The Cactus Framework

Tom Goodale
LSU Center for Computation and Technology
and Cardiff University
goodale@cct.lsu.edu
Introduction
What is Cactus?

CACTUS is a generic, freely available, modular, portable and manageable environment for collaboratively developing parallel, high-performance multi-dimensional simulations.

Modularity: "Plug-and-play" Executables

Computational Thorns
- PUGH
- Carpet
- CartGrid3D
- Time
- ElisOR
- IOFlexIO
- IOHDF5
- IOUtil
- HTTPD

Numerical Relativity Thorns
- ADMConstraint
- ALPGEN
- BANANA
- ADM_BSSN
- SimpleExcision
- FishEye
- IOA
- IDAnalysis
- IDLinear
- IDRIPET

ISCO with AMP ??
Faster elliptic solver ??
Excision
Cactus in a Nutshell

- Cactus acts as the “main” routine of your code, it takes care of e.g. parallelism, IO, checkpointing, parameter file parsing for you (if you want), and provides different computational infrastructure such as reduction operators, interpolators, coordinates, elliptic solvers, …
- Everything Cactus “does” is contained in thorns (modules), which you need to compile-in. If you need to use interpolation, you need to find and add a thorn which does interpolation.
- It is very extensible, you can add your own interpolators, IO methods etc.
- Not all the computational infrastructure you need is necessarily there, but hopefully all of the APIs etc are there to allow you to add anything which is missing, and the community is always adding
- We’re trying to provide an easy-to-use environment for collaborative, high-performance computing, from easy compilation on any machine, to easy visualization of your output data.
Why Use a Framework

- **Portability**
  - Framework authors generally deal with the nasty parts of porting to a new architecture. E.g. Linking Fortran and C

- **Modularity**
  - Enforces clear interfaces between different modules, e.g. Components solving different physical systems

- **Access to components from a wider community**
  - Much commonality across a range of different fields, E.g. Coordinate systems, I/O, Visualisation, ...

- **Community building and standardisation**
  - Picking a framework makes people think about interfaces, data structures, who is doing what, how it all fits together, etc

- **Future proofing**
  - Forces discipline and documentation
  - Easy to add new functionality
  - Change some functionality without breaking everything.
Why Not Use a Framework?

- Small code
- Learning curve
  - Using any framework involves learning a lot of things
- Large base of existing code
  - Every framework has its own conventions, and there will need to be some changes to your code to fit it into the framework
  - Thus can be a lot of work to port into framework
- Performance loss
  - This is normally negligible, especially for codes with lots of operations, but if you want the last 1% of speed from a machine, hand-coding will be better
- Loss of control
  - You now rely on an external team to maintain a key part of your code base
History

- Cactus originated in the astrophysics community as a framework for numerical relativity (black hole/neutron star/gravitational wave) simulations.

- Built from the codes, experiences, requirements and expertise of a wide range of researchers from both physics and computer science.

- Early versions were specific for numerical relativity, but it was realized that a generic framework used by several application domains would better serve the community in the longterm.

- Simultaneous computer science research and incorporation of new technologies from the very start.

www.cactuscode.org
Project Philosophy

- Open code base and community contributions crucial
- Strict quality control for base framework
- Development always driven by real users requirements
- Application driver for computer science projects
- Leverage other projects where possible
- Support and develop for a wide range of application domains
- Provide tools for a complete working environment
- Everything from an Ipaq to Playstation to the TeraGrid
- Adherence to standards and a flexible make system means that the main code base is extremely portable
- Have yet to find a platform where we cannot run

<table>
<thead>
<tr>
<th>Machines</th>
<th>Operating Systems</th>
<th>Processors</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC</td>
<td>Linux</td>
<td>IA32, IA64</td>
</tr>
<tr>
<td>PC</td>
<td>Windows 2000/NT/XP (Cygwin)</td>
<td>IA32</td>
</tr>
<tr>
<td>PC</td>
<td>OpenBSD/FreeBSD</td>
<td>IA32</td>
</tr>
<tr>
<td>Fujitsu VP</td>
<td>OSF/Linux</td>
<td>Alpha</td>
</tr>
<tr>
<td>HP/Compaq</td>
<td>Unicos</td>
<td>Alpha</td>
</tr>
<tr>
<td>Cray T3E</td>
<td>HP-UX</td>
<td>PA8500</td>
</tr>
<tr>
<td>HP Exemplar (V2500)</td>
<td>MacOS 10/Linux</td>
<td>PowerPC</td>
</tr>
<tr>
<td>Macintosh</td>
<td>SuperUX</td>
<td></td>
</tr>
<tr>
<td>NEC SX-5</td>
<td>Solaris</td>
<td>Sparc II, Sparc III</td>
</tr>
<tr>
<td>Sun</td>
<td>AIX</td>
<td>RS-6000</td>
</tr>
<tr>
<td>IBM SP2</td>
<td>Irix</td>
<td>R8000, R10000, R12000</td>
</tr>
<tr>
<td>SGI Origin, O2</td>
<td></td>
<td>PowerPC</td>
</tr>
<tr>
<td>Hitachi SR8000-F1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Cactus Developer Community
Developing Computational Infrastructure

The Users
- LSU Cactus Group
- Argonne National Laboratory
- Konrad-Zuse Zentrum
- U. Chicago
- Compaq/HP
- NCSA
- Clemson
- Global Grid Forum
- Wash U
- Lawrence Berkeley Laboratory
- AEI Cactus Group
- TAC
- U. Kansas
- AEI Cactus Group

Grants and Projects
- DFN TiKSL/GriKSL
- EU GridLab
- NSF KDI ASC
- NSF GrADS

Many Benefits
- Visualization
- Parallel I/O
- Remote Computing
- Portal
- Optimization
- Experts
Design
Abstract Design Goals

- Generalization
  - Meta-code that can be applied to any numerical system

- Abstraction
  - Identify key concepts that can be abstracted

- Encapsulation
  - Modularize, protect thorn developers from each other

- Extension
  - Prepare for future concepts and technologies
Top Design Requirements

■ Highly portable
■ Support code development in Fortran/C/C++
■ Enable collaborations to work successfully together
■ Provide complete environment for code development (debugging, performance, visualization)
■ Robust interoperability of modules, but still allow for fast prototyping
■ Able to incorporate other packages (drivers, elliptic solvers)
■ Good support, flexibility and functionality for parallelism and IO
■ As easy to use as possible
Flesh and Thorns

Plug-In “Thorns” (modules)
- driver
- input/output
- interpolation
- SOR solver
- wave evolvers
- multigrid
- coordinates

Core “Flesh”
- extensible APIs
- ANSI C
- parameters
- scheduling
- error handling
- make system
- grid variables
- utilities

- remote steering
- Fortran/C/C++
- equations of state
- Your Physics !!
- Your Computational Tools !!
- black holes
Thorn Architecture

Thorn

Parameter Files and Testsuites

Configuration Files

Fortran Routines

C Routines

C++ Routines

Source Code

Documentation!

Make Information

www.cactuscode.org
Thorn Collections

- For organizational convenience, thorns are grouped into *arrangements*
  - may have related functionality (e.g. IO or Maxwell solvers)
  - may have the same author
  - may contain everything needed for one problem

- We call a collection of arrangements, a **toolkit** e.g.
  - Cactus Computational Toolkit
  - Cactus Relativity Toolkit

---

Cactus Computational Toolkit

- CactusBase
- CactusPUGH
- CactusElliptic
- CactusPUGHIO

+ ...
Application View

Application Toolkit

Computational Toolkit

CCTK(...) Flesh CST
Flesh-Thorn Interface
Configuration files

- Each thorn provides 3 configuration files, detailing its interface with the Flesh and with other thorns
- CCL: Cactus Configuration Language
- `interface.ccl`
  - implementation, this thorn’s variables and variables used from other thorns, functions provided or used from other thorns
- `param.ccl`
  - this thorn’s parameters, parameters used and extended from other thorns
- `schedule.ccl`
  - when and how this thorn’s routines should be executed, optionally with respect to routines from other thorns
Scheduling

- Defined in schedule.ccl
- The Cactus Flesh contains a flexible rule based scheduler which controls the program flow.
- The scheduler calls routines from thorns, the order in which the routines are called is prescribed in a thorn configuration file.
- Scheduler also takes care of telling the Driver to assign memory for or to synchronize variables.
- (The main calling routines in the Flesh are also overloadable, providing a flexible and configurable mechanism for customising program flow)
Thorns typically register their routines to be run in one of the standard time bins.

- Can define own time bins.
- Many additional features: while loops, schedule as etc.
- Scheduling dependent on parameters … want to develop more flexible script based system.
Thorns specify which functions are to be called at which time, and in which order.

Rule based scheduling system

Routines are either **before** or **after** other routines (or don't care).

Routines can be grouped, and whole group scheduled.

Functions or groups can be scheduled **while** some condition is true.

Flesh sorts all rules and flags an error for inconsistent schedule requests.
Simulation Initialisation

CCTK_ParamCheck

[Invalid Parameter Combinations] Terminate Simulation

[Parameter Combinations Valid]

CCTK_BaseGrid

[Not Recovering From Checkpoint] CCTK_Init

[Recovering From Checkpoint] CCTK_RecoverVariables

CCTK_PostRecoverVariables

CCTK_CPInitial

CCTK_Analysis

Output Data

Continue Simulation
Evolution

Check Termination Condition

[Simulation Over]

[Continue Simulation]

Update Time and Rotate Timelevels

CCTK_PreStep

CCTK_Evol

CCTK_PostStep

CCTK_Checkpoint

CCTK_Analysis

Output Data
Variables

- Defined in interface.ccl
- Essentially these are just variables which your thorn tells the Cactus infrastructure about, to get parallelisation, IO, interpolation, communication, checkpointing etc.
- Public, restricted or private, depending on who should see them.
- Can be arbitrarily sized grid arrays, or grid functions fixed to the size of the computational domain, or grid scalars.
- Many other available features: any dimension, distribution type etc.
Parameters

- Defined in param.ccl
- Range checking and validation
  - combats old problem of setting parameters to values which didn’t exist:
    - `evolution_method = "super stable fast 10th order shock capturing"
  - thorn writers must now specify ranges and descriptions for all parameters
  - checked at run time
- Steerable
  - new implementation of steerable/changeable parameters for remote steering
  - must define steerable … only if it makes sense to do so
Compilation

- Autoconf based configuration process for different architectures, compilers and libraries
- CST (Cactus Specification Tool): Perl based parsing of files written in Cactus Configuration Language (CCL) generates thorn wrappers, argument lists (Fortran!), parameters, variables, scheduling, etc.
- Cactus creates different configurations from the same source code
  - different architectures on common file system
  - use different compiler options, debug-modes, communication protocols
  - different thorn lists
Testing

- Test Suite Checking Technology
  - Thorns can contain parameter files and their output
  - Running Cactus “test suite” program re-executes these parameter files checking against the original output.
  - RemoteTest
- Advantages
  - Checks development work on your own thorns
  - Protects against changes in other thorns
  - Compares e.g multiprocessor runs against single processor runs, or the use of new thorns
  - Compares different architectures

- Warning Levels
  - CCTK_Warn
  - Warning level can be chosen at run time
Flesh APIs
Flesh API

- Abstract Flesh API for
  - Driver functions (storage, communication)
  - Interpolation
  - Reduction
  - IO, Checkpointing

- In general, thorns “overload” or “register” their capabilities with the Flesh, agreeing to provide a function with the correct interface

- e.g. CCTK_Interp(…,”second order interpolator”)
- e.g. CCTK_OutputVar(“variable”,“IOASCII”)
- e.g. CCTK_CoordRange(lower,upper,”x”,“cart3d”)
Drivers/Parallelism

- **Driver thorns:**
  - Decompose computational grid/tasks across processors
  - Set up grid variables (grid functions, grid arrays, grid scalars)
  - Handle storage
  - Provide communication/synchronization of grid variables

- **Associated thorns provide hyper slabbing, interpolation, reduction … anything requiring communication across processors**

- **Application thorns rarely need to talk to the driver:**
  - CCTK_MyProc
  - CCTK_nProcs
  - CCTK_Barrier
IO Thorns

- Register the IO method they can provide with the flesh
- Provide functions for the flesh to call for IO
- Decide themselves how and where to output
- If applicable they also provide functions for checkpointing

Most users perform output simply by setting parameters in a parameter file for a run

Can also trigger output directly from application code (useful for debugging)

- CCTK_OutputVar
- CCTK_OutputVarByMethod
- And others
Other Infrastructure

- Similar procedure for thorns providing interpolation and reduction operations
- CCTK_Reduce, CCTK_Interp

- Other infrastructure (Coordinates, Elliptic Solvers, Boundary Conditions) is provided by thorns.
- Typically as consensus builds up a master thorn provides basic infrastructure
  - IOUtil, ADMBase, EllBase, CoordBase
The Thorns
# Cactus Computational Toolkit

## CactusBase
- Boundary, IOUtil, IOBasic, CartGrid3D, LocalInterp, IOASCII, Time

## CactusBench
- BenchADM, BenchIO

## CactusConnect
- HTTPD, HTTPDExtra, Socket

## CactusExamples
- HelloWorld, FleshInfo, DemoInterp, TimerInfo, SampleIO, WaveToy1DF77, WaveToy2DF77

## CactusElliptic
- EllBase, EllPETSc, EllSOR

## CactusPUGH
- PUGH, PUGHInterp, PUGHReduce, PUGHSlab

## CactusPUGHIO
- IOFlexIO, IOHDF5, IOHDF5Util, IOPanda, IOStreamedHDF5, IsoSurfacer

## CactusTest
- TestArrays, TestCoordinates, TestInclude1, TestInclude2, TestComplex, TestInterp, TestReduce

## CactusWave

## CactusExternal
- FlexIO, jpeg6b

---

[www.cactuscode.org](http://www.cactuscode.org)
Application specific thorns: numerical relativity, chemical engineering, astrophysics, MHD.

Computational Infrastructure:
- Numerical Techniques: Elliptic solvers, Method of Lines
- Drivers: Adaptive Mesh Refinement, Fixed Mesh Refinement
- “Grid” tools: file transfer, notification, SOAP, announce, email
Standard Driver: PUGH

- MPI based for finite difference grids
- Any dimension space (1D, 2D, 3D, …)
- Automatically determines an optimal topology and load balancing
- Topology and load can also be given by hand
Numerical Methods

Different drivers needed for different methods:

- Finite differencing (PUGH)
- Unstructured grids (planned)
- Multi-model (planned)
- Adaptive mesh refinement (PAGH [dev])
- Fixed/Adapted mesh refinement (Carpet)
- Multiblock (Carpet) (Others planned)
- Particle methods (planned)
- Spectral methods
- Monte Carlo
IO and Visualization
**IO Methods**

Most work in 1D/2D or 3D
- Scalars/Reductions to screen
- Scalars/Reductions
- ASCII
- FlexIO
- HDF5
- Streaming HDF5
- Panda parallel IO
- JPEGs
- Isosurfaces
- Geodesics

Very configurable:
- One file per processor
- One file per n-processors
- Chunked/Unchunked
- Single/Double precision
- Hyper-slabbing
- Customise for each variable
- Optimised for parallel IO

Checkpointing and restart supported with FlexIO and HDF5
Visualization

- OpenDX
- LCAVision
- Amira
- IsoView
- xgraph
- gnuplot

www.cactuscode.org/VizTools
Hello World Example
Hello World Thorn

Standard example, print the words Hello World to the screen.
Standalone version in C:

```c
#include <stdio.h>

int main(void)
{
    printf("Hello World!");
    return 0;
}
```

CactusExamples/HelloWorld

**HelloWorld**

- CCL files
  - interface.ccl
  - param.ccl
  - schedule.ccl
- Source files
- Make file
- Example parameter file
- Documentation (latex)
Configuration Files

interface.ccl

```c
implements: helloworld
```

Just printing “Hello World” to the screen … no variables etc, but need to give what the thorn does a name ….

schedule.ccl

```c
schedule HelloWorld at CCTK_EVOL {
   LANG: C
} "Print message to screen"
```

Run the routine called HelloWorld (written in C) during the evolution time bin.
#include "cctk.h"
#include "cctk_Arguments.h"

void HelloWorld(CCTK_ARGUMENTS)
{
    DECLARE_CCTK_ARGUMENTS
    CCTK_INFO("Hello World !");
    return;
}

# Source files in this directory
SRCS = HelloWorld.c

- CCTK_INFO is used to print to standard output
- (allows us to control output on multiple processors, write to file, customize etc.)

- Need to tell the Cactus make system which files to compile
Running it ...

Compile a Cactus executable which contains the thorn HelloWorld

Parameter file:

```
ActiveThorns = "HelloWorld"
Cactus::cctk_itlast = 10
```

Run:
```
<cactus exe> <parameter file>
```
Scalar Wave Example
Scalar waves in 3D are solutions of the hyperbolic wave equation:

\[-\phi_{tt} + \phi_{xx} + \phi_{yy} + \phi_{zz} = 0\]

**Initial value problem**: given data for \(\phi\) and its first time derivative at initial time, the wave equation says how it evolves with time.
Numerical Method

Numerical solve by discretising on a grid, using explicit finite differencing (centred, second order)

\[
\phi^{n+1}_{i,j,k} = 2\phi^{n}_{i,j,k} - \phi^{n-1}_{i,j,k} + \frac{\delta t^2}{\delta x^2}(\phi^{n}_{i+1,j,k} - 2\phi^{n}_{i,j,k} + \phi^{n}_{i-1,j,k}) + \frac{\delta t^2}{\delta y^2}(\phi^{n}_{i,j+1,k} - 2\phi^{n}_{i,j,k} + \phi^{n}_{i,j-1,k}) + \frac{\delta t^2}{\delta z^2}(\phi^{n}_{i,j,k+1} - 2\phi^{n}_{i,j,k} + \phi^{n}_{i,j,k-1})
\]
Finite grid, so need to apply outer boundary conditions

Main parameters:
- grid spacings: $\Delta t$, $\Delta x$, $\Delta y$, $\Delta z$, which coords?, which initial data?

Simple problem, analytic solutions, but contains many features needed for modelling more complex problems
Stand Alone Code: Main.f

program WaveToy

Fortran 77 program for 3D wave equation.
Explicit finite difference method.

Global variables in include file
include "WaveToy.h"
integer i,j,k

SET UP PARAMETERS
nx = 30
[MORE PARAMETERS]

SET UP COORDINATE SYSTEM AND GRID
x_origin = (0.5 - nx/2)*dx
y_origin = (0.5 - ny/2)*dy
z_origin = (0.5 - nz/2)*dz

do i=1,nx
    do j=1,ny
        do k=1,nz
            x(i,j,k) = dx*(i-1) + x_origin
            y(i,j,k) = dy*(j-1) + y_origin
            z(i,j,k) = dz*(k-1) + z_origin
            r(i,j,k) = sqrt(x(i,j,k)**2+y(i,j,k)**2+z(i,j,k)**2)
        end do
    end do
end do

OPEN OUTPUT FILES
open(unit=11,file="out.xl")
open(unit=12,file="out.yl")
open(unit=13,file="out.zl")

SET UP INITIAL DATA
call InitialData
call Output

EVOLVING
do iteration = 1, nt
    call Evolve
    if (mod(iteration,10).eq.0) call Output
end do
stop
end
Standalone Program

Setting up parameters
Setting up grid and coordinate system
Opening output files
Setting up initial data
Performing iteration 10
Performing iteration 20
Performing iteration 30
Performing iteration 40
Performing iteration 50
Performing iteration 60
Performing iteration 70
Performing iteration 80
Performing iteration 90
Performing iteration 100
Done
Wave Equation Program

- Developed as single processor 3D program (~1hr)
- Should you use Cactus? Not needed for such a simple program but if you need:
  - parallelism for more resolution?
  - to share development with colleagues?
  - to run on a variety of platforms?
  - 2D/3D output and checkpointing?
  - to solve an elliptic equation for initial data?
  - adaptive mesh refinement?, remote monitoring? ….
  - to be ready for future technologies (the Grid!)
- All can be provided by Cactus
CactusWave Example
Arrangement

Initial Data:
- IDScalarWaveCXX
- IDScalarWaveC
- IDScalarWave
- IDScalarWaveElliptic

Evolution:
- WaveToyF90
- WaveToyC
- WaveToyFreeF90
- WaveToyF77
- WaveBinarySource
- WaveToyCXX
Cactus Flesh Provides

- Parameter parser with types, ranges, checking

- Scheduling of routines

- Make system on many architectures

- Dynamic argument lists

- Utilities such as program checking test suites
Cactus Thorns Provide

Coordinates
  (Cartesian, various domains with symmetries)

Boundary conditions
  (Fixed, flat, radiation,...)

IO
  (screen, 0D-3D, IsoSurfaces, JPEG, streaming, configurable)

Driver
  (create grid variables, handle storage, communications)
c ==============================================================
program WaveToy

minimum
Fortran 77 program for 3D wave equation.
Explicit finite difference method.
c ==============================================================
c Global variables in include file
include "WaveToy.h"
integer i,j,k
c SET UP PARAMETERS
nx = 30
[MORE PARAMETERS]
c SET UP COORDINATE SYSTEM AND GRID
x_origin = (0.5 - nx/2)*dx
y_origin = (0.5 - ny/2)*dy
z_origin = (0.5 - nz/2)*dz
do i=1,nx
do j=1,ny
do k=1,nz
x(i,j,k) = dx*(i-1) + x_origin
y(i,j,k) = dy*(j-1) + y_origin
z(i,j,k) = dz*(k-1) + z_origin
r(i,j,k) = sqrt(x(i,j,k)**2+y(i,j,k)**2+z(i,j,k)**2)
end do
c OPEN OUTPUT FILES
open(unit=11,file="out.xl")
open(unit=12,file="out.yl")
open(unit=13,file="out.zl")
c SET UP INITIAL DATA
call InitialData
call Output
c EVOLVING
do iteration = 1, nt
call Evolve
if (mod(iteration,10).eq.0) call Output
end do
stop
c ==============================================================
Converting Your Code ...

- Decide on arrangement/thorn structure
- Write thorn configuration files
  - param.ccl, interface.ccl, schedule.ccl
- Add Cactus “stuff” to source code
- Add source code filenames to make.code.defn
- Compile and debug ...
- Write a parameter file
- Run and debug ...
- Create a test suite to be sure it keeps working
- If you want, make your thorns available to friends/colleagues/the world
Could just write one thorn containing all the routines

Better to split the code in different thorns
- IDScalarWave (Initial data)
- WaveToyF77 (Evolving routines)

Make use of Toolkit thorns for Coordinates, Boundaries, IO.

Why separate thorns?
- Easier to develop and share
- Better code planning - modularity
Thorn Configuration Files

- Need to decide:
  - What are the grid variables:
    - interface.ccl
  - What are the parameters:
    - param.ccl
  - In which order should the routines be scheduled:
    - schedule.ccl
  - Do I use grid variables, parameters or scheduling from other thorns?

- These configuration files are parsed (Perl) during compilation, generating code for argument lists, parameters, program flow, etc.
Parameters: set in a parameter file, written by the guy running the code, read and verified at run time.

Thorn writer defines which parameters can be set and the values they can take.

Example: How many grid points? What initial data? What to output?

Specify:
- **data type** (real, integer, keyword, boolean, string)
- **range** (nx>0, initial_data is “gaussian wave” or “plane wave”)
- **description**
- **visibility** to other thorns (private, restricted, global)
- **default value**
- **parameters used from other thorns**
- **steerable?**
# Parameter definitions for thorn WaveToyF77

```c++
private:

KEYWORD bound "Type of boundary condition to use"
{
    "none" :: "No boundary condition"
    "flat" :: "Flat boundary condition"
    "static" :: "Static boundary condition"
    "radiation" :: "Radiation boundary condition"
    "robin" :: "Robin boundary condition"
    "zero" :: "Zero boundary condition"
} "none"
```

Only the evolver needs to know about the boundary condition.

Evolver knows about all these different boundary conditions.

Default condition is parameter isn’t set in parameter file.
# Parameter definitions for thorn IDScalarWave

```
shares: grid
USES KEYWORD type

restricted:
KEYWORD initial_data "Type of initial data"
{
    "plane" :: "Plane wave"
    "gaussian" :: "Gaussian wave"
    "box" :: "Box wave"
} "gaussian"

private:
REAL radius "The radius of the gaussian wave"
{
    0:* :: "Radius must be positive"
} 0.0
```

IDScalarWave uses the parameter type from CartGrid3D

Other thorns can use or extend initial_data

Only for IDScalarWave
Thorn configuration: interface.ccl

- Object orientated concepts:
  - Implementation, Inherits, Friends

- Grid variables
  - group name (many flesh functions act on groups)
  - group type (grid array, grid function, grid scalar)
  - variable type (real, int, complex)
  - dimension
  - description
  - visibility to other thorns (private, protected, public)
# Interface definition for WaveToyF77

```
# Interface definition for WaveToyF77

implements: wavetoy

public:

cctk_real scalarevolve type = GF
timelevels=3
{
    phi
} "The evolved scalar field"
```

- Implements: describes what this thorn “does”, WaveToyF77 can be replaced by any other thorn which “does” the same thing and has the same public interface.

- Timelevels: finite difference method is a 3 time level scheme, \( \phi_n, \phi, \phi_p \). Time levels are rotated at each iteration.

- Scope: grid variables can be public, protected or private.
# Interface definition for IDScalarWave

**implements**: idscalarwave

**inherits**: wavetoy grid

- Inherit what IDScalar wave 
takes from other thorns 
(implementations)

- Needs phi from 
WaveToyF77 (wavetoy) to 
fill in initial data

- Needs coordinate grid 
functions and parameters from CartGrid3D (grid)
CactusWave Implementations

Initial Data:
- idscalarwave
- idscalarwave
- idscalarwave
- idscalarwave
- idscalarwaveelliptic

Evolution:
- wavetoy
- wavetoy
- wavetoy
- wavetoy
- wavebinarysource
- wavetoy
CactusWave Implementations

- All the thorns implementing wavetoy look the same to other thorns, and do the same thing.
- Internally they are different (written in different languages: C, C++, F77, F90, Free Form F90).
- You can include as many implementations of wavetoy as you like when you compile Cactus (e.g. include all 5).
- When you run Cactus you choose which one you want to use in your parameter file.
Schedules when the thorn routines are run, and when storage (and communications) for variables are activated.

WaveToyF77 has one base routine, WaveToyF77_Evol:

- when should it be run?
- should it be BEFORE or AFTER any other routine?
- what language (Fortran or C) is it written in?
- is variable storage or communication needed?
- is the variable storage needed just for the routine, or for the whole run?
# Schedule definitions for thorn WaveToy77

**STORAGE:** scalarevolve

```
schedule WaveToyF77_Evolution as WaveToy_Evolution at EVOL
{
    LANG: Fortran
    SYNC: scalarevolve
}
"Evolution of 3D wave equation"
```

```
schedule WaveToyF77_Boundaries as WaveToy_Boundaries at EVOL
{
    LANG: Fortran
}
"Boundaries of 3D wave equation"
```

Storage always assigned

Synchronize group on exit

AFTER WaveToy_Evolution

Function alias
# Schedule definitions for thorn IDScalarWave

schedule IDScalarWave_CheckParameters at CCTK_PARAMCHECK
{
    LANG: Fortran
} "Check parameters"

schedule IDScalarWave_InitialData at CCTK_INITIAL
{
    STORAGE: wavetoy::scalarevolve
    LANG: Fortran
} "Initial data for 3D wave equation"

Should already be on, but make sure
WaveToy Scheduling

if (recover initial data)
    Recover parameters
endif

Startup routines
    CartGrid3D: Register GH Extension for GridSymmetry
    CartGrid3D: Register coordinates for the Cartesian grid
    PUGH: Startup routine
    IOUtil: Startup routine
    IOASCII: Startup routine
    IOBasic: Startup routine
    PUGHReduce: Startup routine
    WaveToyF77: Register banner

Parameter checking routines
    CartGrid3D: Check coordinates for CartGrid3D
    IDScalarWave: Check parameters

Initialisation
    CartGrid3D: Set up spatial 3D Cartesian coordinates
    IOASCII: Choose 1D output lines
    IOASCII: Choose 2D output planes
    PUGH: Report on PUGH set up
    Time: Initialise Time variables
    Time: Set timestep based on Courant condition
    WaveToyF77: Schedule symmetries

if (NOT (recover ID AND recovery_mode'strict'))
    IDScalarWave: ID for 3D wave equation
endif
if (recover initial data)
endif
if (checkpoint initial data)
endif
if (analysis)
endif
Do periodic output of grid variables
do loop over timesteps
    Rotate timelevels
    iteration = iteration + 1
    t = t+dt
    WaveToyF77: Evolution of 3D wave equation
    WaveToyF77: Boundaries of 3D wave equation
    if (checkpoint)
        endif
    if (analysis)
        endif
    Do periodic output of grid variables
enddo

Termination routines
    PUGH: Termination routine

Shutdown routines

www.cactuscode.org
#include "cctk.h"  dx2i = 1.0/dx2
#include "cctk_Parameters.h"  dy2i = 1.0/dy2
#include "cctk_Arguments.h"  dz2i = 1.0/dz2

subroutine WaveToyF77_Evolution(CCTK_ARGUMENTS)

implicit none  istart = 2

c Declare variables in argument list  jstart = 2
DECLARE_CCTK_ARGUMENTS  kstart = 2

INTEGER i,j,k,ierr
iend = cctk_lsh(1)-1
INTEGER istart, jstart, kstart, iend, jend, kend  jend = cctk_lsh(2)-1
CCTK_REAL dx,dy,dz,dt  kend = cctk_lsh(3)-1
CCTK_REAL dx2,dy2,dz2,dt2  factor = 2*(1 - (dt2)*(dx2i + dy2i + dz2i))
CCTK_REAL dx2i,dy2i,dz2i

istart = 2
dx2 = dx*dx
dy2 = dy*dy
dz2 = dz*dz
dt2 = dt*dt
dz2i = 1.0/dz2

c Set up shorthands  return

CCTK_REAL factor  end

c ---------------

dx = CCTK_DELTA_SPACE(1)

phi(i,j,k) = factor*phi_p(i,j,k) -

dy = CCTK_DELTA_SPACE(2)  &
& (phi_p(i+1,j,k)+phi_p(i-1,j,k))*dx2i

dz = CCTK_DELTA_SPACE(3)  &
& +(phi_p(i,j+1,k)+phi_p(i,j-1,k))*dy2i

dt = CCTK_DELTA_TIME  &
& +(phi_p(i,j,k+1)+phi_p(i,j,k-1))*dz2i

do k = kstart, kend  end do
  do j = jstart, jend
    do i = istart, iend
      phi(i,j,k) = factor*phi_p(i,j,k) -

      & (phi_p(i+1,j,k)+phi_p(i-1,j,k))*dx2i
      & +(phi_p(i,j+1,k)+phi_p(i,j-1,k))*dy2i
      & +(phi_p(i,j,k+1)+phi_p(i,j,k-1))*dz2i

      end do
    end do
  end do
end
Cactus runs from a parameter file

```bash
> exe/cactus_wave wavetoy.par
```

All thorns you need must be activated using the `ActiveThorns` parameter

Parameters must be qualified with the implementation or thorn name, depending on their type

```plaintext
ActiveThorns = "idscalarwave time wavetoyf77 pugh pughreduce cartgrid3d pughslab ioutil ioascii"

time::dtfac = 0.5
idscalarwave::initial_data = "gaussian"
idscalarwave::sigma = 2.8
idscalarwave::radius = 0
wavetoyf77::bound = "zero"
grid::type = "BySpacing"
grid::dxyz = 0.6
driver::global_nx = 30
driver::global_ny = 30
driver::global_nz = 30
cctk_itlast = 100
IOASCII::out1D_every = 10
IOASCII::out1D_vars = "wavetoy::phi"
IOASCII::outinfo_every = 10
IOASCII::outinfo_vars = "wavetoy::phi"
IO::outdir = "StandAlone"
```
Cactus for Collaborations
Collaborative Working

- Framework designed explicitly for widely distributed collaborations: for example Numerical Relativity
- Many design features for collaborative working
  - Modularity makes it easy to share thorns
  - Testsuites for checking they work
  - Multiple implementations to upgrade and compare thorns
- Development and distribution via CVS
- Scripts for finding and downloading thorns
- Interactive collaborative monitoring, steering and visualization
Collaborative Simulations

- Many tools being developed which allow whole collaborations to
  - Monitor and steer simulations from any where in the world
  - View remote data with different visualization programs
  - Receive notifications of events via Email, SMS etc.
  - Share parameter files and data
  - Work together from portals
Remote Monitoring/Steering:

- Thorn which allows simulation to any to act as its own web server
- Connect to simulation from any browser anywhere ... collaborate
- Monitor run: parameters, basic visualization, ...
- Change *steerable* parameters
- Running example at www.CactusCode.org
- Wireless remote viz, monitoring and steering
Cactus Portals

- Thin Client !!
  - Any Web Browser
- Portal (will) provides:
  - Single access to all resources
  - Locate/build executables
  - Central/collaborative parameter files, thorn lists etc.
  - Job submission/tracking
  - Access to new Grid Technologies
- Developing interfaces for mobile devices
Cactus Community

- Mailing lists:
  - users@cactuscode.org,
  - developers@cactuscode.org,
  - news@cactuscode.org

- Configuration files for many different HPC academic machines:

- Flesh and Toolkits all Open Source
  - Contributions
  - Suggestions

- Bug Tracking and Feature Requests
Where to Go Next
Cactus Web Page Highlights

- Lots of information on web pages, index at
- Slides from this tutorial and others
- Users Guides and Thorn Guides
- Quick Start guides, FAQs and HOWTOs
- Visualization tools, where to get them, how to use them
- Architecture notes, configuration options
- Benchmarks
- Download information
- Mail list archives
Test Drive ...

- Email cactusmaint@cactuscode.org for advice
- Check out and run the simple WaveToy example
- Try it out with a toy example of your own

- Weigh up the advantages & disadvantages of using Cactus, other frameworks, or writing your own code
- (Whichever, try and write your code to be easily ported into or between frameworks, look at the common features)
- Get involved in a community, lots of help, advice and friends out there!