

Kranc

Generating Cactus Thorns from Mathematica

Sascha Husa

Albert Einstein Institute, Potsdam, Germany

April 30 2004

Kranc has been written by Ian Hinder (U Southampton), Christiane Lechner (AEI) and myself, and has been released under the GPL (<http://numrel.aei.mpg.de/Research/Kranc>).

Kranc: a Mathematica application to generate numerical codes for tensorial evolution equations,
<http://arxiv.org/abs/gr-qc/0404023>

Content

- What is Kranc?
- Motivation / Objectives
- Code generation strategy
- What does Kranc consist of?
- How is a Kranc code built up?
- Performance issues
- Future developments
- Conclusions

What is Kranc?

Kranc is a suite of Mathematica-based computer-algebra packages, which comprise a toolbox to convert (tensorial) systems of partial differential evolution equations to parallelized C or Fortran code.

- “rapid prototyping” system for scientists handling very complicated systems of partial differential equations,
- but through integration into the Cactus computational toolkit we can also produce efficient parallelized production codes.

Central ideas:

Take programming to the limit and work at the highest possible level, but without sacrificing flexibility, performance and transparency down to lower levels.

Modular instead of monolithic; style resembles Unix rather than Windows.

Motivation and Objectives

Our work is motivated by the field of numerical relativity, i.e. the numerical solution of the Einstein equations.

We aim to address the problems of handling complexity and of dealing with tensorial (or other multi-component) quantities in a transparent way.

Kranc opens up some new perspectives on performance optimization and collaborative working.

Kranc's code generation strategy

Designing numerical codes to solve tensorial equations, 2 principal alternatives come to mind:

- (i) Implement high-level tensor operations in a language like C++ or Fortran 90 using tensorial operations like contractions which operate on tensor objects.

- (ii) Use computer algebra to expand all tensorial expressions into components, and then use only standard arithmetics in the numerical code.

First approach avoids mixing of technologies, second approach uses specialized technologies to deal with parts of the problem.

We adopt the second strategy – numerical code is component oriented.

Strategy motivation:

- needed a **framework that extends beyond code generation**, and can also be used as a tool for the analytical part of our work. → **use CA system.**
- Including the transition from tensors to components and the process of code generation in the CA system is **easy to implement**, and greatly **simplifies the structure of the resulting numerical code.**

Using a language like PERL for this step would have demanded knowledge of (at least) 3 different languages: Mathematica, PERL and C/Fortran.

- The component oriented design streamlines performance optimization as the code is **readable and straightforward** down to the level of standard arithmetics. The resulting **code can in principle be implemented in any suitable language**, and we provide the option of generating either Fortran 90 or C.
- The **full power of the chosen computer algebra system can be utilized in the process of code generation**, e.g. to perform optimizations which rely on the structure of the equations.

Kranc's code generation strategy II

With Kranc, a code is built from *complete Kranc-generated Cactus thorns*.

Currently, we only support evolution problems with periodic boundaries!

Evolution algorithm: Ian Hawke's AlphaThorns/MoL (method of lines) thorn.

Spatial discretization is implemented in a centralized and flexible way: All finite difference formulas are collected in a single header file, currently via C-style macros.

We denote derivatives using an abstract notation; e.g. a first derivative of the variable f in the x -direction would be denoted as $D1[f]$.

At compile time, the user has two choices to make:

- Should the derivatives be precomputed at the start of the grid loop?
(-DNOPRECOMPUTE)
- What sort of finite differencing should be performed?
(-DFD_C2, -DFD_C4, -DFD_C0)

Kranc's code generation strategy III

On the user level Kranc consists of Mathematica functions that generate an entire thorn. The basic tasks of a thorn are:

- define Cactus GFs;
- assign values to GFs;
- set GF attributes (e.g. symmetries or boundary conditions)
- define Cactus parameters;

Kranc thorns are classified into 5 types, with specific characteristics regarding their functionality are: *base*, *setter*, *translator*, *evaluator* and *MoL* thorns.

While the base thorn only defines parameters, GFs and their properties, all the other thorns also perform one or more calculations, i.e. they assign new values to one or more GFs.

Kranc's thorn types

All thorns may define their own parameters and share parameters from other thorns. In addition, Kranc thorns implement the following tasks:

- Base thorn:
 - ★ Register GFs & their symmetries, discriminate between evolved (2 timelevels) and “primitive” (1 timelevel) GFs.

- Setter thorn:
 - ★ Set GFs to values, e.g. to set initial data or to update auxiliary GFs. Functions are scheduled in the POSTINITIAL and EVOL time bins at appropriate times relative to the time stepping functions.

- MoL evolve thorn:
 - ★ Register GFs with MoL thorn. During registration, the GFs need to be designated as either “evolved” or “primitive”, with respect to this thorn.
 - ★ Generate code to apply boundary conditions.
 - ★ Set MoL RHSs through a function scheduled in the EVOL bin

- Translator thorn:
 - ★ A specialized version of a setter thorn which translates between Kranc variables and some other set of variables.
This interface is essential for compatibility with existing numerical relativity thorns using the ADM variables via the ADMBase thorn.

- Evaluator thorn:
 - ★ Define new GFs.
 - ★ Assign values to new GFs in routines scheduled in ANALYSIS.

What does Kranc consist of?

Kranc is composed of several Mathematica packages that perform distinct functions: KrancThorns, TensorTools, CodeGen, Thorn and MapLookup.

The user only needs to be concerned with calling functions from one package: *KrancThorns* – contains functions for creating the different types of Kranc thorn.

The user directly calls `Create*Thorn` functions from a Mathematica script or interactively in a command line or notebook session.

Internally, KrancThorns uses the Thorn package to create “generic” Cactus thorns.

TensorTools

Tensorial expressions are entered in MathTensor syntax. For example, T_a^b would be written as `T[1a,ub]`.

Before using a tensor, it must be registered with the TensorTools package using the `DefineTensor` function:

```
DefineTensor[T]
```

Entering a tensorial expression causes it to be rendered in an easy to read form:

```
In := T[1a,1b]
```

```
Out =  $T_{ab}$ 
```

Expansion of tensorial expressions into components

The function `MakeExplicit` converts an expression containing abstract tensors into a list of component expressions:

```
In := MakeExplicit[T[la,
lb]g[ub, uc]]
```

```
Out = { g11 T11 + g21 T12 + g31 T13, g12 T11 + g22 T12 + g32 T13,
        g13 T11 + g23 T12 + g33 T13, g11 T21 + g21 T22 + g31 T23,
        g12 T21 + g22 T22 + g32 T23, g13 T21 + g23 T22 + g33 T23,
        g11 T31 + g21 T32 + g31 T33, g12 T31 + g22 T32 + g32 T33,
        g13 T31 + g23 T32 + g33 T33}
```

CodeGen: High Level Code Generation

Code generation is based on Mathematica functions to represent each block of text.

Fundamental objects are CodeGen “blocks”: `string` or a list of CodeGen blocks.

All CodeGen functions return CodeGen blocks, all lists are flattened and all strings concatenated when the final source file is generated.

Many programming constructs are naturally block-structured; for example, C for loops need braces after the block of code to loop over. For this reason, it was decided that CodeGen functions could take as arguments any blocks of code which needed to be inserted on the inside of such a structure.

By default, Kranc generates C code. To generate Fortran code the command `SetSourceLanguage"Fortran"` has to be issued after loading `KrancThorns.m`.

Thorn: Constructing the most general Cactus thorn

The Kranc system is designed to be as modular as possible, with as much code re-use as possible. To this end, there is a package called “Thorn” which takes as input a high level description of a Cactus thorn, and which creates the necessary thorn files. This maximizes code reuse for the different types of Kranc thorn.

The Thorn package provides every aspect of thorn generation that is not specific to the Kranc thorns we have defined, and could be used for creating any type of thorn.

Example: ADM equations

$$\partial_t h_{ij} = \mathcal{L}_\beta h_{ij} + 2\alpha K_{ij}$$

$$\partial_t K_{ij} = \mathcal{L}_\beta K_{ij} + D_i D_j \alpha + 2\alpha h^{lm} K_{lj} K_{mi} - \alpha K K_{ij} - \alpha R_{ij}.$$

$$C_H \equiv R - h^{lm} h^{pq} K_{lp} K_{mq} + K^2 = 0$$

$$C_{Mi} \equiv h^{lm} D_l K_{im} - D_i K = 0.$$

254 lines of Mathematica / 9 thorns / 690 lines of CCL files / 3131 lines of C:

```
(* set path *)
```

```
$Path = Flatten[{"../..../Tools/CodeGen", "../..../Tools/MathematicaMisc",
"../..../Tools/External", ".", $Path}];
```

```
Get["TensorTools.m"];
```

```
Get["KrancThorns.m"];
```

```
(* switch off spelling warnings *)
```

```

Off[General::spell, General::spell1];

(*.....some functions needed later.....*)
(* function to create set of variables without duplicates*)
MakeExplicitSet[T_] := RemoveDuplicates[MakeExplicit[T]];

(* function to create group structure from tensor*)
CreateGroupFromTensor[T_] :=
  List[ToString[Delete[Cases[{T}, _Symbol, {1, 2}], 0]],
    MakeExplicitSet[T]
  ];

(* Register the tensor quantities with the TensorTools package *)
Map[DefineTensor, {h, hInv, K, beta, Ricci, gamma, CM, trgama}];

(* Set tensor symmetries *)
AssertSymmetricIncreasing[hInv[ua, ub]];
Map[AssertSymmetricDecreasing, {h[la, lb], K[la, lb], Ricci[la, lb]}];
AssertSymmetricDecreasing[gamma[ua, lb, lc], lb, lc];

(* define connection *)
DefineConnection[CD, gamma];

```

```

(* Determinant of the metric in terms of their components *)
hDet = Det[MatrixOfComponents[h[1a,1b]]];

(* .....GroupDefinition structures for variables and primitives.... *)

ADMvars= Map[CreateGroupFromTensor, {h[1a,1b], K[1a,1b]}];
ADMgauges = Map[CreateGroupFromTensor, {alpha, beta[ua]}];
ADMevals = Map[CreateGroupFromTensor, {CH, CM[1a]}];
ADMRicci = Map[CreateGroupFromTensor, {gamma[ua,1b,1c], Ricci[1a,1b]}];
ADMtrK = {CreateGroupFromTensor[trK]};
ADMprimitives = Flatten[{ADMgauges,ADMRicci, ADMtrK},1];

(* .....Equations.....*)

<<ADMEqs.m;

(* evolution equations *)
EvolEqs = EvolEqs/.{LieD -> Lie};
ADMMainEvolEqs = Map[MakeExplicitSet,EvolEqs];

(* constraint equations*)

```

```
CHEqs = Map[Expand,MakeExplicit[Flatten[ConstrEqs][[1]],2];
```

```
CMEqs = MakeExplicit[Flatten[ConstrEqs][[2]]];
```

```
(* equations for primitives*)
```

```
ChristoffelEqs = MakeExplicitSet[ChristoffelRules];
```

```
RicciStandardEqs = Map[Simplify,MakeExplicitSet[RicciStandardRules]];
```

```
RiccifromMetricEqs = MakeExplicitSet[RiccifromMetricRules];
```

```
trKEqs = {MakeExplicit[trKRule]};
```

```
ZeroShiftEqs = Map[MakeExplicit,{beta[ua]->0}];
```

```
(* shorthands *)
```

```
ScalarCurvShorthandEqs = MakeExplicit[ScalarCurvShorthandRule];
```

```
InvMetricEqs = Map[MakeExplicitSet,{
    deth -> hDet,
    invdeth -> 1 / deth,
    hInv[ua,ub] -> invdeth hDet MatrixInverse[h[ua,ub]]}];
```

```
(* relation between evol. variables and ADMBase variables *)
```

```
TranslatorEqs = {h11 == gxx, h21 == gxy, h31 == gxz,
                  h22 == gyy, h32 == gyz,
                  h33 == gzz,
```

```
K11 == -kxx, K21 == -kxy, K31 == -kxz,  
      K22 == -kyy, K32 == -kyz,  
      K33 == -kzz};
```

```
(* collect list*)
```

```
ADMCollectList = {alpha, hInv33, hInv23, hInv22, hInv13, hInv12, hInv11};
```

```
(* from now on we do not make use of TensorTools anymore*)
```

```
SetEnhancedTimes[False];
```

```
(*.....Create Cactus Thorns.....*)
```

```
ADMSystemDescription="ADM";
```

```
ADMSystemName = "KrancADM";
```

```
commonArgs = {SystemName -> ADMSystemName, DeBug -> False};
```

```
(* to generate Fortran code: *)
```

```
(* SetSourceLanguage["Fortran"]; *)
```

```
(*.....create Base Thorn.....*)
```

```

baseThornInfo = Timing[CreateBaseThorn[
  Join[ADMvars,ADMprimitives],
  Map[First,ADMvars],
  Map[First,ADMprimitives],
  SystemDescription -> ADMSystemDescription,
  Apply[Sequence,commonArgs]]];

(*.....create Mol Thorn.....*)

ADMMoLShorthands=Flatten[{InvMetricEqs}][[All, 1]];

MainMoLCalculation={Shorthands->ADMMoLShorthands,
CollectList -> ADMCollectList,
  Equations->{Flatten@{InvMetricEqs, ADMMainEvolEqs}}};

mainMoLThornInfo = Timing[CreateMoLThorn[
  MainMoLCalculation,
  Join[ADMvars,ADMprimitives],
  PrimitiveGroups-> Map[First,ADMprimitives],
  ThornName -> ADMSystemName<>"MoL",
  SystemDescription -> ADMSystemDescription,
  Apply[Sequence,commonArgs]]];

```

```

(*..... create Translator Thorn .....*)

GFsinternal = TranslatorEqs[[All,1]];
GFsexternal = Map[Delete[#, 0] &, Map[Abs,TranslatorEqs[[All,2]]]];

externalGroups =
  {"metric",Take[GFsexternal,6]},{ "curv",Take[GFsexternal,-6]};

InCalculation = {Equations -> Solve[TranslatorEqs, GFsinternal]};
OutCalculation = {Equations -> Solve[TranslatorEqs,GFsexternal]};

translateThornInfo = Timing[CreateTranslatorThorn[
  Join[ADMvars,externalGroups],
  TranslatorInCalculation -> InCalculation,
  TranslatorOutCalculation -> OutCalculation,
  Apply[Sequence,commonArgs]]];

(* .....create Setter Thorn for Christoffels and Ricci.....*)

(*.....Ricci from Metric .....*)
RiccifromMetricCalculation = {

```

```

CollectList -> ADMCollectList,
Shorthands -> Flatten[InvMetricEqs][[All,1]],
Equations ->{Flatten@{InvMetricEqs, ChristoffelEqs,RiccifromMetricEqs}}};

```

```

setRiccifromMetricThornInfo = Timing[
  CreateSetterThorn[RiccifromMetricCalculation,
    Join[ADMRicci,{ADMvars[[1]]}],
    ThornName -> ADMSystemName<>"setRiccifromMetric",
    SetTime -> "initial_and_poststep",
    Apply[Sequence,commonArgs]]];

```

(*.....Ricci from Christoffel*)

```

RicciStandardCalculation = {
  CollectList -> ADMCollectList,
  Shorthands -> Flatten[InvMetricEqs][[All,1]],
  Equations ->{Flatten@{InvMetricEqs, ChristoffelEqs},Flatten@{RicciStandardE

```

```

setRicciStandardThornInfo = Timing[
  CreateSetterThorn[RicciStandardCalculation,
    Join[ADMRicci,{ADMvars[[1]]}],
    ThornName -> ADMSystemName<>"setRicciStandard",
    SetTime -> "initial_and_poststep",

```

```

        Apply[Sequence,commonArgs]]];

(* .....create Setter Thorns for the gauge .....*)
ZeroShiftCalculation = {Equations -> ZeroShiftEqs};

setZeroShiftThornInfo = Timing[
    CreateSetterThorn[ZeroShiftCalculation,
        {ADMgauges[[2]]},
        ThornName -> ADMSystemName<>"setZeroShift",
        SetTime -> "initial_and_poststep",
        Apply[Sequence,commonArgs]]];

HarmonicLapseEqs = {{deth -> hDet, alpha -> Sqrt[deth]}};

HarmonicLapseCalculation = {
    Shorthands -> {deth},
    Equations -> HarmonicLapseEqs};

setHarmonicLapseThornInfo = Timing[
    CreateSetterThorn[HarmonicLapseCalculation,
        {ADMgauges[[1]],ADMvars[[1]]},
        ThornName -> ADMSystemName<>"setHarmonicLapse",

```

```

    SetTime -> "initial_and_poststep",
    Apply[Sequence,commonArgs]]];

```

```
(*.....create Setter Thorn for trK.....*)
```

```

trKCalculation = {Shorthands -> Flatten[InvMetricEqs][[All,1]],
                  Equations -> {Flatten@{InvMetricEqs,trKEqs}}};

```

```

settrKThornInfo = Timing[
  CreateSetterThorn[trKCalculation,
    Join[ADMtrK,ADMvars],
    ThornName -> ADMSystemName<>"settrK",
    SetTime -> "initial_and_poststep",
    Apply[Sequence,commonArgs]]];

```

```
(*.....create Evaluator Thorn for constraints..... *)
```

```

CHShorthands = Flatten[{InvMetricEqs,ScalarCurvShorthandEqs}][[All,1]];
CMShorthands = Flatten[InvMetricEqs][[All,1]];

```

```

HamCalculation = {
  Shorthands-> CHShorthands,

```

```

CollectList -> ADMCollectList,
Equations -> {Flatten@{InvMetricEqs,ScalarCurvShorthandEqs,CHEqs}}};

MomCalculation = {
  Shorthands->    CMShorthands,
  CollectList -> ADMCollectList,
  Equations -> {Flatten@{InvMetricEqs,CMEqs}}};

EvaluationDefinitions = {"CH", HamCalculation}, {"CM", MomCalculation}};

evaluateThornInfo = Timing[
  CreateEvaluatorThorn[EvaluationDefinitions,
    Join[ADMevals,ADMvars,ADMprimitives],
    ThornName->ADMSystemName<>"evalConstraints",
    Apply[Sequence,commonArgs]]];

(* ..... generate ThornList.....*)

thornInfo = {baseThornInfo, mainMoLThornInfo, translateThornInfo,
  setRicciStandardThornInfo,setRiccifromMetricThornInfo,
  setZeroShiftThornInfo,
setHarmonicLapseThornInfo, settrKThornInfo,evaluateThornInfo};

```

```
thorns=Flatten[thornInfo[[All,2]],1];  
CreateThornList[thorns,SystemName->ADMSystemName];
```

Performance Issues

We compare 2 codes using the same numerical methods: our ADM example code & AEI CactusEinstein/ADM code (carefully handcoded former production code of the AEI astrophysical numerical relativity group).

code version	compile time	user runtime	cactus timing
CactusEinstein ADM, Intel 8	184	145.2	145.6
CactusEinstein ADM, VAST f90 & gcc	154	155.0	155.2
Kranc ADM, C, Intel 8	119	154.9	155.4
Kranc ADM, C, VAST f90 & gcc	89	166.8	167.4
Kranc ADM, F90, Intel 8	129	159.7	159.5
Kranc ADM, F90, VAST f90 & gcc	92	168.3	168.8

Table 1: Timing results from a Dell D600 Laptop running Redhat Linux 9.0, with a 1.6 GHz Intel Pentium M CPU and 1GByte of RAM, comparing results obtained with the Intel 8 compilers vs. a combination of Pacific Sierra VAST F90 and gcc 3.2.2 compilers. Optimization options for the Intel compilers are `-O3 -xN -ip`, and `-O3` for VAST and gcc.

Future Developments

- Boundaries!
- improve performance, e.g. add option to generate to F90 array language style code
- interface to elliptic solver
- test interface to AEI excision code
- multipatch
- code up more systems
-

Conclusions

- Should I use Kranc?

Yes – if you consider using Cactus, and you have a coding problem where complexity of the mathematical structure is the dominating problem, and you can't find a more suitable tool.

- How to get started?

download code, read paper and work through examples

<http://numrel.aei.mpg.de/Research/Kranc>

- I like the idea, but Kranc does not yet support my needs . . .

hand coded code can be added as for any other collection of thorns

modyfing Kranc to extend its capability is easy – collaborators welcome!