MathGR: a tensor and GR computation package to keep it simple

Yi Wang
Kavli Institute for the Physics and Mathematics of the Universe (WPI),
Tokai Institutes for Advanced Study, University of Tokyo,
5-1-5 Kashiwanoha, Kashiwa, Chiba 277-8583, Japan

We introduce the MathGR package, written in Mathematica. The package can manipulate tensor and GR calculations with either abstract or explicit indices, simplify tensors with permutational symmetries, decompose tensors from abstract indices to partially or completely explicit indices and convert partial derivatives into total derivatives. Frequently used GR tensors and a model of FRW universe with ADM type perturbations are predefined. The package is built around the philosophy to “keep it simple”, and makes use of latest tensor technologies of Mathematica.

PROGRAM SUMMARY

Program title: MathGR
Program obtainable by: Send an email to tririverwangyi@gmail.com, with title “MathGR acquire package”.
Licensing provisions: GPLv3
Programming language: Mathematica (Minimal: 6.0. Tensor simplification with symmetry: 9.0 or later.)
Operating system: Linux, Windows and MacOS
Number of processors used: 1
Keywords: symbolic tensor, general relativity, cosmological perturbations
Classification: 1.5
Nature of problem: simplify symbolic tensors, use cases include general relativity and cosmic perturbations

I. INTRODUCTION

Tensor computations play a crucial roll in the studies of general relativity (GR). However, in typical cases those calculations are tediously complicated. Fortunately, with the development of symbolic calculation in computer technology, a number of computer packages are built to calculate the GR tensors, including the GRTensor [1], xAct [2], Ricci [3], and so on.

Those packages are well developed for years, which works great for a huge variety of applications. General relativists would most likely find whatever functionality they need in those packages and work with them.

However, in case the researchers do have a need to modify some internals of the packages, or hope to have a complete understanding of the package they use, things become complicated. Although many packages are kindly provided open source, they have in general a heavy code base. For example, the latest xAct package has over 20,000 lines of code (LOC), where the xTensor.m file alone has more than 8,000 LOC. As another example, the Ricci package has over 7,000 LOC.

There is definitely nothing wrong with heavy weight packages which provide as many functionalities as possible. But on the other hand, it would also be helpful to have lightweight packages which are easy for the users to understand the underlying mechanisms and modify them when needed. The MathGR package is built for this purpose. Currently, MathGR has less than 400 LOC, which is 20 KB of code 1, and those numbers will not grow significantly in the future.

1 This count including comments but excluding standalone documentation, examples and unit tests. The pure functional code is smaller
The package is also built modularly, separating the tensor manipulation, GR definitions, integration by parts, specific model definition, general utilities, display parser into different files.

While keeping to be lightweight, MathGR still provides competing functionalities for GR computations, with fast speed. The functionalities of MathGR include:

- Tensor simplifications with symmetries. Symmetries, asymmetries for any subset of indices, and any cyclic symmetries defined by the Mathematica Cycles can be brought into unique forms.

- Tensor calculations with either abstract or explicit (or mixed) indices, and decomposition from abstract indices to explicit ones. For example, tensors with abstract tensor indices such as the Ricci scalar (where the dummy indices denotes summation throughout this paper and assumed by the package)

  \[ R = -g^{\alpha\beta} g^{\gamma\delta} \partial_\gamma \partial_\delta g_{\alpha\beta} + \cdots \]  

  can be calculated and simplified, and later decomposed if needed, with builtin command into

  \[ R = -g^{00} g^{00} \dot{g}_{00} - g^{00} g^{0i} \partial_i \dot{g}_{00} + \cdots , \]

  where dot denotes derivative with respect to the 0 index, and \( \alpha, \cdots \) run from 0 to \( n \) and \( i, \cdots \) run from 1 to \( n \). Alternatively, one can also decompose the \( \alpha, \cdots \) indices into completely explicit indices 0,1,...,\( n \), or decompose \( i, \cdots \) indices into explicit indices 1,...,\( n \).

- Simplification with total derivatives. When calculating an action of a physical system, total derivatives can either be dropped or reduced into boundary terms. MathGR can try to reduce a given expression into total derivatives plus the rest of the terms which are minimal under various conditions.

- Cosmic perturbations. There is a builtin model calculating GR tensors for the FRW metric with ADM type perturbations. This model can on the one hand be used directly for the research of inflationary cosmology, and on the other hand provide an example for writing models.

- Make use of Mathematica’s builtin tensor engine. Since Mathematica 9.0, a symbolic tensor engine is introduced. This engine is powerful and fast in reducing tensors with symmetries into unique forms. However, the interface of the tensor engine is in the coordinate independent form. The input looks like:

  \[ \$Assumptions = \{ T \in \text{Arrays}\{\{\text{Dim}, \text{Dim}, \text{Dim}\}, \text{Complexes}, \text{Antisymmetric}\{\{1, 2\}\}\} \}; \]

  TensorReduce[TensorContract[T \otimes T, \{2, 6\}]]

  And the corresponding output is

  \[-\text{TensorTranspose}\text{TensorContract}[T \otimes T, \{2, 6\}, \{1, 2, 4, 3\}]\]

  For many physicists, the above notation is not obvious, especially in case those notations come together with an expression with hundreds of terms. MathGR makes use of the above engine, but the input and output now takes the form

  \[ \text{Simp}[T_{abc} T_{mnb}] \rightarrow -T_{bac} T_{mnb} \]

  Complicated polynomial expressions of tensors taking the above form can be simplified similarly.

  On the other hand, for people using Mathematica version lower than 9.0, MathGR provides a simple tensor simplification engine, which can still do some calculations such as cosmic perturbations, but is not powerful enough to simplify tensor with symmetries (except totally symmetric) or permutation of indices.

With the lightweight code base and modularity, we hope MathGR is helpful for researchers who has a demand to modify the internal functionality of a package, or those who want to take part of the internal technology from the package to build their own tools. Moreover, the development is (partially) test driven. Unit tests and integrated tests are provided thus the modifications by users are safer if all tests are passed.
II. MODULES OF MATHGR

Here we introduce the functions provided by each module. To make the notation less clustered, we have used the upper and lower indices in the tensor notations. For example, the InputForm of $T_{dabc}$ should be inputted as

$$T[UP["d"], DN["a"], DN["b"], DN["c"]]$$

(7)

One can load the look.m package (to be introduced later) using

```mathematica
Get["MathGR/look.m"]
```

(8)
after which the InputForm (7) is displayed as $T_{dabc}$. One may either copy this $T_{dabc}$ to future input to increase readability, or keep the style of InputForm (7).

Alternatively, one can choose not to load look.m, but instead run resources/display.nb, which makes use of the Notation package in Mathematica. But not all the tensor appearances are defined in display.nb.

A. The core tensor calculations (tensor.m)

The module tensor.m provides functions for general tensor calculation and simplification. To make use of tensor.m, we first load the package using

```mathematica
Get["MathGR/tensor.m"]
```

(9)
The publicly defined functions and symbols are

- **Declare tensor symmetry (DeclareSym):** If a tensor has no symmetry, one can use the tensor directly without declaration. On the other hand, for tensors with symmetry, one had better to declare the symmetry such that MathGR can make use of this symmetry to simplify the tensor.

The symmetry of a tensor $T$ can be declared using

```mathematica
DeclareSym[T, {list of indices}, symmetry]
```

(10)
Here the list of indices are a sequence of UP and DN, representing upper and lower indices respectively. The symmetry could be either Symmetric or Antisymmetric of some slots, or some generic cyclic symmetries. The grammar for "symmetry" here is identical to the one defined in the Mathematica symbolic tensor system thus one can check Mathematica documentation for more details.

For example,

```mathematica
DeclareSym[T, {DN, DN, DN}, Antisymmetric[{1,2}]]
```

(11)
defines a tensor

$$T_{abc},$$

(12)
where the indices $abc$ can be any other indices, but upper and lower are distinguished here, as usual in GR. Here the tensor is asymmetric with permutation of $ab$ indices. Internally, this declaration is appended to the systemwide assumption variable $\$Assumptions, in a form similar to equation (3). In case of total symmetric tensor (declared by Symmetric[All]), DeclareSym also set the attribute of $T$ to be Orderless, thus the simplifications are made automatically before sending to the simplification engine.

There is no need to declare indices before using. Here it is not necessary to use strings ("a", etc.) as indices. One can also use DN[a] where a is a variable. However, once a is a variable and assigned to a value like $a = b + c$, the indices will have the same replacement and will not work as desired (unless one does it by purpose). Thus it is in general recommended to use strings as indices (or Clear and Protect the indices before usage if variables are preferred).

The MathGR package should be copied to a path that Mathematica can find, which is stored in the global variable $\$Path in Mathematica. For Linux the default path is `/usr/local/Mathematica/Applications/`.

---

4 There is no need to declare indices before using. Here it is not necessary to use strings ("a", etc.) as indices. One can also use DN[a] where a is a variable. However, once a is a variable and assigned to a value like $a = b + c$, the indices will have the same replacement and will not work as desired (unless one does it by purpose). Thus it is in general recommended to use strings as indices (or Clear and Protect the indices before usage if variables are preferred).

5 The MathGR package should be copied to a path that Mathematica can find, which is stored in the global variable $\$Path in Mathematica. For Linux the default path is `/usr/local/Mathematica/Applications/`
• Simplify tensors (Simp, SimpQ, SimpM, SimpH): Considering simplification is important for tensor calculation, 4 different commands are provided. Typically Simp is the one to use. For example, for the tensor $T$ defined above, one can run

$$T_{abc}T_{mnb} - T_{bac}T_{nmb} \// \text{Simp}$$

A result of 0 is obtained as desired. We shall explain the internal realization of Simp after discussing other simplification commands. Less trivial examples are considered when introducing gr.m, with simplification of curvature tensors.

There is also a list named SimpHook. This is a list of rules that the user wants to apply before and after simplification. The rules are in the format of Rule or RuleDelayed, e.g.

$$\text{SimpHook} = \{T_{abc} \rightarrow t \cdot F_{abc}\}$$

The SimpQ command stands for a quick simplification mechanism (and the only available mechanism that works for Mathematica with version lower than 9.0). In this method, the dummy indices are replaced by a standard set of indices specified by the variable StdIdx. For example,

$$\text{StdIdx} = \text{LatinIdx}$$

chooses $a, b, c, \cdots$ (which are strings instead of variables) as indices. Two other choices are also predefined. GreekIdx sets the standard indices to $\alpha, \beta, \gamma, \cdots$. And UniqueIdx sets the standard indices to a set of unique variables, to avoid the dummy variables to coincide with other variables.

The SimpM command calls Mathematica’s TensorReduce for doing simplification. The tensors with explicit indices are first translated into tensors without choice of bases, and then TensorReduce is called and the result is parsed back to tensors with explicit indices. This translation can be done in a fast way making use of the pattern matchings provided by Mathematica.

The SimpH command provides a hybrid simplification method: It first calls SimpQ, which does minimal simplification and is one order of magnitude faster than SimpM. Then SimpM is called for complete simplification. On the other hand, on Mathematica with version lower than 9.0, SimpH is set to SimpQ directly. Finally, Simp calls SimpH. But before and after calling, the SimpHook is applied.

• Partial derivative (Pd): Here we define the partial derivative Pd customly, instead of modifying the systemwide partial derivative D. The partial derivative can be called as

$$\text{Pd}[f[\text{indices}], \text{DN}[\text{index}]]$$

which displayed as $\partial_{\text{index}} f \cdots$, where $\partial$ is actually \[\text{\textit{\text{\textsc{CapitalSampi}}}}\] in Mathematica, which looks like $\partial$ (we try to avoid modifying system symbols such as the real $\partial$).

The derivative Pd has linearity and Leibniz rules builtin.

By defaults, the partial derivative acting on all tensors or scalars are nonzero. To define constants with vanishing partial derivative, one has to declare explicitly. For example

$$\text{Pd}[f1[f2[\_\_], \_]] := 0$$

defines both $f1$ and $f2[\_\_\_]$ as constants.

Note that this “define constant” approach is different from the builtin derivative in Mathematica, where without explicit function dependence, the variable is by default considered as constant. We consider the former to be safer for our purpose, otherwise one may forget to define non-constants.

• Decomposition of indices to lower dimensions (Decomp): This function decomposes the dummy indices of $m + 1 + 1 + \cdots + 1$ dimensional tensors into $m$ ($m \geq 0$) dimensional indices and other explicit indices. In case of $m = 0$, all indices are explicit. The explicit indices are marked as UE[n] and DE[n] for upper and lower indices respectively, where n is a number. The usage is

$$\text{Decomp}[\text{expression}, \text{rule}, \text{indices}]$$

If indices are not given, all dummy indices are decomposed. The rules are of the form, for example,

$$\{\{\text{DN}[]\rightarrow\text{DE}[], \text{UP}[]\rightarrow\text{UE}[]\}\&}, \{\{\text{DN}[]\rightarrow\text{DN}[]\}, \text{UP}[]\rightarrow\text{UP}[]\}\&$$

(19)
where the lower or upper index is decomposed into one explicit index \( DE[0] \) or \( UE[0] \), and another abstract index. There are a number of predefined decomposition schemes, namely

- Decomp0i[expression, indices]: decompose indices into 0 and i components. The rule is as illustrated above.
- Decomp01i[expression, indices]: decompose indices into 0, 1, i components.
- Decomp0123[expression, indices]: decompose indices into all explicit indices, 0, 1, 2, 3.
- Decomp1i[expression, indices]: decompose indices into 1, i components.
- Decomp123[expression, indices]: decompose indices into all explicit indices, 1, 2, 3.

Again if the indices are not explicitly specified, all dummy indices are decomposed. The free indices, if exist, are not touched.

To further ease the calculation, there is a list DecompHook, which contains a set of replacement rules to be applied after the decomposition. The typical use case is to specify the explicit form of the higher dimensional metric using those set of rules, as illustrated in the model file frwadm.m.

Before moving on, let us mention a subtly of Decomp family functions. In \( MathGR \), the engine does not know which set of indices the user is using, the higher dimensional one or the lower dimensional one. The users should take care of the difference themselves, for example, switch to a consistent StdIdx after decomposition to avoid confusion.

Also, one may define some “homogeneous and isotropic background” quantities with

\[
Pd[b, _DN]=0 \quad \text{but } Pd[b, DE[0]] \text{ is not zero} \tag{20}\]

In this case, Decomp family functions automatically screen the definition \( Pd[b, _DN]=0 \) until the decomposition is done.

- (Anti-)symmetrize tensors (Sym, Asym): The command

\[
\text{Sym}[expression, indices] \tag{21}\]

symmetrizes the tensor. When indices are not given, all free indices are symmetrized. For example, \( \text{Sym}[f_{\mu\nu}] \) gives \( f_{\mu\nu} + f_{\nu\mu} \). Note that we do not add factors as 1/2 here. The function Asym does similar things, only that a sign is added in front of each terms, determined by if the permutation is even or odd.

- The \( \delta \) symbol (Dta): The simplification of quantities like \( Dta_{\alpha c}f_{bc} \) is automatic. Without the need of calling Simp, this quantity is directly evaluated into \( f_{ba} \). On the other hand, if there are standalone \( Dta_{ab} \) which cannot be simplified, the displayed \( Dta_{ab} \) is in fact replaced by \( \text{HoldForm}[Dta]_{ab} \), which displays the same but have different InputForm. This HoldForm is released when doing Simp. Finally, \( Dta^{a}_{a} \) gives Dim, which denotes dimension of space or spacetime, but without predefined value.

### B. The GR definitions (gr.m)

This package includes definitions which have made use of a metric. To load the package, use

\[
\text{Get["MathGR/gr.m"]} \tag{22}\]

- Metric definition and usage: One may use a metric without any definition. However considering the symmetry and the associated inverse metric, it is convenient to define a metric before usage. Moreover, there is also a default metric called Metric, which is used to calculate metric contractions and curvature tensors. By defining a metric the metric can be assigned to this default Metric. A metric can be defined either globally or locally. The global definition of a metric \( g \) is

\[
\text{UseMetric}[g] \tag{23}\]

With this definition, \( MathGR \) declares symmetry for both the metric \( g_{\mu\nu} \) and the inverse metric \( g^{\mu\nu} \). Also \( g_{\mu}^{\mu} \) and \( g_{\alpha}^{\beta}g_{\beta}^{\alpha} \) are both aliased to \( \delta_{\mu}^{\nu} \). The derivative acting on the inversed metric \( \partial_{\lambda}g^{\mu\nu} \) is replaced using the identity \( \partial_{\lambda}g^{\mu\nu} = -g^{\rho\nu}g^{\sigma\lambda}\partial_{\lambda}g_{\rho\sigma} \). Also, the system’s default metric (with variable name Metric, and used for contraction, and calculating the affine symbol and curvature tensors) is assigned to \( g \).
Alternatively, one can use a metric locally, without affecting the global metric. There are two related function usages:

One can define a metric, say $h$, with all metric proprieties but without assigning the system’s default metric to $h$, using

$$\text{UseMetric}[h, \text{False}]$$  \hspace{1cm} (24)

Also, one can temporarily define a metric and do some calculation with it, using, e.g.

$$\text{WithMetric}[h, (\text{some tensors})]$$  \hspace{1cm} (25)

Here the (some tensors) are calculated using metric $h_{\mu \nu}$, but outside the WithMetric environment, the default metric is still $g$.

- **Curvature related symbols and tensors**: The affine connection $\Gamma^a_{bc}$ (named Affine in MathGR), intrinsic curvature tensors $R^{a\,bcd}, R_{abcd}, R^a_b, R[]$, extrinsic curvature tensors $K_{ab}, K[]$ are defined and will be calculated when called. There is also a tensor RADM[], which differs from the Ricci scalar $R[]$ by a total derivative.

For example,

$$\text{WithMetric}[g, R[]] \text{ // Simp}$$  \hspace{1cm} (26)

calculates and simplifies the Ricci Scalar with a temporary metric $g$. Note that by default, everything is calculated directly down to metrics. On the other hand, in case one wants to, for example, calculate $R[]$ to affine connection and stops, one can locally rename the affine connection to block the calculation going on. For example,

$$\text{Block}[\{\text{Affine=Gam}\}, R[] \text{ // Simp}]$$  \hspace{1cm} (27)

The covariant derivative is also defined, via

$$\text{CovD}[\text{expression}, \text{DN}[\text{index}]]$$  \hspace{1cm} (28)

as is standard in GR.

For example, one can check the second Bianchi identity for the curvature tensor

$$\text{WithMetric}[g, \text{CovD}[R_{abcd}, \text{DN}[e]] + \text{CovD}[R_{abde}, \text{DN}[c]] + \text{CovD}[R_{abec}, \text{DN}[d]] \text{ // Simp}]$$  \hspace{1cm} (29)

This gives 0 as expected, in about 1 second. The same test takes xAct (with pre-compiled C code) 10 seconds to complete.

- **Contraction with metric**: One can definitely use the straightforward way to do contraction. For example,

$$f[] = g[\text{UP}[\#1], \text{UP}[\#2]] \, f[\text{DN}[\#1], \text{DN}[\#2]] \& \text{[Unique[]], Unique[]}$$  \hspace{1cm} (30)

Note that here we have introduced unique variables. Otherwise there is a danger that if we use $f$ elsewhere with other tensors, the dummy indices of $f$ may coincide with the indices of other tensors and cause mistake.

To make the Unique[] function easier to use, we defined Uq[n] to stand for a sequence of unique variables. Thus the above equation can be written as

$$f[] = g[\text{UP}[\#1], \text{UP}[\#2]] \, f[\text{DN}[\#1], \text{DN}[\#2]] \& \text{Uq}[2]$$  \hspace{1cm} (31)

However, the above expression is still long, and the length will grow quickly with lots of contractions. To save writing, we provide a function MetricContract, to allow contraction with the default metric. The above example can be rewritten as

$$f[] = f[DG[1], DG[1]] \text{ // MetricContract}$$  \hspace{1cm} (32)

where DG and UG are a new type of indices, which is parsed by MetricContract. The same labels are contracted using the default metric. Multiple contractions can be calculated similarly, for example

$$f3[] = f[DG[1], DG[2]] \, f[DG[2], DG[3]] \, f[DG[3], DG[1]] \text{ // MetricContract}$$  \hspace{1cm} (33)

calculates $f_b^a f^c_c f^c_a$, in case only $f_{ab}$ with lower indices are defined.

---

6 The test is run on an Intel i5-2500 (3.30GHz) powered computer, running archlinux.

7 Note that in MathGR we do not assume relations between upper indices and lower indices (i.e. if the indices are raised and lowered by the metric, or by which metric) unless explicitly defined. This is why MetricContract is handy.
C. Integration by parts (ibp.m)

One typical use case for a GR package is to calculate a gravitational action

$$ S = \int_M d^d x \sqrt{-g} \mathcal{L}, $$

(34)

where if $M$ does not have boundary, total derivatives in $\mathcal{L}$ can be dropped. If $M$ has a boundary, those total derivatives can be reduced into boundary terms on the boundary $\partial M$. For a non-compact manifold, either could be true depending on if the boundary terms decouples.

For this purposes, we developed a module to factor $\mathcal{L}$ into total derivatives and the rest part. As usual, integration-like operations needs more intelligence than derivative-like operations. This module does not guarantee to find the total derivative for complicated cases. Nevertheless it works for simple cases and provide convenience for research.

In case that $\mathcal{L}$ is a pure total derivative, the final target result is unique. However, typically, $\mathcal{L}$ is a total derivative plus some rest part. Here depending on use cases, we designed different criteria to try minimizing the rest part:

- **Eliminate derivatives on a variable**: This can be used in a variation principle. For example,

  
  ```math
  Get["MathGR/ibp.m"]
  Ibp[y Pd[x, DN["i"]], IbpVar[x]]
  ```

  (35)

  (36)

  tries to eliminate derivatives acting on $x$, and gives a result

  $$ -x Pd[y, DN["i"]] + PdHold[x y, DN["i"]] $$

  (37)

  Here PdHold is a function defined to hold total derivatives. One can release this held total derivative by replacing it to Pd, or alternatively set it to zero by PdHold=0 if the manifold does not have a boundary.

- **Bring the rest part into standard form of a second order action**: In this case Ibp will try to eliminate terms in the rest part with more than two time derivatives on it. And the terms like $f x \dot{x}$ is transformed into $-\dot{x} f x^2/2 + PdHold[f x^2/2, DE[0]]$. The usage is

  ```math
  Ibp[f x Pd[x, DE[0]], IbpStd2]
  ```

  (38)

- **Leaf count**: If no criteria is given to Ibp, i.e. Ibp is called as Ibp[expression], the LeafCount function is applied to compare the rest part.

Internally, the Ibp function works as follows: First, a set of rules with wildcards are defined for integration by parts. Then every possible rule is applied to the expression and the result is sorted. The one with simplest rest part is chosen and the same set of rules are tried repeatedly on the new result until a fixed point is reached.

It is easy to extend the above algorithm such that the rules are applied multiple times before the result is sorted and selected. However in this case the time complexity increase quickly. In typical use cases, Ibp may deal with expressions with of order 1000 or more terms (considering the complexity of the gravitational action). In this case multiple-step rules are not realistic.

Here we have introduced the Ibp function motivated by calculating the gravitational action. On the other hand, this function can be certainly used for other purposes, as long as total derivatives are wanted.

D. Parsing of output (look.m)

As we mentioned at the beginning of this section, the output of *MathGR* can be parsed and brought into a better looking form. For this purpose, the package look.m should be loaded. No functions are provided in look.m. Instead, this module modify the global variable $\$Post to monitor output. Once there are new tensors appearing in the output, the module define the looking style of this tensor with the help of commands Format and Interpretation, to bring the display into superscript and subscript forms. After that the tensor name is added to a list TensorFormatList to prevent duplicate parsing.

There are also predefined looking styles for partial derivatives. Partial derivative acting on an abstract index is displayed as Capital Sampi, and time derivative (w.r.t $DE[0]$) is denoted by dot; derivative w.r.t. $DE[1]$ is denoted by prime.
E. Public utilities (util.m)

To ease some typical calculations, some utilities are provided. Those utilities are not directly about tensor calculation but can save some writing for those calculations. The utilities can be loaded using

\[ \text{Get["MathGR/util.m"]} \]  

and provide

- **SolveExpr**: By default, the variable to be inputted to the Mathematica command `Solve` should be atomic or a simple function. Expressions with head `Plus`, `Times` and `Power` are not allowed. `SolveExpr` solves this problem. For example, one can use

\[ \text{SolveExpr}[x^2 + y == 0, x^2] \]

(40)

to find a solution of \( x^2 \). To realize this, `SolveExpr` first replace \( x^2 \) by a unique temporary variable, solve the equation and replace the temporary variable back with \( x^2 \).

- **Series expansion and coefficients**: In `MathGR`, the default variable to control orders of perturbations is named `Eps`. In case of a perturbation theory calculation, one multiplies every perturbation variable by `Eps` and expand them together (an example can be found in `frwadm.m`). The series expansion and extracting the coefficients simply makes use of `Mathematica` functions `Series` and `Coefficient`. To save some writing, one can use

\[ \text{expression // SS}[n] \]

(41)

where \( n \) is an explicit integer, to expand expression up to \( n \)th order in `Eps`, or

\[ \text{expression // OO}[n] \]

(42)

to extract the order \( \text{Eps}^n \) terms in the expression and disregard all other terms. Simplification function `Simp` is called automatically after expansion or extraction of coefficients.

- **Fourier transformation**: In the calculation of second order action, doing Fourier transformation can eliminate total derivatives and ease calculations such as solving gravitational constraints. For example a term is transformed as

\[
\int d^4x \partial_i f(x) \partial_i f(x) \rightarrow \int dt \frac{d^3k}{(2\pi)^3} k^2 f_k(t)f_{-k}(t)
\]  

(43)

The transformation for a second order action (and only for second order action) is provided by `Fourier2[expression]`.

III. A SAMPLE CALCULATION: SECOND ORDER COSMIC PERTURBATIONS

To illustrate an explicit use case of the package, here we calculate the cosmic perturbations of inflationary cosmology up to second order. The result is well known for decades [4] (for a review with the same notation used here, see [5]). Nevertheless to present a standard and familiar calculation for illustration purpose may be more useful for a manual compared with presenting a new and unfamiliar calculation.

Here we present the input, the explanations and the results. The intermediate results are long and is available in the file `resources/Example_Cosmo.nb`

The model specification of a FRW universe with ADM type perturbations can be loaded by

\[ \text{Get["MathGR/frwadm.m"]} \]

(44)

Here the metric \( g_{\mu\nu} \) is defined by

\[
ds^2 = -N^2 dt^2 + h_{ij}(N^i dt + dx^i)(N^j dt + dx^j)
\]

(45)
where

\[ N = 1 + \text{Eps}\, \alpha, \quad N_i = \text{Eps}(b_i + \partial_i\beta) \quad h_{ij} = a^2 \exp(2\text{Eps}\, \zeta)\delta_{ij}, \]  

(46)

where \( \partial_i b_i = 0 \). Note that for simplicity only the scalar sector is considered in \( h_{ij} \). We have yet one gauge degree of freedom in the scalar sector. We can fix the gauge by either set \( \zeta = 0 \), or set the inflaton field to be homogeneous and isotropic. We shall consider the latter case as an example.

The action up to second order can be calculated with

\[
P\text{dHold}[_\ldots] := 0 \quad (* \text{Total derivatives can be neglected here.} *\)  
\[
P\mathbf{d}[\phi, _{DN}] := 0 \quad (* \text{The inflaton perturbation is gauged away.} *\)  
\[
s012 = \text{Sqrtg} \left( \frac{\text{RADM}[1]}{2} + \text{DecompG2H}[X[\phi]] - V[\phi] \right) / / \text{SS}[2]  
\]

(47)

(48)

(49)

Here DecompG2H is a function provided in frwadm.m (may move to more general places in case other models also need this function). This function calls Decomp0i to decompose a 4-dimensional quantity into 3+1 dimensions, where in the 4-dimensional quantities the metric \( g \) is used and in the 3-dimensional quantities the metric \( h \) is used. Contraction indices labelled by \( UG[n] \) and \( DG[n] \) are contracted using the metric \( g \) before decomposition.

The background equation of motion can be derived by the first order action. For this purpose, we extract the first order action and consider the variation principle:

\[
s1 = s012 // O0[1] \]  
\[
solBg = \text{SolveExpr}\{D[s1, \alpha]==0, D[Ibp[s1, IbpVar[\zeta]], \zeta]==0\}, \{V[\phi], Pd[\phi, \text{DE@0}^2]\} \]  
\[
\text{SimpHook} = \text{Union}[\text{SimpHook, solBg[[1]]}] \]  

(50)

(51)

(52)

where on the second line the background equation of motion is solved. Note that SolveExpr is used because we want to eliminate a composed expression \( \dot{\phi}^2 \). Also note that there are derivatives on \( \zeta \) in \( s1 \). Thus we should first do integration by parts before applying the variation principle \( D[\ldots, \zeta]=0 \). On the third line the background solution is added to SimpHook. Thus it will be automatically applied when simplifying the second order action.

Now we can work on the second order perturbations:

\[
s2 = s012 // O0[2] // \text{Fourier2} \]  
\[
\text{cons} = \text{Solve}\{D[s2, \alpha]==0, D[s2, \beta]==0, D[s2, b[DN@"a"]]==0\}, \{\alpha, \beta, b[DN@"a"]\} [[1]] \]  
\[
s2Solved = \text{Ibp}[s2 /. \text{cons, IbpStd2}] \]  

(53)

(54)

(55)

Here in \( s2Solved \), we shall recover the well known result

\[ a^3\dot{\zeta}^2 - a\dot{k}^2\dot{\zeta}^2 \]  

(56)

as the second order action, with all constraints solved.

**IV. CONCLUSION AND FUTURE DIRECTIONS**

Here we presented the usage of a new tensor package, \textit{MathGR}, which is simple and lightweight, such that people can understand and modify the internal more easily.

We shall keep the simplicity of the package. While new functionalities are expected to be added, we shall not add functionalities which significantly increase the complexity of the package, especially for the core parts tensor.m and gr.m.

We hope to keep the current existing application programming interface (API) stable. In other words, we shall try our best to keep the currently working calculations work for the future versions. We shall extend the current API by adding new functions, and add options with default values to the current functions. (The arXiv version of) this paper will keep updating for the API extensions (and in case of API change).

However, there is one exception. We hope to improve look.m in the future, such that the result parsed by look.m can not only be re-evaluated, but also be edited. Currently only the former is possible and this extension, when happens, may break the result parsed by older versions of look.m. Nevertheless, the InputForm of the expression will not be modified. For users who want a stable API from now, they can avoid using look.m, or keep the current version of look.m while updating other files to future versions.

We shall in the future add more comments to the existing code, and add broader coverage for unit tests and integrated tests. Those efforts will help for the users who want to hack and fork the package.

Finally but most importantly, as the package is being tested and used in realistic research, we expect to encounter bugs and provide bug fixes. As always, the result from \textit{MathGR} should be checked by independent calculations before being trusted and used in research.
Acknowledgments

This research is supported by fundings from Kavli IPMU (WPI), the University of Tokyo.

[1] Peter Musgrave, Denis Pollney and Kayll Lake, [http://grtensor.phy.queensu.ca/](http://grtensor.phy.queensu.ca/) Peter Musgrave, Denis Pollney and Kayll Lake 1996 “GRTensorII: A package for General Relativity” Fields Institute Communications 15 313-318

[2] J. M. Martin-Garcia, [http://www.xact.es/](http://www.xact.es/) J. M. Martin-Garcia, “xPerm: fast index canonicalization for tensor computer algebra”, Comp. Phys. Commun. 179 (2008) 597-603, [arXiv:0803.0862](http://arxiv.org/abs/0803.0862) [cs.SC].

[3] John M. Lee, [http://www.math.washington.edu/~lee/Ricci/](http://www.math.washington.edu/~lee/Ricci/)

[4] V. F. Mukhanov and G. V. Chibisov, “Quantum Fluctuation and Nonsingular Universe. (In Russian),” JETP Lett. 33, 532 (1981) [Pisma Zh. Eksp. Teor. Fiz. 33, 549 (1981)].

[5] Y. Wang, [arXiv:1303.1523](http://arxiv.org/abs/1303.1523) [hep-th].