Merging Projects within HAD infrastructure

Iñaki Olabarrieta
Department of Physics and Astronomy, Louisiana State University – 202 Nicholson Hall Tower Drive, Baton Rouge, LA 70803
(Dated: February 8, 2007)

These notes explain the basic procedure about how to merge two different independent projects within the HAD infrastructure.

I. INTRODUCTION

The idea is that we have two different codes that solves different sets of equations and wish to merge them. We will assume that the different codes have different right hand sides with some coupling on the equations of motion which should be predefined beforehand. An example of such a problem is the merger of a gravitational code with a matter one, in these notes we will use the merging of hyperGR with hyperFlower as an example. In this case the coupling comes through the stress-energy tensor for the gravitational code, and through the metric coefficients for the matter code. The independent codes should have these functions defined even though they could be not dynamical (i.e. static metric for example or a stress-energy which is exactly zero).

II. IMPLEMENTATION

A. New Files

At the time this document is written there are a minimum set of files that every proyect should have. We are going to list them and comment which kind of modifications should be done in order to produce the merger of the codes.

- **setup:** Special care needs to be taken for the setup file in the case of a project merger. When declaring the `ufuncs` and `vfuncs` (and their derivatives `dufuncs` and `dvfuncs`) it should be done in two different parts, in the case of the hyperGRFlower code it is defined in the following way:

```
ufuncs[G]
{
g11,g12,g13, <...>
}
ufuncs[M]
{
D[FV][diss=0], Sx[FV][diss=0], Sy[FV][diss=0], <...>
}
dufuncs[G]
{
dx_g11, dy_g11, dz_g11, <...>
}
dufuncs[M]
{
}
vfuncs[G]
{
Tmunu00, Tmunu01, Tmunu02, <...>
}
vfuncs[M]
{
}
<...>
```

  where `<...>` denotes that we haven't written in this document all the actual functions for reasons of brevity. We note that the declaration of the ufuncs is done in two steps, the functions for the hyperGR code, which we distinguish by a label `[G]` (Gravity variables) and the hyperFlower code denoted by `[M]` (Matter variables).

Note also that the Matter variables are finite volume `[FV]` variables with no dissipation`[diss=0]`, that may or may not be your case. These labels are going to produce a separate set of pointers that can be used in the implementation to distiguish between the variables from the two different projects. Therefore three different sets of pointers are going to be produced in hyperparam.f90, for example the function `g11` will have associated to it the following pointers: `H_G11`, `G_G11` and `M_G11`. Depending if we consider it as a function of the combined project, the hyperGR project of the matter project. Note that this function is defined in the three projects (not always the case), in hyperGR and hyperGRFlower it is a dynamical (`u` variable) while in the hyperFlower code it is fixed (`v` or `w` variable). We are denoting the number of u variables of the combine code as `NU`, `NU_G` are the number of `U` variables which are denoted by the `G_` label and `NU_M` are the ones with `M_` label. Likelywise with the `v` variables. The following relationships hold for the number of functions of each type:

```
NU   = NU_G + NU_M
NV_M = NV   + NU_G
NV_G = NV   + NU_M
```

Note that the `v[G]` (and alternatively the `v[M]`) are the `v` variables which are in hyperGR and that we are not previously declared as a `u` variable in the combine code, i.e. the metric function `g11` is declared as `u` variable in hyperFlower but as a v variable in hyperFlower, in the combined code it will be declared as a `u[G]` variable and not as a `v[M]` one. Note that in the example above there is no `v[M]` variable this is not because hyperFlower does not have any `v` variable but because all the `v` variables are already declared as `u[G]` variables. In the way we have declared the variables in the setup file the first `NU_G` variables of the `u` vector of the combine code corresponds to the geometric `u` variables and the other `NU_M` corresponds to the matter ones. The number of `v` variables of any independent project is always equal or larger than the number of `v` variables of the combined one (something that could seem counter intuitive).

The parameters of the combined project should have at least the union of the parameters of the two independent codes plus any new parameter that you may include. The declaration of the parameters does not have any special secret in the case of the combined project.

- **Makefile:** Now the Makefile should compile all the files needed for both codes. In order to do the compilation the makefile makes symbolic links to the files on the appropiate directories. Define enviroment variables to store the locations of the directories for the projects that we are merging: For example:

```
GRFLOWER_DIR = $(HEADDIR)/src/hyperGRFlower
FLOWER_DIR   = $(HEADDIR)/src/hyperFlower
GR_DIR       = $(HEADDIR)/src/hyperGR
```

In the previous case we are merging projects hyperFlower and hyperGR into hyperGRFlower. The following gives an example of how to do the symbolic links:

```
        if test ! -f exact_gr.f90; then\
                ln -s $(GR_DIR)/exact.f90 ./exact_gr.f90;\
        fi
```

Note that in the previous example we are linking the file from the hyperGR project to the current directory and we are also changing its name adding `_gr` label to it to denote that this is the "exact.f90" file from the hyperGR code. This needs to be done for every filename that may be duplicated. When doing the symbolic linking do not forget to link also any include file needed to the appropiate subdirectory in the combined project.

Another important modification to the Makefile is the addition to the compilation command of `-DGR_FLOWER -DHYPER` this is done in order to have the variables `GR_FLOWER` and `HYPER` defined for their use in preprocessor conditionals. See more about this in the section related the modification of the Old Files.

```
.f.o:
        $(F90_FIXED) -DGR_FLOWER -DHYPER -c $*.f
.F90.o:
        $(F90_FIXED) -DGR_FLOWER -DHYPER -c $*.F90
.f90.o:
        $(CPP) -DGR_FLOWER -DHYPER $*.f90 $*.f90.f
        $(F90_) -c $*.f90.f
        mv $*.f90.o $*.o
        rm -f $*.f90.f
.c.o:
        $(CC_) -DGR_FLOWER -c $*.c
```

- **main_rhs.f90:** This file could have different names (rhs.f90, maple_rhs.f90 have been used in different HAD projects) the important thing is that this file should contain the routine which computes the right hand side necessary for the rk routine being used in the hyper directory (calrhs and calcrhs2 have being used depending if the routine does the calculation point- or grid-wise). Right now there are three different rk routines avaiable in the hyper directory: rk3.f90, rk3tvd.f90, rk3tvd_fv.f90 with different options each one. You should look in detail how this routine works and which one is more appropiate for your particular project.

  The calcrhs2 routine basically consists in calls to the original calcrhs2 routines for each independent project. Remember that these routines does not know now that `u` and `v` are the ones for the combined project and are expecting the `u` and `v` for each independent code. In order to pass the appropiate piece of the original `u` and `v` vector we declare the following grid functions:

```
    type(gridfunction), dimension(NU_G) :: gr_dtu
    type(gridfunction), dimension(NU_G) :: gr_u, gr_dxu, gr_dyu, gr_dzu
    type(gridfunction), dimension(NV_G) :: gr_v, gr_dxv, gr_dyv, gr_dzv

    type(gridfunction), dimension(NU_M) :: m_dtu
    type(gridfunction), dimension(NU_M) :: m_u, m_dxu, m_dyu, m_dzu
    type(gridfunction), dimension(NV_M) :: m_v, m_dxv, m_dyv, m_dzv
```

  and then initialize them in the following manner:

```
    do i = 1, NU_G
      gr_u  (i) = u  (i)
      gr_dxu(i) = dxu(i)
      gr_dyu(i) = dyu(i)
      gr_dzu(i) = dzu(i)
      gr_dtu(i) = dtu(i)
    end do
    do i = 1, NU_M
      m_u  (i) = u  (NU_G+i)
      m_dxu(i) = dxu(NU_G+i)
      m_dyu(i) = dyu(NU_G+i)
      m_dzu(i) = dzu(NU_G+i)
      m_dtu(i) = dtu(NU_G+i)
    end do
    do i = 1, NV
      gr_v  (i) = v  (i)
      gr_dxv(i) = dxv(i)
      gr_dyv(i) = dyv(i)
      gr_dzv(i) = dzv(i)
    end do
    do i = 1, NU_M
      gr_v  (NV+i) = m_u  (i)
      gr_dxv(NV+i) = m_dxu(i)
      gr_dyv(NV+i) = m_dyu(i)
      gr_dzv(NV+i) = m_dzu(i)
    end do
    do i = 1, NV
      m_v  (i) = v  (i)
      m_dxv(i) = dxv(i)
      m_dyv(i) = dyv(i)
    m_dzv(i) = dzv(i)
  end do
  do i = 1, NU_G
    m_v  (NV+i) = gr_u  (i)
    m_dxv(NV+i) = gr_dxu(i)
    m_dyv(NV+i) = gr_dyu(i)
    m_dzv(NV+i) = gr_dzu(i)
  end do
```

where we have made use of the fact that the `_G` variables are at the begining of the `u` vector and `_M` ones are at the end. Now we can call to the appropiate routines with the appropiate vectors:

```
  call update_gr     (gr_dtu, gr_u,   gr_v,                      &
                      gr_dxu, gr_dyu, gr_dzu,                    &
                      gr_dxv, gr_dyv, gr_dzv, w, imask, par);
  call update_flower (m_dtu,  m_u,    m_v,    w,                 &
                      m_dxu,  m_dyu,  m_dzu,                     &
                      m_dxv,  m_dyv,  m_dzv,   imask, par);
```

We note that we have modified the original names of these routines (which did not have the `_flower` or the `_gr` labels attached to them) in order to not have overloading of subroutine names. The final part of this routine involves joining together the right hand side obtained from each routine into the original `u` vector:

```
  do i = 1, NU_G
     dtu(i) = gr_dtu(i)
  end do
  do i = 1, NU_M
     dtu(i+NU_G) = m_dtu(i)
  end do
```

This is the basic idea to apply in most of the files of the combined project. One more thing to take into account is that now we are making reference to routines that are in modules that are called in the original projects as the module where we are using it. The names of the original modules have to be changed, see the Old Files section, and here you want to add the new names of the modules. For the example above:

```
module rhs_mine
   use GF
   use params
   use flower_rhs
   use gr_rhs
   use HYPER_DERIVS
   implicit none
```

we have added the use of `flower_rhs` and `gr_rhs` which are the new names of the modules containing the rhs for the original projects.

- **boundary.f90:** In the case of hyperGRFlower the boundary conditions for the fluid are built in the calculation of the right hand side. Therefore the only boundary conditions that need to be applied are the ones on the gravitational ones. Note that these can be applied on the right hand side `gr_dtu` or on the vector `gr_u` itself therefore after calling the routine you should put back both of the vectors to the `u` vector of the combined project.
- **comp_amr_error.f, grid_ell.f:** This file probably should be produced independtly of the particular comp_amr_error.f from the orginal projects.
- **exact.f90:** Nothing to do here, move along
- **global_auxvars.f90, auxvars.f90:** This is a good place to calculate some of the coupling between the two codes. In particular in the case of hyperGRFlower global_auxvars has a call to the auxvars routine where the calculation of the stress-energy tensor is performed.

- **outer.f90**: Nothing special to do here.
- **renaming.inc**: This file is new at this point. In this file you should write the relationship between pointers in the two different codes. The problem is that grid functions in the two different projects could be named in a different way. In this file we set the names between the pointers to be equal:

  ```
  integer, parameter:: M_ALPHA = M_LAPSE
  integer, parameter:: M_D111  = M_D1G11
  integer, parameter:: M_D211  = M_D2G11
  ```

  In the example above in one project the `ALPHA` function is called `LAPSE` in the other one, `D111` is called `D1G11` and `D211` is called D2G11. The last modification is adding at the end of hyperparam.F90, once it is created, a line:

  ```
  include 'renaming.inc'
  ```

B. Old Files

In this section we describe the modifications needed in the old projects in order to be able to merge them. Two main modifications need to be done in the files belonging to the original projects. Both modifications should not alter the way the independent project work, and this should be checked.

The first modification involves the change of any pointer that may appear in the original projects. In order to do that the setup file has to be changed plus any explicit appearence of pointers. In the setup file (remember this is the old setup file sitting in the original project directory, i.e. hyperGR and hyperFlower in the case of our example) we want to add the same label we use to denote this particular project variables in the combined code. For example in the case of hyperGR/setup it will look something like:

```
ufuncs [G]
{
g11,g12,g13, <...>
}
vfuncs [G]
{
Tmunu00,Tmunu01,Tmunu02, <...>
}
```

where the only modification is the `[G]` label (there is no `[M]` label at all). Note that this will produce a new set of `G_` pointers which, in this case and since there is no other label, will be exactly the same as the original `H_` pointers. Now every appearance of `H_` pointers in any routine of the hyperGR code should be changed to `G_` pointers. This will have no effect in the case of compiling hyperGR by itself since, again, `H_` pointers are the same as `G_` ones but will reference to the proper location in memory for the combined code. Note that also the size of the arrays have to be changed from `NU` to `NU_G` and `NV` to `NV_G`.

The second main change involves changing the names of any routine that appear more than once in the combined project. A tipical example of this is the `calcrhs2` routines. Both projects that we are combining are going to have the same name for different routines (and it should be that way in the HAD infrastructure if we want to compile that project by itself). In orther not to overload the routine names we make use of preprocessor statements, for example in the case of the routine prim2char in the hyperGR project we have the following:

```
#if defined GR_SF || defined GR_FLOWER
  subroutine prim2char_gr(par,v,w,direction,u,w_in,w_out,w_0,T,uevolved,uexact,dxu,dyu,dzu,sources)
#else
  subroutine prim2char   (par,v,w,direction,u,w_in,w_out,w_0,T,uevolved,uexact,dxu,dyu,dzu,sources)
#endif
<...>
#if defined GR_SF || defined GR_FLOWER
  end subroutine prim2char_gr
#else
  end subroutine prim2char
#endif
```

Note that we are changing the name of the routine in case the variables `GR_FLOWER` or `GR_SF` are defined while compiling, remember the `-D` option we included in the compiling command in the Makefile of the combined project. In this case we are considering the possibility of joining hyperGR with two different codes (`GR_FLOWER` corresponds to the merger of hyperGR with hyperFlower and `GR_SF` corresponds to the merger with a scalar field project). The same has to be done with the module names:

```
#if defined GR_SF || defined GR_FLOWER
module charvars_gr
#else
module charvars
#endif
<...>
#if defined GR_SF || defined GR_FLOWER
  public prim2char_gr, char2prim_gr, what_boundary
#else
  public prim2char, char2prim
#endif
<...>
#if defined GR_SF || defined GR_FLOWER
end module charvars_gr
#else
end module charvars
#endif
```

are the proper modifications in the case of the charvars module in the hyperGR project.