EML4930/EML6934: Lecture 10

Symbolic math with SymPy, latin-hypercube DOE with pyDOE

Charles Jekel

November 2, 2017

Symbolic math with SymPy

SymPy is a Python library for symbolic mathematics. It aims to become a full-featured computer algebra system (CAS) while keeping the code as simple as possible in order to be comprehensible and easily extensible. SymPy is written entirely in Python.

`http://www.sympy.org`

Documentation: `http://docs.sympy.org/latest/index.html`

Tutorial: `http://docs.sympy.org/latest/tutorial/index.html`

What's in SymPy

Way more than what we can cover in a single lecture

- Arbitrary precision
- Integration, integral transforms
- Equations, inequalities, Diophantine equations, differential equations, Recurance relations
- Number theory
- Boolean algebra
- Tensors
- Probability
- Group theory

What's missing from SymPy

- Graph theory
- Quantifier elimination
- Control theory
- Coding theory

For a full comparison of every computer algebra system visit this wiki `https://en.wikipedia.org/wiki/List_of_computer_algebra_systems`

SymPy is a Python library

- SymPy is a pure Python library
- If you can't do it in Python, then you can't do it in SymPy
- SymPy does not change, modify, or add to the Python language
- SymPy will be easy and intuitive if you know Python

Symbolic expression one variable at a time

```
from sympy import *

# create symbolic variables x, y, z
x = Symbol('x')
y = Symbol('y')
z = Symbol('z')
```

It's recommended to assign the variable name to the symbol of the same.

You can assign multiply symbolic variables at once

```
# create symbolic variables w, x, y, z
w, x, y, z = symbols('w x y z')
```

There are a few things to note:

- use

  `Symbol()`

  to create a single symbolic variable
- use

  `symbols()`

  to create multiple symbolic variables where each variable is separated by a space
- all items are case-sensitive
- Symbol() is different from symbol()

Symbolic expressions

```
a = 3.0*x**2 - 2.0*x + 1.0

# we can do math on symbolic expressions
a = a*x +1

# simplify a
a = a.simplify()
# equivalent to a = simplify(a)
```

Symbolic equalities

```
# Eq is an equality object
# this is how to express x+1 = 4 in SymPy
b = Eq(x+1,4)
```

Are two symbolic representations equal?

It's impossible to show that two symbolic expressions are equal in general
https://en.wikipedia.org/wiki/Richardson%27s_theorem

```
a = cos(x)**2 - sin(x)**2
b = cos(2*x)

# to see if a == b we could do
print(simplify(a-b))

# or we can use equals()
print(a.equals(b))

# these methods are not perfect!
```

Rationals in SymPy

```
# let's express 1/3 as a rational
a = Rational(1,3)
b = Integer(1)/Integer(3)
```

Substitution

```
a = x*y**2 + y*x**2

# to evaluate at x = 1, y = 2, use dictionary
a.subs({x:1, y:2})

# to substitute y = x^2
a.subs(y, x**2)

# note that a is immutable! .subs() does not change a!
```

Printing symbolic math in SymPy

There are several printers available in SymPy

- str
- stepr
- ASCII pretty printer
- Uncode pretty printer
- LaTeX
- MathML
- Dot

initialize printing

```
init_printing()
# that's it to start the pretty printing process
```

If you are in a qtconsole

- and have LaTeXinstalled, the printer will use LaTeXto create images of the expressions
- and don't have LaTeXinstalled, the printer will use Matplotlib's MathJax to render LaTeX

If you are in a normal Python session

- unicode printer
- ASCII printer

Printing options and pprint

You can set LaTeXand unicode to false if you just want to use ASCII printing

```
init_printing(use_latex=False, use_unicode=False)
```

on my Windows System (in a normal command prompt) the unicode detector isn't working...

Additionally there is a unicode pretty printer

```
pprint( x**2+3)
```

If you'd rather use ASCII then you'll have to specify

```
pprint(x**2+3, use_unicode=False)
```

Getting the LaTeX from an expression is easy

```
a = latex(x**2*y + x*y**2)
# a is a latex string
print(a)
```

Simplification functions

Function	Description
simplify()	Attempt to intelligently Express the simplest function
expand()	Expanding polynomial expressions
factor()	Factors a polynomial into irreducible factors over rationals
collect()	Collects common powers of a term in an expression
cancel()	Take a rational and put into standard form (no common factor
apart()	Partial fraction decomposition
trigsimp()	Simplify expression using trigonometric identities
expand_trig()	i.e. apply sum or double angle identities

Exponential and log

```
a = ln(x)
# same as a = log(x)

b = exp(x)

# if needed you can specify that x, y are strictly positive with
x, y = symbols('x y', positive=True)

# alternatively you can specify that n is a real number
n = symbols('n', real=True)
```

Special functions

```
x, y, z = symbols('x y z')
k, m, n = symbols('k m n')

# factorial
factorial(x)

# binomial coefficient
binom(n, k)

# gamma function
gamma(z)

# generalized hypergoemetric function
hyper([1,2], [3], z)
```

Let's just take a moment to appreciate Python and SymPy

There is a lot that SymPy has to offer, more so that I can go over today. If you want to learn everything. Work through the (very long Tutorial – it may take you a few hours...)
`http://docs.sympy.org/latest/tutorial/`

Remainder of what I'm going to cover:

- Calculus (derivatives, integrals, limits)
- Solvers
- Matrix operations

derivatives

```
# to take derivatives use the diff function

y = x**3 + 2.0*x**2 - 4.0*x - 10.0

# take the derivative with respect to x (dy/dx)
diff(y,x)

# take the second derivative with respect to x  (d**2y/dx**2)
diff(y,x,x)

# take the third derivative with respect to x (d**3y/dx**3)
diff(y,x,x,x)
```

derivatives - object oriented approach...

```
# to take derivatives use the diff function

y = x**3 + 2.0*x**2 - 4.0*x - 10.0

# take the derivative with respect to x (dy/dx)
y.diff(x)

# take the second derivative with respect to x  (d**2y/dx**2)
y.diff(x,x)

# take the third derivative with respect to x (d**3y/dx**3)
y.diff(x,x,x)

# note y is not changed when you do this (y is immutable)
```

Integrals - indefinite

```
y = cos(x)

# take the integral of y with respect to x
integrate(y,x)

# or
y.integrate(x)

# note this does not include constant!!!
# you must add constants manually!
```

Integrals - definite

```
y = 1 /  x**3

# take the integral of y with respect to x from 1 to infinity (o
# (integration_variable, lower_limit, upper_limit)
integrate(y, (x, 1, oo))
y.integrate((x, 1, oo))

# these two lines are the same!
```

Limits

```
y = 1 /  x**3

# take the limit of y as x goes to infinity
limit(y, x, oo)

# take the limit of y as x goes to zero from the positive
limit(y, x, o, '+')

# take the limit of y as x goes to zero from the negative
limit(y, x, o, '-')
```

Solve an equation

```
y = x**2 - 2.1*x + 4.0

# solve for when y = 0
solve(y,x)

# solve for when y = 4
solve(Eq(y,4),x)
```

There is more! Read
`http://docs.sympy.org/latest/tutorial/solvers.html`

- linear system solvers
- non-linear system solvers
- differential equation solvers

Matrices in SymPy

```
# NOTE: Matrix are the only mutable SymPy object
M = Matrix([[1, 3], [-2, 3]])
N = Matrix([[0, 3], [0, 7]])
# add two matrices
M + N
# Matrix multiplication
M*N

# Multiply a matrix
3*M

# same as M*M
M**2

# invert a Matrix
M**-1
```

Functions for matrix manipulation

Function	Description
rref()	Reduce row echelon form
nullspace()	Find the nullspace of a matrix
eigenvals()	Find the eigenvalues of a matrix
eigtenvects()	Find the eigenvectors of a matrix
det()	Compute the determinant of matrix

You can use these functions as methods of the Matrix object as well...

End of SymPy

SymPy is very powerful!

Hopefully this will be useful.

Design of experiments (DOE) with pyDOE

The pyDOE package is designed to help the scientist, engineer, statistician, etc., to construct appropriate experimental designs.

`https://pythonhosted.org/pyDOE/`

Capabilities

- General Full-Factorial (fullfact)
- 2-Level Full-Factorial (ff2n)
- 2-Level Fractional-Factorial (fracfact)
- Plackett-Burman (pbdesign)
- Box-Behnken (bbdesign)
- Central-Composite (ccdesign)
- Latin-Hypercube (lhs)

latin-hypercube random sampling

lhs(n, samples=10, criterion='c', iterations=5)

Parameters

n : int The number of factors to generate samples for (consider this the number of design variables)

Optional

samples : int

The number of samples to generate for each factor (Default: n)

criterion : str

Allowable values are "center" or "c", "maximin" or "m", "centermaximin" or "cm", and "correlation" or "corr". If no value given, the design is simply randomized.

iterations : int

The number of iterations in the maximin and correlations algorithms (Default: 5).

lhs use

```
from pyDOE import lhs
import matplotlib.pyplot as plt
lhd = lhs(2, samples=5, criterion='cm')
# my design variables are available as
x_1 = lhd[:,0]
x_2 = lhd[:,1]

# NOTE: the outuput of your variables will always be [0,1]
# so you'll need to scale your variables accordingly

# scatter plot of the design over the domain
plt.figure()
plt.plot(x_1,x_2, 'o')
plt.grid(True)
plt.xlim((0,1))
plt.ylim((0,1))
plt.show()
```