SageMath: A Guide for the Curious Physicist

Nishtha Tikalal

© 2025 Nishtha Tikalal

This work is licensed under the
Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License.

To view a copy of this license, visit
`https://creativecommons.org/licenses/by-nc-sa/4.0/`

Contents

Chapter 1

Introduction to SageMath for Physicists

SageMath is an open-source mathematics software system integrating powerful symbolic and numerical computation capabilities, built on top of the Python programming language. It is designed to provide a comprehensive environment for exploring mathematical concepts, performing calculations, and visualizing results across many domains, including physics.

1.1 What is SageMath?

SageMath combines a vast collection of open-source packages and libraries, such as NumPy, SciPy, Maxima, and others, providing a single platform for algebra, calculus, number theory, linear algebra, and more. Its integrated nature allows physicists to tackle complex analytic and numeric problems with a unified interface.

1.2 Installing and Setting Up SageMath

You can install SageMath on your local machine or use it online via platforms such as CoCalc. The recommended approach depends on your operating system and preferences:

- **Local Installation:** Download from `https://www.sagemath.org/` and follow platform-specific installation guides.
- **Online Usage:** Use the browser-based CoCalc environment (`https://cocalc.com/`), which requires no installation and offers collaborative features.

1.3 Using the SageMath Interface

Once set up, you can interact with SageMath through:

- **Command Line Interface:** Enter commands in a terminal.
- **Jupyter Notebooks:** Interactive web-based documents mixing code, text, and visualizations.
- **Sage Worksheets:** The classic Sage notebook interface.

1.4 Basic Mathematical Operations

SageMath supports standard arithmetic and symbolic operations. For example:

```
sage: 2 + 2
4

sage: factor(2007)
3^2 * 223
```

You can define variables symbolically and perform algebraic manipulations:

```
sage: var('x')
x

sage: expand((x + 1)^3)
x^3 + 3*x^2 + 3*x + 1
```

1.5 Helpful Features

- **Tab Completion:** Quickly find commands and functions.
- **In-line Help:** Type `?command` to get documentation.
- **Integration with Python:** Use Python syntax and libraries seamlessly.
- **Visualization:** Create 2D and 3D plots for better understanding.

1.6 Next Steps

This guide will walk you through the essential topics required for physics modeling, beginning with refresher mathematics and proceeding to advanced applications such as differential equations, vector fields, wavefunctions, and simulations—all within SageMath.

Try typing a simple calculation now:

```
sage: sqrt(2)^2
2
```

Chapter 2

Basic Mathematics Refresher

This chapter reviews essential mathematical concepts and operations you will use frequently in your physics modeling work with SageMath. It revisits algebraic expressions, functions, and calculus foundations using SageMath syntax.

2.1 Algebraic Expressions and Arithmetic

SageMath supports standard arithmetic operations:

```
sage: 2 + 3
5
sage: 5 - 7
-2
sage: 4 * 6
24
sage: 8 / 3
8/3
sage: 8 / 3.0  # Decimal approximation
2.66666666666667
sage: 2^5      # Exponentiation
32
```

Note Sage returns exact rational results unless you explicitly request floating point.

Use parentheses to control order:

```
sage: 2 * (4 + 1)^2
```

```
50
sage: 2 * 4 + 1^2
9
```

2.2 Working with Variables and Functions

Declare symbolic variables:

```
sage: var('x y')
(x, y)
```

Define functions:

```
sage: f(x) = x^2 + 3*x + 5
sage: f(2)
15
```

Expand and factor expressions:

```
sage: expand((x + 1)^3)
x^3 + 3*x^2 + 3*x + 1

sage: factor(x^3 - 1)
(x - 1) * (x^2 + x + 1)
```

2.3 Calculus Basics

Differentiate functions symbolically:

```
sage: derivative(f, x)
2*x + 3
```

Integrate symbolically or numerically:

```
sage: integral(f, x)
x^3/3 + (3/2)*x^2 + 5*x

sage: numerical_integral(f, 0, 1)
(9.08333333333333, 1.0e-14)
```

Compute limits:

```
sage: limit(sin(x)/x, x=0)
1
```

2.4 Matrices and Linear Algebra

Define matrices and perform operations:

```
sage: A = Matrix([[1,2],[3,4]])
sage: B = Matrix([[0,1],[1,0]])
sage: A + B
[1 3]
[4 4]

sage: A * B
[2 1]
[4 3]

sage: A.det()
-2
```

Compute eigenvalues and eigenvectors:

```
sage: A.eigenvalues()
[5.37228132326901, -0.372281323269014]
```

2.5 Exercises

- Compute $(3+5)^2 - 4/3$ using SageMath.
- Define $g(x) = \sin(x) + \cos^2(x)$ and compute $g(\pi/4)$.
- Differentiate and integrate the function $h(x) = e^{-x^2}$.
- Define a 3x3 identity matrix and add 2 times the matrix of all ones.

This comprehensive refresher prepares you to smoothly work with mathematical expressions in SageMath, supporting physics modeling ahead.

Chapter 3

Modeling Mechanics with SageMath

This chapter introduces essential modeling techniques for classical mechanics using SageMath, focusing on graphical representations and practical examples such as free body diagrams, vector fields, and simple dynamical systems.

3.1 Free Body Diagrams

Free body diagrams help visualize forces acting on an object. While SageMath does not have built-in free body diagram drawing tools, you can model forces as vectors and annotate plots to represent an FBD.

```
# SageMath code for a simple free body diagram on an inclined plane
from sage.all import *

theta = pi/6
m = 1
g = 9.8
N = vector((0, m*g*cos(theta)))
Fg = vector((-m*g*sin(theta), 0))
resultant = N + Fg
G = Graphics()
# Plot object as a point
G += point((0,0), size=30, rgbcolor='black')
# Plot forces as arrows
G += arrow((0, 0), N, color='blue', thickness=2)
```

```
G += arrow((0, 0), Fg, color='red', thickness=2)
G += arrow((0, 0), resultant, color='green', thickness=2)
G.show(aspect_ratio=1)
```

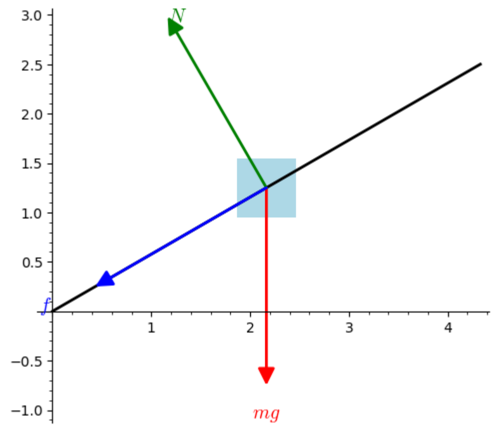

Figure 3.1: Free Body Diagram of a Mass on an Inclined Plane

3.2 Vector Fields and Force Visualization

SageMath can graph vector fields to model forces such as gravitational or electric fields.

```
# SageMath code to plot a central force vector field
from sage.all import *

x, y = var('x y')
def central_force(x_, y_):
    r = sqrt(x_^2 + y_^2)
    if r == 0:
        return (0, 0)
    else:
        return (-x_/r^3, -y_/r^3)

vf = plot_vector_field(central_force, (x, -3, 3), (y, -3, 3), plot_points=20, aspect_rat
vf.show()
```

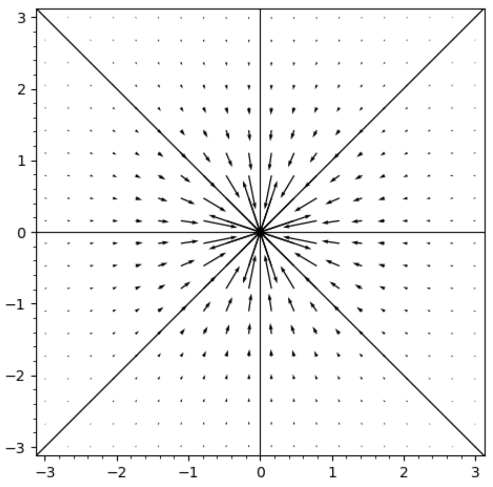

Figure 3.2: Vector Field Representing a Central Force

3.3 Graphing Wavefunctions

Visualize quantum mechanical wavefunctions (real and imaginary parts):

```
# SageMath code to plot ground state wavefunction of quantum harmonic oscillator
from sage.all import *

x = var('x')
alpha = 1
psi0 = (alpha/pi)^(1/4) * exp(-alpha*x^2/2)
plot(psi0, (x, -4, 4), title='Quantum Harmonic Oscillator Ground State',
     axes_labels=['x', '$\\psi_0(x)$']).show()
```

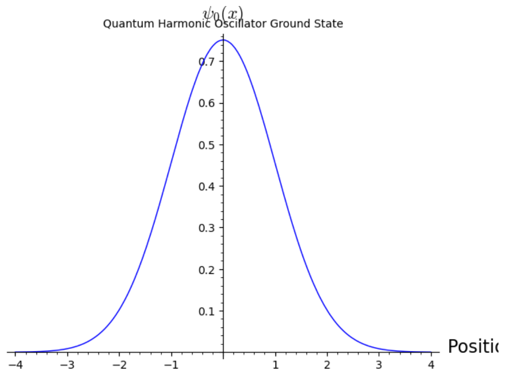

Figure 3.3: Wavefunction Plot for a Quantum Harmonic Oscillator

3.4 Phase Space and Phase Change Diagrams

Plot phase portraits to analyze dynamical systems:

```
# SageMath code for phase space diagram of a simple pendulum
from sage.all import *

theta, omega = var('theta omega')
g = 9.8
l = 1

def pendulum_phase_space(theta_, omega_):
    return (omega_, -g/l*sin(theta_))

pf = plot_vector_field(pendulum_phase_space, (theta, -pi, pi), (omega, -4, 4),
                       plot_points=25, aspect_ratio=1)
pf.show()
```

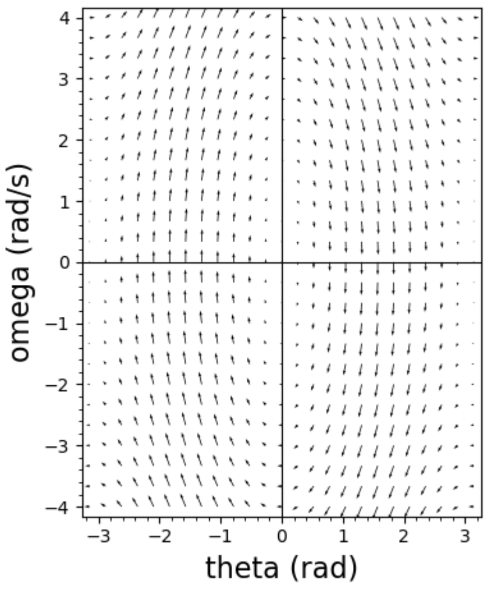

Figure 3.4: Phase Space Diagram of a Simple Pendulum

3.5 Feynman Diagrams

SageMath does not natively support Feynman diagram drawing, but external tools such as TikZ-Feynman can be used to create and import them.

3.6 Collision Modeling

Simulate and visualize elastic collisions between two particles:

```
# SageMath code for 2D elastic collision trajectories
from sage.all import *

A0 = vector((0, 0))
B0 = vector((3, 0))
vA = vector((1, 0))
vB = vector((-1, 0.4))
t_collision = 1.5

vA_after = vector((0.5, 0.6))
vB_after = vector((-0.8, -0.5))

tlist1 = srange(0, t_collision, 0.05)
A_traj1 = [A0 + vA*t for t in tlist1]
B_traj1 = [B0 + vB*t for t in tlist1]

tlist2 = srange(t_collision, 3, 0.05)
A_traj2 = [A0 + vA*t_collision + vA_after*(t-t_collision) for t in tlist2]
B_traj2 = [B0 + vB*t_collision + vB_after*(t-t_collision) for t in tlist2]

P = list_plot(A_traj1, color='blue') + list_plot(A_traj2, color='blue', linestyle='--')
    list_plot(B_traj1, color='red') + list_plot(B_traj2, color='red', linestyle='--')
P.show(axes_labels=['x', 'y'])
```

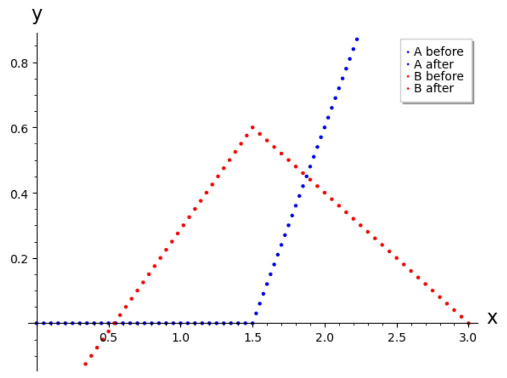

Figure 3.5: Elastic Collision Trajectories

3.7 Simple Systems

Model canonical systems such as mass on an incline and pendulums with ODE solvers and graphical interpretations:

```
# SageMath code: Pendulum motion simulation over time
from sage.all import *

g = 9.81
l = 1
theta0 = 0.6
omega0 = 0

def pendulum(state, t):
    theta, omega = state
    return [omega, -g/l*sin(theta)]

tvals = srange(0, 10, 0.01)
result = desolve_odeint(pendulum, [theta0, omega0], tvals, [var('theta'), var('omega')])

list_plot([(tvals[i], result[i][0]) for i in range(len(tvals))],
          plotjoined=True, axes_labels=['Time (s)', 'Angle (rad)'],
          title='Pendulum Angular Position vs Time').show()
```

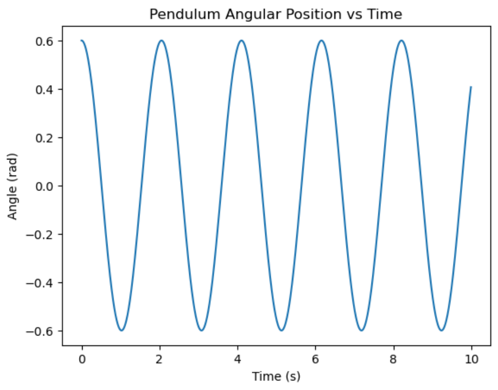

Figure 3.6: Pendulum Motion Simulation

Chapter 4

Vectors

4.1 Vector Operations and Visualization

Vectors are fundamental objects in physics representing quantities with both magnitude and direction. Essential operations include vector addition, scalar multiplication, dot product, and cross product.

The following SageMath code illustrates basic vector operations with 2D arrows:

```
# Vector operations visualization in 2D
v1 = vector([2, 1])
v2 = vector([1, 3])
v_sum = v1 + v2
v_scaled = 1.5 * v1

G = Graphics()
G += arrow((0,0), v1, color='blue', thickness=2)
G += arrow((0,0), v2, color='green', thickness=2)
G += arrow((0,0), v_sum, color='red', thickness=3)
G += arrow((0,0), v_scaled, color='purple', thickness=2, linestyle='--')

G.show(aspect_ratio=1)
```

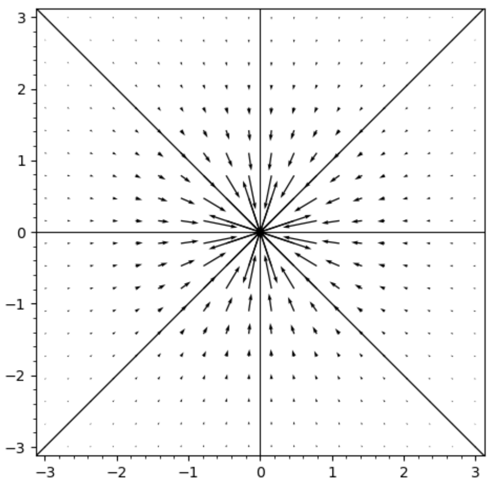

Figure 4.1: Visualization of Basic Vector Operations

Graphical representation of vectors helps gain intuition about their algebraic properties. Visualization tools like SageMath can plot vectors and vector sums in 2D and 3D.

4.2 Modeling Vector Fields in Electromagnetism

Electromagnetic phenomena are often modeled using vector fields, where a vector is assigned to each point in space. For example, the electric field $\mathbf{E}(\mathbf{r})$ and magnetic field $\mathbf{B}(\mathbf{r})$ describe forces exerted on charges and currents.

Here is SageMath code plotting a 2D central-force-like vector field (like an electric field from a point charge):

```
x, y = var('x y')
def E_field(x_, y_):
    R = sqrt(x_^2 + y_^2)
    if R == 0:
        return (0, 0)
    else:
        return (-x_ / R^3, -y_ / R^3)

plot_vector_field(E_field, (x, -3, 3), (y, -3, 3), plot_points=20, aspect_ratio=1).show(
```

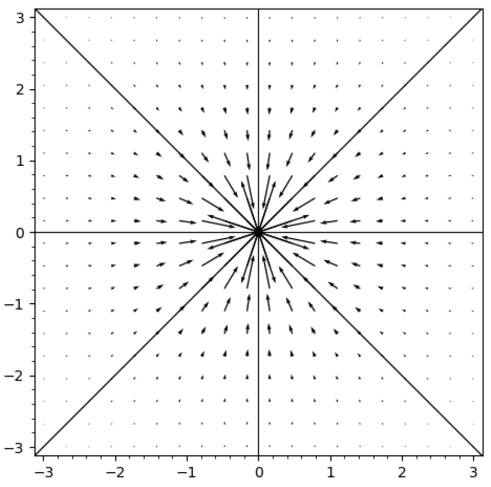

Figure 4.2: Example Electromagnetic Vector Field

Vector calculus tools allow mathematical modeling and manipulation of these fields, including divergence, curl, and gradients, providing insight into field behavior and dynamics.

4.3 Graphing Fields and Flows

Graphing vector fields visually depicts fields and flows, making phenomena such as fluid dynamics, electric/magnetic fields, and force fields more comprehensible.

The SageMath code below plots a rotational flow field:

```
def flow_field(x_, y_):
    return (-y_, x_)

plot_vector_field(flow_field, (x, -3, 3), (y, -3, 3),

plot_points=20, aspect_ratio=1).show()
```

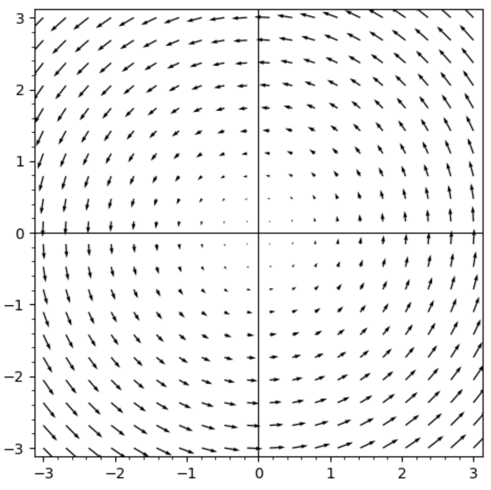

Figure 4.3: Graph of a Vector Field Showing Flow Patterns

Chapter 5

Wavefunctions and Quantum Modeling

5.1 Symbolic Representations of Wavefunctions

Quantum wavefunctions describe the probability amplitudes of quantum states. Symbolically, wavefunctions can be represented using variables and mathematical expressions involving special functions like Hermite polynomials.

The ground state wavefunction $\psi_0(x)$ of the one-dimensional quantum harmonic oscillator is given by

$$\psi_0(x) = \left(\frac{\alpha}{\pi}\right)^{1/4} e^{-\alpha x^2/2},$$

where $\alpha = \frac{m\omega}{\hbar}$.

```
# SageMath code: symbolic wavefunction for ground state quantum harmonic oscillator
x = var('x')
m, omega, hbar = var('m omega hbar')
alpha = m*omega/hbar
psi0 = (alpha/pi)^(1/4) * exp(-alpha*x^2/2)

# Substitute numerical values for plotting example
psi0_num = psi0.substitute(m=1, omega=1, hbar=1)
show(psi0_num)
```

$$\frac{e^{\left(-\frac{1}{2}\,x^{2}\right)}}{\pi^{\frac{1}{4}}}$$

Figure 5.1: Symbolic Representation of the Ground State Wavefunction

5.2 Plotting Real and Complex Wavefunctions

To visualize wavefunctions, you can plot their real, imaginary parts, or magnitude squared (probability density). For example, the first excited state wavefunction $\psi_1(x)$ can be plotted alongside $\psi_0(x)$.

```
# SageMath code to plot real wavefunctions: ground and first excited states
from sage.all import *

x = var('x')
m = 1; omega = 1; hbar = 1
alpha = m*omega/hbar
psi0 = (alpha/pi)^(1/4) * exp(-alpha*x^2/2)
psi1 = (4*alpha^3/pi)^(1/4) * x * exp(-alpha*x^2/2)

p0 = plot(psi0, (x, -3, 3), color='blue', legend_label='$\\psi_0$')
p1 = plot(psi1, (x, -3, 3), color='red', legend_label='$\\psi_1$')

(p0 + p1).show(legend_loc='upper right')
```

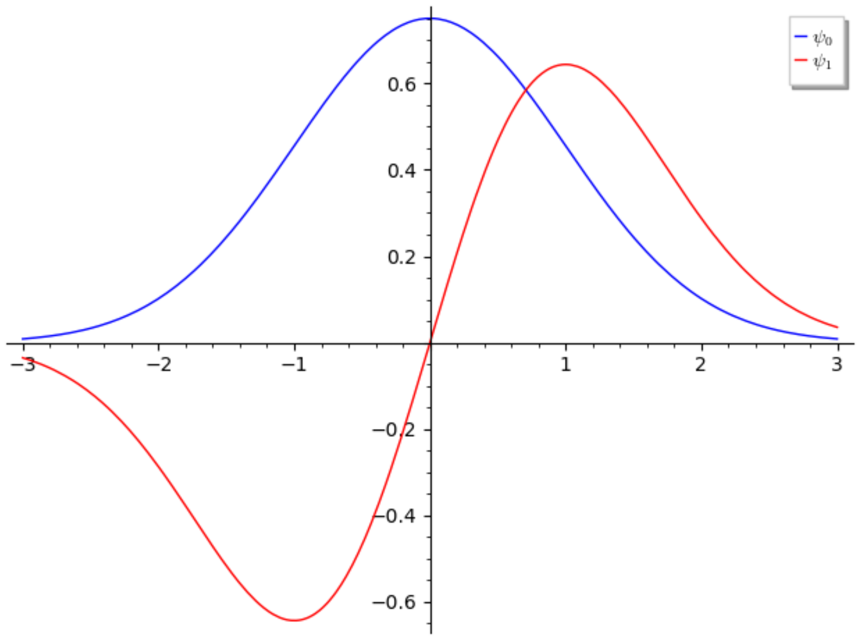

Figure 5.2: Real Parts of Ground and First Excited State Wavefunctions

5.3 Simulating Quantum Systems

Time evolution and eigenstates of quantum systems can be explored using numerical solvers. For example, simulate the probability density $|\psi(x,t)|^2$ for a time-dependent superposition ("wave packet") in the quantum harmonic oscillator.

```
# SageMath code: simple time-dependent quantum harmonic oscillator wave packet
import numpy as np
from sage.all import *

var('x t')
m = 1; omega = 1; hbar = 1
alpha = m*omega/hbar

psi0 = (alpha/pi)^(1/4) * exp(-alpha*x^2/2)
psi1 = (4*alpha^3/pi)^(1/4) * x * exp(-alpha*x^2/2)

# Time-dependent linear combination (normalized)
psi_t = 1/sqrt(2) * (psi0 + exp(I*omega*t) * psi1)
```

```
# Plot probability density at fixed time t=0
f = abs(psi_t.substitute(t=0))^2
plot(f, (x, -3, 3), title=r'$|\psi(x,0)|^2$', axes_labels=['x', 'Probability Density']).
```

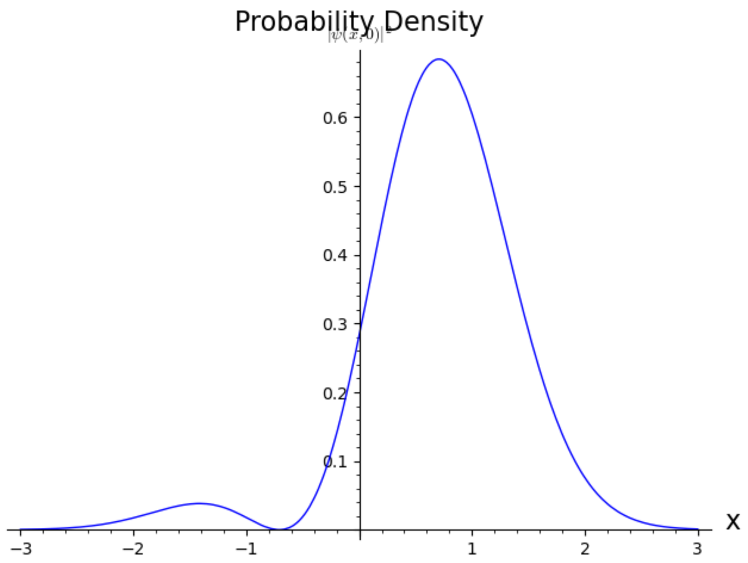

Figure 5.3: Probability Density of a Time-Dependent Quantum Harmonic Oscillator State

Chapter 6

Feynman Diagrams and Particle Interactions

6.1 Introduction to Feynman Diagram Concepts

Feynman diagrams are pictorial representations used in quantum field theory to depict particle interactions and scattering amplitudes. Named after Richard Feynman, they provide an intuitive and calculationally powerful way to understand complex processes between subatomic particles.

These diagrams illustrate: - Particles entering an interaction (initial state) - Vertices where particles emit or absorb force carriers - Particles leaving after interaction (final state) - Virtual particles mediating forces as internal lines

Feynman diagrams are essential tools in modern particle physics and enable simplified calculations of probabilities for particle processes.

6.2 Basic Construction and Representation

To construct a Feynman diagram:

1. Identify the initial and final state particles. 2. Draw lines representing particles: solid lines for fermions (e.g., electrons, quarks), wavy lines for gauge bosons (photons, gluons). 3. Mark interaction points called vertices where particles meet and exchange force carriers. 4. Assign directions with arrows indicating particle flow; antiparticles have arrows reversed. 5. Respect conservation laws at vertices: energy, momentum, charge, and quantum numbers.

Example: Electron-positron annihilation into muon-antimuon pairs involves two vertices linked by a virtual photon propagator.

6.3 Visualization Techniques

Visualizing Feynman diagrams enhances intuition and documentation of particle interactions.

- TikZ-Feynman Package: A LaTeX package providing elegant commands to draw Feynman diagrams with high typographical quality. - Graphical Tools: Software like JaxoDraw, FeynDiagram.org, or python-based packages to interactively create and export diagrams. - Manual Drawing: Using vector graphics software or TikZ libraries to custom draw diagrams for publications.

Example TikZ-Feynman Code Snippet:

```
\begin{tikzpicture}
  \begin{feynman}
    \vertex (a) {\(e^{-}\)};
    \vertex[right=2cm of a] (b);
    \vertex[above right=2cm of b] (f1) {\(\mu^{-}\)};
    \vertex[below right=2cm of b] (f2) {\(\mu^{+}\)};

    \diagram* {
      (a) -- [fermion] (b) -- [fermion] (f1),
      (b) -- [anti fermion] (f2),
      (a) -- [photon, edge label=\(\gamma\)] (b),
    };
  \end{feynman}
\end{tikzpicture}
```

This draws the electron-positron annihilation Feynman diagram with a photon mediator.

Chapter 7

Classical Mechanics Simulation

7.1 Lagrangian and Hamiltonian Formulations

Classical mechanics can be elegantly formulated using the Lagrangian and Hamiltonian formalisms, which replace Newton's vector equations with scalar functions describing system energy.

The **Lagrangian**, L, is defined as the difference between kinetic energy T and potential energy V:

$$L = T - V.$$

Using generalized coordinates q_i and velocities $\dot{q}_i$, the Euler-Lagrange equations provide equations of motion:

$$\frac{d}{dt}\frac{\partial L}{\partial \dot{q}_i} - \frac{\partial L}{\partial q_i} = 0.$$

From the Lagrangian, the **Hamiltonian**, H, can be constructed as the Legendre transform:

$$H = \sum_i p_i \dot{q}_i - L,$$

where $p_i = \frac{\partial L}{\partial \dot{q}_i}$ are the canonical momenta. Hamilton's equations describe the dynamics:

$$\frac{dq_i}{dt} = \frac{\partial H}{\partial p_i}, \quad \frac{dp_i}{dt} = -\frac{\partial H}{\partial q_i}.$$

These formulations are especially useful for complex systems, constraints, and quantum analogues.

7.2 Numerical ODE Solvers

Many classical systems require numerical integration for their equations of motion. Tools such as SageMath offer solvers like `desolve_odeint` to compute trajectories efficiently.

7.3 Simulating Oscillations and Coupled Systems

7.3.1 Simple Pendulum Simulation

The pendulum's motion is governed by

$$\ddot{\theta} + \frac{g}{l}\sin\theta = 0.$$

Using SageMath, the system can be simulated numerically:

```
from sage.all import *

def pendulum(state, t):
    theta, omega = state
    g, l = 9.81, 1.0
    dtheta_dt = omega
    domega_dt = -(g/l)*sin(theta)
    return [dtheta_dt, domega_dt]

tvals = srange(0, 10, 0.01)
result = desolve_odeint(pendulum, [0.6, 0], tvals, [var('theta'), var('omega')])

list_plot([(tvals[i], result[i][0]) for i in range(len(tvals))],
          plotjoined=True, axes_labels=['Time (s)', 'Angle (rad)'],
          title='Simple Pendulum Simulation').show()
```

7.3.2 Coupled Oscillators

Consider two masses connected by springs:

$$m\ddot{x}_1 = -k(x_1 - x_2), \quad m\ddot{x}_2 = -k(x_2 - x_1).$$

SageMath can solve and visualize their dynamics:

```
m, k = 1, 1
def coupled_oscillators(state, t):
    x1, v1, x2, v2 = state
    dx1dt = v1
    dv1dt = -(k/m)*(x1 - x2)
    dx2dt = v2
    dv2dt = -(k/m)*(x2 - x1)
    return [dx1dt, dv1dt, dx2dt, dv2dt]

tvals = srange(0, 20, 0.05)
init_state = [1, 0, -1, 0]
result = desolve_odeint(coupled_oscillators, init_state, tvals,
                       [var('x1'), var('v1'), var('x2'), var('v2')])

p1 = list_plot([(tvals[i], result[i][0]) for i in range(len(tvals))],
               plotjoined=True, color='blue', legend_label='x1(t)')
p2 = list_plot([(tvals[i], result[i][2]) for i in range(len(tvals))],
               plotjoined=True, color='red', legend_label='x2(t)')

(p1 + p2).show(legend_loc='upper right', axes_labels=['Time (s)', 'Position'])
```

Chapter 8

Electromagnetic Field Simulations

8.1 Maxwell's Equations in SageMath

Maxwell's equations govern classical electromagnetism, describing electric and magnetic fields and their interactions. The equations in differential form are:

$$\nabla \cdot \mathbf{E} = \frac{\rho}{\varepsilon_0}, \quad \nabla \cdot \mathbf{B} = 0,$$
$$\nabla \times \mathbf{E} = -\frac{\partial \mathbf{B}}{\partial t}, \quad \nabla \times \mathbf{B} = \mu_0 \mathbf{J} + \mu_0 \varepsilon_0 \frac{\partial \mathbf{E}}{\partial t}.$$

Using SageMath, these vector calculus operations and partial differential equations can be symbolically defined and manipulated, enabling analytical insights and numerical approximations.

```
from sage.all import *

var('x y z t')
E = vector(var('E_x E_y E_z'))
B = vector(var('B_x B_y B_z'))

# Define divergence of E symbolically
div_E = diff(E[0], x) + diff(E[1], y) + diff(E[2], z)
show(div_E)
```

8.2 Visualizing Electromagnetic Waves

Electromagnetic waves can be modeled as time-varying solutions of Maxwell's equations, e.g., plane waves:

$$\mathbf{E}(x,t) = E_0 \sin(kx - \omega t)\hat{y}, \quad \mathbf{B}(x,t) = B_0 \sin(kx - \omega t)\hat{z}.$$

SageMath can plot vector fields evolving over time, illustrating wave propagation.

```
# SageMath code to plot electric field vector wave at fixed time slices
var('x t')
E0, k, omega = 1, 2*pi, 2*pi

E = lambda x_, t_: E0 * sin(k*x_ - omega*t_)
x_vals = srange(0, 2, 0.1)
E_vals = [(x, 0, E(x, 0)) for x in x_vals]  # Electric field along y-axis
B_vals = [(x, E(x, 0), 0) for x in x_vals]  # Magnetic field along z-axis

# Vector plot at time t=0 can be visualized by arrows or line plots
```

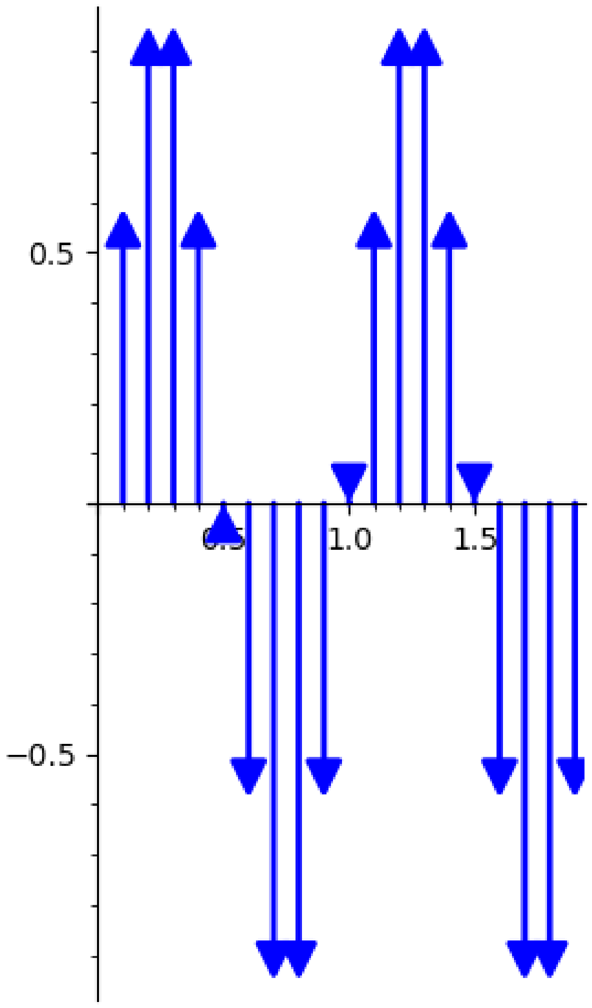

Figure 8.1: Electromagnetic wave vector field at a time slice

8.3 Boundary Problems and Numerical Solutions

Maxwell's equations often require numerical methods in complex geometries or with material boundaries. Techniques such as the Finite Difference Time Domain (FDTD) method or finite element analysis can be implemented using SageMath's PDE solvers or interfacing with specialized libraries.

Example: Numerically solving a scalar wave equation (simplified electromagnetic wave) in one dimension:

```
# Example: 1D Wave equation solver outline in SageMath
from sage.all import *
```

```
x, t = var('x t')
u = function('u')(x, t)
c = 1  # wave speed

PDE = diff(u, t, t) - c^2 * diff(u, x, x) == 0
# Numerical discretization and solution approaches can be implemented here
```

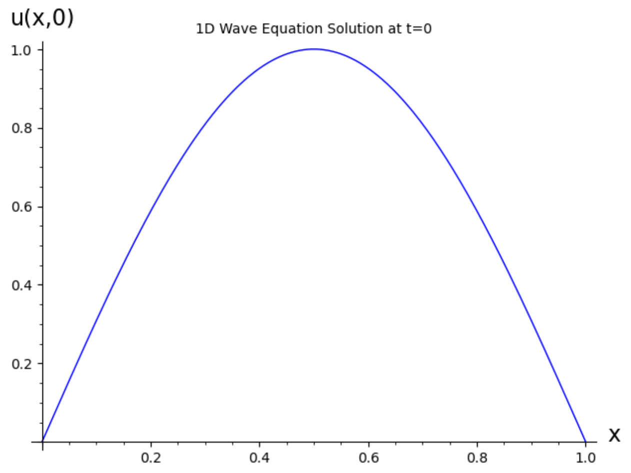

Figure 8.2: Numerical simulation of a 1D electromagnetic wave propagation

Chapter 9

Thermodynamics and Statistical Mechanics

9.1 State Diagrams and Phase Transitions

Thermodynamic state diagrams depict regions of different phases of matter, separated by coexistence curves marking phase transitions such as melting, boiling, and sublimation.

An example is the pressure-temperature phase diagram showing solid, liquid, and gas regions, with a triple point where all three coexist and a critical point ending the liquid-gas boundary.

```
# SageMath example: Simple phase diagram plotting coexistence curves
import matplotlib.pyplot as plt
import numpy as np

T = np.linspace(0, 500, 500)
P_solid_liquid = 10 + 0.05*T  # melting curve example
P_liquid_gas = 50 + 0.1*T     # boiling curve example

plt.plot(T, P_solid_liquid, label='Melting Curve')
plt.plot(T, P_liquid_gas, label='Boiling Curve')
plt.xlabel('Temperature (K)')
plt.ylabel('Pressure (atm)')
plt.title('Simplified Phase Diagram')
plt.legend()
plt.show()
```

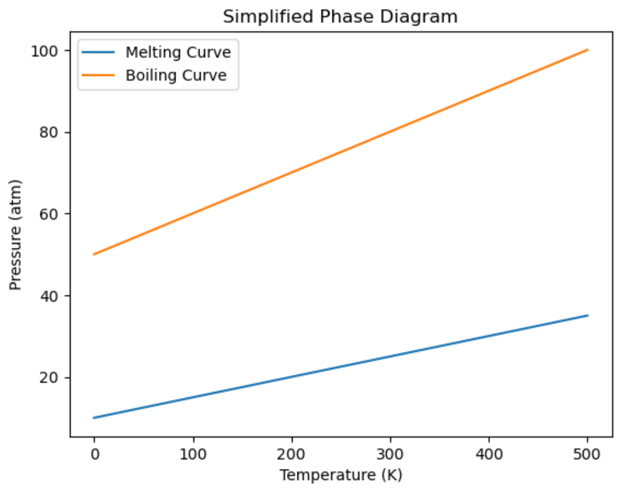

Figure 9.1: Pressure-Temperature Phase Diagram Showing Phase Boundaries

9.2 Partition Functions

The partition function Z encodes statistical properties of a system in thermodynamic equilibrium:

$$Z = \sum_i e^{-\beta E_i},$$

where E_i are energy levels and $\beta = 1/(k_B T)$.

Using SageMath, symbolic and numeric partition functions can be computed for various model systems such as the two-level system or quantum harmonic oscillator.

```
# SageMath code: partition function of a two-level system
var('beta E0 E1')
Z = exp(-beta*E0) + exp(-beta*E1)
show(Z)
```

$$e^{\left(-E_0\beta\right)} + e^{\left(-E_1\beta\right)}$$

Figure 9.2: Symbolic Partition Function for a Two-Level System

9.3 Monte Carlo Simulations

Monte Carlo methods use random sampling to evaluate thermodynamic averages in complex systems, such as the Ising model for magnetism near phase transitions.

Typical SageMath implementations perform Metropolis-Hastings sampling and statistical averaging.

```
# Simplified SageMath outline of Monte Carlo step (conceptual)
import random
def monte_carlo_step(state, beta):
    # compute energy difference if spin flipped
    dE = ...
    if dE < 0 or random.random() < exp(-beta*dE):
        # accept move
        state.flip_spin()
    return state
```

Chapter 10

Programming Essentials in Physics Modeling

10.1 Python Constructs in Sage

SageMath leverages the power of Python, enabling physics modelers to use familiar programming constructs such as loops, conditionals, lists, dictionaries, and comprehensions.

Example: Using a loop with SageMath variables to compute and print squares of numbers.

```
for i in range(1, 6):
    print(f"{i} squared is {i2}")
```

Customizing computations with conditional logic:

```
x = var('x')
for i in range(1, 6):
    expr = x^i
    if i % 2 == 0:
        print(f"Expanding expression for i={i}: {expr.expand()}")
    else:
        print(f"Expression for i={i}: {expr}")
```

10.2 Custom Functions and Classes for Physics

Defining reusable functions helps organize calculations and simulations.

Function example: Kinetic energy calculator

```
def kinetic_energy(mass, velocity):
    return 0.5 * mass * velocity^2

print(kinetic_energy(2, 3))
```

You can also define classes encapsulating physical systems with properties and methods.

```
class SimpleHarmonicOscillator:
    def __init__(self, mass, spring_const):
        self.m = mass
        self.k = spring_const

    def frequency(self):
        from sage.functions.trig import sqrt
        return sqrt(self.k / self.m) / (2 * pi)

    def energy(self, amplitude):
        return 0.5 * self.k * amplitude^2

sho = SimpleHarmonicOscillator(mass=1, spring_const=4)
print(sho.frequency())
print(sho.energy(0.1))
```

10.3 Modular and Scripted Modeling

Building complex models benefits from modular code organization using multiple scripts or functions.

- Save reusable code in separate files and import them with `attach` or Python `import`. - Structure physics computations into functions or classes for clarity. - Use scripts to automate simulations, parameter sweeps, or batch computations.

Example of attaching a separate Sage file:

```
attach('physics_functions.sage')
# Now you can call functions defined in physics_functions.sage
```

Chapter 11

Exporting and Publishing

11.1 Integrating SageMath with LaTeX

SageMath provides seamless integration with LaTeX, allowing direct insertion of symbolic mathematics and computations into LaTeX documents.

Use the `latex()` function to generate LaTeX code for Sage objects:

```
from sage.all import *
var('z')
expr = integrate(z^4, z)
print(latex(expr))
```

This outputs `\frac{1}{5} z`^{5}, *whichcanbeuseddirectlyinmathmode*.

11.2 Creating Dynamic Documents with SageTeX

`The SageTeX package lets you embed Sage code within LaTeX documents, where computations are performed during typesetting, ensuring calculations remain up to date.`

`Minimal example in a LaTeX document:`

```
\documentclass{article}
\usepackage{sagetex}
\begin{document}
The integral of \(x^4\) is
\[
\sage{integrate(x^4, x)}
\]
\end{document}
```

Compile the document with SageTeX support to automatically compute and insert results and plots.

11.3 Exporting Codes and Figures for Publications

- Use Sage commands like save() or export_pdf() to save plots and graphics for inclusion in manuscripts. - Export symbolic expressions with latex() for pristine math typesetting. - Save figures in formats like PNG, PDF, or SVG to meet journal submission requirements:

```
p = plot(sin(x), (x, 0, 2*pi))
p.save('sine_plot.pdf')
```

- Organize your source Sage code and exported figures in a project folder for reproducibility.

Chapter 12

Troubleshooting and Support

12.1 Common Errors and Fixes

When using SageMath, users often encounter typical errors. Awareness of common mistakes and remedies improves efficiency:

- **Implicit Multiplication Errors:** Sage requires explicit multiplication symbols.
  - Correct: 5*x, Incorrect: 5x
  - Enable implicit multiplication with implicit_multiplication(True) if desired.
- **Function vs Expression Confusion:**
  - Defining f = x creates a symbolic expression, not a function.
  - Define functions explicitly: f(x) = x.
- **Parameter Order Mistakes:**
  - Use taylor? or Sage's built-in help to confirm argument order.
- **Syntax and Type Errors:**
  - Ensure correct Sage syntax; check for mixing Python and Sage types.
- **AI Assistance:**
  - SageMath includes an experimental AI feature which can help diagnose errors and suggest fixes interactively.

12.2 Community Forums and Documentation

- **Ask Sage:** https://ask.sagemath.org - Q&A platform with active Sage community.
- **Sage Zulip Chat:** Real-time developer and user discussions.
- **Official Documentation:** https://doc.sagemath.org - Comprehensive manuals, tutorials, and references.
- **Tutorials and Screencasts:** Numerous online resources enhance learning.

Interacting with these communities accelerates problem solving and broadens understandin

12.3 Further Learning Paths and Reading

- Explore SageMath's thematic tutorials and official lessons.
- Investigate specific packages relevant to physics and mathematics.
- Review computational physics literature that incorporates SageMath.
- Engage with open-source contributions for hands-on development experience.

Continued practice and community involvement deepen mastery of SageMath for scientific research and teaching.