NRPyLaTeX: A LaTeX interface to computer algebra systems for general relativity

Kenneth J. Sible[1,†], **Zachariah B. Etienne**[2,3,4,‡]

[1] Department of Computer Science & Engineering, University of Notre Dame, Notre Dame, IN 46556, USA

[2] Department of Physics, University of Idaho, Moscow, ID 83843, USA

[3] Department of Physics & Astronomy, West Virginia University, Morgantown, WV 26506, USA

[4] Center for Gravitational Waves and Cosmology, West Virginia University, Chestnut Ridge Research Building, Morgantown, WV 26505, USA

[†] E-mail: ksible@nd.edu

[‡] E-mail: zetienne@uidaho.edu

Abstract. While each computer algebra system (CAS) contains its own unique syntax for inputting mathematical expressions, LaTeX is perhaps *the* most widespread language for typesetting mathematics. NRPyLaTeX (NL) enables direct LaTeX input of complex tensorial expressions (written in Einstein notation) relevant to general relativity and differential geometry into the SymPy CAS. As SymPy also supports output compatible with the Mathematica and Maple CASs, NL lowers the learning curve for inputting and manipulating tensorial expressions in three widely used CASs. LaTeX however is a typesetting language, and as such is not designed to resolve ambiguities in mathematical expressions. To address this, NL implements a convenient configuration interface that, e.g., defines variables/keywords and assigns properties/attributes to them. Configuration commands appear as LaTeX comments, so that entire NL workflows can fit seamlessly into the LaTeX source code of scientific papers without interfering with the rendered mathematical expressions. Further, NL adopts NRPy+'s rigid syntax for indexed symbols (e.g., tensors), which enables NL output to be directly converted into highly optimized C/C++-code kernels using NRPy+. Finally NL has robust and user-friendly error-handling, which catches common tensor indexing errors and reports unresolved ambiguities, further expediting the input and validation of LaTeX expressions into a CAS.

Keywords: LaTeX, parser-lexer, general relativity, numerical relativity, Einstein notation, differential geometry, NRPy+, SymPy, computer algebra

Submitted to: *Class. Quantum Grav.*

1. Introduction

LaTeX is perhaps *the* canonical markup language for communicating mathematical ideas in scientific papers, and most practitioners of general relativity and differential geometry are proficient in it. Further, computer algebra systems (CASs) like `SymPy` [1], `Mathematica` [2], or `Maple` [3] are often an essential resource in these fields, but to our knowledge there are currently no CASs or CAS add-on packages that support the input of tensorial/indexed LaTeX expressions adopting Einstein notation. Thus new practitioners are often required to learn one language for communicating mathematical ideas in scientific papers, and another to input & manipulate mathematical expressions in a CAS. As a result expressions developed within a CAS often need to be translated into LaTeX or vice-versa when constructing a scientific paper, and human errors in transcription can become a menace to productivity and reproducibility of scientific results.

Various packages have been developed to ease the input of tensorial/indexed expressions using Einstein notation into CASs. For example, `xTensor` [4] is an open-source add-on package for the `Mathematica` CAS, greatly extending its capabilities for performing abstract tensor calculus. In short, `xTensor` enables the definition and manipulation of tensors with arbitrary symmetries on multiple manifolds. Upon declaring a metric for a given manifold, covariant, Lie, and other derivatives may be computed, and associated curvature tensors may be immediately defined. As it is an extremely powerful add-on to one of the most popular and powerful CASs, `xTensor` finds wide use across the community. Despite its success, perhaps the largest barrier to entry for new users is its dependency on the closed-source `Mathematica`, whose software licenses carry a price that can be too high for some users.

`SymPy` is another very powerful CAS used widely across the scientific community. Unlike `Mathematica` or `Maple`, `SymPy` is built upon `Python` and is open-source (under the permissive 3-Clause BSD license). While `SymPy` contains both basic support for tensors and differential geometry, as well as basic, experimental LaTeX input capabilities, LaTeX input of tensorial/indexed expressions is not currently supported[a]. Instead, to input and manipulate tensorial/indexed expressions found in general relativity and differential geometry contexts, a few add-on packages for `SymPy` have been developed. These packages are somewhat analogous to `xTensor`, but they benefit from the fact that `SymPy` possesses no expensive software license. `EinsteinPy` [5] is one such (open-source, free) package, which extends basic tensor support in `SymPy` to handle metrics and construction of their associated connections and curvature tensors (Riemann & Ricci). It also contains friendly `Jupyter` notebook documentation and functionality for e.g., plotting geodesics in various spacetimes. The `GraviPy` [6] add-on package to `SymPy` possesses similar basic tensor symbolic manipulation functionality as `EinsteinPy`, but in addition e.g., supports the construction of the Einstein tensor, as well as user-defined tensors and covariant derivatives.

a as of `SymPy` 1.9

While `xTensor`, `EinsteinPy`, and `GraviPy` are exemplary packages that effectively convert popular CASs into powerful tools for general relativity and differential geometry, each depends on its own custom syntax for inputting tensorial expressions. As discussed, this adds a slight learning curve for new users, and opens the door to LaTeX $\leftrightarrow$ CAS transcription errors when preparing scientific papers.

Our new, open-source[b] `NRPyLaTeX` package provides a multifaceted LaTeX interface to `SymPy`, with general relativity and differential geometry workloads in mind. `NRPyLaTeX` enables the input of tensorial expressions exactly as they would appear in the LaTeX source code of scientific papers, and it possesses native support for e.g., multiple metrics; custom tensors of arbitrary rank & dimension; index raising & lowering; and covariant, Lie, & partial derivatives.

As LaTeX is a typesetting markup language, it is neither designed to resolve ambiguities in mathematical expressions, nor able to point out e.g., errors in Einstein notation. To address ambiguities (e.g., which rank-2 tensor represents the metric), `NRPyLaTeX` contains a robust and user-friendly configuration interface, which enables one to define variables and keywords, assign properties and attributes, ignore commands and substrings, and perform (syntactic) string replacement. For example, this includes the ability to declare variables, define tensors, input into `SymPy` un-rendered equations, declare arbitrary operators, and provide aliases for variable names. Conveniently this configuration interface exists entirely within LaTeX comments, enabling it to be interspersed with the rendered LaTeX without itself being rendered. Thus it provides a means to share CAS worksheets *within the source code of scientific papers*, while minimizing the chances of a transcription error when translating CAS input expressions to a LaTeX-ed scientific paper. It also provides a new means for inputting LaTeX-ed equations (found in e.g., scientific paper source codes on the arXiv) directly into CASs.

To address errors in Einstein notation (e.g., "indexing" errors) and to help guide the user in resolving ambiguities in mathematical expressions, `NRPyLaTeX` contains a comprehensive error handling infrastructure. For example, this infrastructure will return errors when a LaTeX-ed tensorial expression written in Einstein notation contains a repeated "down" index, or an unbalanced free index across the equality, or the use of a covariant derivative without specifying the associated metric.

As its name implies, `NRPyLaTeX` falls under the umbrella of our `SymPy`-based `NRPy+`[c] framework [7, 8]; `NRPyLaTeX` can be used seamlessly for not only symbolic but also numerical workflows using `NRPy+`. In short, `NRPy+` builds upon and extends `SymPy`'s native capabilities, to convert often complex indexed expressions (e.g., tensorial expressions) directly into highly optimized `C/C++`-code kernels with common-subexpression elimination; single-instruction, multiple-data (SIMD); and finite-difference derivative code generation capabilities. `C/C++`-code kernels like these form the heart of numerical relativity codes.

The remainder of the paper is organized as follows. In Sec. 2 we discuss the

b under the permissive 2-Clause BSD license

c `NRPy+` is short for "Python-based code generation for numerical relativity... and beyond!".

basic features and internal structure of NRPyLaTeX; and in Sec. 3 present symbolic and numerical workflows, as well as usage and error handling. Finally in Sec. 4 we summarize the basic results and present plans for future work.

2. Basic Features and Methodology

NRPyLaTeX provides a LaTeX frontend to the SymPy CAS[d], supporting the input of tensorial expressions written in Einstein notation. It includes a number of features with general relativity and differential geometry applications in mind, including covariant and Lie derivatives; spacetime and reference metrics; Levi-Civita and Christoffel symbols; as well as metric inverses and determinants, just to name a few.

At its core the NRPyLaTeX frontend is a lexer/parser[e] with a powerful configuration interface that addresses ambiguities (e.g., specifying which metric is associated with a covariant derivative, or the dimension of a tensor) and defines arbitrary variable/operator macros. Configuration options are input as LaTeX comments so that all information needed to construct symbolic expressions can be shared in, e.g., a scientific paper's LaTeX source code.

NRPyLaTeX adopts the NRPy+ notation for indexed symbols, so that e.g., $g_{\mu\nu}$ is accessible as a nested list of SymPy symbols `gDD[mu][nu]`, where e.g., g_{tt} is stored in `gDD[0][0]`. Here `D` denotes a covariant (*down*) index. Similarly `U` is appended to the tensor name for each contravariant (*up*) index. In fact, for each indexed symbol of dimension n and rank r, NRPyLaTeX will create an r-dimensional array (or symbol) of size $n \times \ldots \times n = n^r$ using NRPy+ notation.

As NRPy+ is designed to convert SymPy expressions into highly optimized C/C++-code kernels, compatibility with SymPy and NRPy+ enables NRPyLaTeX to interface well with symbolic and numerical workloads, respectively. For example, specifying that the partial derivative $g_{\mu\nu,\delta}$ should be evaluated numerically requires that one simply first define the variable `gDD` with a placeholder differentiation type in a NRPyLaTeX configuration line. Once this is completed, all symbolic first derivatives of $g_{\mu\nu}$ will be accessed in the NRPy+-compatible syntax `gDD_dD[mu][nu][delta]`, second derivatives as `gDD_dDD[mu][nu][delta][gamma]`, and so forth. NRPy+ can interpret such partial derivative symbols as requiring numerical (finite-difference) differentiation and generate the appropriate C/C++-code kernel.[f]

Finally, NRPyLaTeX is developed using modern software engineering techniques, including `git` version control[g], robust error handling, interactive Jupyter notebook documentation, convenient IPython/Jupyter interfaces, and more than 80 code-

d SymPy provides a pipeline from NRPyLaTeX to Mathematica/Maple through their Printing module.
e NRPyLaTeX tokenizes the input LaTeX, using substring pattern matching, and parses the resulting token iterator at a syntactic level, including proper handling of Einstein notation.
f NRPyLaTeX extends the NRPy+ notation for differentiation, in order to support Lie derivatives with respect to an arbitrary vector v^i `_ld[v]D..D` and covariant derivatives (`_cdD`, `_cdU`, `_cdDU`, etc.)
g NRPyLaTeX source code can be downloaded from `https://github.com/zachetienne/nrpylatex`, though it is generally advisable to install NRPyLaTeX through `pip` via `pip install nrpylatex` .

validation unit tests. These tests check NRPyLaTeX's error handling, as well as a large number of challenging tensor parsing edge cases and use case examples. Perhaps the most sophisticated such use case is NRPyLaTeX's ability to reproduce the entire covariant BSSN [7, 9–11] system of equations in a NRPy+-compatible form; the generated symbolic expressions have been validated against the handwritten SymPy expressions for the BSSN formalism appearing within NRPy+. Associated with the BSSN validation test is a Jupyter notebook tutorial that demonstrates precisely how to input such equations into NRPyLaTeX. All validation tests are run within a continuous integration infrastructure, which is triggered upon any commit to the git repository. Further NRPyLaTeX exists within the Python Package Index (PyPI) and thus can be installed from a standard Python installation via `pip install nrpylatex`. Though, for the purpose of reproducing results in this paper, one should specify version `1.0.8`.

Next we demonstrate NRPyLaTeX's basic features through real-world examples, highlighting both its ability to handle symbolic (Sec. 3.1) and numerical/NRPy+ (Sec. 3.2) workloads, as well as its exception handling (Sec. 3.3).

3. Examples

3.1. Symbolic Workload Example: Schwarzschild Solution to Einstein's Equations

```
1  \begin{align}
2      % keydef basis [t, r, \theta, \phi]
3      % vardef -const 'G', 'M'
4      % vardef -zero 'gDD' (4D)  %% initialize every component of gDD to zero
5      g_{t t} &= -\left(1 - \frac{2GM}{r}\right) \\ %% g_{t t} = g_{0 0}
6      g_{r r} &=  \left(1 - \frac{2GM}{r}\right)^{-1} \\
7      g_{\theta\theta} &= r^2 \\
8      g_{\phi\phi} &= r^2 \sin^2\theta \\
9      % assign -metric 'gDD'  %% inverse gUU, determinant det(gDD), and connection GammaUDD
10     R^\alpha{}_{\beta\mu\nu} &= \partial_\mu \Gamma^\alpha_{\beta\nu} - \partial_\nu \Gamma^\
           alpha_{\beta\mu} + \Gamma^\alpha_{\mu\gamma} \Gamma^\gamma_{\beta\nu} - \Gamma^\alpha_
           {\nu\sigma} \Gamma^\sigma_{\beta\mu} \\
11     R_{\beta\nu} &= R^\alpha{}_{\beta\alpha\nu} \\
12     R &= g^{\beta\nu} R_{\beta\nu} \\
13     G_{\beta\nu} &= R_{\beta\nu} - \frac{R}{2} g_{\beta\nu} \\
14     K &= R^{\alpha\beta\mu\nu} R_{\alpha\beta\mu\nu} %% automatic index raising & lowering
15 \end{align}
```

Figure 1: Symbolic workload example: Schwarzschild solution of general relativity

In this example, we demonstrate NRPyLaTeX's ability to handle symbolic workloads, by first defining the Schwarzschild metric, and then constructing the corresponding Riemann tensor $R^{\alpha}{}_{\beta\mu\nu}$, Ricci tensor $R_{\beta\nu}$, Ricci scalar R, Einstein tensor $G_{\beta\nu}$, and the Kretschmann scalar K. As shown in Fig. 1, Lns. 2–3, the coordinate basis and all constants (ignored during differentiation) must first be specified in the NRPyLaTeX configuration. Then in Ln. 4, the 4-metric $g_{\mu\nu}$ is initialized to zero (this ensures off-diagonal terms are set).

The remainder of the example is almost entirely boilerplate LaTeX, which one might find in the source code of a journal article. In Lns. 5–8 the diagonal components of the

Schwarzschild metric are specified. Line 9 is the final configuration command, which assigns $g_{\mu\nu}$ as the metric, and in doing so automatically constructs the metric inverse $g^{\mu\nu}$ and the associated connection $\Gamma^{\sigma}_{\mu\nu}$, both of which are essential to define the Riemann curvature tensor $R^{\alpha}{}_{\beta\mu\nu}$, the Ricci curvature tensor $R_{\beta\nu}$, the Ricci scalar R, the Einstein tensor $G_{\beta\nu}$, and the Kretschmann scalar K:

$$R^{\alpha}{}_{\beta\mu\nu} = \partial_\mu \Gamma^{\alpha}_{\beta\nu} - \partial_\nu \Gamma^{\alpha}_{\beta\mu} + \Gamma^{\alpha}_{\mu\gamma}\Gamma^{\gamma}_{\beta\nu} - \Gamma^{\alpha}_{\nu\sigma}\Gamma^{\sigma}_{\beta\mu} \tag{1}$$

$$R_{\beta\nu} = R^{\alpha}{}_{\beta\alpha\nu} \tag{2}$$

$$R = g^{\beta\nu} R_{\beta\nu} \tag{3}$$

$$G_{\beta\nu} = R_{\beta\nu} - \frac{R}{2} g_{\beta\nu} \tag{4}$$

$$K = R^{\alpha\beta\mu\nu} R_{\alpha\beta\mu\nu}. \tag{5}$$

Upon parsing the contents of Fig. 1, NRPyLaTeX generates symbolic (SymPy) expressions for all defined tensors and scalars using NRPy+ notation as described in Sec. 2, and injects these expressions into the local namespace. So for example one can immediately access $g_{\theta\theta}$ from `gDD[2][2]`, finding that indeed it stores r^2. Similarly, one can immediately confirm that each component of the Einstein tensor $G_{\mu\nu}$, stored in the nested list `GDD[mu][nu]` is identically zero (this requires use of SymPy's simplification routine), and that the Kretschmann scalar (stored within the local variable `K`) is equal to the known value for the Schwarzschild solution $K = 48(GM/r)^2$. Thus we are able to confirm that NRPyLaTeX is correctly parsing these complex symbolic expressions, verifying that the Schwarzschild metric indeed solves Einstein's equations and that invariants take values consistent with the literature.

Next we move on to a workload in which the metric is only known numerically, and thus the metric and metric derivatives must be left in a general unspecified form. In this workload, the resulting expressions can be directly input into NRPy+, which outputs highly optimized C/C++-code kernels that can adopt finite-difference representations to approximate partial derivatives.

3.2. Numerical Workload Example: BSSN Hamiltonian Constraint for Vacuum Spacetimes with Unspecified Metric

The 3+1 Baumgarte-Shapiro-Shibata-Nakamura (BSSN) formalism of general relativity [12, 13] is perhaps the most widely used formulation within the numerical relativity community. Transcribing the BSSN equations into a hand-optimized C/C++/Fortran code can be a daunting task by itself, not to mention the debugging process. Here we present a NRPyLaTeX workflow aimed at generating a SymPy expression for the BSSN Hamiltonian constraint (i.e., the Hamiltonian constraint written in terms of BSSN variables) in vacuum[h] The resulting expression can be plugged directly into NRPy+ to generate highly optimized C/C++-code kernels.

h The NRPy+ tutorial contains a Jupyter notebook for parsing the entire BSSN formalism in vacuum: https://github.com/zachetienne/nrpytutorial.

We start by defining the BSSN conformal 3-metric $\bar{\gamma}_{ij}$, which is related to the the physical 3-metric γ_{ij} via

$$\bar{\gamma}_{ij} = e^{-4\phi}\gamma_{ij}. \tag{6}$$

Further the BSSN formalism decomposes the physical extrinsic curvature K_{ij} into its trace K and its trace-free part $\bar{A}_{ij}$:

$$\bar{A}_{ij} = e^{-4\phi}\left(K_{ij} - \frac{1}{3}\gamma_{ij}K\right). \tag{7}$$

Given these definitions, the Hamiltonian constraint is written (Eq. 46 of [7])

$$H = \frac{2}{3}K^2 - \bar{A}_{ij}\bar{A}^{ij} + e^{-4\phi}\left(\bar{R} - 8\bar{D}^i\phi\bar{D}_i\phi - 8\bar{D}^2\phi\right), \tag{8}$$

where $\bar{D}_i$ and $\bar{R}$ are the covariant derivative and Ricci scalar associated with $\bar{\gamma}_{ij}$, respectively.

The following example was written as a validation test against the same expression manually input into `NRPy+` (and independently validated against other trusted BSSN codes). As such, it adopts the `NRPy+` convention referring to the evolved "conformal factor" variable as cf. `NRPy+` natively supports three options for conformal variable, but following e.g., [7, 10, 14] instead of evolving ϕ or $e^{-4\phi}$ directly in a numerical relativity code when constructing the spacetime, the variable cf $= W = e^{-2\phi}$ is evolved. This variable is generally chosen as it is a smoother function near puncture black holes, resulting in lower truncation errors when evaluating numerical derivatives.

```
1  % vardef -diff_type=dD -metric 'gammabarDD' %% append _dD to each partial derivative
2  % vardef -diff_type=dD -symmetry=sym01 'AbarDD', 'RbarDD' %% symmetry [i][j] == [j][i]
3  % parse \bar{R} = \bar{\gamma}^{ij} \bar{R}_{ij} %% internal calculation (no rendering)
4  % vardef -diff_type=dD 'cf'
5  % srepl "e^{-4\phi}" -> "\text{cf}^2" %% syntactic string replacement
6  % srepl -persist "\partial_<1> \phi" -> "\partial_<1> \text{cf} \frac{-1}{2 \text{cf}}"
7  % srepl "\bar{D}^2" -> "\bar{D}^i \bar{D}_i" %% custom operator definition
8  H = \frac{2}{3} K^2 - \bar{A}_{ij} \bar{A}^{ij} + e^{-4\phi} \left(\bar{R} - 8 \bar{D}^i \phi
       \bar{D}_i \phi - 8 \bar{D}^2 \phi\right)
```

Figure 2: Numerical/`NRPy+` workload example: The Hamiltonian constraint of the BSSN formalism

In Fig. 2, we demonstrate the procedure for constructing the Hamiltonian constraint in terms of BSSN variables using as input unspecified spacetime variables W, K, $\bar{A}_{ij}$, $\bar{\gamma}_{ij}$, and $\bar{R}_{ij}$, and any derivatives will be computed numerically (outside of `NRPyLaTeX`) from these given variables. By "unspecified", we mean that e.g., each component of the conformal 3-metric is left completely symbolic (e.g. γ_{01} would be the `SymPy` symbol `gammaDD01`, accessible from the local namespace variable `gammaDD[0][1]`). Further, unlike the example of Sec. 3.1 (Ln. 1), we do not specify a particular basis as this expression is covariant.

Line 1 defines an unspecified, 3-dimensional metric $\bar{\gamma}_{ij}$ and changes the derivative type to append a suffix `_dD` instead of attempting to evaluate

each derivative symbolically. The resulting symbolic expressions involving e.g., `gammabarDD_dD[j][k][i]` (i.e., $\bar{\gamma}_{jk,i}$) can be passed into `NRPy+`, which generates `C/C++`-code kernels that evaluate the derivatives using finite differences. Next (Ln. 2) the symmetric tensors `AbarDD` and `RbarDD` are defined in a similar way (i.e., they each appear in the final expression). As `AbarDD` and `RbarDD` are unspecified (i.e., we assume they are computed independently, and thus used here as input into the expression), their symmetry in the first two indices "`sym01`" (i.e. $\bar{A}_{ij} = \bar{A}_{ji}$ and $\bar{R}_{ij} = \bar{R}_{ji}$) must be manually imposed. Then (Ln. 3) the scalar `Rbar` is constructed by contracting `RbarDD` with itself, i.e. $\bar{R} = \bar{\gamma}^{ij}\bar{R}_{ij}$.[i]

Next, we apply a two-step procedure to replace every instance of ϕ with the chosen evolved "conformal factor" variable $\text{cf} = W = e^{-2\phi}$. First, we define the conformal factor `cf` (Ln. 4) and replace every instance of $e^{-4\phi}$ with cf^2 (Ln. 5). Note that this and all `srepl` replacements are done on the syntactic level, and are *not* string replacements. Second (Ln. 6), we replace each partial derivative of the form $\partial_{\langle 1\rangle}\phi$ for any index $\langle 1\rangle$ with the expression $(-1/2)\text{cf}^{-1}\partial_{\langle 1\rangle}\text{cf}$, as implied by the chain rule:

$$\partial_{\langle 1\rangle}\text{cf} = \partial_{\langle 1\rangle}e^{-2\phi} = -2e^{-2\phi}\partial_{\langle 1\rangle}\phi \Rightarrow \partial_{\langle 1\rangle}\phi = -\frac{1}{2}\text{cf}^{-1}\partial_{\langle 1\rangle}\text{cf}.$$

Line 6 also enables the `-persist` option, which ensures this replacement is made inside of each covariant derivative expansion.

Finally (Ln. 7), `srepl` is used one more time to define a custom contraction operator $\bar{D}^2 := \bar{D}^i\bar{D}_i$. `NRPyLaTeX` expands each covariant derivative using the metric $\bar{\gamma}_{ij}$ and the associated, internally generated connection Γ^i_{jk}. All components of Γ^i_{jk} are available to the user in their local namespace, as well as the covariant derivatives (represented with the suffix `_cd[UD]..[UD]`; the `..` notation here denotes repetition to match the rank, or number of indices, of an indexed symbol). All this defined, the Hamiltonian constraint is immediately constructed in a form ready for `NRPy+ C/C++`-code kernel generation.

i Note, we could also use $R = \bar{R}^i{}_i$ instead of $\bar{R} = \bar{\gamma}^{ij}\bar{R}_{ij}$ since `NRPyLaTeX` performs automatic index raising and lowering using the metric.

3.3. Example 3: NRPyLaTeX Exception Handling and Index Checking

```
In [1]: %load_ext nrpylatex.extension
In [2]: %%parse_latex
   ...: % vardef 'gUD', 'vD'
   ...: v^a = g^a_b v_b
TensorError: illegal bound index 'b' in vU
In [3]: %%parse_latex
   ...: % vardef 'gUU'
   ...: v^a = g^{cb} v_b
TensorError: unbalanced free index {'a', 'c'} in vU
In [4]: %parse_latex v^a = g^{a*} v_b
ParseError: v^a = g^{a*} v_b
                      ^
unexpected '*' at position 10
In [5]: %parse_latex T_{ab} = v_a w_b
ParseError: T_{ab} = v_a w_b
                           ^
cannot index undefined tensor 'wD' at position 32
In [6]: %parse_latex J^a = (4\pi k)^{-1} \nabla_b F^{ab}
ParseError: J^a = (4\pi k)^{-1} \nabla_b F^{ab}
                                ^
cannot generate covariant derivative without defined metric 'g'
```

Figure 3: Examples of error handling within an `IPython` (interactive `Python`)/`Jupyter` session of `NRPyLaTeX`

Figure 3 demonstrates `NRPyLaTeX`'s robust system for index checking and generic exception handling, implemented using its own `TensorError` and `ParseError` algorithms, respectively. In short these algorithms report ambiguities in mathematical expressions, violations of Einstein notation, and problems at the syntactic level with error reporting that is human-friendly, both being descriptive and pointing to the exact location of the error.

`NRPyLaTeX` will raise its `TensorError` exception upon violation of the Einstein summation convention. In Fig 3, Cell 1 we load the custom `IPython`/`Jupyter` extension module `nrpylatex.extension` using the built-in `load_ext` magic command.[j] In Cell 2 we attempt to parse the equation $v^a = g^a{}_b v_b$, which is invalid as Einstein notation requires that a repeated ("bound") index may appear only once as a superscript and exactly once as a subscript in any given term. `NRPyLaTeX` thus raises a `TensorError` exception for an 'illegal bound index', which occurred since the index `b` appeared twice in the *down* position. Next, in Cell 3 we attempt to parse the equation $v^a = g^{cb} v_b$, proper Einstein summation notation requires a free index to appear in every term with the same ("up" or "down") position and cannot be summed over in any term. `NRPyLaTeX` checks for this and raises a `TensorError` exception for an 'unbalanced free index', which occurred since index `a` and index `c` appeared only once on each side of the equation.

`NRPyLaTeX` will raise the custom `ParseError` exception upon identifying a syntax error in the argument string of the `parse_latex` function or magic command. In Cell 4, we attempt to parse the equation $v^a = g^{a*} v_b$, but `NRPyLaTeX` raised a `ParseError`

j In a standard `Python` environment, we would instead import the `parse_latex` function from the `nrpylatex` module using the `import` keyword.

exception for an 'unexpected' lexeme as NRPyLaTeX did not recognize a valid index. NRPyLaTeX could also raise a ParseError exception for an 'expected' token if only that exact token would satisfy a production rule (e.g. `\sqrt[0.2]{...}` would raise that exception since NRPyLaTeX would expect an integer for the root). In Cell 5, we attempt to parse the equation $T_{ab} = v_a w_b$, but NRPyLaTeX raises a ParseError exception for indexing an 'undefined' symbol wD, which occurred because the indexed symbol wD has not been defined (i.e., NRPyLaTeX did not find the indexed symbol wD in the namespace). This exception can be addressed by defining wD with the vardef macro. In Cell 6 we attempt to parse the equation $J^a = (4\pi k)^{-1}\nabla_b F^{ab}$, but NRPyLaTeX raised a ParseError exception since a covariant derivative cannot be generated without first defining a metric. For covariant derivatives without a diacritic, NRPyLaTeX searches the namespace for the metric gDD (or gUU) and did not find it, thus throwing the error. The Appendix describes more generally which metric is assumed to be associated with a given covariant derivative.

4. Conclusions & Future Work

NRPyLaTeX provides a LaTeX interface for the free and open source SymPy CAS, and includes native support for Einstein notation. As LaTeX is widely used in the scientific community and the NRPyLaTeX infrastructure is completely free and open source, the learning curve for new users is minimized; the financial barrier to entry is eliminated; and anyone is free to modify and extend NRPyLaTeX (within the confines of its permissive 2-Clause BSD license). In addition to Einstein notation, NRPyLaTeX can, provided a metric, dynamically perform index raising and lowering, and automatically generate the metric inverse, determinant, and connection. Moreover, NRPyLaTeX expands covariant and Lie derivatives; and generates the Levi-Civita symbol (of arbitrary rank).

NRPyLaTeX's configuration consists of a command system modeled on the POSIX command syntax, aiming to reduce the learning curve. In brief, the command system defines variables, keywords, and ignore commands; assigns properties and attributes; and performs (syntactic) string replacement. To prevent rendering of NRPyLaTeX commands, we embed each command inside of a LaTeX comment; i.e., in text that follows a single percent symbol. This enables entire NRPyLaTeX workflows to be stored seamlessly within a scientific paper's LaTeX source code.

In summary, use cases for NRPyLaTeX range from symbolic manipulation of complex tensorial expressions in Jupyter notebooks, numerical code generation (e.g., when combined with its sister code NRPy+), and sharing of mathematical workflows directly within the LaTeX source code of a scientific paper.

In the future, we plan on expanding the integration with NRPy+ to include reference metric support. This will enable one to choose from the broad class of curvilinear coordinate systems that NRPy+ supports [7], to output covariant expressions in a particular coordinate system. In addition, we plan to incorporate native support for Post-Newtonian (PN) notation. In the current build of NRPyLaTeX, we are capable

of parsing PN notation using the `srepl` command to remap the PN notation for the dot/cross products of vectors to their definitions in Einstein notation. However, this approach requires too many `srepl` replacements to be both robust and feasible. Instead, we are targeting native PN support for a future release of `NRPyLaTeX`. Finally, we plan on adding a direct pipeline for parsing LaTeX to optimized `C/C++`-code using `NRPy+`.

Acknowledgments

The authors would like to thank S. R. Brandt, R. Haas, and D. Chiang for useful discussions and suggestions during the preparation of `NRPyLaTeX`. Support for this work was provided by NSF awards OAC-2004311, PHY-1806596, as well as NASA awards ISFM-80NSSC18K0538 and TCAN-80NSSC18K1488.

Appendix (`NRPyLaTeX` v1.0.8 Reference Manual)

This Appendix serves both to ensure completeness in the exposition of `NRPyLaTeX` features and to provide a quick reference guide for `NRPyLaTeX`.

Appendix A.1. PARSE_LATEX Function

`parse_latex()`

```
Parameters
----------
sentence : str
    input string (raw string preferred)
reset : bool, default=False
    reset current state by clearing namespace
verbose : bool, default=False
    verbose output and visible traceback
ignore_warning : bool, default=False
    ignore OverrideWarning
Returns
-------
tuple of str or sympy.core.expr.Expr
    tuple of each new variable (inserted into current stackframe) or
    single SymPy expression (sentence must also be an expression)
Raises
------
TensorError
    violation of the Einstein summation convention
ParseError
    invalid input string (related to SyntaxError)
Warns
-----
OverrideWarning
    a variable in the namespace is overridden
```

Figure A1: PARSE_LATEX Function Reference

The `parse_latex` function provides a frontend interface to `NRPyLaTeX`, accepting a LaTeX string, preferably a `Python` *raw string* to avoid unwanted character escaping,

and returning either a SymPy expression or a namespace dictionary containing every variable defined by the vardef command and/or a LaTeX equation. Note, NRPyLaTeX injects every instantiated variable into the local namespace (i.e. the local scope of the parse_latex function call) to avoid redundant variable assignment. Further, to eliminate constant redefining of common symbols/tensors, NRPyLaTeX automatically expands Levi-Civita symbols (of arbitrary rank), covariant derivatives, and Lie derivatives with their appropriate Einstein notation definitions using an operator inference system. In addition to parsing expressions/equations in Einstein notation, NRPyLaTeX also supports the LaTeX align environment for parsing multiple equations in a single parse_latex function call. NRPyLaTeX also supports comma/semicolon notation for specifying partial/covariant derivatives, respectively. Finally, NRPyLaTeX provides an IPython/Jupyter extension module nrpylatex.extension that includes a parse_latex (line and cell) magic command, which aliases to the %%latex cell magic for displaying and the parse_latex function for processing.

Appendix A.2. PARSE Command

parse

```
USAGE
    parse [OPTION] EQUATION
OPTION
    help
        display command usage
EQUATION
    single assignment (in LaTeX)
```

Figure A2: PARSE Command Usage

The parse command parses an equation without typesetting in a .tex document or a Jupyter Notebook. Typically, we use the parse command to perform a intermediate calculation that we deem necessary but trivial; e.g., a contraction to reduce rank.

Appendix A.3. IGNORE Command

ignore

```
USAGE
    ignore [OPTION] SUBSTRING, ...
OPTION
    help
        display command usage
SUBSTRING
    double-quoted string
```

Figure A3: IGNORE Command Usage

The `ignore` command ignores a LaTeX command or substring during parsing, so that one can sanitize LaTeX typesetting for input into `NRPyLaTeX`. Note, the following formatting commands are ignored by default: `\left`, `\right`, `{}`, and `&`. Furthermore, the `ignore` command and the empty replacement (`% srepl "..." -> ""`) are interchangeable. Finally, we remark that an equation cannot be split across a line break, and hence we suggest appending a percent symbol `%` to the end of a line break, creating a custom line break `\\%`, and then removing that substring using the ignore command.

Appendix A.4. VARDEF Command

vardef

```
USAGE
    vardef [OPERAND ...] [OPTION] VARIABLE, ... [DIMENSION]
OPERAND
    metric=METRIC
        desc: assign metric for index manipulation
        default: metric associated with diacritic (or lack thereof)
    weight=NUMBER
        desc: assign weight for Lie derivative generation
        default: 0
    diff_type=DIFF_TYPE
        desc: assign derivative type {symbolic | dD | dupD (upwind)}
        default: symbolic
    symmetry=SYMMETRY
        desc: assign (anti)symmetry
        default: nosym
        example(s):  sym01 -> [i][j] = [j][i], anti01 -> [i][j] = -[j][i]
OPTION
    const
        label variable type: constant
    kron
        label variable type: delta function
    metric
        label variable type: metric
    zero
        assign zero to each component
    help
        display command usage
VARIABLE
    single-quoted alphabetic string
    example(s): 'vU', 'gDD', 'alpha'
DIMENSION
    variable dimension (array length)
    default: (3D)
```

Figure A4: VARDEF Command Usage

The `vardef` command defines a variable, including optional assignment of a metric (for index raising/lowering), derivative type, tensor weight, and (anti)symmetry option. In addition to assigning attributes and properties, we can label the variable a scalar constant, Kronecker delta, or metric tensor. Note, the `metric` option symmetrizes the variable and generates the metric inverse, determinant, and connection. In `NRPyLaTeX`, every variable in the namespace has an associated metric to automatically perform index raising and lowering on that variable, with the default metric following from the

diacritic (or lack thereof) in the variable name (e.g., $v_i, \hat{v}_i, \bar{v}_i, \tilde{v}_i$). Note, NRPyLaTeX also supports diacritics on covariant derivatives (e.g., $D_i, \hat{D}_i, \ldots, \tilde{\nabla}_i$), which determines the metric used in the expansion of those covariant derivatives.

If we wanted to change the derivative type of a specific subexpression, we could use the vphantom command, a pseudo in-line LaTeX comment for NRPyLaTeX, to prefix that subexpression with a derivative type. However, a priority system need be established between the derivative type of a variable and that specified by a vphantom command. In general, vphantom takes precedence over each variable in the subexpression with the exception of a variable explicitly assigned a symbolic derivative type. Finally, if v and vU, for example, are both in the namespace and we attempt to parse the expression v^2, the output from NRPyLaTeX will be vU[2] (the third component of vU). To resolve that indexing ambiguity between vU[2] and v**2 (v squared), we suggest using the notation v^{{2}} since v^2 and v^{{2}} are typeset identically in LaTeX. Similarly, if v and vD are both in the namespace and we are parsing the symbol v_2, we suggest replacing v_2 with \text{v_2} using the srepl macro to construct a compound symbol in NRPyLaTeX and preserve the LaTeX typesetting of v_2.

Appendix A.5. KEYDEF Command

keydef

```
USAGE
    keydef OPERAND [OPTION] VARIABLE
OPTION
    help
        display command usage
OPERAND
    basis=BASIS
        desc: define basis (or coordinate system)
        example(s): [x, y, z], [r, \phi], default
    index=RANGE
        desc: override index range
        example(s): i (4D), [a-z] (2D)
VARIABLE
    single-quoted alphabetic string
    example(s): 'vU', 'gDD', 'alpha'
```

Figure A5: KEYDEF Command Usage

The keydef command overrides global properties of NRPyLaTeX, called **keywords**, e.g. the coordinate system or index ranges (for looping or summation). To enable symbolic differentiation without imposing a specific coordinate system, NRPyLaTeX implements a generalized coordinate system $\{x_0, \ldots, x_n\}$ capable of dynamic dimension adjustment, with a default dimension of three. However, a specific coordinate system can be imposed on NRPyLaTeX using the keydef command, allowing for coordinate indexing and symbolic differentiation with respect to a specific coordinate system. In addition to defining a coordinate system, the keydef command can also override the

default range of an index used in Einstein summation notation, making it useful for mixed dimension indexing.

Appendix A.6. ASSIGN Command

assign

```
USAGE
    assign [OPERAND ...] [OPTION] VARIABLE, ...
OPTION
    help
        display command usage
OPERAND
    metric=VARIABLE
        desc: assign metric for index manipulation
        default: metric associated with diacritic (or lack thereof)
    weight=NUMBER
        desc: assign weight for Lie derivative generation
        default: 0
    diff_type=DIFF_TYPE
        desc: assign derivative type {symbolic | dD | dupD (upwind)}
        default: symbolic
    symmetry=SYMMETRY
        desc: assign (anti)symmetry
        default: nosym
        example(s):  sym01 -> [i][j] = [j][i], anti01 -> [i][j] = -[j][i]
VARIABLE
    single-quoted alphabetic string
    example(s): 'vU', 'gDD', 'alpha'
```

Figure A6: ASSIGN Command Usage

The `assign` command assigns a `vardef` option to an already existing variable in the namespace, which can be useful for assigning attributes to variables created through equation parsing.

Appendix A.7. SREPL Command

srepl

```
USAGE
    srepl [OPTION] RULE, ...
OPTION
    persist
        apply rule(s) to every subsequent input of the parse() function (internal or external)
    help
        display command usage
RULE
    syntax: "..." -> "..."
    remark (1): whitespace ignored
    remark (2): substring can include a lexeme capture group
        syntax: <i> (single), <i..> (continuous) where i = 0, 1, 2, ...
```

Figure A7: SREPL Command Usage

The `srepl` command performs string replacement, ignoring whitespace, with lexeme capture group support, allowing for dynamic variable renaming, expansion/replacement of subexpressions, custom operator definitions, custom superscript/subscript definitions (e.g., `'`, `+`, or `-`), and much more. However, unlike their regular expression counterpart, `srepl` capture groups capture either a single lexeme or a sequence of lexemes[k]. If we use a continuous capture group, then `srepl` captures lexemes continuously until reaching the lexeme immediately following the capture group in the replacement pattern, e.g. `x^{<1..>}` captures everything inside of the curly braces. Finally, the `persist` option instructs `NRPyLaTeX` to perform (syntactic) string replacement on each `parse_latex` function call, including those functions calls internal to `NRPyLaTeX` that generate LaTeX.

References

[1] A. Meurer, C. P. Smith, M. Paprocki, O. Čertík, S. B. Kirpichev, M. Rocklin, A. Kumar, S. Ivanov, J. K. Moore, S. Singh, et al., *PeerJ Computer Science* **2017**, *3*, e103.

[2] Mathematica website (Wolfram Research, Inc.) `https://www.wolfram.com/mathematica`, **2021**.

[3] Maple website (Waterloo Maple, Inc.) `https://www.maplesoft.com/products/Maple/`, **2021**.

[4] J. M. Martín-García, xTensor website, `http://www.xact.es/xTensor/index.html`, **2021**.

[5] EinsteinPy Development Team, EinsteinPy: Python library for General Relativity, `https://einsteinpy.org/`, **2021**.

[6] GraviPy: Tensor Calculus Package for General Relativity based on SymPy, `https://github.com/wojciechczaja/GraviPy`, **2021**.

[7] I. Ruchlin, Z. B. Etienne, T. W. Baumgarte, *Phys. Rev. D* **2018**, *97*, 064036.

[8] NRPy+'s webpage, `http://nrpyplus.net/`.

[9] J. D. Brown, *Phys. Rev. D* **2009**, *79*, 104029:1–6.

[10] P. J. Montero, I. Cordero-Carrion, *Phys. Rev. D* **2012**, *85*, 124037.

[11] T. W. Baumgarte, P. J. Montero, I. Cordero-Carrion, E. Müller, *Phys. Rev. D* **2013**, *87*, 044026:1–14.

[12] T. W. Baumgarte, S. L. Shapiro, *Phys. Rev. D* **1999**, *59*, 024007:1–7.

[13] M. Shibata, T. Nakamura, *Phys. Rev. D* **1995**, *52*, 5428–5444.

[14] P. Marronetti, W. Tichy, B. Bruegmann, J. Gonzalez, U. Sperhake, *Phys. Rev. D* **2008**, *77*, 064010.

k Consider the following pattern: `\partial_t \phi = <1..> \\`. Now, suppose that instead of the `srepl` capture group `<1..>` we use a regex capture group `([^\\]+)`. In that case, the string replacement would fail for substrings containing backslashes (a common occurrence in LaTeX).