

# Calculus and Diffy Q's

Ádám Gyenge

## Basic Differentiation

SymPy can perform differentiation of mathematical expressions symbolically. This is useful for obtaining exact representations of derivatives.

```
# Define symbols and an expression
x, y = sp.symbols('x y')
expression = x**2 + sp.sin(x*y)
```

```
# Compute the partial derivative with respect to x
derivative_x = sp.diff(expression, x)
```

```
# Compute the partial derivative with respect to y
derivative_y = sp.diff(expression, y)
```

## Basic Differentiation Result

The partial derivative with respect to  $x$  is  $2x + y$ , and the partial derivative with respect to  $y$  is  $x \cos(xy)$ .

```
>>> derivative_x  
2*x + y*cos(x*y)
```

```
>>> derivative_y  
x*cos(x*y)
```

# Higher-Order Differentiation

For higher-order derivatives, you can either pass the variable multiple times or use the number of derivatives. Example:

```
>>> sp.diff(x**5, x, x)
20*x**3
```

```
>>> sp.diff(x**5, x, 2)
20*x**3
```

# Mixed Partial Derivatives

SymPy also computes mixed partial derivatives.

```
>>> x, y, z = sp.symbols('x y z')
```

```
>>> sp.diff(sp.exp(x*y*z), x, y, y, z, z)
```

```
x*(x**3*y**3*z**3 + 8*x**2*y**2*z**2 + 14*x*y*z + 4)  
*exp(x*y*z)
```

# Symmetry of second derivatives

## Theorem (Young's theorem)

*Exchanging the order of partial derivatives of a twice-differentiable multivariate function*

$$f(x_1, x_2, \dots, x_n)$$

*does not change the result. That is, the second-order partial derivatives satisfy the identities*

$$\frac{\partial}{\partial x_i} \left( \frac{\partial f}{\partial x_j} \right) = \frac{\partial}{\partial x_j} \left( \frac{\partial f}{\partial x_i} \right).$$

# Implicit functions

- ▶ An *implicit function* is defined by an equation of the form  $F(x, y) = 0$ , where  $F$  is a function of two variables, and  $y$  is implicitly related to  $x$ .
- ▶ Unlike explicit functions, where  $y$  is expressed directly in terms of  $x$ , implicit functions require solving the equation  $F(x, y) = 0$  to determine  $y$  as a function of  $x$ .
- ▶ In many cases, it is challenging or impossible to express  $y$  explicitly, yet the implicit function theorem guarantees the existence of  $y$  as a differentiable function of  $x$  under certain conditions.

# Implicit Differentiation

To differentiate an implicit function, we apply implicit differentiation. Taking the total derivative of  $F(x, y)$  with respect to  $x$ , we use the chain rule:

$$\frac{\partial F}{\partial x} + \frac{\partial F}{\partial y} \frac{dy}{dx} = 0.$$

Rearranging, we solve for  $\frac{dy}{dx}$ :

$$\frac{dy}{dx} = -\frac{\frac{\partial F}{\partial x}}{\frac{\partial F}{\partial y}},$$

provided that  $\frac{\partial F}{\partial y} \neq 0$ . This allows us to compute the derivative without explicitly solving for  $y$ .

# Implicit Differentiation

To differentiate implicit functions, use SymPy's `idiff()` function.  
Example:

```
# Define an implicit equation
implicit_eq = x**5 + y**2 + z**4 - 8*x*y*z

# Differentiate implicitly
implicit_derivative = sp.idiff(implicit_eq, y, x)
```

# Implicit Differentiation Result

The derivative of  $y$  with respect to  $x$  is:

```
>>> implicit_derivative  
(5*x**4/2 - 4*y*z)/(4*x*z - y)
```

```
>>> implicit_derivative.simplify()  
(5*x**4 - 8*y*z)/(2*(4*x*z - y))
```

## Unevaluated Derivatives

You can also create unevaluated derivatives using the Derivative class. Example:

```
# Create an unevaluated derivative
deriv = sp.Derivative(sp.exp(x*y*z), x, y, 2, z, 2)
```

Result:

```
>>> deriv
Derivative(exp(x*y*z), x, (y, 2), (z, 2))
```

We get nicer result if we initialize pretty printing:

```
[>>> sp.init_printing()
[>>> deriv
      5
      ∂      ( x·y·z )
      ----- e
      2      2
∂z  ∂y  ∂x
```

# Unevaluated Derivative

- ▶ Unevaluated derivatives can be useful when we want to perform complex operations.
- ▶ To evaluate the derivative, use the `doit()` function:

```
>>> deriv.doit()  
x*(x**3*y**3*z**3 + 8*x**2*y**2*z**2 + 14*x*y*z + 4)  
  *exp(x*y*z)
```

# Antiderivatives

SymPy supports symbolic integration. Here's an example of computing an indefinite integral:

```
# Define a simple expression
expression = x**2 + sp.sin(x)

# Compute the indefinite integral
indefinite_integral = sp.integrate(expression, x)
```

Result:

```
>>> indefinite_integral
x**3/3 - cos(x)
```

## Definite Integration

For definite integrals, you can specify the limits:

```
# Define the limits of integration
```

```
lower_limit = 0
```

```
upper_limit = sp.pi
```

```
# Compute the definite integral
```

```
definite_integral =
```

```
    sp.integrate(expression, (x, lower_limit, upper_limit))
```

The result of the definite integral from 0 to  $\pi$  is:

```
>>> definite_integral
```

```
2 + pi**3/3
```

# Improper Integrals

To compute improper integrals, use `oo` to denote infinity:

```
# Define an improper integral
improper_integral = sp.integrate(sp.exp(-2*x),
    (x, 0, sp.oo))
```

The result of the improper integral is:

```
>>> improper_integral
1/2
```

# Integration of Special Functions

SymPy supports the integration of functions like  $\sin(x)$  and  $e^x$ :

```
# Define an expression with special functions
```

```
special_expression = sp.sin(x) * sp.exp(x)
```

```
# Compute the integral
```

```
integral_special = sp.integrate(special_expression, x)
```

The integral of  $\sin(x) \cdot e^x$  is:

```
>>> integral_special
```

```
exp(x)*sin(x)/2 - exp(x)*cos(x)/2
```

## Multiple integrals

To compute

$$\int_{-\infty}^{\infty} \int_{-\infty}^{\infty} e^{-x^2-y^2} dx dy$$

one can type the following:

```
# Define a multiple integral
multiple_integral = integrate(sp.exp(-x**2 - y**2),
                              (x, -sp.oo, sp.oo), (y, -sp.oo, sp.oo))
```

The result will be  $\pi$ .

```
>>> multiple_integral
pi
```

# Unevaluated Integrals

As with derivatives, we can create unevaluated integrals:

```
# Define unevaluated integral  
expr = sp.Integral(x**2, x)
```

This gives:

```
>>> expr  
Integral(x**2, x)
```

```
>>> expr.doit()  
x**3/3
```

## Limit Computations

To compute limits, use the `limit()` function. Example:

```
# Define the variable and expression
```

```
expression = sp.sin(x) / x
```

```
# Compute the limit
```

```
limit_result = sp.limit(expression, x, 0)
```

The limit as  $x \rightarrow 0$  of  $\frac{\sin(x)}{x}$  is:

```
>>> limit_result
```

```
1
```

## One-Sided Limits

SymPy can compute one-sided limits using the `dir` parameter.

Example:

```
# Define the piecewise function
f = sp.Piecewise((1, x > 0), (-1, x < 0))

# Compute one-sided limits
limit_right = sp.limit(f, x, 0, dir='+')
limit_left = sp.limit(f, x, 0, dir='-')
```

The results for the right and left-hand limits are:

```
>>> limit_right
1

>>> limit_left
-1
```

## Multivariate Limits

SymPy can compute multivariate limits. Example:

```
# Define variables and the expression
```

```
x, y = sp.symbols('x y')
```

```
expression = (x**2 + y**2) / (x**2 + 2*y**2)
```

```
# Compute the multivariate limit
```

```
multivariate_limit = sp.limit(expression, x, 0, y, 0)
```

The result of the multivariate limit is:

```
>>> multivariate_limit
```

```
1/2
```

# Taylor Series Expansion

SymPy can compute Taylor series expansions. Example:

```
# Define the variable and expression  
expression = sp.sin(x)
```

```
# Compute the series expansion  
taylor_series = sp.series(expression, x, 0, 6)
```

Here we take the expansion around 0 up to order 6.

Result:

```
>>> taylor_series  
x - x**3/6 + x**5/120 + O(x**6)
```

## Laurent Series

Laurent series expansions can also be computed. Example:

```
# Define the expression  
expression = 1 / (x - 1)
```

```
# Compute the Laurent series  
laurent_series = sp.series(expression, x, 0, 5)
```

The result of the Laurent series expansion is:

```
>>> laurent_series  
-1/x + 0(x)
```

## Asymptotic expansion

Asymptotic expansion is possible at infinity.

For instance, to compute the asymptotic expansion of  $\ln(1 + x)$  as  $x \rightarrow \infty$ :

```
# Define the expression
expression = sp.log(1 + x)

# Compute the asymptotic expansion
asymptotic_series = sp.series(expression, x, sp.oo, 3)
```

The result is:

```
>>> asymptotic_series
log(x) + 1/x - 1/(2*x**2) + O(1/x**3)
```

## Order terms and truncation

- ▶ Series expansions in SymPy include an  $O()$  term, which represents higher-order terms.
- ▶ The `removeO()` method can be used to truncate the series by removing the  $O()$  term.
- ▶ For example:

```
# Remove the order term
truncated_series = taylor_series.removeO()
```

The result is:

```
>>> truncated_series
x - x**3/6 + x**5/120
```

# Defining Differential Equations

## Symbolic Representation

- ▶ Use Eq objects to define equations.
- ▶ Declare variables and functions beforehand.

```
import sympy as sp
```

```
x = sp.symbols('x')
```

```
y = sp.Function('y')
```

```
eq = sp.Eq(sp.Derivative(y(x), x, x) + y(x), 0)
```

Example:  $y'' + y = 0$

```
>>> eq
```

```
Eq(y(x) + Derivative(y(x), (x, 2)), 0)
```

# Functions in SymPy

## Using Function class

- ▶ Functions of one or more variables are represented using the class `Function`.

```
y = sp.symbols('y', cls=sp.Function)
```

- ▶ Functions allow symbolic differentiation and equation manipulation.

## Importance of Declaring Dependencies

- ▶  $y(x)$  must be declared as a function of  $x$  for:
  - ▶ Derivatives
  - ▶ Symbolic manipulations

# Solving Equations

## General Solutions with `dsolve()`

- ▶ Use `dsolve()` to solve differential equations symbolically.

```
solution = sp.dsolve(eq, y(x))
```

Solution for  $y'' + y = 0$ :

```
>>> solution
```

```
Eq(y(x), C1*cos(x) + C2*sin(x))
```

## Extracting and Manipulating Solutions

- ▶ Extract LHS and RHS:

```
>>> solution.lhs  
y(x)
```

```
>>> solution.rhs  
C1*sin(x) + C2*cos(x)
```

- ▶ Substitute constants:

```
>>> solution.rhs.subs({'C1': 1, 'C2': 2})  
sin(x) + 2*cos(x)
```

# Verifying Solutions

**Using** `checkodesol()`

- ▶ Verify that a solution satisfies the original equation:

```
check_result = sp.checkodesol(eq, solution)
```

Output:

```
>>> check_result  
(True, 0)
```

# Initial Conditions and Boundary Value Problems

## Solving with Initial Conditions

- ▶ Example: Solve  $y'' + y = 0$  with:
  - ▶  $y(0) = 1$
  - ▶  $y'(0) = 0$

```
solution_ic = sp.dsolve(eq, y(x), ics={y(0): 1,  
    y(x).diff(x).subs(x, 0): 0})
```

Result:

```
>>> solution_ic  
Eq(y(x), cos(x))
```

## Plotting Solutions

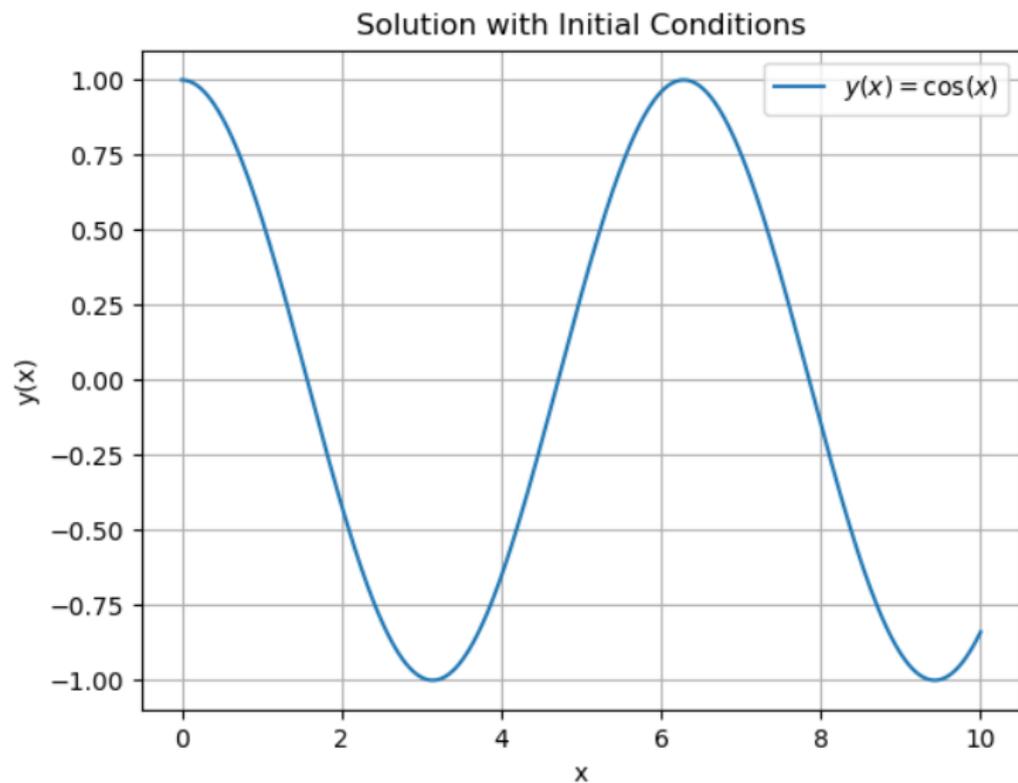
We can then plot using `lambdify`:

```
import numpy as np
import matplotlib.pyplot as plt

x_vals = np.linspace(0, 10, 500)
sol_func_ic = sp.lambdify(x, solution_ic.rhs, 'numpy')
y_vals = sol_func_ic(x_vals)

plt.plot(x_vals, y_vals, label='$y(x) = \cos(x)$')
plt.xlabel('x')
plt.ylabel('y(x)')
plt.title('Solution with Initial Conditions')
plt.legend()
plt.grid()
plt.show()
```

# Plotting Solutions



# Systems of ODEs

## Solving Coupled Equations

- ▶ Example System:

$$y_1' = y_2, \quad y_2' = -y_1$$

```
y1, y2 = sp.symbols('y1 y2', cls=sp.Function)
eq1 = sp.Eq(y1(x).diff(x), y2(x))
eq2 = sp.Eq(y2(x).diff(x), -y1(x))
solution_system = sp.dsolve([eq1, eq2])
```

Solution:

```
>>> solution_system
[Eq(y1(x), C1*sin(x) + C2*cos(x)),
 Eq(y2(x), C1*cos(x) - C2*sin(x))]
```

# Partial Differential Equations (PDEs)

## Defining PDEs

- ▶ Example: First-order linear PDE:

$$2\frac{\partial u}{\partial x} + 3\frac{\partial u}{\partial y} = 5$$

```
x, y = sp.symbols('x y')
u = sp.Function('u')(x, y)
pde = sp.Eq(2 * sp.Derivative(u, x)
            + 3 * sp.Derivative(u, y), 5)
```

# Partial Differential Equations (PDEs)

## Solving PDEs

- ▶ Use `pdsolve()`:

```
solution = sp.pdsolve(pde)
```

General solution:

```
>>> solution  
Eq(u(x, y), 10*x/13 + 15*y/13 + F(3*x - 2*y))
```

Here  $F$  is an unknown function.

## Adding Initial Conditions to PDEs

- ▶ Example: Given  $u(x, 0) = x^2$ :

```
u0 = x**2
```

```
specific_condition = solution.subs(y, 0).rhs - u0
```

```
F_general = sp.simplify(specific_condition)
```

Particular solution:

```
>>> F_general
```

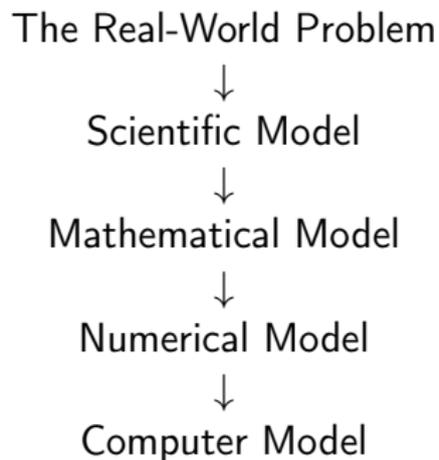
```
-x**2 + 10*x/13 + F(3*x)
```

The meaning of this is that

$$-x^2 + \frac{10x}{13} + F(3x) = 0.$$

We can then substitute  $F$  back into the general solution to obtain the particular solution.

# The Modeling Process



## Example: The Period of a Pendulum

**Problem:** Determine the period of oscillation of a pendulum.

**Scientific (Physical) Model Assumptions:**

- ▶ Point mass
- ▶ Massless string of length  $l$
- ▶ No friction or air resistance
- ▶ Gravitational acceleration  $g$

Using conservation of energy:

$$\frac{1}{2}ml^2(\varphi'(t))^2 + mgl(1 - \cos \varphi(t)) = mgl(1 - \cos \alpha)$$

# From Physics to Mathematics

After rearranging:

$$\varphi'(t) = -\sqrt{\frac{2g}{l}} \sqrt{\cos \varphi(t) - \cos \alpha}$$

To compute the period  $T$ :

$$T = 4\sqrt{\frac{l}{g}} \int_0^{\pi/2} \frac{1}{\sqrt{1 - \sin^2(\alpha/2) \sin^2 \vartheta}} d\vartheta$$

**This is the mathematical model.**

- ▶ The integral is an elliptic integral.
- ▶ No closed elementary form exists.

# Numerical and Computer Models

## Numerical Model:

- ▶ Approximate the elliptic integral using numerical integration.

## Computer Model:

- ▶ Perform computations on a computer.

Example ( $l = 1\text{m}$ ,  $g = 9.8\text{m/s}^2$ ):

- ▶  $\alpha = 5^\circ \Rightarrow T \approx 2.0080\text{s}$
- ▶  $\alpha = 90^\circ \Rightarrow T \approx 2.3690\text{s}$

# Small-Angle Approximation

Expand:

$$\frac{1}{\sqrt{1-x}} \quad \text{around } x = 0$$

This yields:

$$T = 2\pi\sqrt{\frac{l}{g}} \left( 1 + \frac{1}{4} \sin^2 \frac{\alpha}{2} + \dots \right)$$

For small  $\alpha$ :

$$T \approx 2\pi\sqrt{\frac{l}{g}}$$

Example:

$$T \approx 2.0071\text{s (for } \alpha = 5^\circ)$$

Very good approximation!

# Sources of Error in Modeling

Errors are unavoidable in modeling.

- ▶ Our goal is to:
  - ▶ Estimate errors
  - ▶ Control errors
- ▶ If results disagree with reality:
  - ▶ Refine the model.

**Important:** The presence of error is not always bad. Sometimes simplification makes exact solutions possible.

# Main Sources of Error

- ▶ **Model Error**

- ▶ Scientific model does not perfectly describe reality.
- ▶ Example: neglecting air resistance, string mass.

- ▶ **Measurement Error**

- ▶ Inaccurate parameters ( $l$ ,  $g$ , etc.).

- ▶ **Formula Error**

- ▶ Truncating series expansions.

# Further Error Sources

- ▶ **Discretization Error**
  - ▶ Derivatives  $\rightarrow$  difference quotients
  - ▶ Integrals  $\rightarrow$  finite sums
- ▶ **Rounding and Representation Error**
  - ▶ Finite precision number representation
  - ▶ Rounding during computations

# Error Propagation Through Models

Real-World Problem

↓ (model + measurement error)

Scientific Model

↓ (formula error)

Mathematical Model

↓ (discretization error)

Numerical Model

↓ (rounding error)

Computer Model