A Primer on Input Files and Obtaining Initial UEDGE Solutions

Tom Rognlien
Updated 03/19/15

This document is for the user who wants to create a new UEDGE case from a different MHD equilibrium-based mesh or for a user who wants to learn a general procedure to create new cases without relying entirely on the previous cases. Nevertheless, for either situation, the user should find it useful to use the existing test case closest to the desired new case as a rough guide, although that will not be assumed here (and thus the length of this document). For users using the Python version of UEDGE from the FACETS project, there is a set of standard input files for various test cases located in the FACETS SVN directory uedge/test/Forthon_cases (see the file README-FORTHON_cases in this directory for a description of each case).

If instead, you are using a Basis version of UEDGE, the instructions given below are the same except the following: Firstly, in the Basis version, one does not need to used the prefix on variable names (e.g., bbb.mhdgeo can be truncated to mhdgeo), though you may use the prefix and Basis will understand. Secondly, all Python arrays begin from 0, whereas in Basis, only arrays explicitly dimensioned as starting at 0 will be treated this manner. Also, Python uses [] brackets, whereas Basis uses () brackets for array indexes. In the text below, any Basis-type array notation is shown at the end of the comment for each variable. Also note that in calling a subroutine with no arguments from Basis, the () can be omitted, e.g., bbb.exmain() is equivalent to exmain.

A more complete description of the options and features of the UEDGE code is given in the manual available as http://www.mfescience.org/mfedocs/uedge_man_V4.41.pdf.

STEP 1 - MHD equilibrium and geometry type:

Obtain the ASCII-formatted "a" and "n or g" EFIT format files for the desired MHD equilibrium; UEDGE uses information in both of these files. Here we treat the most common (and default) case of a single-null equilibrium. The following input parameters are then added to the input file:

```
bbb.mhdgeo=1                          # Specifies tokamak toroidal geometry
bbb.isfixlb=0                         # 0 for standard inner divertor plate BCs
bbb.isfixrb=0                         # 0 for standard outer divertor plate BCs
os.system('cp –f aXXX123 aeqdsk')     # aeqdsk file name; for Basis omit os.system and ()
os.system('cp –f gXXX123 neqdsk')     # neqdsk file name; for Basis omit os.system and ()
```

Double null geometries can be treated as well. Here the simplest case is to consider up/down symmetry about the magnetic axis, and the only change is to add the input variable

```
bbb.geometry = "dnbot"        # Default is "snull"
```

UEDGE can also treat 2D slab geometry (bbb.mhdgeo=0) and cylindrical (r,z) geometry (bbb.mhdgeo=-1) with internal mesh algorithms; see the general user manual.

STEP 2 - Mesh setup:

The are many input variables that can be used to optimize the mesh cell spacing as described in the user manual; here we give a basic set for an orthogonal mesh where two of the cell faces are

aligned with the poloidal flux surfaces (only using a linear curve) and the other two are orthogonal to the first two.

The initial mesh should probably be of modest size because it is then faster and more trouble-free to obtain an initial steady-state UEDGE solution, and finer-mesh solutions can subsequently by using UEDGE's ability to interpolate a solution of one mesh to another - using essential arbitrary size ratios - as an initial guess on the finer mesh. But it will be best to consider increasing the mesh size incrementally for efficiency. A second consideration of the initial mesh size is if one has a previous solution for an even remotely related problem saved as an HDF5 (or PDB) file. Targeting this pre-existing-case mesh size can help lead to a new solution quickly. However, even this pre-existing case should preferably be of moderate size (one can use the UEDGE interpolation to decrease mesh size on the pre-existing problem), as the new case will need to begin with the mesh size of the pre-existing case. More detail is given below for determining the initial guess (starting values) for the solution.

The user controls the radial size of the simulated region by specifying the minimum and maximum values of the poloidal flux (normalized to unity at the magnetic separatrix). There are two minimum flux values used, one each for the core and private-flux regions. The poloidal boundaries are typically (but not always) given by the position of the divertor plates in the MHD equilibrium "g" files, although these can be overridden by user input. To determine the radial size, set

```
flx.psi0min1 = 0.95          #core minimum
flx.psi0min2 = 0.98          #private flux minimum
flx.psi0max  = 1.05          #maximum flux at wall
```

In you want (or need) to specify the location that the separatrix intersects the divertor plates, you only need to give the intersection of the major radius (grd.rstrike) with the separatrix on the inner and outer divertor plates (thus two values); thus, set

```
grd.isspnew = 1              # Turns on user-specified strike point
grd.rstrike =[1.77, 2.5]     # R_major [inner, outer] at sep intersec; for Basis (1.77,2.5)
```

To determine the mesh size, the domain is divided into four poloidal sections moving clockwise poloidally from the inner divertor plate: inner divertor leg, inner core, outer core, and outer divertor leg. Radial, there are two segments moving outward from the core boundary: core and scrape-off layer (SOL). The number of cells in each of these regions is set as follows:

```
com.nxleg[0,0] = 8    # Poloidal cells in inner leg; for Basis (1,1)
com.nxleg[0,1] = 8    # Poloidal cells in outer leg; for Basis (1,2)
com.nxcore[0,0] = 8   # Poloidal cells in inner core; for Basis (1,1)
com.nxcore[0,1] = 8   # Poloidal cells in outer core; for Basis (1,2)
com.nycore[0] = 10    # Radial cells in core/PF regions; for Basis (1)
com.nysol[0] = 16     # Radial cells in SOL region; for Basis (1)
```

The user can control the radial spacing of the cells by setting

```
flx.alfcy = 2.        # if <<1, most uniform; > 1 increase sep. concentration
```

Modest control of near-X-point poloidal cells by setting

```
grd.slpxt = 1.1       # Typical 1.0->1.2; larger giving concentr. near X-point
```

STEP 3 - Specify variables to be evolved:

In UEDGE, one can turn evolution of variables on and off through various switches. The normal default is for one hydrogenic ion species with density and parallel velocity equal to an assumed electron species (ambipolar transport), electron and ion temperatures (separate), and a diffusive neutral species. Typically, the diffusive neutral transport in the poloidal direction is augmented by also solving for a full momentum equation in that direction, which requires the settings listed below:

```
bbb.isnion[0] = 1       #(=1 default) for evolving ion density; for Basis (1)
bbb.isupon[0] = 1       #(=1 default) for evolving ion parallel momentum; for Basis (1)
bbb.isteon = 1          #(=1 default) for evolving electron temperature
bbb.istion = 1          #(=1 default) for evolving ion temperature
bbb.isngon[0] = 0       #(=1 default) for evolving diffusive neutrals; for Basis (1)
bbb.nhsp = 2            #(=1 default) is number of hydrogenic species (i+g)
bbb.ziin[1] = 0         #(=1 default) is ion charge for second H species; for Basis (2)
bbb.isnion[1] = 1       #(=0 default) for evolving inertial neutrals; for Basis (2)
bbb.isupgon[0] = 1      #(=0 default) for evolving inertial neutrals; for Basis (1)
```

If any variable is turned off (=0), it is frozen at is present value.

STEP 4 - Set boundary conditions:

There are a large number of possible boundary conditions combinations as detailed in the UEDGE manual. Here we describe a standard basic set.

For the core boundary:

```
bbb.isnicore[0] = 1         #=1 uses ncore for density BC; for Basis (1)
bbb.ncore[0] = 6.0e19       #value of core density if isnicore=1; for Basis (1)
bbb.isupcore[0] = 0         #(=0 default) sets u_par = 0; for Basis (1)
bbb.iflcore = 1             #specify core power
bbb.pcoree = 5.0e7          #electron power across core
bbb.pcorei = 5.0e7          #ion power across core
bbb.isngcore[0]=2           # use ionization scale length for gas; for Basis (1)
bbb.isupcore[1] = 0         #(=0 default) sets gas u_par = 0; for Basis (2)
```

For the divertor plates:

```
isextrnp = 0                #(=0 default) sets dni/dx=0;if=1, extrapolate
isupss = 0                  #(=0 default) sets u_par=cs;if=1, extrapolate
ibctepl = 1                 #(=1 default) elec eng flux=bcee*Te*part_flux
ibctipl = 1                 #(=1 default) ion eng flux=bcee*Te*part_flux
bbb.recycp = 1.0            #ion recycling coeff. giving gas flux in
bbb.recycm = 0.1            #neut. up = -recycm*(ion up)
```

For private-flux walls:

```
bbb.isnwconi = 3            #PF wall density scale length bbb.lyni(1)
bbb.lyni[0] = 0.05          #PF wall density scale length; for Basis (1)
bbb.isupwi = 1              #(=1 default) zero ni & ng par momem-dens flux
bbb.istepfc = 3             #priv. wall has Te scale length bbb.lyte(1)
```

```
bbb.istipfc = 3          #priv. wall has Ti scale length bbb.lyti(1)
bbb.recycw = 0           #ion recycling coeff. giving gas flux in
```

For outer walls:

```
bbb.isnwcono = 3         #outer wall has fixed density scale length
bbb.lyni[1] = 0.05       #outer wall density scale length; for Basis (2)
bbb.isupwo = 1           #(=1 default) zero ni & ng par momem-dens flux
bbb.istewc=3             #outer wall has Te scale length bbb.lyte(2)
bbb.istiwc=3             #outer wall has Te scale length bbb.lyti(2)
bbb.recycw = 0           #ion recycling coeff. giving gas flux in
```

STEP 5 – Setting radial transport coefficients:

The simplest model for radial transport is that they are spatially constant, and the user may then set the following in the input file (for Basis, omit []).

```
bbb.difni[]              # Ion density diff. coeff. for each species (all m**2/s)
bbb.travis[]             # Ion parallel velocity (radial) coefficient
bbb.kye                  # Electron energy coefficient (Chi_e)
bbb.kyi                  # Ion energy coefficient (Chi_i)
bbb.vcony[]              # Ion density radial convection velocity (m/s)
```

One can also add to these coefficients a corresponding set of spatially varying coefficients that can be specified after a call to allocate to properly dimension the arrays:

```
bbb.dif_use[ix,iy,ifld]      # Ion density coeff.;3 indices are poloidal, radial, species
bbb.dif_travis[ix,iy,ifld]   # Ion parallel velocity (radial) coeff.
bbb.kye_use[ix,iy]           # Electron energy coefficient
bbb.kyi_use[ix,iy]           # ion energy coefficient
bbb.vy_use[ix,iy,ifld]       # Ion density radial convection velocity (m/s)
```

The values of these arrays can be use to better-fit profiles to experimental values. An interpretive transport mode of UEDGE (using scripts) can help specify the coefficient profiles needed to fit the plasma profiles in the core region assuming that plasma variables are approximately constant on magnetic flux surfaces.

Yet another option that is used is to specify the radial variation of the coefficients at the outer midplane, and then UEDGE provides the poloidal variation as some power of 1/B_toroidal to easily simulate "ballooning" transport, i.e., a peaking at the outer midplane. To use this option, set

```
bbb.isbohmcalc = 3       # Flag to use poloidal variation of (1/B)**inbdif
bbb.difniv[iy,ifld]      #  Ion density coeff.; indices are radial and species
bbb.travisv[iy,ifld]     # Ion parallel velocity (radial) coefficient
bbb.kyev[iy]             # Electron energy coefficient (Chi_e)
bbb.kyiv[iy]             # Ion energy coefficient (Chi_i)
bbb.vconyv[iy,ifld]      # Ion density radial convection velocity (m/s)
```

Note that for this option, the constant diffusion coefficients (e.g., bbb.difni,…) will be added, so set them to zero if a constant value is not also wanted.

STEP 6 - Choosing initial conditions:

To begin a new case, the user can either start from a set of generic profiles by setting bbb.restart = 0. However, as mentioned above, it is usually preferable to restart from an existing HDF5 or PDB solutions from another case, BUT WITH THE SAME NUMBER OF RADIAL AND POLOIDAL MESH CELLS. Even if the previous case is not from the same tokamak, this approach is usually easier. It is even better if the boundary conditions are the same or close (i.e., same core density or same core power), but this is not necessary. In this case, the input file will contain the lines

```
bbb.restart = 1
bbb.allocate()
uefacets.restore('yourfile.h5')  # or 'yourfile.pdb' for a PDB "save" file; for Basis use the line restore yourfile.pdb
```

Note: if you have a previous case that is on a different size mesh, you can use the UEDGE interpolator to produce an approximate solution on the desired mesh size by first running the prior case interactively as follows:

a) Start up by having UEDGE read the prior-case input file, but don't execute bbb.exmain yet (doesn't really hurt to do so, just takes time)

b) Then set the following:

```
bbb.issfon = 0          # Don't calculate initial Jacobian
bbb.ftol = 1.e50        # Allowed tolerance for an "accepted" solution
bbb.exmain()
```

c) Now change the mesh values (bbb.nxleg, bbb.nxcore, bbb.nycore, bbb.nysol) to the desire values and type (UEDGE automatically interpolates between meshes)

```
bbb.exmain()
```

d) Finally save the "accepted" solution on the new mesh into a HDF5 or PDB file for use as initial conditions in the new case

Lastly note that it is possible to set the initial UEDGE profiles manually after a call to the allocate routine that allocates the nis, ups, etc. arrays. At this point, one can fill these arrays interactively (i.e., from PYTHON or BASIS command line) with any profiles you want, although this does require knowledge of the various geometry regions (core, SOL, private flux) to obtain sensible profiles.

STEP 7 - Obtaining an initial solution:

After creating the desire input file as described above, it is best to develop a new steady-state solution by running UEDGE interactively. The approach taken is a conservative one, outlining a series of substeps taken if the new case is substantially different than a previous case used as a template. If the new case is similar to a previous case, the procedure can often be shortened. As the user becomes more experienced, or similar. As mentioned above, it is prudent to begin new cases with modest- sized meshes, and then interpolate to finer meshes. Granted, this procedure can be time-consuming, but it generally is the most efficient and effective strategy.

7.1) Startup UEDGE and read the input file

7.2) Set

```
bbb.dtreal = 1e-9       # UEDGE time-step (s)
bbb.itermx = 15         # Maximum number on nonlinear iterations
bbb.exmain()
```

Note: if the new case is similar to a previous case, the user may be more aggressive in choosing the initial time-step, say bbb.dtreal = 1e-5, 1.e-3, or in some cases bbb.dtreal=1e20, in which case the steady-state solution is obtained in this step alone (the skip directly to 7.5).

7.3) If convergence is not obtain (printed iterm .ne. 1) then skip and go to 7.4. If convergence is obtained at this single small time-step, then run UEDGE to steady-state using

```
bbb.rundt()   # for older Basis versions, use the scripts rdinitdt and rdcontdt
```

This subroutine advances the time-step progressively over a number of time-steps to reach a steady-state solution. Completing this, save the new solution - you are done with the initial case; go to 7.5.

7.4) If the initial try in 7.2 does not converge, then

```
bbb.isbcwdt = 1         # Turns on the time-step for boundary equations
bbb.dtreal = 1.e-10
bbb.exmain()
```

If UEDGE convergences, type

```
bbb.t_stop = 1.e-7
bbb.rundt()  # for older Basis versions, use the scripts rdinitdt and rdcontdt
```

If this finishes, you will be at a total accumulated time of 1e-7 secs. Then type

```
bbb.isbcwdt = 0         # Return boundary condition equations algebraic
bbb.dtreal = 1.e-9
bbb.t_stop = 10.
bbb.rundt()     # for older Basis versions, use the scripts rdinitdt and rdcontdt
```

As under 7.3, this will advance the solution over many (expanding) time-steps to 10 secs (accumulated), generally long enough to have a essentially a steady-state solution. Completing this, save the new solution - you are done with the initial case; go to 7.5.

If UEDGE does not converge after the initial try of 7.4, or subsequently in this section, you can try turning off (and then back on) subsets of the equations with bbb.isnion, bbb.isupon, bbb.isteon, and bbb.istion, and repeat step 7.4. However, if you reach this stage, you probably should consult with an experienced UEDGE user.

7.5) If you started on a modest-sized mesh as recommended, you probably want to increase the mesh. At first, you can probably succeed in doubling the mesh in one direction, and returning to step 7.2 - this step should go more smoothly now. Then double the mesh in the second direction and repeat. As the mesh gets finer [say, (nx,ny)~(50,30)], especially if the case has many impurity species, you may find that doubling the mesh is too aggressive, and you will need to increment more slowly.

Once you have the desired mesh-size, the new test case can be used as to give the initial conditions to efficiently survey the sensitivity of the solution to power input, gas input, core density, impurity fraction, and other parameters of interest.