Introduction to Numerical Relativity III

Erik Schnetter, Pohang, July 2007

Lectures Overview

I. The Einstein Equations
(Formulations and Gauge Conditions)

II. Analysis Methods
(Horizons and Gravitational Waves)

III. Numerical methods
(Cactus and Mesh Refinement)

Numerical Methods: Cactus & Mesh Refinement

1. Mesh Refinement
2. Cactus, a software framework
3. Kranc, a code generator
4. CCATIE, a free BSSN code

Please interrupt and ask questions at any time

Part I:
Mesh Refinement

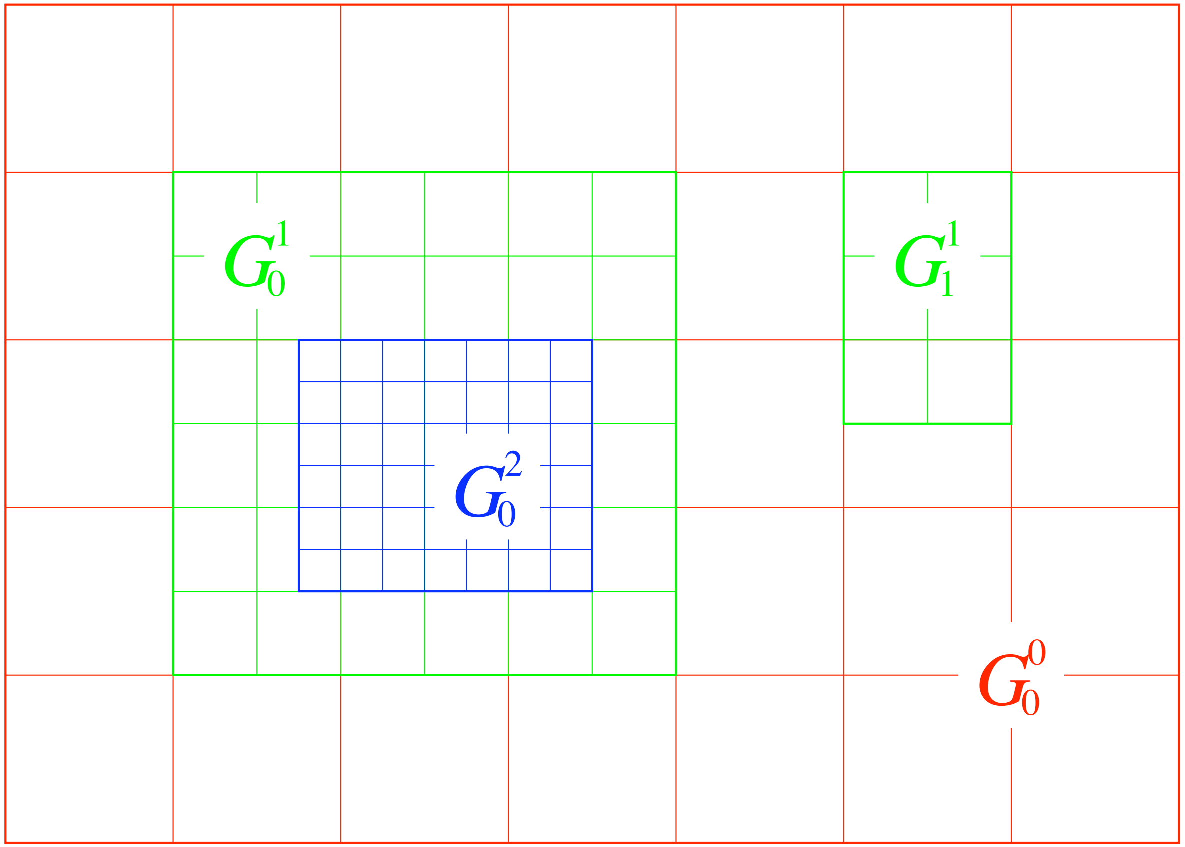
Mesh Refinement

- Central idea: use different resolutions in different parts of the simulation domain
- E.g. in BBH simulations, need to cover about 3 orders of magnitude in resolution
- Carpet: [Schnetter, Hawley, Hawke: Class. Quantum Grav. **21**, 1465 (2004)]
- http://www.carpetcode.org/

Typical grid structure

Coarse,
medium,
and fine grids

Grids are
aligned

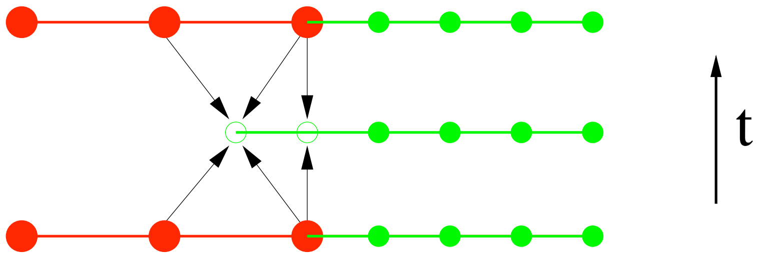


Berger-Oliger mesh refinement

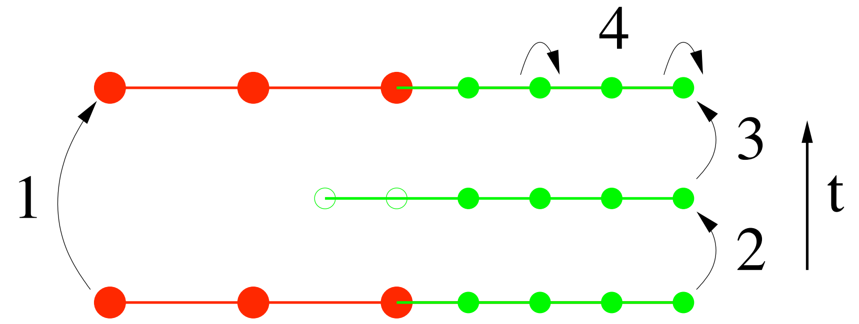
- Fine grids *overlap* coarse grids (coarse grids don't have holes)
- During evolution, fine grid boundary condition is interpolated from coarser grids
- Finer grids need to take smaller time steps (*subcycling in time*; but cf. *global time stepping*)
- Each grid is cuboid (rectangular), which makes it efficient

Time stepping

Prolongation:
fine grid boundary condition

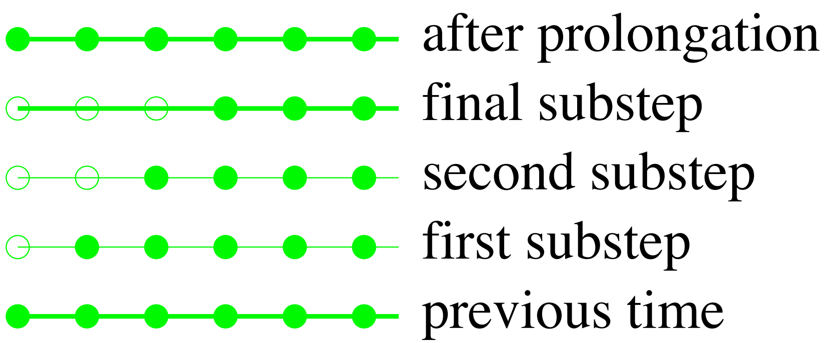


Time evolution sequence
(including restriction)



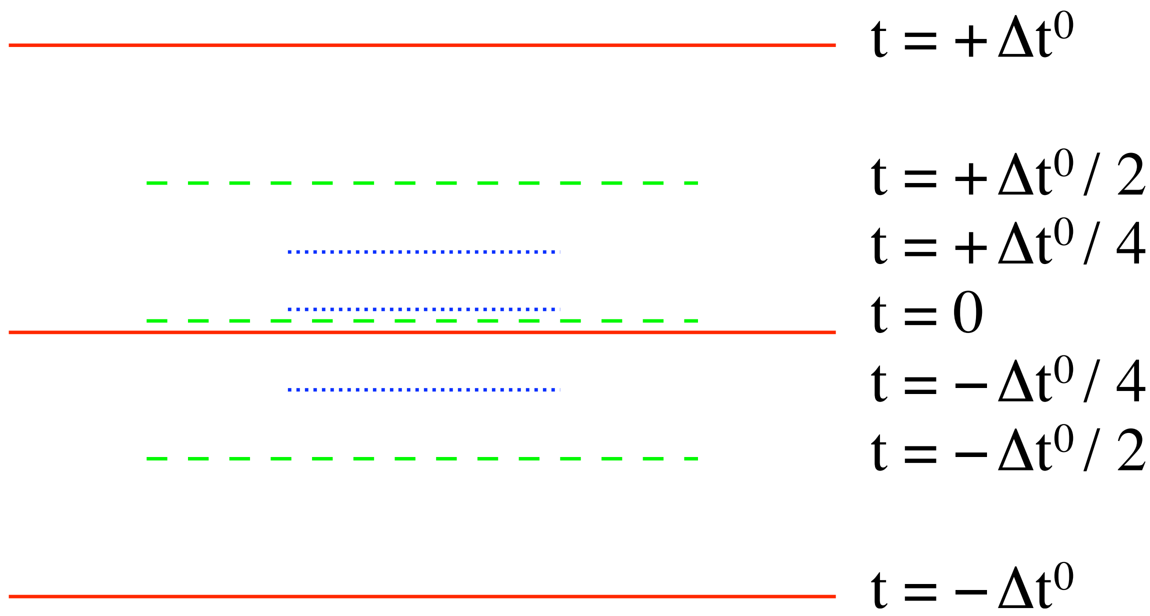
Note: time interpolation requires multiple time levels

Buffer zones



It is unstable to prolongate during the substeps of a time integrator if there are second spatial derivatives. Instead, use *buffer zones*.

Initial data generation



Need to set up initial data on multiple time levels.
Method: evolve both forwards and backwards in time, generating an intermediate *hourglass* structure.

Typical usage

- Refined regions typically track black holes or neutron stars
- Typical parameters:
  - 10 levels
  - 3 ghost zones, 9 buffer zones
  - 5th order spatial, 2nd order temporal interpolation

“Typical” simulation

- 64 processors,
  64 GByte memory
- Outer boundary at 500M,
  finest resolution 0.02M
- Total run time: several days/one week
  (using checkpointing and restart)

Please interrupt and ask questions at any time

Part II: Cactus, a software framework

Code Ingredients

- Evolution system, gauges (BSSN)
- Constraints, analysis quantities
- Horizon finder
- Wave extraction
- Initial data
- Test cases

Code Ingredients II

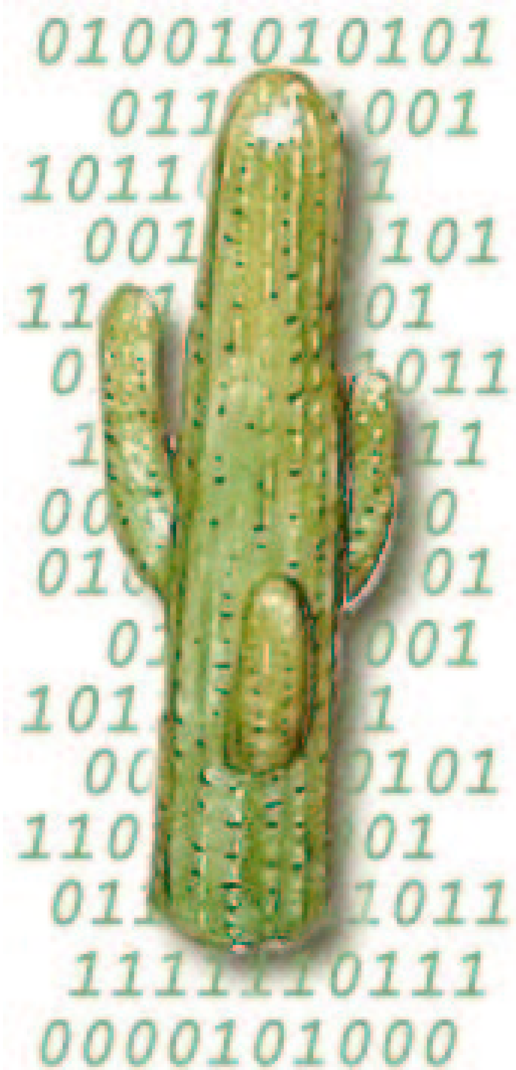
- Finite differencing
- Time integration
- Mesh refinement
- Parallelisation
- I/O (fast, platform-independent)
- “Something to make all this work together”

Components and frameworks

- Large software is typically split into *components*, which are large, independent pieces
- A *framework* puts the components together
- Cactus is a framework
- A framework doesn't do anything "useful" by itself -- it's like a bookshelf

Cactus

- <u>http://www.cactuscode.org/</u>
- In Cactus speak, the framework is called *flesh*, and the components *thorns*
- There are many public thorns which use Cactus, especially for numerical relativity



Saguaro

(Carnegiea gigantea)

Einstein Toolkit

- A common infrastructure for all relativity codes
- Defines common variables, common schedule events, etc.
- Comes with public thorns for basic tasks (simple initial data, simple analysis methods)
- There are least five production level relativity codes based on Cactus, all but one private, all using the Einstein Toolkit
- Three-level structure:

Physics code

Einstein Toolkit

Computational Toolkit

Anatomy of a Thorn

- A thorn in Cactus contains:
  - Cactus declarations (CCL language)
  - source code (C, C++, Fortran)
  - makefile fragments
  - documentation
  - test cases
  - example parameter files
- Thorns are the basic modular units
- Usually, each thorn is in a separate CVS repository

interface.ccl

- Declares *thorn name* and *implementation name*
- Declares *grid functions*
- Can *inherit* public grid functions from other implementations
- Declares *routines* (APIs provided/used by the thorn)

```
IMPLEMENTS: ADMConstraints
INHERITS: ADMBase

CCTK_REAL Hamiltonian TYPE=gf
{
  ham
} "Hamiltonian Constraint"

CCTK_REAL Momentum TYPE=gf
{
  momx momy momz
} "Momentum Constraint"
```

schedule.ccl

- Calls routines at certain times, e.g. *initial* or *evol* or *analysis*
- *Schedule groups* introduce a hierarchical structure
- Rule-based: schedule *AFTER*, *BEFORE*, *WHILE*
- Allocates storage for grid variables
- Synchronises variables

```
SCHEDULE ADMConstraints_Calculate AT analysis
{
  LANG: Fortran
  STORAGE: Hamiltonian Momentum
  SYNC: Hamiltonian Momentum
  TRIGGERS: Hamiltonian Momentum
} "Calculate the constraints"
```

param.ccl

- Declares parameters
- Five types: integer, real, boolean, keyword, string
- Allowed ranges need to be declared
- Can “inherit” public parameters from other implementations, possibly extending ranges

```
SHARES: ADMBase

EXTENTS KEYWORD initial_data
{
  "gaussian" :: "Gaussian pulse"
}

PRIVATE:

CCTK_REAL gaussian_amplitude \
  "Amplitude"
{
  0.0:* :: "must be nonnegative"
} 1.0
```

Example Source Code

```
#include “cctk.h”
#include “cctk_Arguments.h”

subroutine ADMConstraints_calculate (CCTK_ARGUMENTS)
  implicit none
  DECLARE_CCTK_ARGUMENTS

  CCTK_REAL :: dx, dy, dz
  integer   :: i, j, k

  dx = CCTK_DELTA_SPACE(1)
  ...

  do i = 2, cctk_lsh(1)-1
    ...
    ham(i,j,k) = (gxx(i+1,j,k) - gxx(i-1,j,k)) / (2*dx)
    ...
```

Parameter Files

- At run time, parameter files activate thorns and specify parameter values
- Not all compiled thorns need to be active

```
ActiveThorns = "PUGH CartGrid3D ADMBase IDSimple ADMConstraints"

driver::global_nx = 101
...
grid::xmin =  0.0
grid::xmax = 30.0
...
grid::type = "octant"

ADMBase::initial_data = "Minkowski"
```

Driver

- A *driver* is a special thorn that handles memory management and parallelisation
- Two drivers exist: *PUGH* (uniform grid) and *Carpet* (AMR, multi-block)
- Two more AMR drivers in development, based on *SAMRAI* and *Paramesh*
- Interpolation, reduction, and hyperslabbing operations closely tied to driver
- I/O (efficient and parallel) and checkpointing/ recovery also somewhat driver specific

Metadata and Data Preservation

- Thorn *Formaline* collects meta-data about a simulation (and sends them to a server)
- Collects machine name, user name, parameters, current simulation time, special events, etc.
- Allows real-time overview about currently running simulations by all people on all machines
- Some simulation results are later semi-automatically staged to be permanently stored in an archive

Please interrupt and ask questions at any time

Part III:
Kranc,
a code generator

Coding equations

- Coding equations is very tedious; the BSSN equations contain thousands of terms

Example:

$$
\begin{aligned}
\partial_0 \gamma_{ij} &= -2\alpha K_{ij} \\
\partial_t \gamma_{ij} &= -2\alpha K_{ij} + \gamma_{kj}\partial_i\beta^k + \gamma_{ik}\partial_j\beta^k + \beta^k\partial_k\gamma_{ij} \\
\forall_{ij} : \partial_t \gamma_{ij} &= -2\alpha K_{ij} + \sum_k \gamma_{kj}\partial_i\beta^k + \sum_k \gamma_{ik}\partial_j\beta^k + \sum_k \beta^k\partial_k\gamma_{ij}
\end{aligned}
$$

60 terms

Hand-coding

- Hand-coding takes a long time
- It is easy to make errors
- It is difficult to change the equations later
- It is also difficult to optimise the code
- Main problem: we think about the equations on a high level, but need to code on a much lower level

Automatic coding

- Therefore, these days most people use Maple or Mathematica to generate code
- E.g., Mathematica generates code fragments...
- ...and a wrapper is added by hand
- The wrapper e.g. declares variables, loads from/stores into arrays, calculates finite differences, etc.

Kranc

- Kranc is a Mathematica package which generates complete Cactus thorns from equations
- [Husa, Hinder, Lechner, Comput. Phys. Comm. **174**, 983 (2006); Lechner, Alic, Husa, arXiv:cs.SC/0411063 (2004)]
- http://numrel.aei.mpg.de/Research/Kranc/

Advantages of Kranc

- No need to write wrappers, since whole routines including declarations are generated
- Both discretisation (finite differences) and equations are generated
- Very easy to change equations

Disadvantages of Kranc

- Kranc adds another ingredient: not just Fortran/C code and executable, but also Mathematica script
- Need Mathematica knowledge to understand errors in Kranc script
- Less flexible than hand-coding (e.g. cannot generate HRSC formulation)

Demo

1. Look at hand-written code [CactusWave]
2. Look a Mathematica script which uses Kranc [SW.m]
3. Look at Kranc-generated code [KrancSW]

Please interrupt and ask questions at any time

Part IV:
CCATIE,
a free BSSN code

Demo

1. Look at BBH parameter file

Please interrupt and ask questions at any time