, further simulations were not possible.

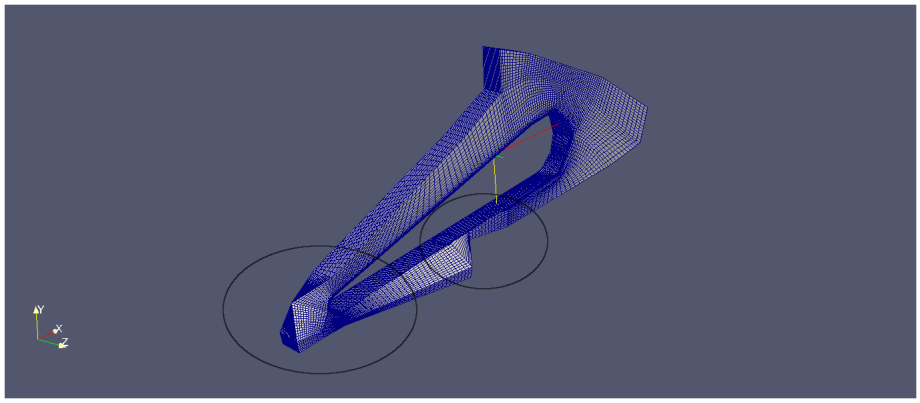

Figure 4.1: Generated grid visualized in Paraview.

Chapter 5

Conclusions and recommendations

Even though it was not possible to run a full simulation with Eilmer3 using the grid converted from CGNS to VTK, a number of things were achieved and there are some lessons that can be drawn from this assignment which can be used for future grid importing research at the UQ turbomachinery department. The conversion software was succesful in writing both VTK files containing the grid information and files containing the boundary conditions. The Eilmer3 simulation showed that the output files from the conversion software can be picked up and used to set up a simulation. Whether the coverted grid could also be used for CFD simulations could unfortunately not be verified because of the collapsed grid.

From the results it also became evident that the freedom with which CGNS files can be constructed makes writing general conversion software challenging. For future work it is therefore recommended to be consistent with the use of a CGNS file structure, for it will make developing conversion software easier. Being more consistent with file structures will likely make working with CGNS much less frustrating. It is also recommended to add an additional boundary condition for symmetry surfaces. If the symmetry surfaces are labled cleverly, such as 'left symmetry boundary' and 'right symmetry boundary' existing Eilmer3 code can be used to map the symmetry surfaces onto each other and facilitate setting up Eilmer3 simulations considerably.

The conversion software was largely based on existing software, which indicates that there are more people dealing with similar issues. Even though it is not well documented, there seems to be a lot of software that can be found which saves time and work developing new software. Documentation in general is an issue with CGNS, especially for someone with little experi-

ence with grid importing or programming in general it can be challenging to get used to CGNS software.

Overall, working with CGNS could be very frustrating at times. However, by the end of the assignments the benefits of CGNS became clearer. Once used to the standard CGNS functions, it becomes easier to work with the CGNS files. The freedom to of writing CGNS files however, makes working with CGNS files more time intensive.

In the end I believe the choice has to be made to either continue working with CGNS or investigate other grid writing options. If the decision is made to continue working with CGNS it will be necessary dedicate more time to CGNS and ensure that the CGNS files are written in a way that will facilitate further work. There are also, many alternatives that can be considered. There are both commercial packages such as GridPro and Pointwise or open source packages such as OpenFOAM that can work with grids and which can be considerd as options instead of CGNS for future work with grids in the turbomachinery group.

Bibliography

[1] M. Wright: *CFD Calculations for Turbomachinery Using OpenFoam* (2012)

[2] T. Salter: *CFD Calculations for Turbomachinery Using OpenFoam* (2013)

[3] C.A. de Miranda Ventura: *Aerodynamic Design and Performance Estimation of Radial Inflow Turbines for Renewable Power Generation Applications* (2012)

[4] `http://www.grc.nasa.gov/WWW/cgns/CGNS_docs_current/cgnstools/index.html`

Appendices

Appendix A

Source code for CGNS to Eilmer converter

```
/*
 * cgns2eilmer - read CGNS file, write grid in VTK format, and BCs
     in text format
 */

/*
 * This is an altered version of cgns_to_vtk.c, which is a part of
     cgnstools.
 * The alterations have been made for reading CGNS files and
     writing Eilmer3 friendly formatted files.
 * The code reads the grid from a provided CGNS file and writes the
     grid in an ASCII format VTK file.
 * The boundary conditions are read and written into a seperate
     text file.
 * The code assumes that the CGNS library is installed and is newer
     than version 2.5.
 *
 * For a more general cgns to vtk converter I would suggest taking
     a look at cgns_to_vtk.c, which can be downloaded from
     http://cgns.sourceforge.net/download.html.
 *
 * The code can be compiled and ran using the following:
 *
 * $ cc cgns2eilmer.c -o cgns2eilmer.x -lm -lcgns -lhdf5
 * $ ./cgns2eilmer.x file.cgns
 *
 * There are more advanced options for running the code, which are
     given in the usage message.
 */

#include <stdio.h>
```

```
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <math.h>
#include <sys/types.h>
#include <sys/stat.h>
#ifdef _WIN32
# include <io.h>
# include <direct.h>
# define chdir _chdir
#else
# include <unistd.h>
#endif

/* include CGNS library for use of CGNS functions */
#include "cgnslib.h"

#ifndef CG_MODE_READ
# define CG_MODE_READ MODE_READ
#endif

/* options for cgns2eilmer */
static char options[] = "b:z:s";
/* usage message for cgns2eilmer */
static char *usgmsg[] = {
    "usage : cgns2eilmer [options] CGNSfile [outdir]"
    "options:"
    "   -b<base> = base number (default 1)",
    "   -z<zone> = zone number (default 0 - all)",
    "   -s<soln> = solution number (default 1)",
    "<outdir> is the output directory for the VTK files.",
    "If not specified, it defaults to the current directory.",
    NULL
};

typedef float Node[3];
typedef struct {
    int cnt;
    char name[33];
} Variable;

static int nzones;
static int nbases;
static int cgnsfn;
/* The code only reads the first base, if another base has to be
    read it should be specified in the options */
static int cgnsbase = 1; /* standard setting for base */
static int cgnszone = 0; /* standard setting for zone */
static int cgnssol = 1;  /* standard setting for solution */
```

```
static int CellDim, PhyDim;

static int nnodes;
static Node *nodes;

static CGNS_ENUMT(GridLocation_t) varloc;
static int nvars, ndata;
static Variable *vars;
static cgsize_t varrng[2][3];

/*------- usage - display usage message and exit -------*/

void print_usage (char **usgmsg, char *errmsg)
{
    int n;

    if (NULL != errmsg)
        fprintf (stderr, "ERROR: %s\n", errmsg);
    for (n = 0; NULL != usgmsg[n]; n++)
        fprintf (stderr, "%s\n", usgmsg[n]);
    exit (NULL != errmsg);
}

/*------- getargs
    -----------------------------------------------------
 * get option letter from argument vector or terminates on error
 * this is similar to getopt()
 *--------------------------------------------------------------------*/

int argind = 0; /* index into argv array */
int argerr = 1; /* error output flag */
char *argarg;  /* pointer to argument string */

int getargs (int argc, char **argv, char *ostr)
{
    int argopt;
    char *oli;
    static char *place;
    static int nextarg;

    /* initialization */

    if (!argind)
        nextarg = 1;

    if (nextarg) { /* update scanning pointer */
        if (argind >= argc || ++argind == argc) {
            argarg = NULL;
            return (-1);
```

```
    }
    if ('-' != argv[argind][0]) {
        argarg = argv[argind];
        return (0);
    }
    place = argarg = &argv[argind][1];
    if (!*place) {
        if (++argind == argc) {
            argarg = NULL;
            return (-1);
        }
        argarg = argv[argind];
        return (0);
    }
    nextarg = 0;
}

/* check for valid option */

if ((argopt = *place++) == ':' || argopt == ';' ||
    (oli = strchr (ostr, argopt)) == NULL) {
    if (argerr) {
        fprintf (stderr, "invalid option - `%c'\n", argopt);
        exit (-1);
    }
    return (argopt);
}

/* don't need argument */

if (*++oli != ':') {
    if (*place && *oli == ';') { /* optional argument */
        argarg = place;
        nextarg = 1;
    }
    else {
        argarg = NULL;
        if (!*place)
            nextarg = 1;
    }
    return (argopt);
}

/* get argument */

if (!*place) {
    if (++argind >= argc) {
        if (!argerr) return (':');
```

```
            fprintf (stderr, "missing argument for option `%c'\n",
                argopt);
            exit (1);
        }
        place = argv[argind];
    }
    argarg = place;
    nextarg = 1;
    return (argopt);
}

/*---------- FATAL ------------------------------------------------
 * exit with error message
 *-----------------------------------------------------------------*/

static void FATAL (char *errmsg)
{
    if (NULL == errmsg)
        fprintf (stderr, "CGNS error:%s\n", cg_get_error());
    else
        fprintf (stderr, "%s\n", errmsg);
    exit (1);
}

/*---------- create_filename--------------------------------------
 * create valid filename
 *-----------------------------------------------------------------*/

static void create_filename (char *str, char *fname)
{
    int n = 0;
    char *p;

    for (p = str; *p; p++) {
#ifdef _WIN32
        if (strchr ("\\/:*?\"<>|", *p) == NULL)
#else
        if (isspace(*p)) continue;
        if (strchr ("\\/:*?\"<>|[]()", *p) == NULL)
#endif
            fname[n++] = *p;
        else
            fname[n++] = '_';
    }
    fname[n] = 0;
}

/*---------- fix_name ---------------------------------------------
 * remove invalid characters from variable name
```

```
 *-----------------------------------------------------------------*/

static void fix_name (char *str, char *name)
{
    int n = 0;
    char *p;

    for (p = str; *p; p++) {
        if (!isspace(*p))
            name[n++] = *p;
    }
    name[n] = 0;
}

/*------- write_ints -----------------------------------------------
 * write integers to VTK file
 *-----------------------------------------------------------------*/

static void write_ints (FILE *fp, int cnt, int *data)
{
    fprintf (fp, "%d", *data);
    while (--cnt > 0) {
        data++;
        fprintf (fp, " %d", *data);
    }
    putc ('\n', fp);
}

/*------- write_floats ---------------------------------------------
 * write floats to VTK file
 *-----------------------------------------------------------------*/

static void write_floats (FILE *fp, int cnt, float *data)
{
    fprintf (fp, "%g", *data);
    while (--cnt > 0) {
        data++;
        fprintf (fp, " %g", *data);
    }
    putc ('\n', fp);
}

/*------- get_nodes ------------------------------------------------
 * read zone nodes
 *-----------------------------------------------------------------*/

static int get_nodes (int nz, CGNS_ENUMT(ZoneType_t) zonetype,
    cgsize_t *sizes)
{
```

```
    int i, j, n, ncoords;
    int rind[6];
    cgsize_t nn, rng[2][3];
    CGNS_ENUMT(DataType_t) datatype;
    float *xyz;
    double rad, theta, phi;
    char name[33], coordtype[4];

    /* get number of coordinates */

/* CGNS functions like these are used quite often throughout the
    code. This one returns number of coordinates as ncoords.
    Definitions of the functions can be found in the cgnslib.h
    file. */
    if (cg_ncoords (cgnsfn, cgnsbase, nz, &ncoords))
        FATAL (NULL);
    if (ncoords < PhyDim)
        FATAL ("less than PhyDim coordinates");

    /* check for rind */

    if (cg_goto (cgnsfn, cgnsbase, "Zone_t", nz,
        "GridCoordinates_t", 1, "end"))
        FATAL (NULL);
    if ((i = cg_rind_read (rind)) != CG_OK) {
        if (i != CG_NODE_NOT_FOUND)
            FATAL (NULL);
        for (n = 0; n < 6; n++)
            rind[n] = 0;
    }

    /* get grid coordinate range */

    if (zonetype == CGNS_ENUMV(Structured)) {
        for (n = 0; n < 3; n++) {
            rng[0][n] = 1;
            rng[1][n] = 1;
        }
        nn = 1;
        for (n = 0; n < CellDim; n++) {
            rng[0][n] = rind[2*n] + 1;
            rng[1][n] = rind[2*n] + sizes[n];
            nn *= sizes[n];
        }
    }
    else {
        nn = sizes[0] + rind[0] + rind[1];
        rng[0][0] = 1;
        rng[1][0] = nn;
```

```
    }
    nnodes = (int)nn;

    /* read the nodes */

    strcpy (coordtype, " ");
    xyz = (float *) malloc (nnodes * sizeof(float));
    nodes = (Node *) malloc (nnodes * sizeof(Node));
    if (xyz == NULL || nodes == NULL)
        FATAL ("malloc failed for nodes");

    for (i = 1; i <= ncoords; i++) {
        if (cg_coord_info (cgnsfn, cgnsbase, nz, i, &datatype, name)
            ||
            cg_coord_read (cgnsfn, cgnsbase, nz, name,
                CGNS_ENUMV(RealSingle), rng[0], rng[1], xyz))
            FATAL (NULL);
        if (0 == strcmp (name, "CoordinateX") ||
            0 == strcmp (name, "CoordinateR"))
            j = 0;
        else if (0 == strcmp (name, "CoordinateY") ||
                 0 == strcmp (name, "CoordinateTheta"))
            j = 1;
        else if (0 == strcmp (name, "CoordinateZ") ||
                 0 == strcmp (name, "CoordinatePhi"))
            j = 2;
        else
            continue;
        if (coordtype[j] == ' ' || strchr ("XYZ", name[10]) != NULL)
            coordtype[j] = name[10];
        for (n = 0; n < nnodes; n++)
            nodes[n][j] = xyz[n];
    }
    free (xyz);

    /* change coordinate system to cartesian */

    if (0 == strncmp (coordtype, "RTZ", PhyDim)) {
        for (n = 0; n < nnodes; n++) {
            rad   = nodes[n][0];
            theta = nodes[n][1];
            nodes[n][0] = (float)(rad * cos (theta));
            nodes[n][1] = (float)(rad * sin (theta));
        }
    }
    else if (0 == strcmp (coordtype, "RTP")) {
        for (n = 0; n < nnodes; n++) {
            rad   = nodes[n][0];
            theta = nodes[n][1];
```

```
            phi   = nodes[n][2];
            nodes[n][0] = (float)(rad * sin (theta) * cos (phi));
            nodes[n][1] = (float)(rad * sin (theta) * sin (phi));
            nodes[n][2] = (float)(rad * cos (theta));
        }
    }
    else if (strncmp (coordtype, "XYZ", PhyDim))
        FATAL ("unknown coordinate system");

    return nnodes;
}

/*------- sort_variables
    ----------------------------------------
 * sort variables by name
 *---------------------------------------------------------------------*/

static int sort_variables (const void *v1, const void *v2)
{
    Variable *var1 = (Variable *)v1;
    Variable *var2 = (Variable *)v2;

    return (strcmp (var1->name, var2->name));
}

/*------- get_variables
    ----------------------------------------
 * get the solution variables
 *---------------------------------------------------------------------*/

static int get_variables (int nz, CGNS_ENUMT(ZoneType_t) zonetype,
    cgsize_t *sizes)
{
    char name[33];
    int n, len, nv, nsols;
    int rind[6];
    CGNS_ENUMT(DataType_t) datatype;

    nvars = 0;
    if (cg_nsols (cgnsfn, cgnsbase, nz, &nsols))
        FATAL (NULL);
    if (cgnssol < 1 || cgnssol > nsols) return 0;
    if (cg_sol_info (cgnsfn, cgnsbase, nz, cgnssol, name, &varloc)
        ||
        cg_nfields (cgnsfn, cgnsbase, nz, cgnssol, &nv))
        FATAL (NULL);
    if (nv < 1) return 0;
    if (varloc != CGNS_ENUMV(Vertex) && varloc !=
        CGNS_ENUMV(CellCenter)) return 0;
```

```
    nvars = nv;

    /* check for rind */

    if (cg_goto (cgnsfn, cgnsbase, "Zone_t", nz,
        "FlowSolution_t", cgnssol, "end"))
        FATAL (NULL);
    if ((n = cg_rind_read (rind)) != CG_OK) {
        if (n != CG_NODE_NOT_FOUND)
            FATAL (NULL);
        for (n = 0; n < 6; n++)
            rind[n] = 0;
    }

    /* get solution data range */

    if (zonetype == CGNS_ENUMV(Structured)) {
        nv = varloc == CGNS_ENUMV(Vertex) ? 0 : CellDim;
        for (n = 0; n < 3; n++) {
            varrng[0][n] = 1;
            varrng[1][n] = 1;
        }
        ndata = 1;
        for (n = 0; n < CellDim; n++) {
            varrng[0][n] = rind[2*n] + 1;
            varrng[1][n] = rind[2*n] + sizes[n+nv];
            ndata *= (int)sizes[n+nv];
        }
    }
    else {
        nv = varloc == CGNS_ENUMV(Vertex) ? 0 : 1;
        ndata = (int)sizes[nv];
        varrng[0][0] = rind[0] + 1;
        varrng[1][0] = rind[0] + ndata;
    }

    /* get variable names */

    vars = (Variable *) malloc (nvars * sizeof(Variable));
    if (vars == NULL)
        FATAL ("malloc failed for variable names");

    for (nv = 0; nv < nvars; nv++) {
        if (cg_field_info (cgnsfn, cgnsbase, nz, cgnssol, nv+1,
                &datatype, name))
            FATAL (NULL);
        vars[nv].cnt = 1;
        strcpy (vars[nv].name, name);
    }
```

```
    qsort (vars, nvars, sizeof(Variable), sort_variables);

    /* get number of scalars and vectors */

    for (nv = 2; nv < nvars; nv++) {
        len = (int)strlen(vars[nv].name) - 1;
        if (vars[nv].name[len] == 'Z') {
            strcpy (name, vars[nv].name);
            name[len] = 'Y';
            if (0 == strcmp (name, vars[nv-1].name)) {
                name[len] = 'X';
                if (0 == strcmp (name, vars[nv-2].name)) {
                    vars[nv-2].cnt = 3;
                    vars[nv-1].cnt = 0;
                    vars[nv].cnt = 0;
                }
            }
        }
    }

    return nvars;
}

/*---------- write_volume_cells---------------------------------
 * write volume cell data to VTK file
 *-------------------------------------------------------------*/

static void write_volume_cells (FILE *fp, int nz)
{
    int i, n, ns, nsect, nn, ip;
    int elemcnt, elemsize;
    int *types, cell[9];
    cgsize_t is, ie, nelems, maxsize, maxelems;
    cgsize_t size, *conn;
    CGNS_ENUMT(ElementType_t) elemtype, et;
    char name[33];

    if (cg_nsections (cgnsfn, cgnsbase, nz, &nsect))
        FATAL (NULL);
    if (nsect < 1) FATAL ("no sections defined");

    maxsize = maxelems = 0;
    for (ns = 1; ns <= nsect; ns++) {
        if (cg_section_read (cgnsfn, cgnsbase, nz, ns,
                name, &elemtype, &is, &ie, &nn, &ip) ||
            cg_ElementDataSize (cgnsfn, cgnsbase, nz, ns, &size))
            FATAL (NULL);
        nelems = ie - is + 1;
        if (maxelems < nelems) maxelems = nelems;
```

```
        if (maxsize < size) maxsize = size;
    }
    if (maxsize > CG_MAX_INT32) FATAL("too many elements for 32-bit
        integer");

    conn = (cgsize_t *) malloc ((size_t)maxsize * sizeof(cgsize_t));
    if (conn == NULL)
        FATAL ("malloc failed for element connectivity");

    /* count volume cells */

    elemcnt = elemsize = 0;
    for (ns = 1; ns <= nsect; ns++) {
        if (cg_section_read (cgnsfn, cgnsbase, nz, ns,
                name, &elemtype, &is, &ie, &nn, &ip))
            FATAL (NULL);
        if (elemtype < CGNS_ENUMV(TETRA_4) || elemtype >
            CGNS_ENUMV(MIXED)) continue;
        nelems = ie - is + 1;
        if (elemtype == CGNS_ENUMV(MIXED)) {
            if (cg_elements_read (cgnsfn, cgnsbase, nz, ns, conn,
                NULL))
                FATAL (NULL);
            for (i = 0, n = 0; n < nelems; n++) {
                et = (int)conn[i++];
                switch (et) {
                    case CGNS_ENUMV(TETRA_4):
                    case CGNS_ENUMV(TETRA_10):
                        elemcnt++;
                        elemsize += 5;
                        break;
                    case CGNS_ENUMV(PYRA_5):
                    case CGNS_ENUMV(PYRA_14):
                        elemcnt++;
                        elemsize += 6;
                        break;
                    case CGNS_ENUMV(PENTA_6):
                    case CGNS_ENUMV(PENTA_15):
                    case CGNS_ENUMV(PENTA_18):
                        elemcnt++;
                        elemsize += 7;
                        break;
                    case CGNS_ENUMV(HEXA_8):
                    case CGNS_ENUMV(HEXA_20):
                    case CGNS_ENUMV(HEXA_27):
                        elemcnt++;
                        elemsize += 9;
                        break;
                    default:
```

```
                    break;
                }
                if (cg_npe (et, &nn) || nn == 0)
                    FATAL ("invalid element type in mixed");
                i += nn;
            }
        }
        else {
            switch (elemtype) {
                case CGNS_ENUMV(TETRA_4):
                case CGNS_ENUMV(TETRA_10):
                    nn = 5;
                    break;
                case CGNS_ENUMV(PYRA_5):
                case CGNS_ENUMV(PYRA_14):
                    nn = 6;
                    break;
                case CGNS_ENUMV(PENTA_6):
                case CGNS_ENUMV(PENTA_15):
                case CGNS_ENUMV(PENTA_18):
                    nn = 7;
                    break;
                case CGNS_ENUMV(HEXA_8):
                case CGNS_ENUMV(HEXA_20):
                case CGNS_ENUMV(HEXA_27):
                    nn = 9;
                    break;
                default:
                    nn = 0;
                    break;
            }
            if (nn) {
                elemcnt += (int)nelems;
                elemsize += (nn * (int)nelems);
            }
        }
    }

    if (elemcnt == 0) {
        free (conn);
        return;
    }

    types = (int *) malloc (elemcnt * sizeof(int));
    if (types == NULL)
        FATAL ("malloc failed for cell types");

    /* write the elements */
```

```
    fprintf (fp, "CELLS %d %d\n", elemcnt, elemsize);

    elemcnt = 0;
    for (ns = 1; ns <= nsect; ns++) {
        if (cg_section_read (cgnsfn, cgnsbase, nz, ns,
                name, &elemtype, &is, &ie, &nn, &ip))
            FATAL (NULL);
        if (elemtype < CGNS_ENUMV(TETRA_4) || elemtype >
            CGNS_ENUMV(MIXED)) continue;
        nelems = ie - is + 1;
        if (cg_elements_read (cgnsfn, cgnsbase, nz, ns, conn, NULL))
            FATAL (NULL);
        et = elemtype;
        for (i = 0, n = 0; n < nelems; n++) {
            if (elemtype == CGNS_ENUMV(MIXED)) et = (int)conn[i++];
            switch (et) {
                case CGNS_ENUMV(TETRA_4):
                case CGNS_ENUMV(TETRA_10):
                    nn = 4;
                    types[elemcnt++] = 10;
                    break;
                case CGNS_ENUMV(PYRA_5):
                case CGNS_ENUMV(PYRA_14):
                    nn = 5;
                    types[elemcnt++] = 14;
                    break;
                case CGNS_ENUMV(PENTA_6):
                case CGNS_ENUMV(PENTA_15):
                case CGNS_ENUMV(PENTA_18):
                    nn = 6;
                    types[elemcnt++] = 13;
                    break;
                case CGNS_ENUMV(HEXA_8):
                case CGNS_ENUMV(HEXA_20):
                case CGNS_ENUMV(HEXA_27):
                    nn = 8;
                    types[elemcnt++] = 12;
                    break;
                default:
                    nn = 0;
                    break;
            }
            if (nn) {
                cell[0] = nn;
                for (ip = 0; ip < nn; ip++)
                    cell[ip+1] = (int)conn[i+ip] - 1;
                write_ints (fp, nn+1, cell);
            }
            if (cg_npe (et, &nn) || nn == 0)
```

```
                FATAL ("invalid element type");
            i += nn;
        }
    }
    free (conn);

    /* write the element types */

    fprintf (fp, "CELL_TYPES %d\n", elemcnt);
    write_ints (fp, elemcnt, types);

    free (types);
}

/*------------ read_bc ----------------------------------------
 * read boundary conditions
 *-----------------------------------------------------------------*/

static void read_bc (FILE *fp, int nz)
{

    int ib, nbocos;
    char boconame[33], famname[33];

    int normalindex[3], ndataset;
    int normallist;
    BCType_t ibocotype;
    PointSetType_t iptset;
    DataType_t normaldatatype;
    cgsize_t ipnts[2][3];
    cgsize_t npts, normallistflag;

    /* find out number of BCs that exist under this zone */
    cg_nbocos(cgnsfn,cgnsbase,nz,&nbocos);
    /* do loop over the total number of BCs */
    for (ib=1; ib <= nbocos; ib++) {
    /* get BC info */
        cg_boco_info(cgnsfn,cgnsbase,nz,ib,boconame,&ibocotype,
                  &iptset,&npts,normalindex,&normallistflag,&normaldatatype,&ndataset);

        if (iptset != PointRange)
            FATAL ("For this program, BCs must be set up as
                PointRange type %s\n");

        fprintf(fp, "\nzone number: %d\n", nz);
        fprintf(fp, "BC number: %i\n",ib);
        fprintf(fp, "  name= %s\n",boconame);
```

```
        /* Check if BC is UserDefined. If so read BC info from the
            node Family Name, if not print BCType.
           THIS IS EXTREMELY SPECIFIC TO CERTAIN CGNS FILES WHERE BC
               INFO IS UNDER NODE
               /Base_t/Zone_t/ZoneBC_t/BC_t/FamilyName.
           IF THE BC'S ARE NOT READ CORRECTLY THE PATH FOR cg_goto
               MIGHT HAVE TO BE ADJUSTED */
        if (ibocotype == 1) {                       // bocotype 1 is
            UserDefined
            if (cg_goto (cgnsfn, cgnsbase, "Zone_t", nz, "ZoneBC_t",
                1, "BC_t", ib, "end"))
                FATAL("BC_t node not found");
                if (cg_famname_read(famname))
                    FATAL("Family name not found");

                fprintf(fp, "  type= %s\n", famname);
        }
        else fprintf(fp, " type= %s\n",BCTypeName[ibocotype]);
    /* read point range in here */
        cg_boco_read(cgnsfn,cgnsbase,nz,ib,ipnts[0],&normallist);
        fprintf(fp,"  i-range= %i ,
            %i\n",(int)ipnts[0][0],(int)ipnts[1][0]);
        fprintf(fp,"  j-range= %i ,
            %i\n",(int)ipnts[0][1],(int)ipnts[1][1]);
        fprintf(fp,"  k-range= %i ,
            %i\n",(int)ipnts[0][2],(int)ipnts[1][2]);
    }
}

/*========== main ============================================*/

int main (int argc, char *argv[])
{
    int n, nz;
    char name[33], BCname[35], outfile[37], outfileBC[37];
    cgsize_t sizes[9];
    CGNS_ENUMT(ZoneType_t) zonetype;
    struct stat st;
    FILE *fp, *fpBC;

    /* print usage message if there are not sufficient arguments
        given */
    if (argc < 2)
        print_usage (usgmsg, NULL);

    /* get options from arguments */

    while ((n = getargs (argc, argv, options)) > 0) {
        switch (n) {
```

```
            case 'b':
                cgnsbase = atoi (argarg);
                break;
            case 'z':
                cgnszone = atoi (argarg);
                break;
            case 's':
                cgnssol = atoi (argarg);
                break;
        }
    }

    if (argind >= argc)
        print_usage (usgmsg, "filename not specified");
    if (stat (argv[argind], &st)) {
        fprintf (stderr, "can't stat <%s>\n", argv[argind]);
        exit (1);
    }
    if (S_IFREG != (st.st_mode & S_IFMT)) {
        fprintf (stderr, "<%s> is not a regular file\n",
            argv[argind]);
        exit (1);
    }

    /* open CGNS file */

    printf ("reading CGNS file from \"%s\"\n", argv[argind]);
    fflush (stdout);
    if (cg_open (argv[argind], CG_MODE_READ, &cgnsfn))
        FATAL (NULL);
    if (cg_base_read (cgnsfn, cgnsbase, name, &CellDim, &PhyDim))
        FATAL (NULL);
    printf (" using base %d - %s\n", cgnsbase, name);
    fflush (stdout);
    if (PhyDim != 3 /*|| (CellDim != 1 && CellDim != 3)*/)
        FATAL ("cell and/or physical dimension invalid");

    /* check amount of zones */
    if (cg_nzones (cgnsfn, cgnsbase, &nzones))
        FATAL (NULL);
    if (nzones == 0)
        FATAL ("no zones in the CGNS file");
    if (cgnszone && cgnszone > nzones)
        FATAL ("zone number invalid");

    /* file output directory */

    if (++argind < argc) {
        if (stat (argv[argind], &st) &&
```

```
#ifdef _WIN32
            _mkdir (argv[argind])) {
#else
            mkdir (argv[argind], S_IRWXU|S_IRWXG|S_IROTH|S_IXOTH)) {
#endif
            cg_close (cgnsfn);
            fprintf (stderr, "couldn't create the directory <%s>\n",
                argv[argind]);
            exit (1);
        }
        if (chdir (argv[argind])) {
            cg_close (cgnsfn);
            fprintf (stderr, "couldn't chdir to <%s>\n",
                argv[argind]);
            exit (1);
        }
        printf ("writing ASCII VTK files to directory \"%s\"\n",
            argv[argind]);
    }
    else
        printf ("writing ASCII VTK files to current directory\n");

    for (nz = 1; nz <= nzones; nz++) {
        if (cgnszone && nz != cgnszone) continue;
        if (cg_zone_type (cgnsfn, cgnsbase, nz, &zonetype) ||
            cg_zone_read (cgnsfn, cgnsbase, nz, name, sizes))
            FATAL (NULL);
        if (zonetype == CGNS_ENUMV(Structured)) {
            if (sizes[0]*sizes[1]*sizes[2] > CG_MAX_INT32)
                FATAL("too many coordinates for 32-bit integer");
        }
        else if (zonetype == CGNS_ENUMV(Unstructured)) {
            if (sizes[0] > CG_MAX_INT32)
                FATAL("too many coordinates for 32-bit integer");
        }
        else
            FATAL ("invalid zone type");

    /* grid reading and writing */

        /* create VTK file for grid */
        create_filename (name, outfile);
        strcat (outfile, ".vtk");
        printf ("writing zone %d as %s to \"%s\"\n", nz,
            cg_ZoneTypeName(zonetype), outfile);

        fflush (stdout);
        if ((fp = fopen (outfile, "w+b")) == NULL) {
```

```
            fprintf (stderr, "couldn't open <%s> for output\n",
                outfile);
            exit (1);
        }

        fprintf (fp, "# vtk DataFile Version 2.0\n");
        fprintf (fp, "zone %d - %s\n", nz, name);
        fprintf (fp, "ASCII\n");
        if (zonetype == CGNS_ENUMV(Structured)) {
            fprintf (fp, "DATASET STRUCTURED_GRID\n");
            fprintf (fp, "DIMENSIONS %d %d %d\n",
                (int)sizes[0], (int)sizes[1], (int)sizes[2]);
        }
        else
            fprintf (fp, "DATASET UNSTRUCTURED_GRID\n");

        /* write data */

        get_nodes (nz, zonetype, sizes);

        fprintf (fp, "POINTS %d float\n", nnodes);
        for (n = 0; n < nnodes; n++)
            write_floats (fp, 3, nodes[n]);

        if (zonetype == CGNS_ENUMV(Unstructured))
            write_volume_cells (fp, nz);

        fclose (fp);

        free (nodes);
        if (nvars) free (vars);

    /* BC reading and writing */

        /* create file for BC's */
        create_filename (name, outfileBC);
        strcat (outfileBC, "BC");
        printf ("writing BC's for zone %d to \"%s\"\n", nz,
            outfileBC);

        /* read out BC's */

        if ((fpBC = fopen (outfileBC, "w+b")) == NULL) {
            fprintf (stderr, "couldn't open <%s> for output\n",
                outfileBC);
            exit (1);
        }

        read_bc (fpBC, nz);
```

```
        fclose (fpBC);
    }

    cg_close (cgnsfn);
    return 0;
}
```

Appendix B

Python code

```
# stator8test.py
# test case for importing grid from VTK file

job_title = "Stator8 cgns2vtk test case"
print job_title

# Accept defaults for air giving R=287.1, gamma=1.4
select_gas_model(model='ideal gas', species=['air'])

# Set up flow conditions
from math import pi, sin, cos
alpha = -pi # angle of attack in radians
initial = FlowCondition(p=1000.0, u=0.0, T=300.0)
M_inf = 1.5
u_inf = M_inf * initial.flow.gas.a
inflow_condition = FlowCondition(p=50.0e3, u=-u_inf*cos(alpha),
                         v=u_inf*sin(alpha), T=300.0)

# Do a little more setting of global data.
gdata.dimensions = 3
gdata.title = job_title
gdata.viscous_flag = 0 # inviscid flow
gdata.flux_calc = ADAPTIVE
gdata.max_time = 5.0e-3 # seconds
gdata.max_step = 100
gdata.dt = 1.0e-7
gdata.dt_plot = 1.5e-3
gdata.dt_history = 10.0e-5

# Create blocks
blk_list = []

for n in range(1, 10):
    fp = open("domain.0000%d.vtk" % n, 'r')
```

```
    grid = StructuredGrid()
    grid.read_block_in_VTK_format(fp)
    fp.close
    label = "BLOCK-%d" % n
    blk_list.append( Block3D(grid=grid, label=label,
        fill_condition=initial))

for n in range(10, 31):
    fp = open("domain.000%d.vtk" % n, 'r')
    grid = StructuredGrid()
    grid.read_block_in_VTK_format(fp)
    fp.close
    label = "BLOCK-%d" % n
    blk_list.append( Block3D(grid=grid, label=label,
        fill_condition=initial))

identify_block_connections(exclude_list = [(blk_list[0],
    blk_list[15]), (blk_list[7], blk_list[14])])

# Boundary conditions

# function that assigns CGNS BC to corresponding Eilmer BC
def correct_bc_type(bctype):
    if bctype == 'INLET':
        return 'SUP_IN'
    elif bctype == 'OUTLET':
        return 'SUP_OUT'
    elif bctype == 'HUB':
        return 'ADIABATIC'
    elif bctype == 'SHROUD':
        return 'ADIABATIC'
    elif bctype == 'BLADE':
        return 'ADIABATIC'
    elif bctype == 'ORPHAN':
        return 'SLIP_WALL'
    else:
        return 'SLIP_WALL' # The default option

# function to read lines and find specified target. Used for both
    BC number and i, j, k ranges.
def locate_target(target, f):
    found = False; tokens = []
    while not found:
        line = f.readline()
        if len(line) == 0: break # end of file
        line = line.strip()
        if target.lower() in line.lower():
            tokens = line.split()
            found = True; break
```

```
    if not found:
        raise RuntimeError("Did not find %s while reading VTK grid
            file" % target)
    return tokens


def read_bc(z, fp):
    # loop to find all 6 BC numbers
    n = 1
    while n <= 6:
        # assign target to find BC conditions in order
        target = "BC number: %d" % (n)
        locate_target(target, fp)
        # read BC type
        target = "type="
        BCtype = locate_target (target, fp) # BCtype[1] will contain
            the type i.e. 'HUB' or 'BLADE'

    # read i, j and, k ranges and assign BCtype
        target = "i-range="
        tks = locate_target (target, fp)
        imin = tks[1]; imax = tks[3] # assumes the line read is of
            the form i-range= imin , imax

    #
    # if imin = imax, we know the boundary is either EAST or WEST
        if imin == imax and imin == "1":
            blk_list[z].set_BC("WEST", correct_bc_type(BCtype[1]))
            n += 1
            continue
        elif imin == imax and imin != "1":
            blk_list[z].set_BC("EAST", correct_bc_type(BCtype[1]))
            n += 1
            continue

        target = "j-range="
        tks = locate_target (target, fp)
        jmin = tks[1]; jmax = tks[3]
        if jmin == jmax and jmin == "1":
            blk_list[z].set_BC("SOUTH", correct_bc_type(BCtype[1]))
            n += 1
            continue
        elif jmin == jmax and jmin != "1":
            blk_list[z].set_BC("NORTH", correct_bc_type(BCtype[1]))
            n += 1
            continue

        target = "k-range="
        tks = locate_target (target, fp)
```

```
        kmin = tks[1]; kmax = tks[3]
        if kmin == kmax and kmin == "1":
            blk_list[z].set_BC("BOTTOM", correct_bc_type(BCtype[1]))
            n += 1
            continue
        elif kmin == kmax and kmin != "1":
            blk_list[z].set_BC("TOP", correct_bc_type(BCtype[1]))
            n += 1
            continue

# two loops are necessary because of filenames
for z in range(1, 10):
    # open BC file
    fp = open('domain.0000%dBC' % z, 'r')
    z -= 2
    read_bc (z, fp)
    z += 2

for z in range(10, 31):
    # open BC file
    fp = open('domain.000%dBC' % z, 'r')
    z -= 2
    read_bc (z, fp)
    z -= 2
```