EllBase

Gerd Lanfermann

Date: 2002/08/18 20:10:28

Abstract

Infrastructure for standard elliptic solvers

1 Introduction

Following a brief introduction to the elliptic solver interfaces provided by `EllBase`, we explain how to add a new class of elliptic equations and how to implement a particular solver for any class. We do not discuss the individual elliptic solvers here since these are documented in their own thorns.

1.1 Purpose of Thorn

Thorn EllBase provides the basic functionality for

- registering a class of elliptic equations
- register a solver for any particular class

The solvers are called by the user through a unique interface, which calls the required elliptic solver for a class using the name under which the solver routine is registered.

`EllBase` itself defines the elliptic classes

1. **flat:** `Ell_LinFlat`
   solves a linear elliptic equation in flat space: $\nabla\phi + M\phi + N = 0$
2. **metric:** `Ell_LinMetric`
   solves a linear elliptic equation for a given metric: $\nabla_g\phi + M\phi + N = 0$
3. **conformal metric:** `Ell_LinConfMetric`
   solves a linear elliptic equation for a given metric and a conformal factor: $\nabla_{cg}\phi + M\phi + N = 0$
4. **generic:** solves a linear elliptic equation by passing the stencil functions. There is support for a maximum of 27 stencil functions (3^3). *This is not implemented, yet.*

2 Technical Specification

- Implements: `ellbase`
- Inherits from: `grid`
- Tested with thorns:
  `CactusElliptic/EllTest`,
  `CactusWave/IDScalarWaveElliptic`

3 ToDo

- Add more standard equation classes.
- The method for passing boundary conditions into the elliptic solvers has not fully consolidated. We have some good ideas on what the interface should look like, but the implementation will take some time. If you are worried about BCs, please contact me.

4 Solving an elliptic equation

EllBase provides a calling interface for each of the elliptic classes implemented. As a user you must provide all information needed for a particular elliptic class. In general this will include

- the gridfunction(s) to solve for
- the coefficient matrix or source terms
- information on termination tolerances
- the name of the solver to be used

Motivation: At a later stage you might want to compile with a different solver for this elliptic class: just change the name of the solver in your elliptic interface call. If somebody improves a solver you have been using, there is no need for you to change any code on your side: the interface will hide all of that. Another advantage is that your code will compile and run, even though certain solvers are not compiled in. In this case, you will have to do some return value checking to offer alternatives.

4.1 `Ell_LinFlat`

To call this interface from **Fortran**:

```
 call Ell_LinFlatSolver(ierr, cctkGH, phi_gfi, M_gfi, N_gfi,
.                       AbsTol, RelTol, "solvername")
```

To call this interface from **C**:

```
 ierr = Ell_LinFlatSolver(GH, phi_gfi, M_gfi, N_gfi,
                          AbsTol, RelTol, "solvername");
```

Argument List:

- `ierr`: return value: "0" for success.
- `cctkGH`: the Fortran "pointer" to the grid function hierachy.
- `GH`: the C pointer to the grid hierarchy, type: `pGH *GH`.
- `phi_gif`: the integer *index* of the grid function to solve for.
- `M_gfi`: the integer *index* of the grid function which holds M.
- `N_gif`: the integer *index* of the grid function which holds N.
- `AbsTol`: array of size 3: holding *absolute* tolerance values for the L_1, L_2, L_∞ norm. Check if the solver side supports these norms. The interface side does not guarantee that these norms are actually implemenented by a solver. See the section on norms: 6.
- `RelTol`: array of size 3: holding *relative* tolerance factors for the L_1, L_2, L_∞. Check if the solver side supports these norms. The interface side does not guarantee that these norms are actually implemenented by a solver. See the section on Norms: 6.

- `"solvername"`: the name of a solver, which is registered for a particular equation class. How does one find out the names? Either check the documentation of the elliptic solvers or check for registration infomation outputted by a cactus at runtime.

Example use in Fortran, as used in the WaveToy arrangement: `CactusWave/IDScalarWave`:

```
c     We derive the grid function indicies from the names of the
c     grid functions:
      call CCTK_VarIndex (Mcoeff_gfi, "idscalarwaveelliptic::Mcoeff")
      call CCTK_VarIndex (Ncoeff_gfi, "idscalarwaveelliptic::Ncoeff")
      call CCTK_VarIndex (phi_gfi,    "wavetoy::phi")

c     Load the Absolute Tolerance Arrays
      AbsTol(1)=1.0d-5
      AbsTol(2)=1.0d-5
      AbsTol(3)=1.0d-5

c     Load the Relative Tolerance Arrays, they are not
c     used here: -1
      RelTol(1)=-1
      RelTol(2)=-1
      RelTol(3)=-1

c     Call to elliptic solver, named ``sor''
      call Ell_LinFlatSolver(ierr, cctkGH,
     .     phi_gfi, Mcoeff_gfi, Ncoeff_gfi, AbsTol, RelTol,
     .     "sor")

c     Do some error checking, a call to another solver
c     could be coded here
      if (ierr.ne.0) then
         call CCTK_WARN(0,"Requested solver not found / solve failed");
      endif
```

4.2 Ell_LinMetric

To call this interface from **Fortran**:

```
 call Ell_LinMetricSolver(ierr, cctkGH, Metric_gfi,
.             phi_gfi, M_gfi, N_gfi,
.                         AbsTol, RelTol, "solvername")
```

To call this interface from **C**:

```
 ierr = Ell_LinMetricSolver(GH, Metric_gfi,
              phi_gfi, M_gfi, N_gfi,
                           AbsTol, RelTol, "solvername");
```

Argument List:

- `ierr`: return value: "0" success
- `cctkGH`: the Fortran "pointer" to the grid function hierachy.
- `GH`: the C pointer to the grid hierarchy, type: `pGH *GH`
- `Metric_gfi`: array of size 6, containing the *index* components of the metric g: g_{11}, g_{12}, g_{13}, g_{22}, g_{23}, g_{33}. The **order** is important.

- `phi_gif`: the integer *index* of the grid function so solver for.
- `M_gfi`: the integer *index* of the grid function which holds M.
- `N_gif`: the integer *index* of the grid function which holds N
- `AbsTol`: array of size 3: holding *absolute* tolerance values for the L_1, L_2, L_∞ Norm. Check, if the solver side supports these norms.The interface side does not guarantee that these norms are actually implemenented by a solver. See the section on Norms: 6.
- `RelTol`: array of size 3: holding *relative* tolerance factors for the L_1, L_2, L_∞. Check, if the solver side supports these norms. The interface side does not guarantee that these norms are actually implemenented by a solver. See the section on Norms: 6.
- `"solvername"`: the name of a solver, which is registered for a particular equation class. How to find out the names ? Either check the documentation of the elliptic solvers or check for registration infomation outputted by a cactus at runtime.

4.3 Ell_LinConfMetric

To call this interface from **Fortran**:

```
 call Ell_LinMetricSolver(ierr, cctkGH, MetricPsi_gfi,
.             phi_gfi, M_gfi, N_gfi,
.                         AbsTol, RelTol, "solvername")
```

To call this interface from **C**:

```
 ierr = Ell_LinMetricSolver(GH, MetricPsi_gfi,
              phi_gfi, M_gfi, N_gfi,
                           AbsTol, RelTol, "solvername");
```

Argument List:

- `ierr`: return value: "0" success
- `cctkGH`: the Fortran "pointer" to the grid function hierachy.
- `GH`: the C pointer to the grid hierarchy, type: `pGH *GH`
- `MetricPsi_gfi`: array of size 7, containing the *grid function index* of the metric components and the *grid function index* of the conformal factor Ψ: g_{11}, g_{12}, g_{13}, g_{22}, g_{23}, g_{33}, Ψ. The **order** is important.
- `phi_gif`: the integer *index* of the grid function so solver for.
- `M_gfi`: the integer *index* of the grid function which holds M.
- `N_gif`: the integer *index* of the grid function which holds N
- `AbsTol`: array of size 3: holding *absolute* tolerance values for the L_1, L_2, L_∞ Norm. Check, if the solver side supports these norms.The interface side does not guarantee that these norms are actually implemenented by a solver. See the section on Norms: 6.
- `RelTol`: array of size 3: holding *relative* tolerance factors for the L_1, L_2, L_∞. Check, if the solver side supports these norms. The interface side does not guarantee that these norms are actually implemenented by a solver. See the section on Norms: 6.
- `"solvername"`: the name of a solver, which is registered for a particular equation class. How to find out the names ? Either check the documentation of the elliptic solvers or check for registration infomation outputted by a cactus at runtime.

5 Extending the elliptic solver class

EllBase by itself does not provide any elliptic solving capabilities. It merely provides the registration structure and calling interface.

The idea of a unified calling interface can be motivated as follows: assume you a have elliptic problem which conforms to one of the elliptic classes defined in EllBase.

5.1 Registration Mechanism

Before a user can successfully apply a elliptic solver to one of his problems, two things need to be done by the author who programs the solver.

- **Register a class of elliptic equations** Depending on the elliptic problem This provides the unique calling, the solving routines needs to have specific input data. The interface, which is called by the user, has to reflect these arguments. EllBase already offers several of these interfaces, but if you need to have a new one, you can provide your own.
- **Register a solver for a particular elliptic equation class** Once a class of elliptic equations has been made available as described above, the author can now register solvers for that particular class. Later a user will access the solver calling the interface with the arguments needed for the elliptic class and a name, under which a solver for this elliptic problem has been registered.

The registration process is part of the authors thorn, not part of EllBase. There is no need to change code in EllBase. Usually, a author of solver routines will register the routines that register an elliptic equation class and/or an elliptic solver in the STARTUP timebin. If a author registers both, class and solver, you must make sure, that the elliptic class is registered *before* the solver registration takes place.

5.2 EllBase Programming Guide

Here we give a step by step guide on how to implement an new elliptic solver class, its interface and provide a solver for this class. Since some of the functionality needed in the registration code can only be achieved in C, a basic knowledge of C is helpful.

- **Assumption**:
  - The elliptic equation class will be called "*SimpleEllClass*": it will be flat space solver, that only takes the coefficient matrix M: Note that this solver class is already provided by EllBase.
  - The name of the demonstration thorn will be "`ThornFastSOR`". Since I will only demonstrate the registration principle and calling structure, I leave it to the interested reader to write a really fast SOR solver.
  - The solver for this elliptic equation will be called "`FastSOR_solver`" and will be written in Fortran. Since Fortran cannot be called directly by the registration mechanism, we need to have C wrapper function "`FastSOR_wrapper`".
- **Elliptic class declaration**: `SimpleEllThorn/src/SimpleEll_Class.c`
- **Elliptic solver interface**: `src/SimpleEll_Interface.c`

```
#include ‘‘cctk.h’’
#include ‘‘cctk_Parameters.h’’

#include ‘‘cctk_FortranString.h’’
#include ‘‘StoreNamedData.h’’
```

```
static pNamedData *SimpleEllSolverDB;

void Ell_SimpleEllSolverRegistry(void (*solver_func), const char *solver_name)
{
  StoreNamedData(&SimpleEllSolverDB,solver_name,(void*)solver_func);
}
```

The routine above registers the solver (or better the function pointer of the solver routine "*solve_func") for the equation class *SimpleEllClass* by the name `solver_name` in the database `SimpleEllSolverDB`. This database is declared in statement `static pNamedData....`

Next, we write our interface in the same file `./SimpleEll_Interface.c`:

```
void Ell_SimpleEllSolver(cGH *GH, int *FieldIndex, int *MIndex,
         CCTK_REAL *AbsTol, CCTK_REAL *RelTol,
                         const char *solver_name) {

/* prototype for the equation class wrapper:
   grid hierarchy(*GH), ID-number of field to solve for (*FieldIndex),
   two arrays of size three holding convergence information (*AbsTol, *RelTol)
*/
  void (*fn)(cGH *GH, int *FieldIndex, int *AbsTol, int *RelTol);

  /* derive the function name from the requested name and hope it is there */
  fn = (void(*)) GetNamedData(LinConfMetricSolverDB,solver_name);
  if (!fn) CCTK_WARN(0,''Ell_SimpleEllSolver: Cannot find solver! ‘‘);

  /* Now that we have the function pointer to our solver, call the
     solver and pass through all the necessary arguments */
  fn( GH, FieldIndex, MIndex, AbsTol, RelTol);
}
```

The interface `Ell_SimpleEllSolver` is called from the user side. It receives a pointer to the grid hierarchy, the ID-number of the field to solver for, two arrays which the used upload with convergence test info, and finally, the name of the solver the user want to employ `*solver_name`. **Note:** all these quantities are referenced by pointers, hence the "*".

Within the interface, the solver_name is used to get the pointer to function which was registered under this name. Once the function is known, it called with all the arguments passed to the interface.

To allow calls from Fortran, the interface in C needs to be "wrapped". (This wrapping is different from the one necessary to make to actual solver accessible by the elliptic registry).

```
/* Fortran wrapper for the routine Ell_SimpleEllSolver */
void CCTK_FCALL CCTK_FNAME(Ell_SimpelEllSolver)
     (cGH *GH, int *FieldIndex, int *MIndex,
     int *AbsTol, int *RelTol, ONE_FORTSTRING_ARG) {
  ONE_FORTSTRING_CREATE(solver_name);

  /* Call the interface */
  Ell_SimpleEllSolver(GH, FieldIndex, MIndex, AbsTol, RelTol, solver_name);
  free(solver_name);
}
```

- **Elliptic solver**:`./src/FastSOR_solver.F`
  Here we show the first lines of the Fortran code for the solver:

```
      subroutine FastSOR_solver(_CCTK_ARGUMENTS,
     .   Mlinear_lsh,Mlinear,
     .   var,
     .   abstol,reltol)

      implicit none

      _DECLARE_CCTK_ARGUMENTS
      DECLARE_CCTK_PARAMETERS
      INTEGER CCTK_Equals

      INTEGER Mlinear_lsh(3)
      CCTK_REAL Mlinear(Mlinear_lsh(1),Mlinear_lsh(2),Mlinear_lsh(3))
      CCTK_REAL var(cctk_lsh(1),cctk_lsh(2),cctk_lsh(3))

      INTEGER Mlinear_storage

c     We have no storage for M if they are of size one in each direction
      if ((Mlinear_lsh(1).eq.1) .and.
     .   (Mlinear_lsh(2).eq.1)  .and.
     .   (Mlinear_lsh(3).eq.1)) then
         Mlinear_storage=0
      else
         Mlinear_storage=1
      endif
```

This Fortran solver receives the following arguments: the "typical" CCTK_ARGUMENTS: `_CCTK_ARGUMENTS`, the *size* of the coefficient matrix: `Mlinear_lsh`, the coefficient matrix `Mlinear`, the variable to solve for: `var`, and the two arrays with convergence information.

In the declaration section, we declare: the cctk arguments, the Mlinear size array, the coefficient matrix, by the 3-dim. size array, the variable to solve for. Why do we pass the size of Mlinear explicitly and do not use the `cctk_lsh` (processor local shape of a grid function) as we did for `var` ? The reason is the following: while we can expect the storage of `var` to be *on* for the solve, there is no reason (in a more general elliptic case) to assume, that the coefficient matrix has storage allocated, perhaps it is not needed at all! In this case, we have to protect ourself against referencing empty arrays. For this reason, we also employ the flag `Mlinear_storage`.

- **Elliptic solver wrapper**:`./src/FastSOR_wrapper.c`
  The Fortran code can not be used within the elliptic registry directly. Instead the Fortran code is called through a wrapper:

```
void FastSOR_wrapper(cGH *GH, int *FieldIndex, int *MIndex,
     int *AbsTol,int *RelTol) {

  CCTK_REAL *Mlinear=NULL, *var=NULL;
  int Mlinear_lsh[3];
  int i;

  var = (CCTK_REAL*) CCTK_VarDataPtrI(GH,0,*FieldIndex);

  if (*MIndex>0) Mlinear   = (CCTK_REAL*) CCTK_VarDataPtrI(GH,0,*MIndex);


  if (GH->cctk_dim>3)
```

```
    CCTK_WARN(0,''This elliptic solver implementation does not do dimension>3!'');

  for (i=0;i<GH->cctk_dim;i++) {
    if((*MIndex<0))  Mlinear_lsh[i]=1;
    else             Mlinear_lsh[i]=GH->cctk_lsh[i];
  }

  /* call the fortran routine */
  CCTK_FNAME(SimpleEll_Solver)(_PASS_CCTK_C2F(GH),
         Mlinear_lsh, Mlinear, var,
 AbsTol, RelTol);
}
```

The wrapper `FastSOR_wrapper` takes these arguments: the indices of the field to solve for (`FieldIndex`) and the coefficient matrix (`MIndex`), the two arrays containing convergence information (`AbsTol, RelTol`). In the body of the program we provide two CCTK_REAL pointers to the data section of the field to solver (`var, Mlinear`) by means of `Get_VarDataPtrI`. For Mlinear, we only do this, if the index is non-negative. A negative index is a signal by the user that the coefficient matrix has no storage allocated.(For more general elliptic equation cases, e.g. no source terms.) To make this information of a possibly empty matrix available to Fortran, we load a 3-dim. and pass this array through to Fortran. See discussion above.

- **Elliptic solver startup**: `./src/Startup.c`
  The routine below in `Startup.c` performs the registration of our solver wrapper `FastSOR_wrapper` under the name "*fastsor*" for the elliptic class "*Ell_SimpleEll*". We do not register with the solver interface `Ell_SimpleEllSolver` directly, but with the class. In `Startup,c` we have:

```
#include ‘‘cctk.h’’
#include ‘‘cctk_Parameters.h’’

void FastSOR_register(cGH *GH) {

  /* protoype of the solver wrapper: */
  void FastSOR_wrapper(cGH *GH, int *FieldIndex, int *MIndex,
                       int *AbsTol, int*RelTol);

  Ell_RegisterSolver(FastSOR_wrapper,''fastsor'',''Ell_SimpleEll'');
}
```

  Note that more solver registration code could be put here (registration for other classes, etc.)

- **Elliptic solver scheduling**: `schedule.ccl` We schedule the registration of the fast SOR solver at CCTK_BASE, by this time, *the elliptic class* `Ell_SimpleEll` has already been registered.

```
schedule FastSOR_register at CCTK_INITIAL
{
  LANG:C
} ‘‘Register the fast sor solver’’
```

6 Norms