Method of Lines

Ian Hawke

Date: 2006/09/20 08:55:54

Abstract

The Method of Lines is a way of separating the time integration from the rest of an evolution scheme. This thorn is intended to take care of all the bookwork and provide some basic time integration methods, allowing for easy coupling of different thorns.

1 Purpose

The Method of Lines (MoL) converts a (system of) partial differential equation(s) into an ordinary differential equation containing some spatial differential operator. As an example, consider writing the hyperbolic system of PDE's

$$\partial_t \mathbf{q} + \mathbf{A}^i(\mathbf{q})\partial_i \mathbf{B}(\mathbf{q}) = \mathbf{s}(\mathbf{q}) \tag{1}$$

in the alternative form

$$\partial_t \mathbf{q} = \mathbf{L}(\mathbf{q}), \tag{2}$$

which (assuming a given discretization of space) is an ODE.

Given this separation of the time and space discretizations, well known stable ODE integrators such as Runge-Kutta can be used to do the time integration. This is more modular (allowing for simple extensions to higher order methods), more stable (as instabilities can now only arise from the spatial discretization or the equations themselves) and also avoids the problems of retaining high orders of convergence when coupling different physical models.

MoL can be used for hyperbolic, parabolic and even elliptic problems (although I definitely don't recommend the latter). As it currently stands it is set up for systems of equations in the first order type form of equation (2). If you want to implement a multilevel scheme such as leapfrog it is not obvious to me that MoL is the thing to use. However if you have lots of thorns that you want to interact, for example ADM_BSSN and a hydro code plus maybe EM or a scalar field, and they can easily be written in this sort of form, then you probably want to use MoL.

This thorn is meant to provide a simple interface that will implement the MoL inside Cactus as transparently as possible. It will initially implement only the optimal Runge-Kutta time integrators (which are TVD up to RK3, so suitable for hydro) up to fourth order and iterated Crank Nicholson. All of the interaction with the MoL thorn should occur directly through the scheduler. For example, all synchronization steps should now be possible at the schedule level. This is essential for interacting cleanly with different drivers, especially to make Mesh Refinement work.

For more information on the Method of Lines the most comprehensive references are the works of Jonathan Thornburg [1, 2] - see especially section 7.3 of the thesis. From the CFD viewpoint the review of ENO

methods by Shu, [3], has some information. For relativistic fluids the paper of Neilsen and Choptuik [4] is also quite good.

2 How to use

2.1 Thorn users

For those who used the old version of MoL, this version is unfortunately slightly more effort to use. That is, for most methods you'll now have to set 4 parameters instead of just one.

If you already have a thorn that uses the method of lines, then there are four main parameters that are relevant to change the integration method. The keyword `MoL_ODE_Method` chooses between the different methods. Currently supported are `RK2`, `RK3`, `ICN`, `ICN-Avg` and `Generic`. These are second order Runge-Kutta, third order Runge-Kutta, Iterative Crank Nicholson, Iterative Crank Nicholson with averaging, and the generic Shu-Osher type Runge-Kutta methods. To switch between the different types of generic methods there is also the keyword `Generic_Type` which is currently restricted to `RK` for the standard TVD Runge-Kutta methods (first to fourth order) and `ICN` for the implementation of the Iterative Crank Nicholson method in generic form.

Full descriptions of the currently implemented methods are given in section 4.

The parameter `MoL_Intermediate_Steps` controls the number of intermediate steps for the ODE solver. For the generic Runge-Kutta solvers it controls the order of accuracy of the method. For the `ICN` methods this parameter controls the number of iterations taken, which **does not check for stability**. This parameter defaults to 3.

The parameter `MoL_Num_Scratch_Levels` controls the amount of scratch space used. If this is insufficient for the method selected there will be an error at parameter checking time. This parameter defaults to 0, as no scratch space is required for the efficient ICN and Runge-Kutta 2 and 3 solvers. For the generic solvers this must be at least `MoL_Intermediate_Steps - 1`.

Another parameter is `MoL_Memory_Always_On` which switches on memory for the scratch space always if true and only during evolution if false. This defaults to true for speed reasons; the memory gains are likely to be limited unless you're doing something very memory intensive at initialization or analysis.

There is also a parameter `MoL_NaN_Check` that will check your RHS grid functions for NaNs using the `NaNChecker` thorn from CactusUtils. This will make certain that you find the exact grid function computing the first NaN; of course, this may not be the real source of your problem.

The parameter `disable_prolongation` only does anything if you are using mesh refinement, and in particular `Carpet`. With mesh refinement it may be necessary to disable prolongation in intermediate steps of MoL. This occurs when evolving systems containing second spatial derivatives. This is done by default in MoL. If your system is purely first order in space and time you may wish to set this to `"no"`.

Ideally, initial data thorns should always set initial data at all time levels. However, sometimes initial data thorns fail to do this. In this case you can do one of three things:

- Fix the initial data thorn. This is the best solution.
- If you're using Carpet, it has some facilities to take forward/backward time steps to initialize multiple time levels. See the Carpet parameters `init_each_timelevel` and `init_3_timelevels`

for details.

- Finally, if you set (the MoL parameter) `initial_data_is_crap`, MoL will copy the current time level of all variables it knows about (more precisely, using the terminology of section 2.2, all evolved, save-and-restore, and constrained variables which have multiple time levels) to all the past time levels. **Note that this copies the same data to each past time level; this will be wrong if your spacetime is time-dependent!**

  If enabled, the copy happens in the `CCTK_POSTINITIAL` schedule bin. By default this happens *before* the `MoL_PostStep` schedule group; the parameter `copy_ID_after_MoL_PostStep` can be used to change this to *after* `MoL_PostStep`.

2.2 Thorn writers

To port an existing thorn using the method of lines, or to write a new thorn using it, should hopefully be relatively simple. As an example, within the MoL arrangement is WaveMoL which duplicates the WaveToy thorn given by CactusWave in a form suitable for use by MoL. In this section, "the thorn" will mean the user thorn doing the physics.

We start with some terminology. Grid functions are split into four cateogories.

1. The first are those that are evolved using a MoL form. That is, a right hand side is calculated and the variable updated using it. These we call *evolved* variables.
2. The second category are those variables that are set by a thorn at every intermediate step of the evolution, usually to respect the constraints. Examples of these include the primitive variables in a hydrodynamics code. Another example would be the gauge variables if these were set by constraints at every intermediate step (which is slightly artificial; the usual example would be the use of maximal slicing, which is only applied once every N complete steps). These are known as *constrained* variables.
3. The third category are those variables that a thorn depends on but does not set or evolve. An example would include the metric terms considered from a thorn evolving matter. Due to the way that MoL deals with these, they are known as *Save and Restore* variables.
4. The final category are those variables that do not interact with MoL. These would include temporary variables for analysis or setting up the initial data. These can safely be ignored.

As a generic rule of thumb, variables for which you have a time evolution equation are *evolved* (obviously), variables which your thorn sets but does not evolve are *constrained*, and any other variables which your thorn reads during evolution is a *Save and Restore* variable.

MoL needs to know every GF that falls in one of the first three groups. If a GF is evolved by one thorn but is a constrained variable in another (for example, the metric in full GR Hydro) then each thorn should register the function as they see it. For example, the hydro thorn will register the metric as a Save and Restore variable and the spacetime thorn will register the metric as an evolved variable. The different variable categories are given the priority evolved, constrained, Save and Restore. So if a variable is registered as belonging in two different categories, it is always considered by MoL to belong to the category with the highest priority.

MoL needs to know the total number of GFs in each category *at parameter time*. To do this, your thorn needs to use some accumulator parameters from MoL. As an example, here are the paramaters from WaveMoL:

```
shares: MethodOfLines

USES CCTK_INT MoL_Num_Evolved_Vars
USES CCTK_INT MoL_Num_Constrained_Vars
USES CCTK_INT MoL_Num_SaveAndRestore_Vars

restricted:

CCTK_INT WaveMoL_MaxNumEvolvedVars \
    "The maximum number of evolved variables used by WaveMoL" \
    ACCUMULATOR-BASE=MethodofLines::MoL_Num_Evolved_Vars
{
  5:5           :: "Just 5: phi and the four derivatives"
} 5

CCTK_INT WaveMoL_MaxNumConstrainedVars \
    "The maximum number of constrained variables used by WaveMoL" \
    ACCUMULATOR-BASE=MethodofLines::MoL_Num_Constrained_Vars
{
  0:1           :: "A small range, depending on testing or not"
} 1

CCTK_INT WaveMoL_MaxNumSandRVars \
    "The maximum number of save and restore variables used by WaveMoL" \
    ACCUMULATOR-BASE=MethodofLines::MoL_Num_SaveAndRestore_Vars
{
  0:1           :: "A small range, depending on testing or not"
} 1
```

This should give the *maximum* number of variables that your thorn will register.

Every thorn should register every grid function that it uses even if you expect it to be registered again by a different thorn. For example, a hydro thorn would register the metric variables as Save and Restore, whilst the spacetime evolution thorn would register them as evolved (in ADM) or constrained (in ADM_BSSN), both of which have precedence. To register your GFs with MoL schedule a routine in the bin `MoL_Register` which just contains the relevant function calls. For an evolved variable the GF corresponding to the update term ($\mathbf{L}(\mathbf{q})$ in equation (2)) should be registered at the same time. The appropriate functions are given in section 5.

These functions are provided using function aliasing. For details on using function aliasing, see sections B10.5 and F2.2.3 of the UsersGuide. For the case of real GFs, you simply add the following lines to your `interface.ccl`:

```
########################################
### PROTOTYPES - DELETE AS APPLICABLE! ###
########################################

CCTK_INT FUNCTION MoLRegisterEvolved(CCTK_INT EvolvedIndex, CCTK_INT RHSIndex)
CCTK_INT FUNCTION MoLRegisterConstrained(CCTK_INT ConstrainedIndex)
CCTK_INT FUNCTION MoLRegisterSaveAndRestore(CCTK_INT SandRIndex)
CCTK_INT FUNCTION MoLRegisterEvolvedGroup(CCTK_INT EvolvedIndex, \
                                          CCTK_INT RHSIndex)
```

```
CCTK_INT FUNCTION MoLRegisterConstrainedGroup(CCTK_INT ConstrainedIndex)
CCTK_INT FUNCTION MoLRegisterSaveAndRestoreGroup(CCTK_INT SandRIndex)
CCTK_INT FUNCTION MoLChangeToEvolved(CCTK_INT EvolvedIndex, CCTK_INT RHSIndex)
CCTK_INT FUNCTION MoLChangeToConstrained(CCTK_INT ConstrainedIndex)
CCTK_INT FUNCTION MoLChangeToSaveAndRestore(CCTK_INT SandRIndex)
CCTK_INT FUNCTION MoLChangeToNone(CCTK_INT RemoveIndex)

############################################
### USE STATEMENT - DELETE AS APPLICABLE! ###
############################################

USES FUNCTION MoLRegisterEvolved
USES FUNCTION MoLRegisterConstrained
USES FUNCTION MoLRegisterSaveAndRestore
USES FUNCTION MoLRegisterEvolvedGroup
USES FUNCTION MoLRegisterConstrainedGroup
USES FUNCTION MoLRegisterSaveAndRestoreGroup
USES FUNCTION MoLChangeToEvolved
USES FUNCTION MoLChangeToConstrained
USES FUNCTION MoLChangeToSaveAndRestore
USES FUNCTION MoLChangeToNone
```

Note that the list of paramters not complete; see the section on parameters for the use of arrays or complex variables. However, the list of functions is, and is expanded on in section 5. MoL will check whether a group or variable is a GF or an array and whether it is real or complex. Note that currently complex variable support is disabled.

Having done that, one routine (or group of routines) which we'll here call `Thorn_CalcRHS` must be defined. This does all the finite differencing that you'd usually do, applied to $\mathbf{q}$, and finds the right hand sides which are stored in $\mathbf{L}$. This routine should be scheduled in `MoL_CalcRHS`. The precise order that these are scheduled should not matter, because no updating of any of the user thorns $\mathbf{q}$ will be done until after all the RHSs are calculated. **Important note:** all the finite differencing must be applied to the most recent time level $\mathbf{q}$ and not to the previous time level $\mathbf{q}_p$ as you would normally do. Don't worry about setting up the data before the calculation, as MoL will do that automatically.

Finally, if you have some things that have to be done after each update to an intermediate level, these should be scheduled in `MoL_PostStep`. Examples of things that need to go here include the recalculation of primitive variables for hydro codes, the application of boundary conditions[1], the solution of elliptic equations (although this would be a very expensive place to solve them, some sets of equations might require the updating of some variables by constraints in this fashion). When applying boundary conditions the cleanest thing to do is to write a routine applying the symmetries to the appropriate GFs and, when calling it from the scheduler, adding the `SYNC` statement to the appropriate groups. An example is given by the routine `WaveToyMoL_Boundaries` in thorn WaveMoL.

Points to note. The thorn routine `Thorn_CalcRHS` does not need to know and in fact should definitely not know where precisely in the MoL step it is. It just needs to know that it is receiving *some* intermediate data stored in the GFs $\mathbf{q}$ and that it should return the RHS $\mathbf{L}(\mathbf{q})$. All the book-keeping to ensure that it is passed the correct intermediate state at that the GFs contain the correct data at the end of the MoL step will be dealt with by the MoL thorn and the flesh.

[1] It is possible to alter the calculation of $\mathbf{L}$ so that boundary conditions are automatically updated and do not need setting. This is slightly tricksy. For an example of how this would work see the new radiative boundary condition in ADM_BSSN. For more on this see section 7.3.4 of [1].

2.3 Evolution method writers

If you want to try adding a new evolution method to MoL the simplest way is to use the generic table option to specify it completely in the parameter file - no coding is required at all.

All the generic methods evolve the equation

$$\partial_t \mathbf{q} = \mathbf{L}(\mathbf{q}) \tag{3}$$

using the following algorithm for an N-step method:

$$\begin{aligned} \mathbf{q}^{(0)} &= \mathbf{q}^n, \\ \mathbf{q}^{(i)} &= \sum_{k=0}^{i-1} \left(\alpha_{ik} \mathbf{q}^{(k)} \right) + \Delta t \beta_{i-1} \mathbf{L}(\mathbf{q}^{(i-1)}), \qquad i = 1, \ldots, N, \\ \mathbf{q}^{n+1} &= \mathbf{q}^{(N)}. \end{aligned} \tag{4}$$

This method is completely specified by N (`GenericIntermediateSteps`) and the α (`GenericAlphaCoeffs`) and β (`GenericBetaCoeffs`) arrays. The names in parentheses give the keys in a table that MoL will use. This table is created from the string parameter `Generic_Method_Descriptor`.

As an example, the standard TVD RK2 method that is implemented both in optimized and generic form is written as

$$\mathbf{q}^{(1)} = \mathbf{q}^n + \Delta t \mathbf{L}(\mathbf{q}^n), \tag{5}$$

$$\mathbf{q}^{n+1} = \frac{1}{2} \left(\mathbf{q}^n + \mathbf{q}^{(1)} + \Delta t \mathbf{L}(\mathbf{q}^{(1)}) \right). \tag{6}$$

To implement this using the generic table options, use

```
methodoflines::MoL_Intermediate_Steps = 2
methodoflines::MoL_Num_Scratch_Levels = 1
methodoflines::Generic_Method_Descriptor = \
           "GenericIntermediateSteps = 2 \
            GenericAlphaCoeffs = { 1.0 0.0 0.5 0.5 } \
            GenericBetaCoeffs = { 1.0 0.5 }"
```

The number of steps specified in the table must be the same as `MoL_Intermediate_Steps`, and the number of scratch levels should be at least `MoL_Intermediate_Steps - 1`.

The generic methods are somewhat inefficient for use in production runs, so it is frequently better to write an optimized version once you are happy with the method. To do this you should

- write your code into a new file, called from the scheduler under the alias `MoL_Add`,
- make certain that at each intermediate step the correct values of `cctk_time` and `cctk_delta_time` are set in `SetTime.c` for mesh refinement, boundary conditions and so on,
- make certain that you check for the number of intermediate steps in `ParamCheck.c`.

3 Example

As a fairly extended example of how to use MoL I'll outline how ADM_BSSN works in this context. The actual implementation of this is given in the thorn `AEIThorns/BSSN_MoL`.

As normal the required variables are defined in the `interface.ccl` file, together with the associated source terms. For example, the conformal factor and source are defined by

```
real ADM_BSSN_phi type=GF timelevels=2
{
  ADM_BS_phi
} "ADM_BSSN_phi"

real ADM_BSSN_sources type=GF
{
...,
  adm_bs_sphi,
...
}
```

Also in this file we write the function aliasing prototypes.

Once the sources are defined the registration with MoL is required, for which the essential file is `MoLRegister.c`. In the ADM_BSSN system the standard metric coefficients g_{ij} are not evolved, and neither are the standard extrinsic curvature components K_{ij}. However these are used by ADM_BSSN in a number of places, and are calculated from evolved quantities at the appropriate points. In the MoL terminology these variables are *constrained*. As the appropriate storage is defined in thorn ADMBase, the actual calls have the form

```
 ierr += MoLRegisterConstrained(CCTK_VarIndex("ADMBase::kxx"));
```

The actual evolved variables include things such as the conformal factor. This, and the appropriate source term, is defined in thorn ADM_BSSN, and so the call has the form

```
 ierr += MoLRegisterEvolved(CCTK_VarIndex("adm_bssn::ADM_BS_phi"),
                            CCTK_VarIndex("adm_bssn::adm_bs_sphi"));
```

As well as the evolved variables, and those constrained variables such as the metric, there are the gauge variables. Precisely what status these have depends on how they are set. If harmonic or 1+log slicing is used then the lapse is evolved:

```
 ierr += MoLRegisterEvolved(CCTK_VarIndex("ADMBase::alp"),
                            CCTK_VarIndex("adm_bssn::adm_bs_salp"));
```

If maximal or static slicing is used then the lapse is a constrained variable[2]:

[2]Note that this is actually a bit of a hack. The rational for *Save and Restore* variables was to deal with maximal slicing. However it turned out that I hadn't thought it through correctly and that the treatment for constrained variables was required.

```
ierr += MoLRegisterConstrained(CCTK_VarIndex("ADMBase::alp"));
```

Finally, if none of the above apply we assume that the lapse is evolved in some unknown fashion, and so it must be registered as a Save and Restore variable:

```
ierr += MoLRegisterSaveAndRestore(CCTK_VarIndex("ADMBase::alp"));
```

However, it is perfectly possible that we may wish to change how we deal with the gauge during the evolution. This is dealt with in the file `PreLoop.F`. If the slicing changes then the appropriate routine is called. For example, if we want to use 1+log evolution then we call

```
call CCTK_VarIndex(lapseindex,"ADMBase::alp")
call CCTK_VarIndex(lapserhsindex,"adm_bssn::adm_bs_salp")
ierr = ierr + MoLChangeToEvolved(lapseindex, lapserhsindex)
```

It is not required to tell MoL what the lapse is changing *from*, or indeed if it is changing at all; MoL will work this out for itself.

Finally there are the routines that we wish to apply after every intermediate step. These are `ADM_BSSN_removetrA` which enforces various constraints (such as the tracefree conformal extrinsic curvature remaining trace free), `ADM_BSSN_Boundaries` which applies symmetry boundary conditions as well as various others (such as some of the radiative boundary conditions). Note all the calls to `SYNC` at this point. We also convert from the updated BSSN variables back to the standard ADM variables in `ADM_BSSN_StandardVariables`, and also update the time derivative of the lapse in `ADM_BSSN_LapseChange`.

4 Time evolution methods provided by MoL

The default method is Iterative Crank-Nicholson. There are many ways of implementing this. The standard `"ICN"` and `"Generic"`/`"ICN"` methods both implement the following, assuming an N iteration method:

$$\begin{aligned}
\mathbf{q}^{(0)} &= \mathbf{q}^{n}, && (7)\\
\mathbf{q}^{(i)} &= \mathbf{q}^{(0)} + \frac{\Delta t}{2}\mathbf{L}(\mathbf{q}^{(i-1)}), \quad i = 1, \ldots, N-1, && (8)\\
\mathbf{q}^{(N)} &= \mathbf{q}^{(N-1)} + \Delta t \mathbf{L}(\mathbf{q}^{(N-1)}), && (9)\\
\mathbf{q}^{n+1} &= \mathbf{q}^{(N)} && (10)
\end{aligned}$$

The "averaging" ICN method `"ICN-avg"` instead calculates intermediate steps before averaging:

$$\begin{aligned}
\mathbf{q}^{(0)} &= \mathbf{q}^{n}, && (11)\\
\tilde{\mathbf{q}}^{(i)} &= \frac{1}{2}\left(\mathbf{q}^{(i)} + \mathbf{q}^{n}\right), \quad i = 0, \ldots, N-1 && (12)\\
\mathbf{q}^{(i)} &= \mathbf{q}^{(0)} + \Delta t \mathbf{L}(\tilde{\mathbf{q}}^{(N-1)}), && (13)\\
\mathbf{q}^{n+1} &= \mathbf{q}^{(N)} && (14)
\end{aligned}$$

The Runge-Kutta methods are those typically used in hydrodynamics by, e.g., Shu and others — see [3] for example. Explicitly the first order method is the Euler method:

$$\begin{aligned} \mathbf{q}^{(0)} &= \mathbf{q}^n, & (15)\\ \mathbf{q}^{(1)} &= \mathbf{q}^{(0)} + \Delta t \mathbf{L}(\tilde{\mathbf{q}}^{(0)}), & (16)\\ \mathbf{q}^{n+1} &= \mathbf{q}^{(1)}. & (17) \end{aligned}$$

The second order method is:

$$\begin{aligned} \mathbf{q}^{(0)} &= \mathbf{q}^n, & (18)\\ \mathbf{q}^{(1)} &= \mathbf{q}^{(0)} + \Delta t \mathbf{L}(\mathbf{q}^{(0)}), & (19)\\ \mathbf{q}^{(2)} &= \frac{1}{2}\left(\mathbf{q}^{(0)} + \mathbf{q}^{(1)} + \Delta t \mathbf{L}(\mathbf{q}^{(1)})\right), & (20)\\ \mathbf{q}^{n+1} &= \mathbf{q}^{(2)}. & (21) \end{aligned}$$

The third order method is:

$$\begin{aligned} \mathbf{q}^{(0)} &= \mathbf{q}^n, & (22)\\ \mathbf{q}^{(1)} &= \mathbf{q}^{(0)} + \Delta t \mathbf{L}(\mathbf{q}^{(0)}), & (23)\\ \mathbf{q}^{(2)} &= \frac{1}{4}\left(3\mathbf{q}^{(0)} + \mathbf{q}^{(1)} + \Delta t \mathbf{L}(\mathbf{q}^{(1)})\right), & (24)\\ \mathbf{q}^{(3)} &= \frac{1}{3}\left(\mathbf{q}^{(0)} + 2\mathbf{q}^{(2)} + 2\Delta t \mathbf{L}(\mathbf{q}^{(2)})\right), & (25)\\ \mathbf{q}^{n+1} &= \mathbf{q}^{(3)}. & (26) \end{aligned}$$

The fourth order method, which is not strictly TVD, is:

$$\begin{aligned} \mathbf{q}^{(0)} &= \mathbf{q}^n, & (27)\\ \mathbf{q}^{(1)} &= \mathbf{q}^{(0)} + \frac{1}{2}\Delta t \mathbf{L}(\mathbf{q}^{(0)}), & (28)\\ \mathbf{q}^{(2)} &= \mathbf{q}^{(0)} + \frac{1}{2}\Delta t \mathbf{L}(\mathbf{q}^{(1)}), & (29)\\ \mathbf{q}^{(3)} &= \mathbf{q}^{(0)} + \Delta t \mathbf{L}(\mathbf{q}^{(2)}), & (30)\\ \mathbf{q}^{(4)} &= \frac{1}{6}\left(-2\mathbf{q}^{(0)} + 2\mathbf{q}^{(1)} + 4\mathbf{q}^{(2)} + 2\mathbf{q}^{(3)} + \Delta t \mathbf{L}(\mathbf{q}^{(3)})\right), & (31)\\ \mathbf{q}^{n+1} &= \mathbf{q}^{(4)}. & (32) \end{aligned}$$

5 Functions provided by MoL

All the functions listed below return error codes in theory. However at this current point in time they always return 0 (success). Any failure to register or change a GF is assumed fatal and MoL will issue a level 0 warning stopping the code. This may change in future, in which case negative return values will indicate errors.

These are all *aliased* functions. You can get the functions directly through header files, but this feature may be phased out. Using function aliasing is the recommended method.

Tells MoL that the given GF is in the evolved category with the associated update GF.

Synopsis

C
```
CCTK_INT ierr = MoLRegisterEvolved(CCTK_INT EvolvedIndex,
                                   CCTK_INT RHSIndex)
```

Fortran
```
CCTK_INT ierr = MoLRegisterEvolved(CCTK_INT EvolvedIndex,
                                   CCTK_INT RHSIndex)
```

Result

Currently if there is an error, MoL will issue a level 0 warning. No sensible return codes exist.

0	success

Parameters

`EvolvedIndex`	Index of the GF to be evolved.
`RHSIndex`	Index of the associated update GF.

Discussion

Should be called in a function scheduled in `MoL_Register`.

See Also

`CCTK_VarIndex()`	Get the variable index.
`MoLRegisterSaveAndRestore()`	Register Save and Restore variables.
`MoLRegisterConstrained()`	Register constrained variables.
`MoLChangeToEvolved()`	Change a variable at runtime to be evolved.

Examples

C
```
ierr = MoLRegisterEvolved(CCTK_VarIndex("wavetoymol::phi"),
                          CCTK_VarIndex("wavetoymol::phirhs"));
```

Fortran
```
call CCTK_VarIndex(EvolvedIndex, "wavetoymol::phi")
call CCTK_VarIndex(RHSIndex, "wavetoymol::phirhs")
ierr = MoLRegisterEvolved(EvolvedIndex, RHSIndex)
```

Tells MoL that the given GF is in the constrained category.

Synopsis

C
```
CCTK_INT ierr = MoLRegisterConstrained(CCTK_INT ConstrainedIndex)
```

Fortran
```
CCTK_INT ierr = MoLRegisterConstrained(CCTK_INT ConstrainedIndex)
```

Result

Currently if there is an error, MoL will issue a level 0 warning. No sensible return codes exist.

0	success

Parameters

`ConstrainedIndex`
: Index of the constrained GF.

Discussion

Should be called in a function scheduled in `MoL_Register`.

See Also

`CCTK_VarIndex()`	Get the variable index.
`MoLRegisterEvolved()`	Register evolved variables.
`MoLRegisterSaveAndRestore()`	Register Save and Restore variables.
`MoLChangeToConstrained()`	Change a variable at runtime to be constrained.

Examples

C
```
ierr = MoLRegisterConstrained(CCTK_VarIndex("ADMBase::alp"));
```

Fortran
```
call CCTK_VarIndex(ConstrainedIndex, "ADMBase::alp")
ierr = MoLRegisterConstrained(ConstrainedIndex)
```

Tells MoL that the given GF is in the Save and Restore category.

Synopsis

C `CCTK_INT ierr = MoLRegisterSaveAndRestore(CCTK_INT SandRIndex)`

Fortran `CCTK_INT ierr = MoLRegisterSaveAndRestore(CCTK_INT SandRIndex)`

Result

Currently if there is an error, MoL will issue a level 0 warning. No sensible return codes exist.

0 success

Parameters

`SandRIndex` Index of the Save and Restore GF.

Discussion

Should be called in a function scheduled in `MoL_Register`.

See Also

`CCTK_VarIndex()`	Get the variable index.
`MoLRegisterEvolved()`	Register evolved variables.
`MoLRegisterConstrained()`	Register constrained variables.
`MoLChangeToSaveAndRestore()`	Change a variable at runtime to be Save and Restore.

Examples

C
```
ierr = MoLRegisterSaveAndRestore(CCTK_VarIndex("ADMBase::alp"));
```

Fortran
```
call CCTK_VarIndex(SandRIndex, "ADMBase::alp")
ierr = MoLRegisterSaveAndRestore(SandRIndex)
```

Tells MoL that the given group is in the evolved category with the associated update group.

Synopsis

C
```
CCTK_INT ierr = MoLRegisterEvolvedGroup(CCTK_INT EvolvedIndex,
                                        CCTK_INT RHSIndex)
```

Fortran
```
CCTK_INT ierr = MoLRegisterEvolvedGroup(CCTK_INT EvolvedIndex,
                                        CCTK_INT RHSIndex)
```

Result

Currently if there is an error, MoL will issue a level 0 warning. No sensible return codes exist.

0	success

Parameters

`EvolvedIndex`	Index of the group to be evolved.
`RHSIndex`	Index of the associated update group.

Discussion

Should be called in a function scheduled in `MoL_Register`.

See Also

`CCTK_GroupIndex()`	Get the group index.
`MoLRegisterSaveAndRestoreGroup()`	Register Save and Restore variables.
`MoLRegisterConstrainedGroup()`	Register constrained variables.

Examples

C
```
ierr = MoLRegisterEvolvedGroup(CCTK_GroupIndex("wavetoymol::scalarevolvemol"),
                               CCTK_GroupIndex("wavetoymol::scalarevolvemolrhs"));
```

Fortran
```
call CCTK_GroupIndex(EvolvedIndex, "wavetoymol::scalarevolvemol")
call CCTK_GroupIndex(RHSIndex, "wavetoymol::scalarevolvemolrhs")
ierr = MoLRegisterEvolvedGroup(EvolvedIndex, RHSIndex)
```

Tells MoL that the given group is in the constrained category.

Synopsis

C `CCTK_INT ierr = MoLRegisterConstrainedGroup(CCTK_INT ConstrainedIndex)`

Fortran `CCTK_INT ierr = MoLRegisterConstrainedGroup(CCTK_INT ConstrainedIndex)`

Result

Currently if there is an error, MoL will issue a level 0 warning. No sensible return codes exist.

0 success

Parameters

`ConstrainedIndex`
Index of the constrained group.

Discussion

Should be called in a function scheduled in `MoL_Register`.

See Also

`CCTK_GroupIndex()`	Get the group index.
`MoLRegisterEvolvedGroup()`	Register evolved variables.
`MoLRegisterSaveAndRestoreGroup()`	Register Save and Restore variables.
`MoLChangeToConstrained()`	Change a variable at runtime to be constrained.

Examples

C
```
ierr = MoLRegisterConstrainedGroup(CCTK_GroupIndex("ADMBase::lapse"));
```

Fortran
```
call CCTK_GroupIndex(ConstrainedIndex, "ADMBase::lapse")
ierr = MoLRegisterConstrainedGroup(ConstrainedIndex)
```

Tells MoL that the given group is in the Save and Restore category.

Synopsis

C `CCTK_INT ierr = MoLRegisterSaveAndRestoreGroup(CCTK_INT SandRIndex)`

Fortran `CCTK_INT ierr = MoLRegisterSaveAndRestoreGroup(CCTK_INT SandRIndex)`

Result

Currently if there is an error, MoL will issue a level 0 warning. No sensible return codes exist.

0 success

Parameters

`SandRIndex` Index of the save and restore group.

Discussion

Should be called in a function scheduled in `MoL_Register`.

See Also

`CCTK_GroupIndex()`	Get the group index.
`MoLRegisterEvolvedGroup()`	Register evolved variables.
`MoLRegisterConstrainedGroup()`	Register constrained variables.

Examples

C
```
ierr = MoLRegisterSaveAndRestoreGroup(CCTK_GroupIndex("ADMBase::shift"));
```

Fortran
```
call CCTK_GroupIndex(SandRIndex, "ADMBase::shift")
ierr = MoLRegisterSaveAndRestoreGroup(SandRIndex)
```

Sets a GF to belong to the evolved category, with the associated update GF. Not used for the initial setting.

Synopsis

C
```
CCTK_INT ierr = MoLChangeToEvolved(CCTK_INT EvolvedIndex,
                                   CCTK_INT RHSIndex)
```

Fortran
```
CCTK_INT ierr = MoLChangeToEvolved(CCTK_INT EvolvedIndex,
                                   CCTK_INT RHSIndex)
```

Result

Currently if there is an error, MoL will issue a level 0 warning. No sensible return codes exist.

0	success

Parameters

`EvolvedIndex`	Index of the evolved GF.
`RHSIndex`	Index of the associated update GF.

Discussion

Should be called in a function scheduled in `MoL_PreStep`. Note that this function was designed to allow mixed slicings for thorn ADMBase. This set of functions is largely untested and should be used with great care.

See Also

`CCTK_VarIndex()`	Get the variable index.
`MoLRegisterEvolved()`	Register evolved variables.
`MoLChangeToSaveAndRestore()`	Change a variable at runtime to be Save and Restore.
`MoLChangeToConstrained()`	Change a variable at runtime to be constrained.

Examples

C
```
ierr = MoLChangeToEvolved(CCTK_VarIndex("ADMBase::alp"),
                          CCTK_VarIndex("adm_bssn::adm_bs_salp"));
```

Fortran
```
call CCTK_VarIndex(EvolvedIndex, "ADMBase::alp")
call CCTK_VarIndex(RHSIndex,"adm_bssn::adm_bs_salp")
ierr = MoLChangeToEvolved(EvolvedIndex, RHSIndex)
```

Sets a GF to belong to the constrained category. Not used for the initial setting.

Synopsis

C `CCTK_INT ierr = MoLChangeToConstrained(CCTK_INT EvolvedIndex)`

Fortran `CCTK_INT ierr = MoLChangeToConstrained(CCTK_INT EvolvedIndex)`

Result

Currently if there is an error, MoL will issue a level 0 warning. No sensible return codes exist.

0 success

Parameters

`ConstrainedIndex`
Index of the constrained GF.

Discussion

Should be called in a function scheduled in `MoL_PreStep`. Note that this function was designed to allow mixed slicings for thorn ADMBase. This set of functions is largely untested and should be used with great care.

See Also

`CCTK_VarIndex()`	Get the variable index.
`MoLRegisterConstrained()`	Register constrained variables.
`MoLChangeToSaveAndRestore()`	Change a variable at runtime to be Save and Restore.
`MoLChangeToEvolved()`	Change a variable at runtime to be evolved.

Examples

C
```
ierr = MoLChangeToConstrained(CCTK_VarIndex("ADMBase::alp"));
```

Fortran
```
call CCTK_VarIndex(EvolvedIndex, "ADMBase::alp")
ierr = MoLChangeToConstrained(EvolvedIndex)
```

Sets a GF to belong to the Save and Restore category. Not used for the initial setting.

Synopsis

C	`CCTK_INT ierr = MoLChangeToSaveAndRestore(CCTK_INT SandRIndex)`
Fortran	`CCTK_INT ierr = MoLChangeToSaveAndRestore(CCTK_INT SandRIndex)`

Result

Currently if there is an error, MoL will issue a level 0 warning. No sensible return codes exist.

0	success

Parameters

`SandRIndex`	Index of the Save and Restore GF.

Discussion

Should be called in a function scheduled in `MoL_PreStep`. Note that this function was designed to allow mixed slicings for thorn ADMBase. This set of functions is largely untested and should be used with great care.

See Also

`CCTK_VarIndex()`	Get the variable index.
`MoLRegisterSaveAndRestore()`	Register Save and Restore variables.
`MoLChangeToEvolved()`	Change a variable at runtime to be evolved.
`MoLChangeToConstrained()`	Change a variable at runtime to be constrained.

Examples

C

```
ierr = MoLChangeToSaveAndRestore(CCTK_VarIndex("ADMBase::alp"));
```

Fortran

```
call CCTK_VarIndex(SandRIndex, "ADMBase::alp")
ierr = MoLChangeToSaveAndRestore(SandRIndex)
```

Sets a GF to belong to the "unknown" category. Not used for the initial setting.

Synopsis

C	`CCTK_INT ierr = MoLChangeToNone(CCTK_INT RemoveIndex)`
Fortran	`CCTK_INT ierr = MoLChangeToNone(CCTK_INT RemoveIndex)`

Result

Currently if there is an error, MoL will issue a level 0 warning. No sensible return codes exist.

0	success

Parameters

`RemoveIndex`	Index of the "unknown" GF.

Discussion

Should be called in a function scheduled in `MoL_PreStep`. Note that this function was designed to allow mixed slicings for thorn ADMBase. This set of functions is largely untested and should be used with great care.

See Also

`CCTK_VarIndex()`	Get the variable index.
`MoLChangeToEvolved()`	Change a variable at runtime to be evolved.
`MoLChangeToSaveAndRestore()`	Change a variable at runtime to be Save and Restore.
`MoLChangeToConstrained()`	Change a variable at runtime to be constrained.

Examples

C

```
ierr = MoLChangeToNone(CCTK_VarIndex("ADMBase::alp"));
```

Fortran

```
call CCTK_VarIndex(RemoveIndex, "ADMBase::alp")
ierr = MoLChangeToNone(RemoveIndex)
```

References

[1] J. Thornburg. Numerical Relativity in Black Hole Spacetimes. Unpublished thesis, University of British Columbia. 1993. Available from `http://www.aei.mpg.de/~jthorn/phd/html/phd.html`.

[2] J. Thornburg. A 3+1 Computational Scheme for Dynamic Spherically Symmetric Black Hole Spacetimes – II: Time Evolution. Preprint `gr-qc/9906022`, submitted to *Phys. Rev.* **D**.

[3] C. Shu. High Order ENO and WENO Schemes for Computational Fluid Dynamics. In T. J. Barth and H. Deconinck, editors *High-Order Methods for Computational Physics*. Springer, 1999. A related online version can be found under *Essentially Non-Oscillatory and Weighted Essentially Non-Oscillatory Schemes for Hyperbolic Conservation Laws* at `http://www.icase.edu/library/reports/rdp/97/97-65RDP.tex.refer.html`.

[4] D. W. Neilsen and M. W. Choptuik. Ultrarelativistic fluid dynamics. *Class. Quantum Grav.*, **17**: 733–759, 2000.