Constructing scientific programs using SymPy

Mark Dewing[‡*]

Abstract—We describe a method for constructing scientific programs where SymPy is used to model the mathematical steps in the derivation. With this workflow, each step in the process can be checked by machine, from the derivation of the equations to the generation of the source code. We present an example based on computing the partition function integrals in statistical mechanics.

Index Terms—SymPy, code generation, metaprogramming

Introduction

Writing correct scientific programs is a difficult, largely manual process. Steps in the process include deriving of the constituent equations, translating those into source code, and testing the result to ensure correctness. One challenging aspect of testing is untangling the cause of errors. For example, if the result appears incorrect, it is hard to determine whether the problem is with the algorithm, or a mistake was made in the derivation, or a simple transcription error in writing the source code. Confidence in the correctness of the program can be increased if these steps can be checked by computer.

The process of scientific programming also includes the pursuit of performance. Often the code needs to be heavily modified or rewritten to take best advantage of various target systems. Each modification introduces the possibility of further errors, and must be checked.

A standard approach to create a high-level description of the problem that is specialized to the particular domain, often through a Domain Specific Language (DSL), and then transform this to a representation in a general-purpose programming language. This is used in systems such as FEniCS [FEniCS] (representing PDE's) and the Tensor Contraction Engine [TCE] (representing matrix operations in quantum chemistry).

Since the specifications for scientific software are expressed largely as mathematical equations, our high-level description will be formed from a symbolic mathematics representation. A symbolic mathematics package is well-suited for this representation. In addition to this representation of the high-level description, we will model the derivation steps that lead to a computationally useful form.

The approach taken in this work is to operate on a symbolic representation of the scientific program, and then programmatically transform it into the target system. The specifications for scientific software are expressed largely as equations, and are ideally suited for a symbolic mathematics package. We use SymPy [SymPy], a symbolic mathematics package written in Python, for this part of the process.

The target system is likely a source code representation (C, Fortran, Python, etc), but could encompass more than that. For instance, C might be used to target the CPU or GPU, but the source code might look quite different in those cases. Or the user may call different libraries for the same function to compare performance.

* Corresponding author: markdewing@gmail.com
‡ Intel

Modeling Derivations

The first goal is to model on the computer a set of steps similar to those used when manually performing the derivation. Deriving the equations using in scientific software is similar to a proof where there is a series of logically justified steps connecting each expression until the final result is reached.

The OpenMathDocuments (OMDoc) project [OMDoc] is a representation for mathematics that describes a higher level than expressions in MathML. For instance, it has representations for proofs and lemmas. Similarly, for scientific computation we need to represent structures a higher level. One major difference between proofs and derivations in scientific software is that some steps are approximations.

The steps can be categorized as exact transformations, approximations, or specializations. An exact transform leaves the equality satisfied. Some types of exact transforms are rearranging terms, multiplying by factors, and identities (which operates only on one side of the equation). A specialization is specifying a physical or model parameter, such as the number of spatial dimensions, the number of particles, interaction potential, etc.

Finally, the results can be displayed in rendered mathematics (we use MathML or MathJax in web pages) to make the operation of each step and the results clearly visible.

Implementations

Modeling Derivations

For the implementation, the basic class named `derivation` has a constructor that takes an initial equation (lhs and rhs). The primary method is `add_step`, which takes an operation (or list of operations) to perform and a textual description of the operation(s). There are series of classes for various operations, such `approx_lhs`, which replaces the left-hand side of the equation with a new value. Also there is `add_term`, which adds the same term to both sides of the equation.

Code Generation

It is easy to start generating code by simply printing the statements of the target language. However, for greater generality we will use a model of the target language. Currently this work has (incomplete) language models for Python and C.

At the lowest level of transforming expressions, we developed a pattern-matching syntax that concisely captures some of the SymPy idioms.

The `Match` object matches a SymPy expression. The `__call__` method matches the first argument as the type of the expression. Subsequent arguments are variables to be bound to arguments. If the argument is a tuple, it is matched recursively on that argument. In this way the pattern for a tree structure can be built up concisely.

Variables that can match (for later binding) are members of an `AutoVar` class. This class creates member variables upon first access, and they are bound when the match succeeds.

Here is an example fragment of part of the SymPy to Python expression transformation, that matches addition, subtraction, and the reciprocal. SymPy normalizes subtraction as adding two expressions where the subtractand is multiplied by negative one. (That is, $a-b$ is represented as $-1*b+a$). Matching subtraction requires a nested pattern, which is shown here as well.

```
from sympy import Add, Mul, Pow, S
from derivation_modeling.codegen.lang_py import \
    py_expr, py_num
from derivation_modeling.codegen.pattern_match \
     import AutoVar, Match

class expr_to_py(object):
 def __call__(self, e):
   v = AutoVar()
   m = Match(e)

   # subtraction
   if m(Add, (Mul, S.NegativeOne, v.e1), v.e2):
     return py_expr(py_expr.PY_OP_MINUS,
        expr_to_py(v.e2), expr_to_py(v.e1))

   # addition
   if m(Add, v.e1, v.e2):
        return py_expr(py_expr.PY_OP_PLUS,
            expr_to_py(v.e1), expr_to_py(v.e2))

   # reciprocal
   if m(Pow, v.e2, S.NegativeOne):
        return py_expr(py_expr.PY_OP_DIVIDE,
            py_num(1.0), expr_to_py(v.e2))
```

Examples

Simple derivation

The Euler method is the simplest method for solving a differential equation. The steps involve a finite difference approximation to the derivative, rearranging terms, and the result is

$$f_1 = f_0 + h * 2 * x$$

The derivation is the following code:

```
from sympy import Function, Symbol, diff, sympify
from derivation_modeling import derivation, \
    approx_lhs, mul_factor, add_term

f = Function('f')
x = Symbol('x')
df = diff(f(x),x)
fd = sympify('(f_1 - f_0)/h')
d = derivation(df,2*x)

d.add_step(approx_lhs(fd),
    'Approximate derivative with finite difference')
d.add_step(mul_factor(h),'Multiply by h')
d.add_step(add_term(f0),'Move f_0 term to left side')
```

This can be output to MathML (or MathJax) for display in a web browser, which looks approximately like the following:

$$\frac{\partial}{\partial x}\mathrm{f}(x) = 2 * x$$

Approximate derivative with finite difference

$$\frac{f_1 - f_0}{h} = 2 * x$$

Multiply by h

$$f_1 - f_0 = 2 * xh$$

Move f_0 term to left side to get the final result

$$f_1 = f_0 + 2 * xh$$

Quadrature

For one of the simplest quadrature formulas, we use the trapezoidal rule [Trapezoid]. The derivation part consists of starting from the rule for single interval, and extending it to a series of intervals. (The rules for a single interval can be derived from interpolating polynomials, but we didn't start there)

The starting point for the derivation in Python is to define all the symbols, and the initial expression, then manipulate the expression so the function evaluation of each point is used only once.

```
from sympy import Symbol, Function, IndexedBase, Sum
from derivation_modeling import derivation, identity

i = Symbol('i',integer=True)
n = Symbol('n',integer=True)

I = Symbol('I')
f = Function('f')
h = Symbol('h')
x = IndexedBase('x')

# definitions of split_sum, adjust_limits,
#    peel_terms not shown
# split_sum - expand the sum of terms into a term of sums
# adjust_limits - adjust the expressions in the
#  summation variable.  This allows matching
#  the index used in the summand among different sums.
# peel_terms - move terms from the either end of the sum
#   to be an explicit term this allows the sum limits
#   to match and be combined.

trap = derivation(I, Sum(h/2*(f(x[i])+f(x[i+1])), (i,1,n)))
trap.add_step(identity(split_sum),'Split sum')
trap.add_step(identity(adjust_limits),'Adjust limits')
trap.add_step(identity(peel_terms),'Peel terms')
```

The LaTeX representation for the steps was copied from the generated output.

Start with a sum of single interval formulas

$$I = \sum_{i=1}^{n} \frac{1}{2} h \left(\mathrm{f}(x[i]) + \mathrm{f}(x[i+1])\right)$$

Split into two sums ('Split sum')

$$I = \sum_{i=1}^{n} \frac{1}{2} h \mathrm{f}(x[i]) + \sum_{i=1}^{n} \frac{1}{2} h \mathrm{f}(x[i+1])$$

Adjust the limits so the functions in the sum have compatible indices ('Adjust limits')

$$I = \sum_{i=0}^{n-1} \frac{1}{2} h \mathrm{f}(x[i]) + \sum_{i=1}^{n} \frac{1}{2} h \mathrm{f}(x[i])$$

Peel off some terms so the sum limits match, and combine the sums. ('Peel terms')

$$I = \frac{1}{2} h \mathrm{f}(x[0]) + \frac{1}{2} h \mathrm{f}(x[n]) + 2 \sum_{i=1}^{n-1} \frac{1}{2} h \mathrm{f}(x[i])$$

Now we have the final expression and can move to the transformation step. The approach to multiple dimensional integrals will be iterated one-dimensional integrals.

Partition Function

We start with the partition function from statistical mechanics [Partition]. It incorporates the interactions between particles (think of particles in a box), and contains all the thermodynamic information about a system. The dimension of the integral rises with the number of particles. The complexity for the convergence of grid-based methods is exponential in the number of dimensions, and they quickly become overwhelmed. The convergence of Monte Carlo methods is independent of dimension, and are commonly used to compute these integrals. However, it would be still be useful to use a grid method for a small number of particles as a way to check the Monte Carlo algorithms.

The derivation starts as follows:

```
partition_function =
   derivation(Z,Integral(exp(-V/(k*T)),R))
```

Where V is the inter-particle potential, T is the temperature, k is Boltzmann's constant, and Z is the symbol for the partition function. All of these are defined as SymPy `Symbol`.

Once again, the LaTeX has been copied from the output (although some steps have been combined for space)

$$Z = \int e^{-\frac{V}{Tk}} dR$$

It is conventional to work with the dimensionless inverse temperature, $\beta = kT$. Create the definition and insert into the integral.

```
beta_def = definition(Beta, 1/(k*T), T)
partition.function.add_step(
    replace_definition(beta_def),
    'Insert definition of beta')
```

The rendered output is

$$Z = \int e^{-V\beta} dR$$

To support multiple child derivations branching from a single parent, there is a method to support starting a new derivation from the final step of the previous one. Specialize to two particles - the `specialize_integral` transform replaces the integration variables, and the the `replace` transform replaces the specified variables (using a SymPy `subs`).

```
n2 = partitition_function.new_derivation()
n2.add_step(specialize_integral(R,(r1,r2)),
    'specialize to N=2')
n2.add_step(replace(V,V2(r1,r2)),
    'replace potential with N=2')
```

The rendered output is

$$Z = \int \int e^{-\beta \mathrm{V}(r_1,r_2)} dr_1 dr_2$$

Change variables and switch to a potential that depends only on the magnitude of the interparticle distance

```
r_cm = Vector('r_cm',dim=2)
r_12 = Vector('r_12',dim=2)

r_12_def = definition(r_12, r2-r1)
r_cm_def = definition(r_cm, (r1+r2)/2)

V12 = Function('V')

n2.add_step(specialize_integral(r1,(r_12,r_cm)),
    'Switch variables')
n2.add_step(replace(V2(r1,r2),V12(r_12)),
    'Specialize to a potential that depends only
     on interparticle distance')
n2.add_step(replace(V12(r_12),V12(Abs(r_12))),
    'Depend only on the magnitude of the distance')
```

The rendered output is

$$Z = \int \int e^{-\beta \mathrm{V}(|r_{12}|)} dr_{12} dr_{cm}$$

Integrate out the center of mass (or fixed coordinate) (This step could be performed by SymPy, but isn't right now)

```
Vol = Symbol('Omega')
n2.add_step(do_integral(Vol, [r_12]),
    'Integrate out r_cm (this step is still a hack)')
```

The rendered output is

$$Z = \Omega \int e^{-\beta \mathrm{V}(|r_{12}|)} dr_{12}$$

Decompose into vector components and specify limits. The `identity` transform modifies the right-hand side of the equation without changing its validity. The `decompose` operation takes an expression involving vectors and replaces it with the expression in terms of vector components. The `add_limits` transform adds upper and lower limits to the previously indefinite integral.

```
L = Symbol('L')
n2.add_step(identity(decompose),
    'Decompose into vector components')
n2.add_step(identity(add_limits(-L/2,L/2)),
    'Add integration limits')
```

The rendered output is

$$Z = \Omega \int_{-L/2}^{L/2} \int_{-L/2}^{L/2} e^{-\beta \mathrm{V}\left(\sqrt{r_{12x}^2 + r_{12y}^2}\right)} dr_{12x} dr_{12y}$$

Specialize to the Lennard-Jones potential

```
lj_expr = 4*(1/r**12 - 1/r**6)
lj_pot = derivation(V(r),lj_expr)
n2.add_step(replace_func(V12,lj_pot.final()),
    'Specialize to the LJ potential')
```

$$\mathrm{V}(r) = \frac{4}{r^{12}} - \frac{4}{r^6}$$

And get

$$Z = \Omega \int_{-\frac{1}{2}L}^{\frac{1}{2}L} \int_{-\frac{1}{2}L}^{\frac{1}{2}L} e^{-\beta \left(\frac{4}{\left(r_{12x}^2 + r_{12y}^2\right)^6} - \frac{4}{\left(r_{12x}^2 + r_{12y}^2\right)^3} \right)} dr_{12x} dr_{12y}$$

Insert numerical values for the box size and temperature.

```
L = 2.0
n2.add_step(replace('L',L),
    'Insert value for box size')
n2.add_step(replace('Omega',L*L),
    'Insert value for box volume')
n2.add_step(replace('beta',1.0),
    'Insert value for temperature')
```

$$Z = 4.0 \int_{-1.0}^{1} \int_{-1.0}^{1} e^{-4.0 \frac{1}{\left(r_{12x}^2 + r_{12y}^2\right)^6} + 4.0 \frac{1}{\left(r_{12x}^2 + r_{12y}^2\right)^3}} dr_{12x} dr_{12y}$$

Now we have an integral that is completely specified numerically[1]. It can be evaluated by an existing quadrature routine in SymPy, by another another package (`scipy.quadrature.dblquad`), or by the trapezoidal rule code we derived earlier.

Code Generation

As an example of the language model, the classic 'Hello World' program in python is

```
from derivation_modeling.codegen.lang_py import
    py_expr, py_expr_stmt, py_function_call, \
    py_function_def, py_if, py_print_stmt, \
    py_stmt_block, py_string, \
    py_var


body = py_stmt_block()

hello_func = py_function_def('hello')
hello_func.add_statement(
   py_print_stmt(py_string("Hello, World")))
body.add_statements(hello_func)
main = py_if(
    py_expr(py_expr.PY_OP_EQUAL,
        py_var('__name__'), py_string('__main__')))
main.add_true_statement(
   py_expr_stmt(py_function_call('hello')))
body.add_statements(main)

f = open('hello_py.py','w')
f.write(body.to_string())
f.close()
```

The generated output is

```
def hello():
   print "Hello, World"
if __name__ == "__main__":
   hello()
```

For C, the program is

```
from derivation_modeling.codegen.lang_c import
   c_block, c_function_call, c_function_def, \
   c_func_type, c_int, c_num, c_return, c_stmt, \
   c_string, pp_include

body = c_block()
body.add_statement(pp_include('stdio.h'))
main_body = c_block()

main = c_function_def(
  c_func_type(c_int('main')), main_body)

main_body.add_statement(
  c_stmt(c_function_call("printf",
           c_string("Hello, World\\n"))))

main_body.add_statement(c_return(c_num(0)))
body.add_statement(main)

f = open('hello_c.c','w')
f.write(body.to_string())
f.close()
```

The generated program is

```
#include <stdio.h>
int main(){
  printf("Hello, World\n");
  return 0;
}
```

The code and examples described here can be found in the author's `derivation_modeling` repository on GitHub:
https://github.com/markdewing/derivation_modeling

Discussion

The example derivations presented here are fairly simple and linear. In reality, the connections are more complex. For instance, one is often interested in multiple properties (energy, pressure, distribution functions) that may branch off the original derivation or have a separate thread of steps, but eventually, for efficiency they should all be evaluated in the same integral.

The pattern-matching style makes the lower levels of expression translation fairly clear, but the the translations at the next level up (combining the source code statements) is not very transparent yet. An important future step is enhancing debugging by making the connections between the code generator and the generated code clearer.

Other Work

For solving partial differential equations, there is FEniCS [FEniCS] project and the SAGA (Scientific computing with Algebraic and Generative Abstractions) project [SAGA] .

Ignition [Ignition],[Terrel11]_ is a library that provides support for writing and combining DSL's for describing problems (or aspects of problems)

Part of this work is modeling the target language for code generation. Several other projects for modeling programming languages include Pivot [Pivot], a project for modeling C++. CodeBoost [CodeBoost] is the code transformation portion of the SAGA system. PyCUDA [PyCUDA] is a potential target system, and it also has an associated model of C and CUDA for generation of code [CodePy]

Conclusions

We presented a snapshot of some work on some software blocks necessary for a system of scientific computing, including modeling a derivation, transforming to a source code representation, and code generation.

References

[CodeBoost] http://codeboost.org/
[CodePy] http://mathema.tician.de/software/codepy
[FEniCS] http://www.fenicsproject.org
[Ignition] http://andy.terrel.us/ignition/
[OMDoc] http://www.omdoc.org
[Partition] http://en.wikipedia.org/wiki/Partition_function_%28statistical_mechanics%29
[Pivot] http://parasol.tamu.edu/pivot/
[PyCUDA] http://mathema.tician.de/software/pycuda
[TCE] Tensor Contraction Engine http://www.csc.lsu.edu/~gb/TCE/
[Terrel11] A. Terrel. *From Equations to Code: Automated Scientific Computing* Computing in Science and Engineering 13(2):78-982, March 2011
[Trapezoid] See http://en.wikipedia.org/wiki/Trapezoidal_rule or any numerical analysis textbook
[SAGA] http://www.ii.uib.no/saga/
[SymPy] http://sympy.org/

1. There is a division-by-zero error at $r = 0$ that must be avoided, either by offsetting one limit slightly, or by capping the potential for small r. This latter step has not been added to the definition of the potential yet.