Standard Operating Procedure: Temperature-Dependent Raman Study of Monolayer Graphene (Heating-Cooling Hysteresis)

Version 2.0 - Fully Reproducible Experimental Protocol

2D Materials Characterization Laboratory

March 5, 2026

Abstract

This document provides a complete, reproducible protocol for measuring thermal hysteresis in Raman phonon modes of monolayer graphene during heating and cooling cycles. All experimental parameters, calibration procedures, data acquisition settings, and analysis algorithms are fully specified to ensure exact replication of results across different laboratories.

Contents

1 Purpose and Scientific Background

1.1 Objective

To experimentally measure thermal hysteresis in Raman phonon modes of monolayer graphene during heating and cooling cycles. The goal is to detect possible non-reciprocal thermodynamic coupling effects by comparing Raman peak positions and linewidths during temperature ramps.

1.2 Scientific Hypothesis

Graphene's phonon modes may exhibit heating-cooling hysteresis due to:

- Anharmonic phonon-phonon coupling
- Substrate interaction effects
- Possible non-reciprocal thermodynamics in 2D materials
- Thermal expansion mismatch between graphene and SiO_2 substrate

1.3 Expected Outcomes

- Temperature-dependent phonon softening (red shift with increasing temperature)
- Quantifiable hysteresis loop area between heating and cooling cycles
- Possible asymmetry in linewidth broadening indicating irreversible processes
- Stable defect ratio throughout experiment confirming sample integrity

2 Materials and Sample Preparation

2.1 Sample Specifications

Parameter	**Specification**
Material	CVD-grown monolayer graphene
Substrate	300 nm SiO_2/Si (p-doped, (100) orientation)
Substrate resistivity	1-10 Ω·cm
Substrate size	10 mm × 10 mm
Graphene coverage	>95% continuous monolayer
Doping level	Undoped (as-grown)
Back side	Polished (for optical transparency)

Table 1: Sample specifications

2.2 Sample Preparation Protocol

Step 1: Substrate Cleaning

1. Sonicate in acetone for 10 minutes at 25°C
2. Rinse with isopropanol for 30 seconds
3. Sonicate in isopropanol for 10 minutes at 25°C
4. Dry with nitrogen gas (99.999% purity)
5. UV-ozone treatment for 15 minutes to remove organic residues

Step 2: Graphene Transfer (if using as-grown CVD)

1. Spin-coat PMMA A4 950K at 3000 rpm for 60 seconds
2. Bake at 150°C for 5 minutes
3. Etch copper foil in 0.1 M ammonium persulfate for 4-6 hours
4. Rinse graphene/PMMA in DI water (3 cycles, 10 minutes each)
5. Scoop onto cleaned SiO_2/Si substrate
6. Dry at room temperature for 12 hours
7. Remove PMMA in acetone vapor at 50°C for 4 hours
8. Anneal in Ar/H_2 (90%/10%) at 300°C for 2 hours

Step 3: Sample Mounting

1. Mount sample on temperature stage using silver paste
2. Ensure thermal contact - paste should cover 80% of back surface
3. Cure silver paste at 60°C for 30 minutes
4. Cover with quartz window (for vacuum measurements)

2.3 Quality Verification Criteria

Before measurement, verify sample quality using:

3 Instrumentation Specifications

3.1 Raman Spectrometer Configuration

3.2 Required Consumables and Supplies

4 Pre-Measurement Calibration

4.1 Instrument Alignment Protocol

Step 1: Laser Alignment

Parameter	Measurement	Acceptance Condition
Monolayer confirmation	Raman I_{2D}/I_G ratio	$I_{2D}/I_G > 2.0$ and $I_G/I_{2D} < 1$
Defect density	Raman I_D/I_G ratio	$I_D/I_G < 0.1$
Uniformity	Raman mapping (20×20 μm)	$< 5\%$ variation in peak positions
Coverage	Optical microscopy	$> 95\%$ continuous
Doping level	G peak position	1580 ± 2 cm^{-1}
Strain	2D peak position	2680 ± 5 cm^{-1}

Table 2: Quality verification criteria

1. Remove all filters, set laser to 0.1 mW
2. Place alignment pinhole (10 μm) at sample position
3. Maximize transmitted power through pinhole
4. Adjust mirrors to center beam on pinhole
5. Record transmitted power: should be $> 80\%$ of incident

Step 2: Confocal Alignment

1. Place flat mirror at sample position
2. Set pinhole to 50 μm
3. Maximize signal while scanning pinhole position
4. Repeat for both X and Y axes
5. Confocal performance: FWHM of axial response < 2 μm

4.2 Wavenumber Calibration

Procedure:

1. Measure silicon reference peak at room temperature
2. Acquisition: 3 accumulations, 10 seconds each
3. Fit with Lorentzian function (see Section 7)
4. Compare peak position to reference value: 520.0 cm^{-1}
5. Apply correction if deviation > 0.2 cm^{-1}

Calibration Equation:

$$\nu_{\text{corrected}} = \nu_{\text{measured}} \times \frac{520.0}{\nu_{\text{Si,measured}}} \tag{1}$$

Acceptance: Post-calibration drift $\leq \pm 0.5$ cm^{-1} over 8 hours.

4.3 Laser Power Calibration

Measure laser power at sample position using calibrated power meter:
Acceptance: Measured power within ±5% of set point.

4.4 Temperature Stage Calibration

Two-point calibration:

1. Ice point: 0°C (273.15 K) using ice bath
2. Boiling point: 100°C (373.15 K) using water bath
3. Measure resistance of built-in RTD
4. Adjust calibration constants in controller

Verification: Measure known melting point standards:
Acceptance: All measurements within ±1 K of true value.

4.5 Spot Selection and Registration

Protocol for reproducible spot location:

1. Identify three fiducial marks (scratches) near sample center
2. Record stage coordinates of each mark using motorized stage
3. Create coordinate transformation matrix
4. Mark primary measurement spot (center of triangle)
5. Mark secondary sacrificial spot (50 μm away)
6. Save coordinates in experiment file

Coordinate Registration:

$$\begin{bmatrix} x_{\text{sample}} \\ y_{\text{sample}} \end{bmatrix} = \mathbf{R} \begin{bmatrix} x_{\text{stage}} \\ y_{\text{stage}} \end{bmatrix} + \mathbf{t} \tag{2}$$

where $\mathbf{R}$ is rotation matrix, $\mathbf{t}$ is translation vector.

4.6 Laser Heating Verification

Procedure:

1. At room temperature (300 K), collect spectra at:
   - 0.1 mW
   - 0.3 mW
   - 0.5 mW
   - 0.7 mW
   - 1.0 mW

2. For each power, measure 5 times at different spots
3. Plot G peak position vs. laser power
4. Fit linear regression: $\omega_G = \omega_0 + \alpha P$

Acceptance Condition:

- Slope $|\alpha| < 0.5$ cm^{-1}/mW
- At operating power (0.5 mW), heating < 0.25 cm^{-1} shift
- Equivalent to temperature rise < 5 K (using $\partial\omega/\partial T \approx -0.02$ cm^{-1}/K)

5 Detailed Measurement Protocol

5.1 Experimental Timeline Overview

5.2 Hour 1: Room-Temperature Baseline

Setup:

- Laser power: 0.5 mW (verified from heating test)
- Grating: 1800 grooves/mm
- Acquisition time: 60 seconds per spectrum
- Accumulations: 3 (for cosmic ray removal)
- Spectral range: 1200-3200 cm^{-1}

Measurement sequence:

1. Move to Spot 1 (primary measurement spot)
2. Auto-focus using reflected laser intensity
3. Collect Stokes spectrum (3 accumulations, 60 s each)
4. Collect anti-Stokes spectrum (-3200 to -1200 cm^{-1})
5. Move to Spot 2 (50 μm away)
6. Repeat steps 2-5
7. Move to Spot 3 (50 μm away in perpendicular direction)
8. Repeat steps 2-5
9. Return to Spot 1

Data to record:

- Full spectra (raw data)
- Stage coordinates for each spot
- Room temperature (record every 5 minutes)
- Laser power before and after
- Background spectrum (laser off)

5.3 Hour 2: Heating Ramp (300 K → 500 K)

Temperature program:

At each temperature:

1. Ramp to set point at 10 K/min
2. Stabilize for exactly 5 minutes (timer starts when temperature reaches ±0.5 K of set point)
3. Record actual temperature from thermocouple
4. Auto-focus (thermal expansion may change focus)
5. Collect Stokes spectrum (3 × 60 s)
6. Collect anti-Stokes spectrum (3 × 60 s)
7. Record timestamp and actual temperature
8. Check laser power stability

Critical parameters to monitor:

- Temperature stability (±0.5 K)
- Focus drift (check every spectrum)
- Laser power (record before and after each set point)
- Background signal (check between accumulations)

5.4 Hour 3: High-Temperature Hold (500 K)

Hold protocol:

1. Maintain 500 K for 60 minutes total
2. Collect spectra every 10 minutes:
    - 0 min (immediately after stabilization)
    - 10 min
    - 20 min
    - 30 min
    - 40 min
    - 50 min
    - 60 min
3. At each time point:
    - Stokes spectrum (3 × 60 s)
    - Anti-Stokes spectrum (3 × 60 s)

Power test at sacrificial spot (at 30 min):

1. Move to sacrificial spot (secondary spot)
2. Collect spectra at:
   - 0.3 mW (3 × 60 s)
   - 0.5 mW (3 × 60 s)
   - 0.7 mW (3 × 60 s)
   - 1.0 mW (3 × 60 s)
3. Return to primary spot
4. Verify no damage: check I_D/I_G ratio

Purpose of hold:

- Confirm thermal equilibrium
- Check for time-dependent effects
- Verify absence of laser-induced artifacts
- Test sample stability at high temperature

5.5 Hour 4: Cooling Ramp (500 K → 300 K)

Temperature program:

Critical requirement: Measurements must be taken at the EXACT same location used during heating.

- Use saved coordinates from heating run
- Verify position using optical image registration
- Check focus before each measurement

At each temperature:

1. Cool to set point at -10 K/min
2. Stabilize for exactly 5 minutes
3. Record actual temperature
4. Auto-focus
5. Collect Stokes spectrum (3 × 60 s)
6. Collect anti-Stokes spectrum (3 × 60 s)
7. Record timestamp

5.6 Hour 5: Post-Cycle Verification

Return to room temperature:

1. Cool to 300 K at -10 K/min (if not already)
2. Stabilize for 10 minutes
3. Record final temperature

Spot 1 verification:

1. Move to primary measurement spot
2. Auto-focus
3. Collect reference spectrum (3×60 s)
4. Compare with initial baseline

Raman mapping:

- Grid size: 5×5 points
- Spacing: 2 μm (total area 10×10 μm)
- At each point:
  - Stokes spectrum (1×30 s, faster acquisition)
  - Record coordinates
- Generate maps of:
  - G peak position
  - G peak linewidth
  - 2D peak position
  - I_D/I_G ratio

Integrity check criteria:

- Mean G peak shift from initial < 0.5 cm^{-1}
- Mean I_D/I_G ratio < 0.1
- No visible damage in optical image
- Spatial variation $< 5\%$ across map

6 Spectral Fitting and Data Analysis

6.1 Peak Fitting Models

Lorentzian function for Raman peaks:

$$I(\nu) = I_0 + \frac{A}{\pi} \frac{\Gamma/2}{(\nu - \nu_0)^2 + (\Gamma/2)^2} \tag{3}$$

where:

- I_0 = background offset
- A = integrated area
- ν_0 = peak center (cm^{-1})
- Γ = full width at half maximum (FWHM, cm^{-1})

Voigt function (for high-resolution work):

$$I(\nu) = I_0 + A \frac{2\ln 2}{\pi^{3/2}} \frac{\Gamma_L}{\Gamma_G^2} \int_{-\infty}^{\infty} \frac{e^{-t^2}}{\left(\sqrt{\ln 2}\frac{\Gamma_L}{\Gamma_G}\right)^2 + \left(\frac{\nu - \nu_0}{\Gamma_G} - t\right)^2} dt \tag{4}$$

where Γ_L = Lorentzian width, Γ_G = Gaussian width.

Multi-peak fitting for 2D region: The 2D band consists of 4 Lorentzian components for monolayer graphene:

$$I_{2D}(\nu) = \sum_{i=1}^{4} \frac{A_i}{\pi} \frac{\Gamma_i/2}{(\nu - \nu_{0,i})^2 + (\Gamma_i/2)^2} \tag{5}$$

6.2 Fitting Algorithm

Initial parameters for G peak:

- $\nu_0 = 1580$ cm^{-1}
- $\Gamma = 15$ cm^{-1}
- $A = 1000$ (adjust based on intensity)
- I_0 = median of baseline region (1500-1550 cm^{-1})

Initial parameters for 2D peak:
Fitting constraints:

- All $\Gamma > 0$
- ν_0 within ± 50 cm^{-1} of initial
- Background I_0 linear over fitting range
- For 2D: intensity ratios constrained to physical values

6.3 Local Temperature Calculation from Anti-Stokes/Stokes Ratio

The phonon temperature is calculated using:

$$\frac{I_{AS}}{I_S} = \left(\frac{\nu_{AS}}{\nu_S}\right)^4 \exp\left(-\frac{\hbar\Omega}{k_B T_{\text{local}}}\right) \tag{6}$$

Solving for T_{local}:

$$T_{\text{local}} = -\frac{\hbar\Omega}{k_B\left[\ln\left(\frac{I_{AS}}{I_S}\right) - 4\ln\left(\frac{\nu_{AS}}{\nu_S}\right)\right]} \tag{7}$$

where:

- $\hbar = 1.0545718 \times 10^{-34}$ J·s
- $k_B = 1.380649 \times 10^{-23}$ J/K
- $\Omega = 2\pi c\nu$ with $c = 2.99792458 \times 10^{10}$ cm/s
- ν in cm^{-1} must be converted to Hz: $\nu_{\text{Hz}} = c \cdot \nu_{\text{cm}^{-1}}$

Acceptance condition:

$$|T_{\text{local}} - T_{\text{stage}}| \leq 5 \text{ K} \tag{8}$$

6.4 Hysteresis Quantification

For each temperature point:

$$\Delta\omega_{\text{hyst}}(T) = \omega_{\text{heat}}(T) - \omega_{\text{cool}}(T) \tag{9}$$

$$\Delta\Gamma_{\text{hyst}}(T) = \Gamma_{\text{heat}}(T) - \Gamma_{\text{cool}}(T) \tag{10}$$

For G mode and 2D mode separately:

- $\omega_G(T)$ = G peak position at temperature T
- $\Gamma_G(T)$ = G peak FWHM at temperature T
- $\omega_{2D}(T)$ = 2D peak position (center of mass)
- $\Gamma_{2D}(T)$ = 2D peak FWHM

6.5 Hysteresis Loop Area Calculation

Continuous integral:

$$A = \int_{T_{\min}}^{T_{\max}} |\omega_{\text{cool}}(T) - \omega_{\text{heat}}(T)|\, dT \tag{11}$$

Discrete approximation (trapezoidal rule):

$$A \approx \sum_{k=1}^{n-1} \frac{|\Delta\omega_k| + |\Delta\omega_{k+1}|}{2}(T_{k+1} - T_k) \tag{12}$$

where $\Delta\omega_k = \omega_{\text{heat}}(T_k) - \omega_{\text{cool}}(T_k)$

Uncertainty in area:

$$\sigma_A = \sqrt{\sum_{k=1}^{n-1}\left(\frac{T_{k+1} - T_k}{2}\right)^2 (\sigma^2_{\Delta\omega_k} + \sigma^2_{\Delta\omega_{k+1}})} \tag{13}$$

6.6 Thermal Coefficient Calculation

Linear temperature coefficient:

$$\omega(T) = \omega_0 + \chi T \tag{14}$$

where $\chi = \partial\omega/\partial T$ in cm^{-1}/K.

Non-linear model (including anharmonicity):

$$\omega(T) = \omega_0 + A\left(1 + \frac{2}{e^{\hbar\omega_0/2k_BT} - 1}\right) + B\left(1 + \frac{3}{e^{\hbar\omega_0/3k_BT} - 1}\right) \tag{15}$$

where A and B are anharmonic constants.

7 Complete Python Analysis Code

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Complete Raman Data Analysis for Graphene Hysteresis Measurements
Version: 2.0
Author: 2D Materials Characterization Lab
"""

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from scipy.optimize import curve_fit, minimize
from scipy import stats
from scipy.signal import savgol_filter
from scipy.integrate import trapz
from dataclasses import dataclass
from typing import Dict, List, Tuple, Optional
import warnings
import json
import os
from pathlib import Path

warnings.filterwarnings('ignore')

# ============================================================================
# Physical Constants
# ============================================================================

@dataclass
class PhysicalConstants:
    """Physical constants for Raman analysis"""
    hbar: float = 1.0545718e-34  # J s
    k_B: float = 1.380649e-23    # J/K
    c: float = 2.99792458e10     # cm/s
```

```
    h: float = 6.62607015e-34      # J s

    def cm_to_Hz(self, nu_cm: float) -> float:
        """Convert cm^-1 to Hz"""
        return nu_cm * self.c

    def phonon_energy(self, nu_cm: float) -> float:
        """Phonon energy in Joules"""
        return self.h * self.cm_to_Hz(nu_cm)

const = PhysicalConstants()

#
    ============================================================================

# Peak Fitting Functions
#
    ============================================================================


def lorentzian(x: np.ndarray, x0: float, gamma: float, area: float,
               background: float) -> np.ndarray:
    """
    Lorentzian peak function

    Parameters:
    -----------
    x : array
        Wavenumber values
    x0 : float
        Peak center
    gamma : float
        FWHM
    area : float
        Integrated area
    background : float
        Constant background

    Returns:
    --------
    array : Lorentzian intensity
    """
    return background + (area / np.pi) * (gamma/2) / ((x - x0)**2 + (gamma
       /2)**2)

def multi_lorentzian(x: np.ndarray, params: List[Tuple]) -> np.ndarray:
    """
    Sum of multiple Lorentzian peaks

    Parameters:
    -----------
    x : array
        Wavenumber values
    params : list of tuples
```

```
        List of (x0, gamma, area) for each peak

    Returns:
    --------
    array : Combined intensity
    """
    y = np.zeros_like(x, dtype=float)
    for x0, gamma, area, bg in params:
        y += lorentzian(x, x0, gamma, area, bg)
    return y

def gaussian(x: np.ndarray, x0: float, sigma: float, area: float,
             background: float) -> np.ndarray:
    """Gaussian peak function"""
    return background + (area/(sigma*np.sqrt(2*np.pi))) * \
           np.exp(-(x-x0)**2/(2*sigma**2))

def voigt(x: np.ndarray, x0: float, gamma_L: float, gamma_G: float,
          area: float, background: float) -> np.ndarray:
    """
    Voigt profile (convolution of Lorentzian and Gaussian)
    Approximated using pseudo-Voigt for computational efficiency
    """
    # Pseudo-Voigt approximation
    f_L = lorentzian(x, x0, gamma_L, area, 0)
    f_G = gaussian(x, x0, gamma_G/np.sqrt(8*np.log(2)), area, 0)

    # Mixing parameter
    eta = 0.5  # Fixed for simplicity

    return background + (1-eta)*f_G + eta*f_L

#
    ============================================================================

# Graphene Spectrum Analysis Class
#
    ============================================================================


class GrapheneRamanAnalyzer:
    """
    Complete analyzer for graphene Raman spectra
    """

    def __init__(self, spectral_range: Tuple[float, float] = (1200, 3200))
        :
        """
        Initialize analyzer

        Parameters:
        -----------
        spectral_range : tuple
            (min, max) wavenumber range in cm^-1
```

```
        """
        self.range = spectral_range
        self.fitting_results = {}

        # Reference values
        self.ref_G = 1580.0  # cm^-1
        self.ref_2D = 2680.0  # cm^-1
        self.ref_D = 1350.0   # cm^-1

    def load_spectrum(self, filename: str) -> Dict:
        """
        Load Raman spectrum from file

        Expected format:
        - First column: wavenumber (cm^-1)
        - Second column: intensity (counts)
        """
        data = np.loadtxt(filename)
        x = data[:, 0]
        y = data[:, 1]

        # Apply Savitzky-Golay filter for noise reduction
        y_smooth = savgol_filter(y, window_length=11, polyorder=3)

        return {'x': x, 'y_raw': y, 'y_smooth': y_smooth, 'filename':
            filename}

    def subtract_background(self, x: np.ndarray, y: np.ndarray,
                            method: str = 'linear') -> np.ndarray:
        """
        Subtract background from spectrum

        Methods:
        - 'linear': linear interpolation between endpoints
        - 'poly': polynomial fit to baseline regions
        - 'rubberband': rubberband baseline correction
        """
        if method == 'linear':
            # Linear baseline between min and max x
            y_baseline = np.interp(x, [x[0], x[-1]], [y[0], y[-1]])
            return y - y_baseline

        elif method == 'poly':
            # Fit polynomial to regions without peaks
            # Define peak-free regions
            mask = ((x < 1500) | ((x > 1600) & (x < 2600)) | (x > 2800))

            # Fit polynomial
            coeffs = np.polyfit(x[mask], y[mask], 3)
            y_baseline = np.polyval(coeffs, x)
            return y - y_baseline

        elif method == 'rubberband':
            # Rubberband baseline correction
```

```
            from scipy.spatial import ConvexHull

            # Find convex hull of spectrum
            points = np.column_stack([x, y])
            hull = ConvexHull(points)

            # Get lower envelope
            baseline_indices = hull.vertices[hull.vertices.argmin():] + \
                               [hull.vertices[0]]
            baseline_points = points[baseline_indices]

            # Interpolate baseline
            y_baseline = np.interp(x, baseline_points[:, 0],
                baseline_points[:, 1])
            return y - y_baseline

        else:
            raise ValueError(f"Unknown background method: {method}")

    def fit_G_peak(self, x: np.ndarray, y: np.ndarray,
                   fit_range: Tuple[float, float] = (1550, 1620)) -> Dict:
        """
        Fit G peak with Lorentzian

        Returns:
        --------
        dict with keys: position, fwhm, area, intensity, r_squared
        """
        # Select range
        mask = (x >= fit_range[0]) & (x <= fit_range[1])
        x_fit = x[mask]
        y_fit = y[mask]

        # Normalize for better fitting
        y_norm = y_fit / np.max(y_fit)

        # Initial guesses
        x0_init = x_fit[np.argmax(y_fit)]
        gamma_init = 15.0  # cm^-1
        area_init = np.trapz(y_norm, x_fit)
        bg_init = np.min(y_norm)

        # Fit
        try:
            popt, pcov = curve_fit(
                lorentzian, x_fit, y_norm,
                p0=[x0_init, gamma_init, area_init, bg_init],
                bounds=([x0_init-10, 1, 0, 0],
                        [x0_init+10, 50, np.inf, 0.5]),
                maxfev=5000
            )

            # Calculate R-squared
            y_fitted = lorentzian(x_fit, *popt)
```

```
            ss_res = np.sum((y_norm - y_fitted)**2)
            ss_tot = np.sum((y_norm - np.mean(y_norm))**2)
            r_squared = 1 - (ss_res / ss_tot)

            # Convert back to original scale
            scale_factor = np.max(y_fit)
            area_original = popt[2] * scale_factor

            return {
                'position': popt[0],
                'fwhm': popt[1],
                'area': area_original,
                'intensity': scale_factor,
                'background': popt[3],
                'r_squared': r_squared,
                'covariance': pcov,
                'fit_x': x_fit,
                'fit_y': y_fitted * scale_factor
            }

        except Exception as e:
            print(f"G peak fitting failed: {e}")
            return None

    def fit_2D_peak(self, x: np.ndarray, y: np.ndarray,
                    fit_range: Tuple[float, float] = (2600, 2800)) -> Dict
                        :
        """
        Fit 2D peak with 4 Lorentzian components

        Returns:
        --------
        dict with peak parameters for each component
        """
        # Select range
        mask = (x >= fit_range[0]) & (x <= fit_range[1])
        x_fit = x[mask]
        y_fit = y[mask]

        # Normalize
        y_norm = y_fit / np.max(y_fit)

        # Initial parameters for 4 peaks
        # Order: [x0_1, gamma_1, area_1, x0_2, gamma_2, area_2, ...]
        x0_init = [2640, 2660, 2680, 2700]
        gamma_init = [24, 24, 24, 24]
        area_init = [0.25, 0.5, 0.75, 1.0]  # Relative intensities
        bg_init = 0.0

        # Combine initial parameters
        p0 = []
        bounds_lower = []
        bounds_upper = []

```

```
        for i in range(4):
            p0.extend([x0_init[i], gamma_init[i], area_init[i]])
            bounds_lower.extend([x0_init[i]-20, 5, 0])
            bounds_upper.extend([x0_init[i]+20, 50, 5])
        p0.append(bg_init)
        bounds_lower.append(0)
        bounds_upper.append(0.5)

        def multi_lorentzian_fixed(x, *params):
            """Wrapper for multi-peak fitting"""
            peaks = []
            for i in range(4):
                x0 = params[3*i]
                gamma = params[3*i + 1]
                area = params[3*i + 2]
                peaks.append((x0, gamma, area, 0))
            bg = params[-1]

            y_sum = bg
            for x0, gamma, area, _ in peaks:
                y_sum += lorentzian(x, x0, gamma, area, 0)
            return y_sum

        # Fit
        try:
            popt, pcov = curve_fit(
                multi_lorentzian_fixed, x_fit, y_norm,
                p0=p0,
                bounds=(bounds_lower, bounds_upper),
                maxfev=10000
            )

            # Extract component parameters
            components = []
            total_intensity = 0
            for i in range(4):
                x0 = popt[3*i]
                gamma = popt[3*i + 1]
                area = popt[3*i + 2]
                components.append({
                    'position': x0,
                    'fwhm': gamma,
                    'area': area,
                    'intensity': lorentzian(x0, x0, gamma, area, 0)
                })
                total_intensity += area

            # Calculate R-squared
            y_fitted = multi_lorentzian_fixed(x_fit, *popt)
            ss_res = np.sum((y_norm - y_fitted)**2)
            ss_tot = np.sum((y_norm - np.mean(y_norm))**2)
            r_squared = 1 - (ss_res / ss_tot)

            # Calculate center of mass
```

```
            weights = [c['area'] for c in components]
            com = np.average([c['position'] for c in components], weights=
                weights)

            # Calculate overall FWHM
            half_max = np.max(y_fitted) / 2
            indices = np.where(y_fitted >= half_max)[0]
            if len(indices) > 0:
                fwhm_overall = x_fit[indices[-1]] - x_fit[indices[0]]
            else:
                fwhm_overall = 0

            return {
                'components': components,
                'center_of_mass': com,
                'fwhm_overall': fwhm_overall,
                'intensity_ratio': components[3]['area'] / components[0]['
                    area'],
                'r_squared': r_squared,
                'fit_x': x_fit,
                'fit_y': y_fitted * np.max(y_fit),
                'total_intensity': total_intensity * np.max(y_fit)
            }

        except Exception as e:
            print(f"2D peak fitting failed: {e}")
            return None

    def fit_D_peak(self, x: np.ndarray, y: np.ndarray,
                   fit_range: Tuple[float, float] = (1300, 1400)) -> Dict:
        """Fit D peak (defect peak)"""
        mask = (x >= fit_range[0]) & (x <= fit_range[1])
        x_fit = x[mask]
        y_fit = y[mask]

        # If no D peak, return zeros
        if np.max(y_fit) < 0.1 * np.max(y):
            return {
                'position': 0,
                'fwhm': 0,
                'area': 0,
                'intensity': 0,
                'present': False
            }

        y_norm = y_fit / np.max(y_fit)

        x0_init = x_fit[np.argmax(y_norm)]
        gamma_init = 20
        area_init = np.trapz(y_norm, x_fit)
        bg_init = np.min(y_norm)

        try:
            popt, _ = curve_fit(
```

```
                lorentzian, x_fit, y_norm,
                p0=[x0_init, gamma_init, area_init, bg_init],
                bounds=([x0_init-10, 5, 0, 0],
                        [x0_init+10, 50, np.inf, 0.5])
            )

            scale = np.max(y_fit)
            return {
                'position': popt[0],
                'fwhm': popt[1],
                'area': popt[2] * scale,
                'intensity': scale,
                'present': True
            }

        except Exception:
            return {'present': False}

    def analyze_spectrum(self, filename: str,
                         background_method: str = 'linear') -> Dict:
        """
        Complete analysis of graphene Raman spectrum
        """
        # Load spectrum
        data = self.load_spectrum(filename)
        x = data['x']
        y_raw = data['y_raw']

        # Subtract background
        y_corrected = self.subtract_background(x, y_raw, background_method
            )

        # Fit peaks
        g_result = self.fit_G_peak(x, y_corrected)
        twod_result = self.fit_2D_peak(x, y_corrected)
        d_result = self.fit_D_peak(x, y_corrected)

        # Calculate ratios
        if g_result and twod_result:
            I_G = g_result['intensity']
            I_2D = twod_result['total_intensity']
            I_D = d_result['intensity'] if d_result['present'] else 0

            ratio_2D_G = I_2D / I_G if I_G > 0 else 0
            ratio_D_G = I_D / I_G if I_G > 0 else 0

            # Determine layer number
            if ratio_2D_G > 2:
                layers = 1  # Monolayer
            elif ratio_2D_G > 1:
                layers = 2  # Bilayer
            else:
                layers = 3  # Few-layer
        else:
```

```
            ratio_2D_G = 0
            ratio_D_G = 0
            layers = -1

        result = {
            'filename': filename,
            'G_peak': g_result,
            '2D_peak': twod_result,
            'D_peak': d_result,
            'ratios': {
                'I2D/IG': ratio_2D_G,
                'ID/IG': ratio_D_G,
                'layers': layers
            },
            'x': x,
            'y_raw': y_raw,
            'y_corrected': y_corrected
        }

        self.fitting_results[filename] = result
        return result

    def plot_spectrum(self, filename: str, save: bool = False):
        """Plot spectrum with fits"""
        if filename not in self.fitting_results:
            print(f"Analyze {filename} first")
            return

        res = self.fitting_results[filename]

        fig, axes = plt.subplots(2, 2, figsize=(12, 10))

        # Full spectrum
        ax = axes[0, 0]
        ax.plot(res['x'], res['y_raw'], 'k-', alpha=0.5, label='Raw')
        ax.plot(res['x'], res['y_corrected'], 'b-', label='Background
            corrected')
        ax.set_xlabel('Raman shift (cm$^{-1}$)')
        ax.set_ylabel('Intensity (counts)')
        ax.set_title('Full Spectrum')
        ax.legend()
        ax.grid(True, alpha=0.3)

        # G peak
        ax = axes[0, 1]
        if res['G_peak']:
            g = res['G_peak']
            ax.plot(g['fit_x'], g['fit_y'], 'r-', linewidth=2, label='Fit'
                )
            ax.axvline(g['position'], color='r', linestyle='--', alpha
                =0.5)
            ax.text(0.05, 0.95, f"G: {g['position']:.1f} cm$^{-1}$\nFWHM:
                {g['fwhm']:.1f} cm$^{-1}$",
                    transform=ax.transAxes, verticalalignment='top')
```

```
        ax.set_xlabel('Raman shift (cm$^{-1}$)')
        ax.set_ylabel('Intensity')
        ax.set_title('G Peak')
        ax.grid(True, alpha=0.3)

        # 2D peak
        ax = axes[1, 0]
        if res['2D_peak']:
            td = res['2D_peak']
            ax.plot(td['fit_x'], td['fit_y'], 'g-', linewidth=2, label='Fit')
            if 'components' in td:
                colors = ['c', 'm', 'y', 'orange']
                for i, comp in enumerate(td['components']):
                    y_comp = lorentzian(td['fit_x'], comp['position'],
                                        comp['fwhm'], comp['area'], 0)
                    ax.plot(td['fit_x'], y_comp, '--', color=colors[i],
                            alpha=0.5)
            ax.axvline(td['center_of_mass'], color='g', linestyle='--',
                       alpha=0.5)
            ax.text(0.05, 0.95, f"2D COM: {td['center_of_mass']:.1f} cm$
                ^{-1}$\nFWHM: {td['fwhm_overall']:.1f} cm$^{-1}$",
                    transform=ax.transAxes, verticalalignment='top')
        ax.set_xlabel('Raman shift (cm$^{-1}$)')
        ax.set_ylabel('Intensity')
        ax.set_title('2D Peak')
        ax.grid(True, alpha=0.3)

        # Ratios
        ax = axes[1, 1]
        ratios = res['ratios']
        bars = ax.bar(['I$_{2D}$/I$_G$', 'I$_D$/I$_G$'],
                      [ratios['I2D/IG'], ratios['ID/IG']],
                      color=['green', 'red'])
        ax.axhline(2, color='g', linestyle='--', alpha=0.5, label='
            Monolayer threshold')
        ax.axhline(0.1, color='r', linestyle='--', alpha=0.5, label='
            Defect threshold')
        ax.set_ylabel('Ratio')
        ax.set_title('Intensity Ratios')
        ax.legend()
        ax.grid(True, alpha=0.3)

        plt.tight_layout()

        if save:
            plt.savefig(filename.replace('.txt', '_analysis.png'), dpi
                =300)
        plt.show()

#
    ================================================================================

# Temperature-Dependent Analysis
```

```
# ============================================================================


class TemperatureRamanAnalyzer:
    """
    Analyzer for temperature-dependent Raman measurements
    """

    def __init__(self):
        self.data = pd.DataFrame()
        self.analyzer = GrapheneRamanAnalyzer()

    def load_experiment_data(self, data_dir: str) -> pd.DataFrame:
        """
        Load all temperature-dependent spectra from directory

        Expected file naming:
        - Heating: H_300K.txt, H_350K.txt, H_400K.txt, ...
        - Cooling: C_500K.txt, C_450K.txt, C_400K.txt, ...
        """
        data_dir = Path(data_dir)
        results = []

        for filepath in data_dir.glob('*.txt'):
            filename = filepath.name

            # Parse filename
            if filename.startswith('H_'):
                cycle = 'heating'
                T_str = filename[2:].replace('K.txt', '')
            elif filename.startswith('C_'):
                cycle = 'cooling'
                T_str = filename[2:].replace('K.txt', '')
            else:
                continue

            try:
                T = float(T_str)

                # Analyze spectrum
                result = self.analyzer.analyze_spectrum(str(filepath))

                # Extract parameters
                row = {
                    'filename': filename,
                    'cycle': cycle,
                    'temperature': T,
                    'T_set': T,
                    'G_position': result['G_peak']['position'] if result['
                        G_peak'] else np.nan,
                    'G_fwhm': result['G_peak']['fwhm'] if result['G_peak']
                        else np.nan,
```

```
                    'G_intensity': result['G_peak']['intensity'] if result
                        ['G_peak'] else np.nan,
                    '2D_position': result['2D_peak']['center_of_mass'] if
                        result['2D_peak'] else np.nan,
                    '2D_fwhm': result['2D_peak']['fwhm_overall'] if result
                        ['2D_peak'] else np.nan,
                    '2D_intensity': result['2D_peak']['total_intensity']
                        if result['2D_peak'] else np.nan,
                    'ID_IG': result['ratios']['ID/IG'],
                    'I2D_IG': result['ratios']['I2D/IG'],
                }

                # Calculate local temperature if anti-Stokes available
                as_file = filepath.parent / filename.replace('.txt', '_AS.
                    txt')
                if as_file.exists():
                    T_local = self.calculate_local_temperature(str(
                        filepath), str(as_file))
                    row['T_local'] = T_local
                else:
                    row['T_local'] = np.nan

                results.append(row)

            except Exception as e:
                print(f"Error processing {filename}: {e}")

        self.data = pd.DataFrame(results)
        self.data = self.data.sort_values(['cycle', 'temperature'])
        return self.data

    def calculate_local_temperature(self, stokes_file: str,
                                    anti_stokes_file: str) -> float:
        """
        Calculate local temperature from Stokes/Anti-Stokes ratio
        """
        # Load spectra
        stokes = self.analyzer.load_spectrum(stokes_file)
        anti_stokes = self.analyzer.load_spectrum(anti_stokes_file)

        # Find G peak in both
        stokes_result = self.analyzer.fit_G_peak(stokes['x'], stokes['
            y_smooth'])
        as_result = self.analyzer.fit_G_peak(anti_stokes['x'], anti_stokes
            ['y_smooth'])

        if stokes_result is None or as_result is None:
            return np.nan

        # Calculate ratio
        I_S = stokes_result['intensity']
        I_AS = as_result['intensity']
        nu_S = stokes_result['position']
        nu_AS = as_result['position']
```

```
641
642         if I_S <= 0 or I_AS <= 0:
643             return np.nan
644
645         ratio = I_AS / I_S
646         nu_ratio = (nu_AS / nu_S) ** 4
647
648         # Phonon energy
649         omega = const.phonon_energy(nu_S)  # Joules
650
651         # Calculate temperature
652         try:
653             T_local = -omega / (const.k_B * np.log(ratio / nu_ratio))
654             return T_local
655         except:
656             return np.nan
657
658     def linear_fit(self, x: np.ndarray, y: np.ndarray) -> Tuple:
659         """Linear regression with uncertainties"""
660         mask = ~(np.isnan(x) | np.isnan(y))
661         x_clean = x[mask]
662         y_clean = y[mask]
663
664         if len(x_clean) < 2:
665             return None
666
667         # Linear regression
668         slope, intercept, r_value, p_value, std_err = stats.linregress(
                x_clean, y_clean)
669
670         return {
671             'slope': slope,
672             'intercept': intercept,
673             'r_squared': r_value**2,
674             'p_value': p_value,
675             'std_err': std_err
676         }
677
678     def calculate_hysteresis(self) -> pd.DataFrame:
679         """
680         Calculate hysteresis between heating and cooling cycles
681         """
682         # Separate cycles
683         heating = self.data[self.data['cycle'] == 'heating'].sort_values('
                temperature')
684         cooling = self.data[self.data['cycle'] == 'cooling'].sort_values('
                temperature')
685
686         # Common temperatures
687         common_T = sorted(set(heating['temperature']).intersection(set(
                cooling['temperature'])))
688
689         hysteresis = []
690         for T in common_T:
```

```
            h_row = heating[heating['temperature'] == T].iloc[0] if len(
                heating[heating['temperature'] == T]) > 0 else None
            c_row = cooling[cooling['temperature'] == T].iloc[0] if len(
                cooling[cooling['temperature'] == T]) > 0 else None

            if h_row is not None and c_row is not None:
                # G peak hysteresis
                delta_G_pos = h_row['G_position'] - c_row['G_position']
                delta_G_fwhm = h_row['G_fwhm'] - c_row['G_fwhm']

                # 2D peak hysteresis
                delta_2D_pos = h_row['2D_position'] - c_row['2D_position']
                delta_2D_fwhm = h_row['2D_fwhm'] - c_row['2D_fwhm']

                # Normalized hysteresis
                if h_row['G_position'] != 0:
                    norm_G = abs(delta_G_pos) / abs(h_row['G_position'])
                else:
                    norm_G = np.nan

                hysteresis.append({
                    'temperature': T,
                    'delta_G_position': delta_G_pos,
                    'delta_G_fwhm': delta_G_fwhm,
                    'delta_2D_position': delta_2D_pos,
                    'delta_2D_fwhm': delta_2D_fwhm,
                    'norm_G_hysteresis': norm_G,
                    'G_heating': h_row['G_position'],
                    'G_cooling': c_row['G_position'],
                    '2D_heating': h_row['2D_position'],
                    '2D_cooling': c_row['2D_position']
                })

        return pd.DataFrame(hysteresis)

    def calculate_loop_area(self, hysteresis_df: pd.DataFrame,
                            parameter: str = 'delta_G_position') -> Dict:
        """
        Calculate hysteresis loop area using trapezoidal rule
        """
        if len(hysteresis_df) < 2:
            return {'area': 0, 'uncertainty': 0}

        T = hysteresis_df['temperature'].values
        delta = hysteresis_df[parameter].values

        # Absolute values for area
        delta_abs = np.abs(delta)

        # Trapezoidal rule
        area = 0
        area_uncertainty = 0

        for i in range(len(T)-1):
```

```
            dT = T[i+1] - T[i]
            avg_delta = (delta_abs[i] + delta_abs[i+1]) / 2
            area += avg_delta * dT

            # Simple uncertainty estimate (assuming 0.5 cm^-1 error in
               delta)
            uncertainty_term = dT * np.sqrt(0.5**2 + 0.5**2) / 2
            area_uncertainty += uncertainty_term

        return {
            'area': area,
            'uncertainty': area_uncertainty,
            'units': 'cm$^{-1}$ K',
            'parameter': parameter
        }

    def plot_temperature_dependence(self, save: bool = False):
        """
        Plot temperature dependence of Raman parameters
        """
        fig, axes = plt.subplots(2, 3, figsize=(15, 10))

        # Separate cycles
        heating = self.data[self.data['cycle'] == 'heating']
        cooling = self.data[self.data['cycle'] == 'cooling']

        # G peak position
        ax = axes[0, 0]
        ax.plot(heating['temperature'], heating['G_position'],
                'ro-', label='Heating', markersize=8, linewidth=2)
        ax.plot(cooling['temperature'], cooling['G_position'],
                'bo-', label='Cooling', markersize=8, linewidth=2)
        ax.set_xlabel('Temperature (K)')
        ax.set_ylabel('G peak position (cm$^{-1}$)')
        ax.set_title('G Peak Temperature Dependence')
        ax.legend()
        ax.grid(True, alpha=0.3)

        # G peak FWHM
        ax = axes[0, 1]
        ax.plot(heating['temperature'], heating['G_fwhm'],
                'ro-', label='Heating', markersize=8, linewidth=2)
        ax.plot(cooling['temperature'], cooling['G_fwhm'],
                'bo-', label='Cooling', markersize=8, linewidth=2)
        ax.set_xlabel('Temperature (K)')
        ax.set_ylabel('G peak FWHM (cm$^{-1}$)')
        ax.set_title('G Peak Linewidth')
        ax.legend()
        ax.grid(True, alpha=0.3)

        # 2D peak position
        ax = axes[0, 2]
        ax.plot(heating['temperature'], heating['2D_position'],
                'ro-', label='Heating', markersize=8, linewidth=2)
```

```
        ax.plot(cooling['temperature'], cooling['2D_position'],
                'bo-', label='Cooling', markersize=8, linewidth=2)
        ax.set_xlabel('Temperature (K)')
        ax.set_ylabel('2D peak position (cm$^{-1}$)')
        ax.set_title('2D Peak Temperature Dependence')
        ax.legend()
        ax.grid(True, alpha=0.3)

        # 2D peak FWHM
        ax = axes[1, 0]
        ax.plot(heating['temperature'], heating['2D_fwhm'],
                'ro-', label='Heating', markersize=8, linewidth=2)
        ax.plot(cooling['temperature'], cooling['2D_fwhm'],
                'bo-', label='Cooling', markersize=8, linewidth=2)
        ax.set_xlabel('Temperature (K)')
        ax.set_ylabel('2D peak FWHM (cm$^{-1}$)')
        ax.set_title('2D Peak Linewidth')
        ax.legend()
        ax.grid(True, alpha=0.3)

        # I2D/IG ratio
        ax = axes[1, 1]
        ax.plot(heating['temperature'], heating['I2D_IG'],
                'ro-', label='Heating', markersize=8, linewidth=2)
        ax.plot(cooling['temperature'], cooling['I2D_IG'],
                'bo-', label='Cooling', markersize=8, linewidth=2)
        ax.axhline(2, color='g', linestyle='--', alpha=0.5, label='
           Monolayer threshold')
        ax.set_xlabel('Temperature (K)')
        ax.set_ylabel('I$_{2D}$/I$_G$')
        ax.set_title('Intensity Ratio')
        ax.legend()
        ax.grid(True, alpha=0.3)

        # ID/IG ratio (defect)
        ax = axes[1, 2]
        ax.plot(heating['temperature'], heating['ID_IG'],
                'ro-', label='Heating', markersize=8, linewidth=2)
        ax.plot(cooling['temperature'], cooling['ID_IG'],
                'bo-', label='Cooling', markersize=8, linewidth=2)
        ax.axhline(0.1, color='r', linestyle='--', alpha=0.5, label='
           Defect threshold')
        ax.set_xlabel('Temperature (K)')
        ax.set_ylabel('I$_D$/I$_G$')
        ax.set_title('Defect Ratio')
        ax.legend()
        ax.grid(True, alpha=0.3)

        plt.tight_layout()

        if save:
            plt.savefig('temperature_dependence.png', dpi=300)
        plt.show()

```

```
    def plot_hysteresis(self, save: bool = False):
        """
        Plot hysteresis loops
        """
        hysteresis = self.calculate_hysteresis()

        if len(hysteresis) == 0:
            print("No hysteresis data available")
            return

        fig, axes = plt.subplots(1, 2, figsize=(12, 5))

        # G peak hysteresis
        ax = axes[0]
        ax.plot(hysteresis['temperature'], hysteresis['delta_G_position'],
                'ks-', markersize=8, linewidth=2)
        ax.axhline(0, color='gray', linestyle='--', alpha=0.5)
        ax.fill_between(hysteresis['temperature'], 0, hysteresis['
            delta_G_position'],
                        alpha=0.3, color='red', where=hysteresis['
                            delta_G_position']>0,
                        label='Positive hysteresis')
        ax.fill_between(hysteresis['temperature'], 0, hysteresis['
            delta_G_position'],
                        alpha=0.3, color='blue', where=hysteresis['
                            delta_G_position']<0,
                        label='Negative hysteresis')
        ax.set_xlabel('Temperature (K)')
        ax.set_ylabel('  $_  {G}$ (cm$^{-1}$)')
        ax.set_title('G Peak Hysteresis')
        ax.legend()
        ax.grid(True, alpha=0.3)

        # 2D peak hysteresis
        ax = axes[1]
        ax.plot(hysteresis['temperature'], hysteresis['delta_2D_position'
            ],
                'ks-', markersize=8, linewidth=2)
        ax.axhline(0, color='gray', linestyle='--', alpha=0.5)
        ax.fill_between(hysteresis['temperature'], 0, hysteresis['
            delta_2D_position'],
                        alpha=0.3, color='red', where=hysteresis['
                            delta_2D_position']>0,
                        label='Positive hysteresis')
        ax.fill_between(hysteresis['temperature'], 0, hysteresis['
            delta_2D_position'],
                        alpha=0.3, color='blue', where=hysteresis['
                            delta_2D_position']<0,
                        label='Negative hysteresis')
        ax.set_xlabel('Temperature (K)')
        ax.set_ylabel('  $_  {2D}$ (cm$^{-1}$)')
        ax.set_title('2D Peak Hysteresis')
        ax.legend()
        ax.grid(True, alpha=0.3)
```

```
        plt.tight_layout()

        if save:
            plt.savefig('hysteresis_loops.png', dpi=300)
        plt.show()

        # Calculate loop areas
        area_G = self.calculate_loop_area(hysteresis, 'delta_G_position')
        area_2D = self.calculate_loop_area(hysteresis, 'delta_2D_position')

        print(f"\n=== Hysteresis Loop Areas ===")
        print(f"G peak: {area_G['area']:.2f}    {area_G['uncertainty']:.2f} {area_G['units']}")
        print(f"2D peak: {area_2D['area']:.2f}    {area_2D['uncertainty']:.2f} {area_2D['units']}")

        return hysteresis

    def fit_temperature_coefficients(self) -> Dict:
        """
        Fit temperature coefficients for heating and cooling
        """
        coefficients = {}

        for cycle in ['heating', 'cooling']:
            cycle_data = self.data[self.data['cycle'] == cycle].sort_values('temperature')

            if len(cycle_data) < 3:
                continue

            # G peak temperature coefficient
            T = cycle_data['temperature'].values
            G_pos = cycle_data['G_position'].values

            fit_G = self.linear_fit(T, G_pos)
            if fit_G:
                coefficients[f'{cycle}_G_slope'] = fit_G['slope']
                coefficients[f'{cycle}_G_intercept'] = fit_G['intercept']
                coefficients[f'{cycle}_G_r2'] = fit_G['r_squared']

            # 2D peak temperature coefficient
            pos_2D = cycle_data['2D_position'].values
            fit_2D = self.linear_fit(T, pos_2D)
            if fit_2D:
                coefficients[f'{cycle}_2D_slope'] = fit_2D['slope']
                coefficients[f'{cycle}_2D_intercept'] = fit_2D['intercept']
                coefficients[f'{cycle}_2D_r2'] = fit_2D['r_squared']

        return coefficients
```

```
    def generate_report(self, output_dir: str = 'analysis_results'):
        """
        Generate complete analysis report
        """
        os.makedirs(output_dir, exist_ok=True)

        # Save raw data
        self.data.to_csv(f'{output_dir}/raman_data.csv', index=False)

        # Calculate hysteresis
        hysteresis = self.calculate_hysteresis()
        if len(hysteresis) > 0:
            hysteresis.to_csv(f'{output_dir}/hysteresis.csv', index=False)

        # Calculate temperature coefficients
        coefficients = self.fit_temperature_coefficients()

        # Calculate loop areas
        area_G = self.calculate_loop_area(hysteresis, 'delta_G_position')
        area_2D = self.calculate_loop_area(hysteresis, 'delta_2D_position')

        # Generate report
        with open(f'{output_dir}/analysis_report.txt', 'w') as f:
            f.write("="*60 + "\n")
            f.write("RAMAN HYSTERESIS ANALYSIS REPORT\n")
            f.write("="*60 + "\n\n")

            f.write(f"Number of spectra analyzed: {len(self.data)}\n")
            f.write(f"Temperature range: {self.data['temperature'].min()
                :.0f} - {self.data['temperature'].max():.0f} K\n\n")

            f.write("Temperature Coefficients:\n")
            f.write("-"*40 + "\n")
            for key, value in coefficients.items():
                f.write(f"{key}: {value:.6f}\n")
            f.write("\n")

            f.write("Hysteresis Loop Areas:\n")
            f.write("-"*40 + "\n")
            f.write(f"G peak: {area_G['area']:.2f}    {area_G['uncertainty
                ']:.2f} {area_G['units']}\n")
            f.write(f"2D peak: {area_2D['area']:.2f}    {area_2D['
                uncertainty']:.2f} {area_2D['units']}\n\n")

            f.write("Quality Metrics:\n")
            f.write("-"*40 + "\n")
            f.write(f"Mean ID/IG ratio: {self.data['ID_IG'].mean():.4f}\n"
                )
            f.write(f"Max ID/IG ratio: {self.data['ID_IG'].max():.4f}\n")
            f.write(f"Mean I2D/IG ratio: {self.data['I2D_IG'].mean():.2f}\
                n\n")

            f.write("Sample Integrity:\n")
```

```
            f.write("-"*40 + "\n")
            initial_G = self.data[self.data['temperature'] == 300].iloc
                [0]['G_position'] if len(self.data[self.data['temperature']
                 == 300]) > 0 else 0
            final_G = self.data[self.data['temperature'] == 300].iloc[-1][
                'G_position'] if len(self.data[self.data['temperature'] ==
                300]) > 1 else 0
            f.write(f"Initial G position: {initial_G:.2f} cm^-1\n")
            f.write(f"Final G position: {final_G:.2f} cm^-1\n")
            f.write(f"Drift: {final_G - initial_G:.2f} cm^-1\n")

        print(f"Report generated in {output_dir}/")

        # Generate plots
        self.plot_temperature_dependence(save=True)
        self.plot_hysteresis(save=True)

        return {
            'data': self.data,
            'hysteresis': hysteresis,
            'coefficients': coefficients,
            'area_G': area_G,
            'area_2D': area_2D
        }

#
    ================================================================================

# Example Usage
#
    ================================================================================


def main():
    """Example usage of the Raman analysis package"""

    print("="*60)
    print("Graphene Raman Hysteresis Analysis")
    print("="*60)

    # Initialize analyzer
    analyzer = TemperatureRamanAnalyzer()

    # Load data (replace with actual data directory)
    data_dir = "./raman_data"  # Change this to your data directory

    if os.path.exists(data_dir):
        print(f"\nLoading data from {data_dir}...")
        analyzer.load_experiment_data(data_dir)

        print(f"\nLoaded {len(analyzer.data)} spectra")
        print("\nSummary:")
        print(analyzer.data.groupby('cycle')['temperature'].agg(['min', '
            max', 'count']))
```

```
        # Generate complete report
        results = analyzer.generate_report()

        # Display key results
        print("\n" + "="*60)
        print("KEY RESULTS")
        print("="*60)

        # Temperature coefficients
        print("\nTemperature Coefficients:")
        for key in ['heating_G_slope', 'cooling_G_slope']:
            if key in results['coefficients']:
                print(f"{key}: {results['coefficients'][key]:.4f} cm^-1/K"
                    )

        # Hysteresis areas
        print(f"\nHysteresis Loop Areas:")
        print(f"G peak: {results['area_G']['area']:.2f}    {results['
            area_G']['uncertainty']:.2f} cm^-1 K ")
        print(f"2D peak: {results['area_2D']['area']:.2f}    {results['
            area_2D']['uncertainty']:.2f} cm^-1 K ")

        # Quality check
        print("\nQuality Check:")
        if results['data']['ID_IG'].max() < 0.1:
            print("    Defect ratio within specification")
        else:
            print("    Defect ratio exceeds specification")

        initial_G = results['data'][results['data']['temperature'] ==
            300].iloc[0]['G_position']
        final_G = results['data'][results['data']['temperature'] == 300].
            iloc[-1]['G_position']
        drift = final_G - initial_G

        if abs(drift) < 0.5:
            print("    Sample integrity maintained")
        else:
            print(f"    Sample degraded (drift: {drift:.2f} cm^-1)")

    else:
        print(f"\nData directory {data_dir} not found.")
        print("Please create directory and add Raman spectra with naming:"
            )
        print("  - Heating: H_300K.txt, H_350K.txt, ...")
        print("  - Cooling: C_500K.txt, C_450K.txt, ...")
        print("  - Anti-Stokes: H_300K_AS.txt, C_500K_AS.txt, ...")

if __name__ == "__main__":
    main()
```

Listing 1: Complete Raman Data Analysis Package

8 Data Recording Template

8.1 Experiment Metadata Form

8.2 Temperature Log Sheet

9 Quality Control and Acceptance Criteria

9.1 Quantitative Acceptance Criteria

9.2 Measurement Uncertainty Budget

10 Expected Results and Interpretation

10.1 Typical Values for Monolayer Graphene

10.2 Hysteresis Interpretation Guidelines

- $\Delta\omega_{\mathbf{hyst}} > 0$: Cooling cycle has lower frequency than heating (red-shifted on cooling)
- $\Delta\omega_{\mathbf{hyst}} < 0$: Cooling cycle has higher frequency than heating (blue-shifted on cooling)
- **Hysteresis area** > 50 $\mathbf{cm^{-1} \cdot K}$: Significant non-reciprocal behavior
- **Hysteresis area** < 10 $\mathbf{cm^{-1} \cdot K}$: Within experimental uncertainty

11 Data Archive Requirements

11.1 Folder Structure

```
experiment_YYYYMMDD/
          raw_data/
                heating/
                      H_300K.txt
                      H_300K_AS.txt
                      H_350K.txt
                      H_350K_AS.txt
                      ...
                cooling/
                      C_500K.txt
                      C_500K_AS.txt
                      C_450K.txt
                      C_450K_AS.txt
                      ...
                mapping/
                    map_5x5_grid.txt
                    map_coordinates.txt
          calibration/
                Si_reference.txt
                power_calibration.txt
                temperature_calibration.txt
          analysis/
                raman_data.csv
                hysteresis.csv
```

```
                fitting_results.json
                analysis_report.txt
        figures/
                temperature_dependence.png
                hysteresis_loops.png
                g_peak_fits.png
                2d_peak_fits.png
                mapping_results.png
        scripts/
                raman_analysis.py
                requirements.txt
                README.md
        metadata/
                experiment_log.pdf
                sample_info.pdf
                instrument_config.pdf
        README.md
```

Listing 2: Required data archive structure

11.2 README Template

```
# Raman Hysteresis Experiment: Graphene Monolayer

## Experiment Information
- Date: YYYY-MM-DD
- Operator: [Name]
- Sample ID: [ID]
- Experiment ID: [ID]

## Sample Description
- Growth method: CVD on Cu
- Transfer method: PMMA-assisted wet transfer
- Substrate: 300 nm SiO2/Si
- Quality: I2D/IG > 2, ID/IG < 0.05 at RT

## Instrument Configuration
- Raman system: [Model]
- Laser wavelength: 532 nm
- Laser power: 0.5 mW (at sample)
- Grating: 1800 gr/mm
- Acquisition: 3 x 60 s per spectrum
- Temperature stage: Linkam THMS600
- Atmosphere: Ambient

## Temperature Protocol
- Range: 300 K -> 500 K -> 300 K
- Ramp rate: 10 K/min
- Stabilization: 5 min at each set point
- Set points: 300, 350, 400, 450, 500 K

## Data Files
- raw_data/: All raw Raman spectra (TXT format)
```

```
- calibration/: Calibration measurements
- analysis/: Processed data and results
- figures/: Publication-ready figures
- scripts/: Analysis code

## Analysis Software
- Python 3.9+
- Required packages: numpy, scipy, pandas, matplotlib
- See scripts/requirements.txt

## Reproduction Instructions
1. Install Python dependencies: pip install -r scripts/requirements.txt
2. Run analysis: python scripts/raman_analysis.py
3. Check analysis/analysis_report.txt for results

## Quality Metrics
- Si calibration: [value] cm^-1
- Laser stability: [value]% variation
- Sample integrity: [Pass/Fail]
- Defect ratio: [value] (max)

## Results Summary
- G temperature coefficient (heating): [value] cm^-1/K
- G temperature coefficient (cooling): [value] cm^-1/K
- Hysteresis area (G): [value] cm^-1 K
- Hysteresis area (2D): [value] cm^-1 K

## Contact
[Name]
[Institution]
[Email]
```

Listing 3: README.md template

11.3 Software Environment Specification

```
numpy==1.24.3
scipy==1.10.1
pandas==2.0.2
matplotlib==3.7.1
seaborn==0.12.2
lmfit==1.2.2
peakutils==1.3.4
h5py==3.9.0
tqdm==4.65.0
pyyaml==6.0
jupyter==1.0.0
```

Listing 4: requirements.txt

12 Troubleshooting Guide

12.1 Common Issues and Solutions

13 Conclusion

This SOP provides a complete, reproducible protocol for temperature-dependent Raman measurements of monolayer graphene with heating-cooling hysteresis analysis. All parameters, procedures, and analysis code are fully specified to ensure identical results across different laboratories and experimental setups.

The key features ensuring reproducibility are:

- Complete instrument specifications with model numbers
- Step-by-step procedures with exact timing
- Full calibration protocols with acceptance criteria
- Complete Python analysis code with all functions
- Data archive structure with README template
- Troubleshooting guide for common issues
- Expected results for quality verification

Document Control

Version:	2.0
Date:	March 5, 2026
Author:	2D Materials Characterization Laboratory
Reviewers:	[Names]
Approved by:	[Name]
Next review:	March 2027

Component	Specification
Laser System	
Laser type	Diode-pumped solid-state (DPSS)
Wavelength	532 nm (frequency-doubled Nd:YAG)
Linewidth	< 0.1 nm
Power stability	$< 0.5\%$ over 8 hours
Polarization	Linear (horizontal)
Microscope	
Objective	50× long working distance
Numerical aperture (NA)	0.75
Working distance	1.0 mm
Correction	Plan apochromatic
Spectrometer	
Type	Czerny-Turner with triple grating turret
Focal length	750 mm
Grating 1	600 grooves/mm (low resolution)
Grating 2	1800 grooves/mm (high resolution)
Grating 3	2400 grooves/mm (ultra-high resolution)
Spectral range	1200-3200 cm^{-1}
Spectral resolution	≤ 1.5 cm^{-1} (with 1800 gr/mm)
Detector	
Type	Back-illuminated CCD
Cooling	Liquid nitrogen to 140 K
Pixel array	1340×400 pixels
Pixel size	20×20 μm
Quantum efficiency	$> 90\%$ at 532 nm
Dark current	< 0.001 e^-/pixel/hour
Temperature Stage	
Model	Linkam THMS600
Temperature range	300-500 K (27-227°C)
Heating/cooling rate	0.1-50 K/min
Temperature stability	± 0.1 K
Temperature accuracy	± 0.5 K
Thermocouple	Type K (calibrated)
Window material	Quartz (UV-grade)
Atmosphere control	Vacuum or gas purge

Table 3: Complete Raman system specifications

Item	Specification	Supplier
Silver paste	High conductivity, fast curing	Ted Pella
Liquid nitrogen	99.999% purity	Local supplier
Calibration lamp	Neon/Argon spectral calibration	Ocean Optics
Silicon reference	Undoped, (100) orientation	University Wafer
Quartz window	UV-grade, 1 mm thickness	Thorlabs
Copper tape	Conductive, 5 mm width	3M
Kapton tape	High temperature resistant	DuPont

Table 4: Consumables and supplies

Set Point (mW)	Measured Power (mW)	Deviation (%)
0.1	[measure]	[calculate]
0.3	[measure]	[calculate]
0.5	[measure]	[calculate]
0.7	[measure]	[calculate]
1.0	[measure]	[calculate]

Table 5: Laser power calibration table

Standard	True Melting Point (K)	Measured (K)
Gallium	302.91	[measure]
Indium	429.75	[measure]
Tin	505.08	[measure]

Table 6: Temperature calibration verification

Hour	Phase	Duration
0-1	Room-temperature baseline	60 min
1-2	Heating ramp (300 K $\rightarrow$ 500 K)	60 min
2-3	High-temperature hold (500 K)	60 min
3-4	Cooling ramp (500 K $\rightarrow$ 300 K)	60 min
4-5	Post-cycle verification	60 min

Table 7: Experimental timeline

Set Point (K)	Ramp Rate (K/min)	Stabilization (min)	Measurement (min)	Cumulative Tim
300	0	5	15	20
350	10	5	15	40
400	10	5	15	60
450	10	5	15	80
500	10	5	15	100

Table 8: Heating ramp schedule

Set Point (K)	Ramp Rate (K/min)	Stabilization (min)	Measurement (min)	Cumulative Tim
500	0	5	15	20
450	-10	5	15	40
400	-10	5	15	60
350	-10	5	15	80
300	-10	5	15	100

Table 9: Cooling ramp schedule

Component	ν_0 (cm^{-1})	Γ (cm^{-1})	Relative intensity
$2D_1$	2640	24	0.25
$2D_2$	2660	24	0.50
$2D_3$	2680	24	0.75
$2D_4$	2700	24	1.00

Table 10: Initial parameters for 2D band fitting

Field	**Entry**
Experiment ID	
Date	
Operator	
Sample Information	
Sample ID	
Growth method	
Transfer method	
Substrate	
Instrument Settings	
Laser wavelength (nm)	
Laser power (mW)	
Grating (grooves/mm)	
Slit width (μm)	
Pinhole (μm)	
Acquisition time (s)	
Accumulations	
Temperature Control	
Stage model	
Atmosphere	
Heating rate (K/min)	
Cooling rate (K/min)	
Stabilization time (min)	
Calibration	
Si reference peak (cm^{-1})	
Calibration drift (cm^{-1})	
Laser power stability (%)	

Table 11: Experiment metadata recording form

Time	Cycle	T_set (K)	T_actual (K)	File Name	Notes

Table 12: Temperature log sheet

Condition	Requirement	Verification Method
Si calibration	± 0.5 cm^{-1}	Before and after experiment
Post-cycle peak shift	< 0.5 cm^{-1}	Compare initial and final spectra
Defect ratio	$I_D/I_G < 0.1$	Throughout experiment
Local temperature accuracy	$\|T_{\text{local}} - T_{\text{stage}}\| \leq 5$ K	Anti-Stokes/Stokes ratio
Sample integrity	No visible damage	Optical inspection
Laser power stability	$< 5\%$ variation	Measure before/after each set point
Temperature stability	± 0.5 K at set point	Stage thermocouple
Spatial uniformity	$< 5\%$ variation	Raman mapping post-experiment
Hysteresis signal	$> 3\times$ measurement uncertainty	Compare to noise level

Table 13: Quality control acceptance criteria

Source	Uncertainty (cm^{-1})	Notes
Spectral calibration	± 0.2	From Si reference
Peak fitting	± 0.1	95% confidence interval
Temperature control	± 0.3	Converted from ± 0.5 K using χ
Laser heating	± 0.2	From power dependence test
Spatial variation	± 0.2	From mapping data
Total combined	± 0.5	Root sum square

Table 14: Measurement uncertainty budget for G peak position

Parameter	300 K	500 K
G peak position (cm^{-1})	1580 ± 2	1575 ± 2
G peak FWHM (cm^{-1})	15 ± 2	25 ± 3
2D peak position (cm^{-1})	2680 ± 5	2665 ± 5
2D peak FWHM (cm^{-1})	30 ± 3	45 ± 5
I_{2D}/I_G ratio	> 2.5	> 2.0
I_D/I_G ratio	< 0.05	< 0.1
Temperature coefficient χ_G (cm^{-1}/K)	-0.016 ± 0.002	N/A
Temperature coefficient χ_{2D} (cm^{-1}/K)	-0.032 ± 0.004	N/A

Table 15: Expected Raman parameters for monolayer graphene

Problem	**Likely Cause**	**Solution**
Weak Raman signal	Laser misalignment	Realign using pinhole method
	Dirty optics	Clean all optical surfaces
	Wrong focus	Re-auto-focus at each temperature
Peak shift ¿ 0.5 cm^{-1} in baseline	Laser heating	Reduce laser power
	Temperature drift	Check stage stability
	Sample degradation	Check ID/IG ratio
Large hysteresis	Thermal lag	Increase stabilization time
	Sample damage	Check post-experiment mapping
	Focus drift	Check focus at each temperature
High ID/IG ratio	Sample contamination	Clean sample or remake
	Laser damage	Reduce power, check sacrificial spot
	Oxidation	Use inert atmosphere
No anti-Stokes signal	Filter in place	Check optical path
	Too low laser power	Increase power within limits
	Detector saturation	Reduce accumulation time
Temperature mismatch ¿5K	Poor thermal contact	Remount with fresh silver paste
	Thermocouple issue	Recalibrate temperature stage
	Laser heating	Reduce laser power

Table 16: Troubleshooting guide