README

Dec 12, 2025

This project uses `uv` for Python environment management. It is a program that *does not depend on the local Python*. This means `uv` is the starting point of setting up a project, and it can install Python for the user, and manage both Python versions and packages.

1. Requirements

- The `uv` package manager should be installed following the install guide.
- The python version should already be specified in `pyproject.toml`, for example `requires-python = "~=3.11.0"`.
- Before running `uv sync`, there should be no `.venv` folder in the project directory. You can check this by running `ls -la`, and if you find an existing `.venv`, remove it with `rm -rf .venv`.
- The machine should be running MacOS, Linux x86-64, or Windows x86-64. Note that Linux arm-64 is not supported by `symbolica`.

2. Minimal Working Example

```
# Navigate to the project directory and install dependencies. The
# installation should start by installing python 3.11.x and then proceed
# to install symbolica.

cd extra_credit
uv sync

# Once installation completes, you can run the demonstration scripts to
# see the multipole expansion in action.

uv run run_demo.py
uv run test_higher_orders.py

# OPTIONAL: if on Linux or MacOS, run
./run_demo.sh # to save the terminal output to run_demo.txt
```

The first script demonstrates the basic multipole expansion up to $n = 3$rd order and verifies that the Taylor expansion and Q tensor formulations agree. In its output, there are also comments and references to relevant python scripts.

The second script pushes the implementation to higher orders, computing multipole moments up to $n = 7$.

3. Dive Into the Code

This project folder is organized as follows. The main package `multipole_expansion` contains the core implementation, while test scripts and documentation live at the top level.

```
extra_credit/
├── .venv/                  # Virtual environment (created by uv)
├── pyproject.toml          # Project configuration
├── multipole_expansion/    # Main package
│   ├── multipole_moments.py      # Q tensor construction
│   ├── taylor_expansion.py       # Taylor series approach
│   ├── derivatives.py            # Derivatives of 1/r
│   ├── contraction.py            # Einstein summation
│   └── ...
├── run_demo.py             # Basic demonstration
├── test_higher_orders.py   # High-order verification
└── ...
```

Although `uv run <file>` is the most convenient way to run python files, when viewing code and using the language server features of your code editor (such as "go to definition"), it is helpful to make your editor aware of the locally specified environment.

For this purpose, you can run `source .venv/bin/activate`. After that, the command-line prompt should look like `(multipole-expansion) → multipole_expansion`, and "go to definition" will help you jump between files.

3.1. The Main Script

The python module `multipole_expansion` is written inside the folder with the same name. The key python file is `multipole_expansion/multipole_moments.py`, while other python files provide utility functions, such as contraction for Einstein summation and derivatives for computing the derivatives of $1/r$.

Let us start from `multipole_moments.py` and explore the code base.

Go to line 138. The function `_Q_n_general` returns our desired symmetric traceless tensor. What it does is construct the main product term with coefficient $(2n-1)!!$ and then systematically subtract away the trace correction terms. Let us examine how these terms are constructed.

Go to line 169. The function `_compute_all_traces` computes all the trace correction terms that must be subtracted to make the tensor traceless. For each order n, we can have up to

$$\lfloor n/2 \rfloor \equiv \min\{m \mid m \in \mathbb{Z}, m \geq n/2\} \tag{3.1}$$

pairs of contracted indices, and this function systematically generates all such contractions by summing over

$$k \equiv \text{number of contracted index pairs.} \tag{3.2}$$

Each pairing is implemented as an array with length k,

```
pairing = [ (i_1,j_1) , ... , (i_k,j_k) ]
```

where each element is a tuple, and $\forall \ell \in [1,k], i_\ell < j_\ell$.

Denoting the set of pairings which involve k pairs as $\mathbb{P}_k$, we expand

$$\begin{aligned} \text{All traces} &= \sum_k \quad \text{traces with } k \text{ pairs} \\ &= \sum_k \sum_{\text{pairing} \in \mathbb{P}_k} \text{trace-term}(\text{ pairing }), \end{aligned} \tag{3.3}$$

where the function

$$\text{trace-term}(\bullet) : \text{pairing} \mapsto \delta_{i_1,j_1} \ldots \delta_{i_k,j_k} x^{i_1} \ldots x^{i_k} \times \text{prefactors} \tag{3.4}$$

is implemented after line 189. The function `_trace_with_k_pairs` handles the construction of trace terms with exactly k pairs of Kronecker deltas. For example, when $k = 1$, we get terms like

$$\delta^{ij} x_a^k x_a^l \tag{3.5}$$

multiplied by $|x_a|^2$ and the coefficient $(2n-3)!!$. When $k = 2$, we get two deltas, a factor of $|x_a|^4$, and coefficient $(2n-5)!!$. The coefficient formula coeff $= (2n-2k-1)!!$ is what makes the Taylor expansion and Q tensor formulations mathematically equivalent.

Now, the only remaining task is to construct the set of pairings $\mathbb{P}_k$ for arbitrary order n. This is implemented at line 234, function `_generate_k_pairings`, which is a *recursive algorithm* that

- picks *one* pair of indices from a list of indices,
- shortens the list and updates $k \mapsto k-1$,
- iterates (recursively runs) on the shortened list,
- maps the new indices to the old ones (those in the initial list of indices).

Let us illustrate the last step. For example, if $k = 2$ and we choose among 4 indices, then after the first iteration which picks `[1,2]` out, there will remain 2 indices,

```
remaining = [3,4]; remaining[1] = 3, remaining[2] = 4
```

and the iteration will output a pair `(1,2)` which actually means

```
(remaining[1],remaining[2]) = [3,4]
```

so we need to "map new indices to old ones". Once the condition

```
len(remaining) ⩾ 2 * (k - 1) # See `multipole_moments` line 264
```

is violated, the iteration will stop.

3.2. The Derivative Approach

In addition to the Q tensor construction, we also implement the multipole expansion using the Taylor series derivative approach. Go to `multipole_expansion/derivatives.py`, line 145. The function `_build_derivative_numerator` constructs the polynomial that appears in the numerator when we take the n-th derivative of $1/r$. The formula is

$$\partial^n(1/r) = ((-1)^n/r^{2n+1}) \times P^{i_1 \cdots i_n} \tag{3.6}$$

where P is a polynomial with the same coefficient structure as the Q tensor: $(2n-1)!!$ for the main term, $(2n-3)!!$ for single-pair traces, and so on.

This parallel structure between the derivative and the Q tensor is not a coincidence. It reflects the fundamental mathematical relationship between the two formulations of multipole expansion. When we compute $\phi^{(n)} = ((-1)^n/n!)x_a^{i_1 \cdots i_n}\partial^n(1/r)$ using the Taylor approach, the two $(-1)^n$ factors cancel, leaving us with $(1/n!)$ times a polynomial with the same coefficient structure as Q. After index contraction, both approaches yield identical results.

3.3. Symbolica Basics

Up to now, everything seems perfectly mathematical; we are just telling Python to carry out some mathematical operations. The "symbolic" infrastructure that implements the Kronecker deltas and $\vec{x}, \vec{n}$ vectors are just written as `self.delta`, `self.xa`, etc. But what exactly are these? How did we construct them?

Let us use the Einstein notation module `multipole_expansion/contraction.py` as an example. Go to line 128 of `contraction.py`. The function `_contract_delta` defines the behavior of the Kronecker delta. Because our calculations only involve the position vector x and the unit vector n, the Kronecker delta will be completely defined as long as it can contract with these two vectors. Besides, we also specify that the contraction of a Kronecker delta with itself is $\delta^{ii} = 1 \times 3 = 3$, which is the 3D trace.

In a similar manner, we define in this file other contraction rules, such as those of x with itself or with n. We then wrap all of these contractions into the function at line 55, `contract_indices`, which returns a fully contracted expression. The contraction engine applies pattern matching repeatedly until no more contractions are possible, which is why we sometimes need multiple passes for high-order multipole moments.

In `contraction.py`, you may have noticed the "wildcard" pattern in `symbolica` is similar to that in Wolfram Mathematica: `x_` is a pattern wildcard that matches any "single-atom" expression in the tree-representation. This pattern matching capability is what allows us to define Einstein summation rules declaratively rather than imperatively.

We have seen that symbolic computation boils down to a set of pattern matching and replacing rules, and that with a recursive algorithm, we can construct arbitrarily high-ordered terms from lowest-order ones. The derivative polynomial is built recursively by including all possible delta contractions. This recursive structure extends to arbitrarily high orders, limited only by computational resources rather than conceptual complexity.

3.4. Understanding the Coefficients

The coefficient pattern $(2n-1)!!, (2n-3)!!, (2n-5)!!, \ldots$ is central to the multipole expansion. The double factorial is defined as $n!! = n \times (n-2) \times (n-4) \times \ldots$, continuing down to either 2 or 1. For our purposes, we always work with odd arguments, so $(2n-1)!! = (2n-1) \times (2n-3) \times (2n-5) \times \ldots \times 3 \times 1$.

This sequence is strictly decreasing: $(2n-1)!! > (2n-3)!! > (2n-5)!! > \ldots > 1$. The leading coefficient, which multiplies the main product term $x_a^{i_1} \ldots x_a^{i_n}$, is always $(2n-1)!!$ and is therefore the largest. Each successive trace correction has a smaller coefficient, reflecting the fact that these are perturbative corrections to the main term. For example, at $n = 4$, we have $105 > 15 > 3$, and at $n = 7$, we have $135135 > 10395 > 945 > 105$.

The factor $(-1)^n$ appears in the derivative formula $\partial^n(1/r) = ((-1)^n/r^{2n+1}) \times P$, where P is the polynomial with coefficients $(2n-1)!!, (2n-3)!!$, etc. When we form the Taylor expansion

$$\phi^{(n)} = \frac{(-1)^n}{n!} x_a^{i_1} x_a^{i_2} \ldots x_a^{i_n} \, \partial^n(1/r), \tag{3.7}$$

the two $(-1)^n$ factors cancel, which is why the Q tensor formulation uses only the double factorial coefficients without any sign alternation in the main formula.

4. Troubleshooting

Symbolica is not a small package. Although it has pre-built binaries for the platforms it supports, download and install can take a minute or two. If Symbolica download upon running `uv sync` is slow, you can try installing with pip directly after activating the virtual environment.

```
source .venv/bin/activate
pip install --no-cache-dir symbolica
```

To verify the `symbolica` install, activate the virtual environment and test the import. If everything is working correctly, you should see a confirmation message.

```
source .venv/bin/activate
python -c "from symbolica import Expression; print('Symbolica OK')"
```

You can also run the full demo to see the multipole expansion in action, or run the higher-order tests to verify that the implementation handles arbitrary orders correctly.

```
python run_demo.py
python test_higher_orders.py
```