Determining the spreading law for a picolitre droplet

10090371
Jiachen Guo

School of Physics and Astronomy
The University of Manchester

PHYS20762 Computational Physics
Project 1 Report

February 2018

1 Abstract

A picolitre droplet's spreading dynamics as it is dropped onto a surface of solid substrate is investigated in this experiment. Experimental data of the time variation of two picolitre droplets' radii were obtained- from the top and side view of the first droplet and from only the top view of the second droplet. Using Python, values of contact line speed and contact angle for each droplet at each instant of time were deduced from its recorded time and radii values. Plots of contact line speed against contact angle for each droplet were produced to determine the spreading laws which relate each droplet's contact angles to its contact line speeds. The first droplet was found to have smaller contact angles and its spreading law was best approximated by the de Gennes spreading law with a reduced χ^2 of 0.84. The second droplet was found to have larger contact angles and its spreading law was best approximated by the Cox-Voinov spreading law with a reduced χ^2 of 1.16.

2 Introduction

Microdroplet dispensing has a wide range of applications in biotechnology, microelectronics and microassembly. Rapid developments in these fields produced demands of better understandings of high-precision delivery of smaller and smaller droplets and its applications[3]. It is thus of valid interest to investigate the spreading laws a picolitre droplet on a solid substrate surface.

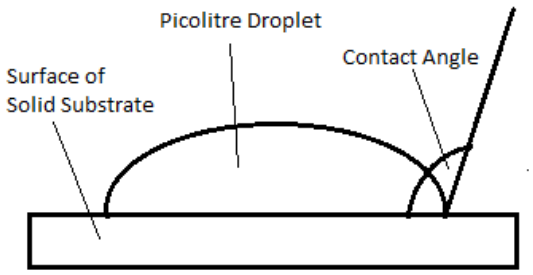

Figure 2.1: Physical illustration of a picolitre droplet when it comes into contact with a surface of solid substrate.

The dynamic spreading of a picolitre droplet on a surface of solid substrate is mainly influenced by its surface tension, contact line hysteresis and fluid viscosity[2]. At every instant, the dynamic state the droplet is in is characterised by its contact line angle, illustrated in Figure 2.1, and the speed of the contact line. Thus a spreading law which relates a droplet's contact angle to contact line speed at every instant of time can provide insights on how the droplet spreads over time on a particular surface of solid substrate. Further analysis can then be done to investigate the microscopic forces that determine the spreading pattern of the droplet.

3 Methodology

Three sets of experimental data for two picolitre droplets of volume $V = 7.6\ pL$ were obtained. The first set consists of 6 runs of 44 recorded time and radii values each, obtained from the side view of the first droplet. The second set consists of 3 runs of 1746 recorded time and radii values each, obtained top view of the first droplet. The third set consists of 2 runs of runs of 460 recorded time and radii values each, obtained from top view of the second droplet. The data for each run is stored in $.txt$ files with the first column containing the time/s values and the second column containing the corresponding radius/$\mu m s^{-1}$ values. The time/s values for every run in each set is identical.

For every set of data, the mean radius $\bar{r}$, of the picolitre droplet at every instant of time is obtained by averaging recorded radii values from all of the runs in that set at that instant of time, as in

$$\bar{r} = \frac{\sum_{i=1}^{n} r_i}{n} \tag{1}$$

where n is the number of runs and r_i is the recorded radius value in each run.

An estimate of the error in mean radius $\bar{r}$ is obtained by the standard error estimation technique of first finding the standard deviation s of the radii values used to calculate $\bar{r}$ and then obtaining the error in the mean radius $\sigma_{\bar{r}}$ through

$$\sigma_{\bar{r}} \approx \frac{s}{\sqrt{n-1}} \tag{2}$$

It is worth noting that the estimation in Equation (2) works best for large values and our n values can hardly be justified as large. However due to limitations in experimental techniques these are the best n values that can be obtained. The error $\sigma_{\bar{r}}$ is then at best a rough estimate of the true error and this has to be taken into account if discrepancies arise in later parts of the data analysis.

Next, to determine the values of contact angles θ from the mean radius $\bar{r}$ values, the height h values of the droplet above the surface of solid substrate are first deduced. This is done by approximating the picolitre droplet as a spherical cap whose volume V is given by

$$V = \frac{\pi h}{6}(3r^2 + h^2) \tag{3}$$

where radius r of the spherical cap here will refer to the mean radius $\bar{r}$ of the droplet. Knowing that the volume of the picolitre droplet $V = 7.6pL$, and $r = \bar{r}$, Equation (3) then becomes a cubic equation in h and can be solved by finding the roots of this cubic equation using Python. The real roots will then be the values of h we want. As the standard error propagation technique on h is tricky since it is a cubic equation, the error in h is chosen to be determined by the technique of varying $\bar{r}$ by its uncertainty $\sigma_{\bar{r}}$ and then recalculating the value of h. As $\bar{r}$ can vary by $\pm\,\sigma_{\bar{r}}$, two new values of h, h_{new} will be obtained. There are then two values of $|h - h_{new}|$. The uncertainty in h, σ_h is then assigned as equal to the larger of the two $|h - h_{new}|$ values.

The contact angle θ values of the picolitre droplet can then be found by

$$\theta = \frac{\pi}{2} - \arctan\left(\frac{\bar{r}^2 - h^2}{2}\right) \tag{4}$$

which also results from the spherical cap approximation. Here, the error in θ, σ_θ can be determined through the standard error propagation technique of combining by quadrature of the error contributions from $\sigma_{\bar{r}}$ and σ_h, as in

$$\sigma_\theta = \left(\frac{\partial\theta}{\partial\bar{r}}\right)^2 {\sigma_{\bar{r}}}^2 + \left(\frac{\partial\theta}{\partial h}\right)^2 {\sigma_h}^2 \tag{5}$$

Next, the contact line speed values at each instant of recorded time are calculated. The average contact line speeds $u_{i+0.5}$ at an instance between every two recorded time values is first calculated through

$$u_{i+0.5} = \frac{r_{i+1} - r_i}{t_{i+1} - t_i} \tag{6}$$

where r_i and r_{i+1} are the mean radius values of two consecutive recorded radii values and t_i and t_{i+1} are the corresponding time values of the r_i and r_{i+1}. With the $u_{i+0.5}$ values, the contact line speed u_i at each instant of time t_i is then calculated by taking the average of two neighbouring $u_{i+0.5}$ values, as in

$$u_i = \frac{u_{i+0.5} + u_{i-0.5}}{2} \tag{7}$$

Note that using this technique, there are no deducible u_i values for the first recorded time value and last recorded time value in each run. This is fine since we still have a large amount of other u_i values that can be used to determine the spreading laws. The error in u_i, σ_{u_i} is easily deducible through the standard error propagation technique of combining by quadrature, in a similar fashion to Equation (5).

With the deduced contact line speeds u, ignoring the subscript i which is only used for numbering purposes in Equations (6) & (7), and the deduced contact angles θ values, graphs of contact line speeds u against contact angles θ are plotted for the side view of the first droplet, the top view of the first droplet and the top view of the second droplet.

Using Python's polyfit function, spreading laws of quadratic, cubic, de Gennes and Cox-Voinov kinds were fitted to the produced graphs. In equation form, these spreading laws are

$$\begin{aligned} &Quadratic && u = c_1\theta^2 + c_2\theta + c_3 \\ &Cubic && u = c_1\theta^3 + c_2\theta^2 + c_3\theta + c_4 \\ &de\ Gennes && u = u_0\left(\theta^2 - {\theta_0}^2\right) \\ &Cox - Voinov && u = u_0\left(\theta^3 - {\theta_0}^3\right) \end{aligned} \tag{8}$$

where c_i and u_0 are constant coefficients. For quadratic and cubic fits, c_1 is the characteristic velocity scale of spreading while for de Gennes and Cox-Voinov fits u_0 is the characteristic velocity scale of spreading.

The suitability of the fits of the spreading laws for each set of deduced contact line speeds u and contact angles θ values were tested by plotting the residuals of the fitted relationship and observing whether uncertainty values are acceptable. χ^2 analysis was also used to reach a conclusion about the suitability of the fitted relationship.

4 Results and Discussion

4.1 Side View of Droplet 1

The side view data of the first picolitre droplet consists of 6 runs, with 44 pairs of recorded time/s and radius/μm values in each run. The time/s values vary over the interval $0.06s - 0.49s$ in increments of $0.01s$. The contact angle $\theta/°$ values varies from $2.5° - 7°$. The contact line speed $u/\mu ms^{-1}$ values vary from $-10\mu ms^{-1}\ to\ 85\mu ms^{-1}$.

The best spreading law relationship over this range of contact angles was determined to be the quadratic fit in (8). This produces a reduced chi-squared $\chi^2_{red} = 0.7$ which is within the acceptable range of $1 - \sqrt{8/Ndof}$ = 0.5 and $1 + \sqrt{8/Ndof}$ = 1.5 where $Ndof$ is the number of degrees of freedom of the fit. Note that only 42 values of contact line speed u were available using Equation (7) so $Ndof = 42 - 3 = 39$.

The other function fits in (8) had also produced desirable χ^2_{red} values of within $0.6 - 1.5$, with the de Gennes function fit producing a χ^2_{red} of 1.02. However further analysis of the residuals of the de

Gennes function fit reveal that residuals do not overlap the x-axis for $\theta > 6°$ and hence is an inappropriate fit.

The relevant plots corresponding to the quadratic function fit are shown in Figure 4.1. This fit produces a characteristic velocity scale of $c_1 = -0.745 \pm 0.554\ \mu m s^{-1} degrees^{-2}$.

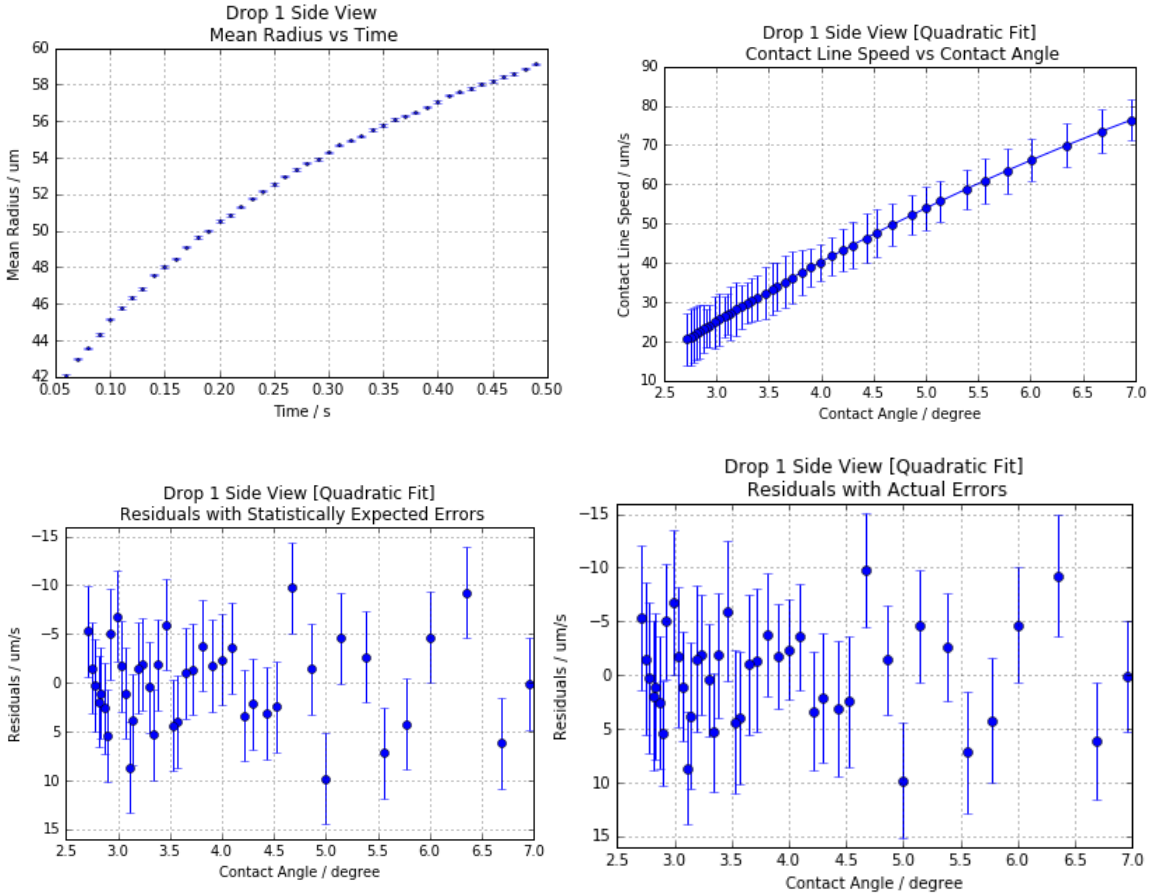

Figure 4.1: *All plots correspond to side view of drop 1. Top Left* Plot showing time-variation of mean radius. *Top Right* Variation of contact line speed u with contact angle θ with quadratic fit. *Bottom Left* Residuals plotted with statistically expected errors. *Bottom Right* Residuals plotted with actual errors. Shows that error estimates are appropriate when compared to bottom left plot with statistically expected errors. Also shows that quadratic fit is acceptable.

4.2 Top View of Droplet 1

The top view data of the first picolitre droplet consists of 3 runs, with 1746 pairs of recorded time$/s$ and radius$/\mu m$ values in each run. The time$/s$ values vary over the interval $0.50s - 17.95s$ in increments of $0.01s$. The contact angle $\theta/°$ values vary from $0.5° - 2.75°$. The contact line speed $u/\mu m s^{-1}$ values vary from $-15\mu m s^{-1} - 30\mu m s^{-1}$.

The best spreading law relationship over this range of contact angles was determined to be the de Gennes fit in (8). This is different from the conclusion for the side view of droplet 1, where a quadratic fit is more appropriate. In this case, the de Gennes fit produces a $\chi^2_{red} = 0.84$. This is within the upper limit of $1 + \sqrt{8/Ndof} = 1.07$ but not within the lower limit of $1 + \sqrt{8/Ndof} = 0.93$. However, the fit of de Gennes function was still chosen out of the four functions in (8) because other function fits also

produced $\chi^2_{red} \approx 0.84$ but the corresponding residual plots were not as good as that of the de Gennes fit. The reason for the χ^2_{red} not within the lower limit might be due to the error estimation of mean radius $\bar{r}$ in Equation (2) since Equation (2) works best for large n values. Here $n = 3$.

Thus we can only conservatively say that out of all 4 relationships in (8), the de Gennes function fit is the best in describing the spreading law for the top view of droplet 1. However, it might not be the most desirable fit and further analysis has to be done to reach a definitive conclusion.

The relevant plots corresponding to the de Gennes function fit are shown in Figure 4.2. This fit produces a characteristic velocity scale of $u_0 = 2.95 \pm 0.16\, \mu m s^{-1} degrees^{-2}$.

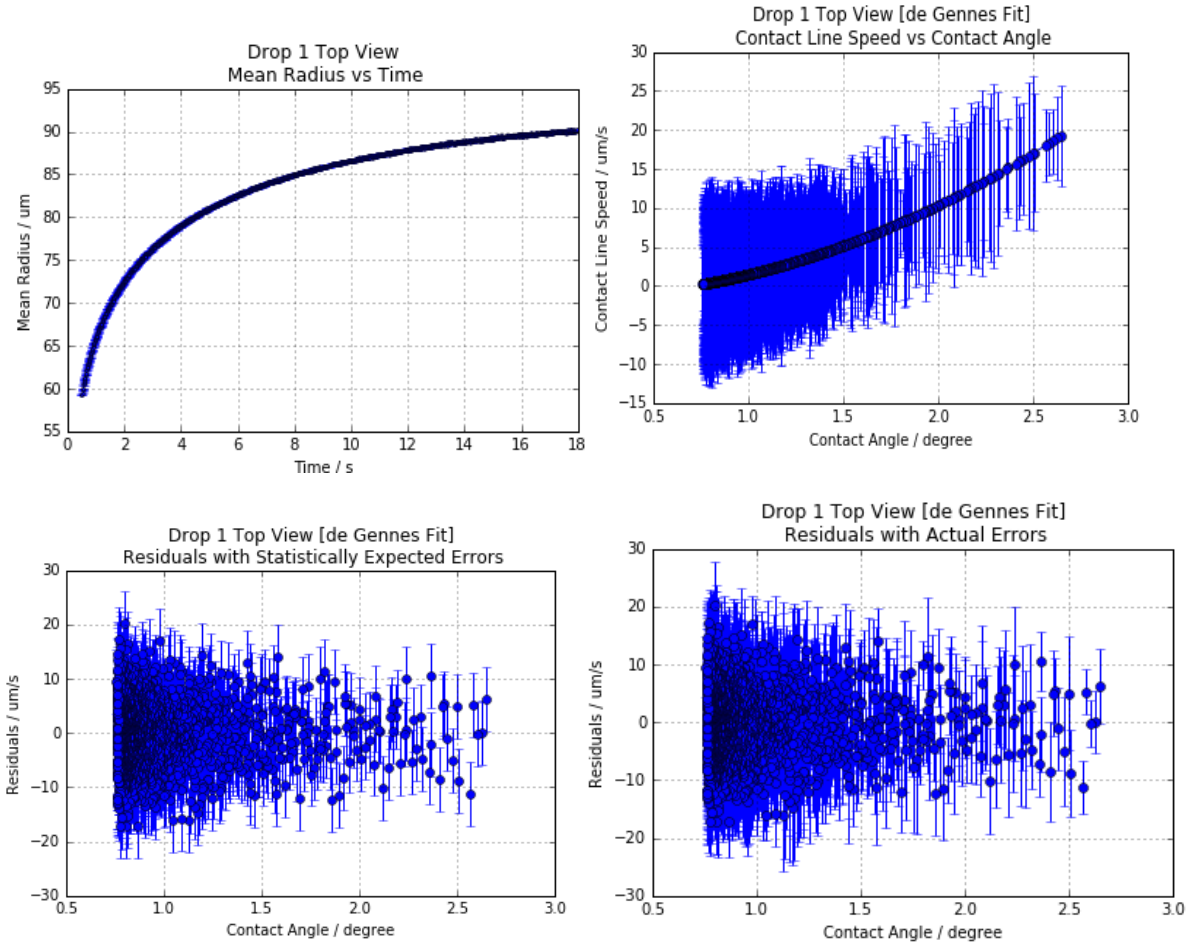

Figure 4.2: *All plots correspond to top view of drop 1. Top Left* Plot showing time-variation of mean radius. *Top Right* Variation of contact line speed u with contact angle θ with de Gennes fit. *Bottom Left* Residuals plotted with statistically expected errors. *Bottom Right* Residuals plotted with actual errors.

4.3 Top View of Droplet 2

The top view data of the second picolitre droplet consists of 2 runs, with 460 pairs of recorded time$/s$ and radius$/\mu m$ values in each run. The time$/s$ values vary over the interval $0.50s - 17.95s$ in increments of $0.01s$. The contact angle $\theta/°$ values vary from $10.5° - 14.5°$. The contact line speed $u/\mu m s^{-1}$ values vary from $-20 \mu m s^{-1}$ *to* $70 \mu m s^{-1}$.

The best spreading law relationship over this range of contact angles was determined to be the Cox-Voinov fit in (8). This produces $\chi^2_{red} = 1.16$. The other function fits also produced similar χ^2_{red} in the range $1.15 - 1.16$. However, the residuals for Cox-Voinov function fit proves to be the most desirable.

Comparing the residual plots of statistically expected errors and residual plots of actual errors, it can be seen that the actual error estimates are of a comparable magnitude to the statistically expected errors. Thus the choice of a Cox-Voinov function fit for the spreading law of the top view of droplet 2 is arrived at.

The relevant plots corresponding to the Cox-Voinov function fit are shown in Figure 4.3. This fit produces a characteristic velocity scale of $u_0 = 0.036 \pm 0.001\ \mu ms^{-1} degrees^{-3}$.

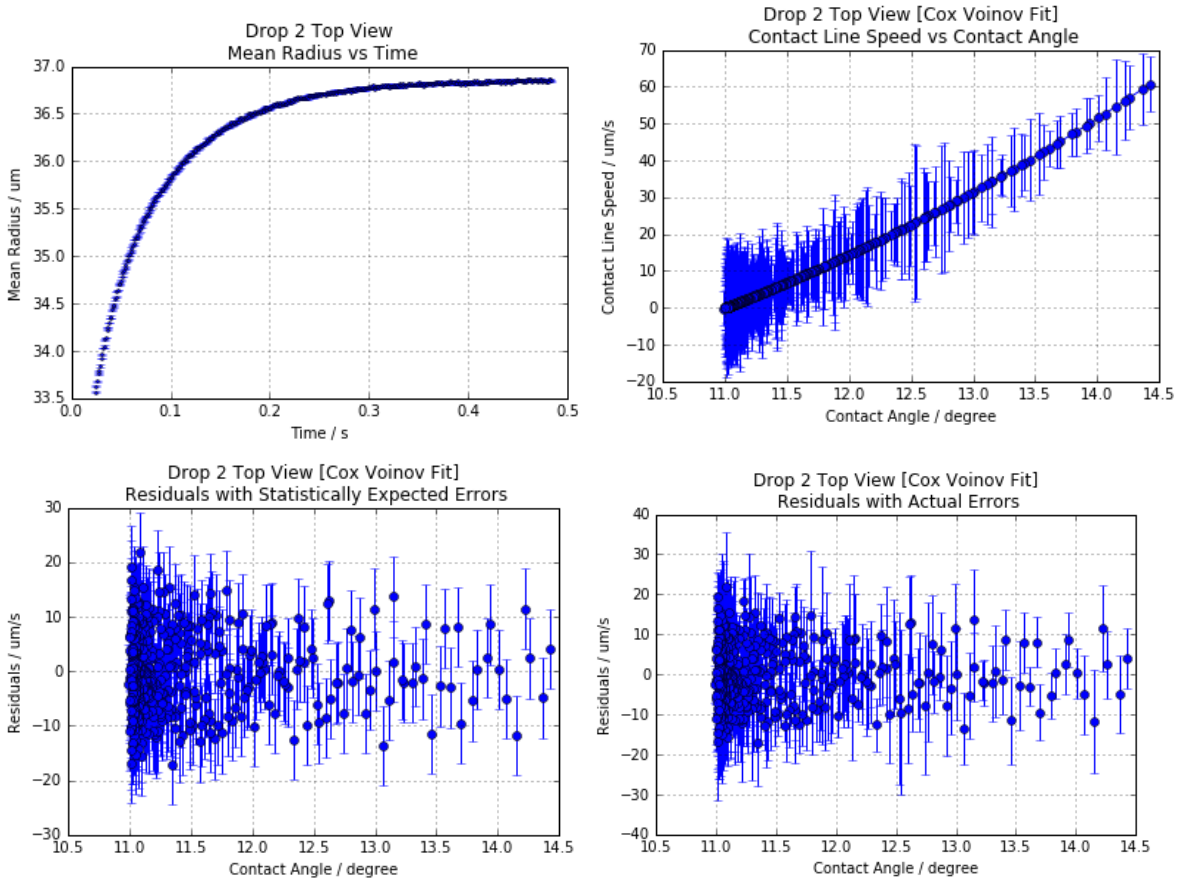

Figure 4.2: *All plots correspond to top view of drop 2. Top Left* Plot showing time-variation of mean radius. *Top Right* Variation of contact line speed u with contact angle θ with Cox-Voinov fit. *Bottom Left* Residuals plotted with statistically expected errors. *Bottom Right* Residuals plotted with actual errors.

5 Conclusion

The conclusions that can be drawn from this investigation are: a de Gennes relationship or a quadratic relationship is the most desirable for describing the spread of a picolitre droplet over a range of smaller dynamic contact angle; a Cox-Voinov relationship is the most desirable for describing the spread of a picolitre droplet over a range of larger dynamic contact angles.

For future anaylsis, the relationship of contact line speed u with contact angle θ for top view of droplet 1 can be further investigated to reach a more definitive conclusion on the best relationship between them.

References

[1] Kant, P. et al. *Controlling droplet spreading with topography, 2017.* School of Mathematics and Manchester Centre for Nonlinear Dynamics, University of Manchester.

[2] Thompson, A.B. et al. *Sequential deposition of overlapping droplets to form a liquid line, 2014.* J. Fluid Mech, vol. 761, pp. 261-281

[3] Mengmeng, H. et al. *Simulation and basic experiment of picoliter droplet micro-dispensing based on piezoelectric drive.* Mechatronics and Automation (ICMA), 2014 IEEE International Conference. DOI: 10.1109/ICMA.2014.6885678

[4] Yingnan, S. et al. *A novel picoliter droplet array for parallel real-time polymerase chain reaction based on double-inkjet printing.* Royal Society of Chemistry. DOI: 10.1039/c4lc00598h

Appendix 1 – Python Script for Analysis of Side View of Droplet 1

```
# -*- coding: utf-8 -*-
"""
Created on Sun Mar  4 20:44:12 2018

@author: mbcxajg2
**************************************
    DATA ANALYSIS: DROP 1 SIDE VIEW
**************************************
"""

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

#=====================================
#           FUNCTION 1
#=====================================
# Returns (contact angle/degree, contact angle error/degree) from inputs: mean radius, mean radius error
def get_theta(mean_radius, mean_radius_error):
    """
    float, float -> tuple
    Returns (contact_angle, contact_angle_error) from mean radius
    """
    r = mean_radius * 1e-06                 # convert into SI units
    r_error = mean_radius_error * 1e-06     # convert into SI units

    # solve for roots of H
    h_roots = np.roots([math.pi/6, 0, math.pi*3*(math.pow(r, 2))/6, (-7.6*1e-15)])
    # only want real root of H_roots
    h = h_roots.real[abs(h_roots.imag)<1e-5]
    theta = (math.pi/2) - math.atan(((r*r)-(h*h))/(2*r*h))

    # calculate h_error by varing r by +- r_error and recalculating h
    # compute h with r = r + r_error
    h_roots_positve_r_error = np.roots([math.pi/6, 0, math.pi*3*(math.pow((r + r_error), 2))/6, (-7.6*1e-15)])
    h_positive_r_error = h_roots.real[abs(h_roots_positve_r_error.imag)<1e-5]
    # compute h with r = r - r_error
    h_roots_negative_r_error = np.roots([math.pi/6, 0, math.pi*3*(math.pow((r - r_error), 2))/6, (-7.6*1e-15)])
    h_negative_r_error = h_roots.real[abs(h_roots_negative_r_error.imag)<1e-5]
    # take the larget h_error obtained
    if abs(h_positive_r_error - h) > abs(h_negative_r_error - h):
        h_error = abs(h_positive_r_error - h)
    else:
        h_error = abs(h_negative_r_error - h)

    # compute dtheta/dr
    dtheta_dr = -(1/(1+math.pow(((math.pow(r,2)-math.pow(h,2))/(2*r*h)),2)))*((1/(2*h))-(h/(2*r*r)))
    # compute dtheta/dh
    dtheta_dh = -(1/(1+math.pow((((r*r)-(h*h))/(2*r*h)),2)))*(-(r/(2*h*h))-(1/(2*r)))
    # compute theta_error
    theta_error = math.sqrt( (dtheta_dr*dtheta_dr*r_error*r_error) + (dtheta_dh*dtheta_dh*h_error*h_error) )

    return (theta*180/math.pi, theta_error*180/math.pi)

#=====================================
#           FUNCTION 2
```

```
#====================================
# Returns (speed/um/s, speed_error/um/s) from inputs:
#   mean radius, previous mean radius, next mean radius,
#   time at mean radius, time at previous mean radius, time at next mean radius,
#   mean radius error, previous mean radius error, next mean radius error
def get_speed(r, prev_r, next_r, t, prev_t, next_t, r_err, prev_r_err, next_r_err):

    """
    float * 9 -> tuple
    Calculates contact line speed for every mean radius value except first and last mean radius values
    Returns (speed/, speed_error)
    """
    v1 = (r - prev_r) / (t - prev_t)  # speed at an instance before r
    v2 = (next_r - r) / (next_t - t)  # speed at an instance after r
    v = (v1 + v2) / 2  # speed at r
    v1_err = math.sqrt( ( (1 / (t - prev_t))**2 ) * (r_err**2 + prev_r_err**2) )
    v2_err = math.sqrt( ( (1 / (next_t - t))**2 ) * (next_r_err**2 + r_err**2) )
    v_err = math.sqrt( ((1/4) * (v1_err**2)) + ((1/4) * (v2_err**2)) )
    return (v, v_err)

#=========================================
#                LOAD DATA
#=========================================
# rx_data[:, 0] -> time/s | rx_data[:, 1] -> radius/um
r1_data = np.loadtxt('Side_view_data_run1.txt')
r2_data = np.loadtxt('Side_view_data_run2.txt')
r3_data = np.loadtxt('Side_view_data_run3.txt')
r4_data = np.loadtxt('Side_view_data_run4.txt')
r5_data = np.loadtxt('Side_view_data_run5.txt')
r6_data = np.loadtxt('Side_view_data_run6.txt')
data_len = len(r1_data)

#=========================================
#                  TIME
#=========================================
time = r1_data[:, 0]        # time values are same for r1_data to r6_data

#=========================================
#     MEAN RADIUS (um) AND ERROR
#=========================================
# Calculate mean radius values
mean_r = (r1_data[:, 1] + r2_data[:, 1] + r3_data[:, 1] + r4_data[:, 1] + r5_data[:, 1] + r6_data[:, 1]) / 6

# Array to store mean radius error values
      mean_r_err = np.zeros(data_len)

      # Calculate mean radius error values
      for i in range(data_len):
          mean_r_err[i] = np.std([r1_data[i][1], r2_data[i][1], r3_data[i][1], r4_data[i][1], r5_data[i][1], r6_data[i][1]]) / math.sqrt(5)

      #=============================================
      #     CONTACT ANGLE (degree) AND ERROR
      #=============================================
      # Arrays to store contact angle and error
      theta = np.zeros(data_len)
      theta_err = np.zeros(data_len)

      # Calculate contact angle and error
      for i in range(data_len):
          theta[i], theta_err[i] = get_theta(mean_r[i], mean_r_err[i])

      #=================================================
      #     CONTACT LINE SPEED (um/s) AND ERROR
```

```
        #================================================
        # Arrays to store contact line speed and  error
        u = np.zeros(data_len)
        u_err = np.zeros(data_len)

        # Calculate contact line speed and error
        for i in range(data_len):
            if i == 0:  # no speed for first data point
                continue
            elif i == data_len - 1: # no speed for last data point
                continue
            else:
                u[i], u_err[i] = get_speed(mean_r[i], mean_r[i-
    1], mean_r[i+1], time[i], time[i-1], time[i+1], mean_r_err[i], mean_r_err[i-
    1], mean_r_err[i+1])

        # Only want arrays with valid speeds
        u = u[1:-1]
        u_err = u_err[1:-1]

        #================================================
        #         PLOT: MEAN RADIUS (UM)
        #                   VS
        #                 TIME (S)
        #================================================
        # Plot
        plt.errorbar(time, mean_r, yerr=mean_r_err, marker='o', markersize=2, linest
    yle='None')
        plt.ylabel('Mean Radius / um')
        plt.xlabel('Time / s')
        plt.grid()
        plt.title('Drop 1 Side View\n Mean Radius vs Time')
        plt.show()

        #================================================
        #           ADJUST CONTACT ANGLE VALUES
        #================================================
        # Only want contact angles with valid contact line speeds for graph of
        # contact lines speed against contact angle
        theta = theta[1:-1]
        theta_err = theta_err[1:-1]

        #================================================
        #     PLOT: CONTACT LINE SPEED (UM/S)
        #                   VS
        #           CONTACT ANGLE (DEGREE)
        #================================================
        # Plot
        plt.errorbar(theta, u, xerr=theta_err, yerr=u_err, marker='o', linestyle='No
    ne')
        plt.ylabel('Contact Line Speed / um/s')
        plt.xlabel('Contact Angle / degree')
        plt.grid()
        plt.title('Drop 1 Side View\n Contact Line Speed vs Contact Angle')
        plt.show()

        #================================================
        #          QUADRATIC FIT AND ANALYSIS
        #================================================
        # Calculate coefficients and error in coefficients
        (q_coeff, q_covr) = np.polyfit(theta, u, 2, cov=True)

        # Calculate u values of quadratic fit
        q_u = np.polyval(q_coeff, theta)

        # Calculate statistically expected errors for the fit
```

```
181.        q_u_expected_err = np.sqrt((1/(42-3))*(np.sum(np.power(u-
    q_u, 2)))) # this value is a constant
182.        q_u_expected_err = q_u_expected_err * np.ones(len(q_u)) # create an array of
     this value
183.
184.        # Plot fitted u against theta
185.        plt.errorbar(theta, q_u, yerr=u_err, marker='o')
186.        plt.ylabel('Contact Line Speed / um/s')
187.        plt.xlabel('Contact Angle / degree')
188.        plt.grid()
189.        plt.title('Drop 1 Side View [Quadratic Fit]\n Contact Line Speed vs Contact
    Angle')
190.        plt.show()
191.
192.        # Plot residuals with statistically expected errors
193.        plt.errorbar(theta, q_u-
    u, yerr=q_u_expected_err, marker='o', linestyle='None')
194.        plt.ylim(16,-16)
195.        plt.ylabel('Residuals / um/s')
196.        plt.xlabel('Contact Angle / degree')
197.        plt.grid()
198.        plt.title('Drop 1 Side View [Quadratic Fit]\n Residuals with Statistically E
    xpected Errors')
199.        plt.show()
200.
201.        # Plot residuals with actual errors
202.        plt.errorbar(theta, q_u-u, yerr=u_err, marker='o', linestyle='None')
203.        plt.ylim(16,-16)
204.        plt.ylabel('Residuals / um/s')
205.        plt.xlabel('Contact Angle / degree')
206.        plt.grid()
207.        plt.title('Drop 1 Side View [Quadratic Fit]\n Residuals with Actual Errors')

208.        plt.show()
209.
210.        # Calculate chi-square value
211.        q_chisq = np.sum(np.power(((u-q_u)/u_err), 2))
212.        q_reduced_chisq = q_chisq/ (42-3)
213.        print('[Quadratic Fit] Chi-Sq:', q_chisq)
214.        print('[Quadratic Fit] Reduced Chi-Sq:', q_reduced_chisq)
215.
216.        # Comment on Fit
217.        print('[Quadratic Fit] Comment: Reduced Chi-
    Sq values within acceptable range. Good fit.')
218.
219.        # Show quadratic equation
220.        print('[Quadratic Fit] Fitted Equation: ax^2 + bx +c')
221.
222.        # Show errors on coefficients
223.        print('[Quadratic Fit] a = %f +-
     %f' % (q_coeff[0], math.sqrt(q_covr[0][0])))
224.        print('[Quadratic Fit] b = %f +-
     %f' % (q_coeff[1], math.sqrt(q_covr[1][1])))
225.        print('[Quadratic Fit] c = %f +-
     %f' % (q_coeff[2], math.sqrt(q_covr[2][2])))
226.
227.        #================================================
228.        #            CUBIC FIT AND ANALYSIS
229.        #================================================
230.        # Calculate coefficients and error in coefficients
231.        (c_coeff, c_covr) = np.polyfit(theta, u, 3, cov=True)
232.
233.        # Calculate u values of quadratic fit
234.        c_u = np.polyval(c_coeff, theta)
235.
236.        # Calculate statistically expected errors for the fit
```

```
237.        c_u_expected_err = np.sqrt((1/(42-3))*(np.sum(np.power(u-
    c_u, 2)))) # this value is a constant
238.        c_u_expected_err = c_u_expected_err * np.ones(len(c_u)) # create an array of
     this value
239.
240.        # Plot fitted u against theta
241.        plt.errorbar(theta, c_u, yerr=u_err, marker='o')
242.        plt.ylabel('Contact Line Speed / um/s')
243.        plt.xlabel('Contact Angle / degree')
244.        plt.grid()
245.        plt.title('Drop 1 Side View [Cubic Fit]\n Contact Line Speed vs Contact Angl
    e')
246.        plt.show()
247.
248.        # Plot residuals with statistically expected errors
249.        plt.errorbar(theta, c_u-
    u, yerr=c_u_expected_err, marker='o', linestyle='None')
250.        plt.ylim(16,-16)
251.        plt.ylabel('Residuals / um/s')
252.        plt.xlabel('Contact Angle / degree')
253.        plt.grid()
254.        plt.title('Drop 1 Side View [Cubic Fit]\n Residuals with Statistically Expec
    ted Errors')
255.        plt.show()
256.
257.        # Plot residuals with actual errors
258.        plt.errorbar(theta, c_u-u, yerr=u_err, marker='o', linestyle='None')
259.        plt.ylim(16,-16)
260.        plt.ylabel('Residuals / um/s')
261.        plt.xlabel('Contact Angle / degree')
262.        plt.grid()
263.        plt.title('Drop 1 Side View [Cubic Fit]\n Residuals with Actual Errors')
264.        plt.show()
265.
266.        # Calculate chi-square value
267.        c_chisq = np.sum(np.power(((u-c_u)/u_err), 2))
268.        c_reduced_chisq = c_chisq/ (42-3)
269.        print('[Cubic Fit] Chi-Sq:', c_chisq)
270.        print('[Cubic Fit] Reduced Chi-Sq:', c_reduced_chisq)
271.
272.        # Comment
273.        print('[Cubic Fit] Comment: Reduce Chi-
    Sq values not as good as quadratic fit.')
274.
275.        print('[Cubic Fit] Fitted Equation: ax^3 + bx^2 + cx + d')
276.
277.        # Show errors on coefficients
278.        print('[Cubic Fit] a = %f +- %f' % (c_coeff[0], math.sqrt(c_covr[0][0])))
279.        print('[Cubic Fit] b = %f +- %f' % (c_coeff[1], math.sqrt(c_covr[1][1])))
280.        print('[Cubic Fit] c = %f +- %f' % (c_coeff[2], math.sqrt(c_covr[2][2])))
281.        print('[Cubic Fit] d = %f +- %f' % (c_coeff[3], math.sqrt(c_covr[3][3])))
282.
283.
284.        #================================================
285.        #            LINEAR FIT AND ANALYSIS
286.        #================================================
287.        # Calculate coefficients and error in coefficients
288.        (l_coeff, l_covr) = np.polyfit(theta, u, 1, cov=True)
289.
290.        # Calculate u values of linear fit
291.        l_u = np.polyval(l_coeff, theta)
292.
293.        # Calculate statistically expected errors for the fit
294.        l_u_expected_err = np.sqrt((1/(42-2))*(np.sum(np.power(u-
    l_u, 2)))) # this value is a constant
```

```
295.        l_u_expected_err = l_u_expected_err * np.ones(len(l_u)) # create an array of
    this value
296.
297.        # Plot fitted u against theta
298.        plt.errorbar(theta, l_u, yerr=u_err, marker='o')
299.        plt.ylabel('Contact Line Speed / um/s')
300.        plt.xlabel('Contact Angle / degree')
301.        plt.grid()
302.        plt.title('Drop 1 Side View [Linear Fit]\n Contact Line Speed vs Contact Ang
    le')
303.        plt.show()
304.
305.        # Plot residuals with statistically expected errors
306.        plt.errorbar(theta, l_u-
    u, yerr=l_u_expected_err, marker='o', linestyle='None')
307.        plt.ylabel('Residuals / um/s')
308.        plt.xlabel('Contact Angle / degree')
309.        plt.grid()
310.        plt.title('Drop 1 Side View [Linear Fit]\n Residuals with Statistically Expe
    cted Errors')
311.        plt.show()
312.
313.        # Plot residuals with actual errors
314.        plt.errorbar(theta, l_u-u, yerr=u_err, marker='o', linestyle='None')
315.        plt.ylabel('Residuals / um/s')
316.        plt.xlabel('Contact Angle / degree')
317.        plt.grid()
318.        plt.title('Drop 1 Side View [Linear Fit]\n Residuals with Actual Errors')
319.        plt.show()
320.
321.        # Calculate chi-square value
322.        l_chisq = np.sum(np.power(((u-l_u)/u_err), 2))
323.        l_reduced_chisq = l_chisq/ (42-2)
324.        print('[Linear Fit] Chi-Sq:', l_chisq)
325.        print('[Linear Fit] Reduced Chi-Sq:', l_reduced_chisq)
326.
327.        # Comment
328.        print('[Linear Fit] Comment: Not as good as quadratic fit.')
329.
330.        print('[Linear Fit] Fitted Equation: ax + b')
331.
332.        # Show errors on coefficients
333.        print('[Linear Fit] a = %f +- %f' % (l_coeff[0], math.sqrt(l_covr[0][0])))
334.        print('[Linear Fit] b = %f +- %f' % (l_coeff[1], math.sqrt(l_covr[1][1])))
335.
336.        #=================================================
337.        #           DE GENNES FIT AND ANALYSIS
338.        #=================================================
339.        # Calculate coefficients and error in coefficients
340.        (g_coeff, g_covr) = np.polyfit(theta*theta, u, 1, cov=True)
341.
342.        # Calculate u values of linear fit
343.        g_u = np.polyval(g_coeff, theta*theta)
344.
345.        # Calculate statistically expected errors for the fit
346.        g_u_expected_err = np.sqrt((1/(42-3))*(np.sum(np.power(u-
    g_u, 2)))) # this value is a constant
347.        g_u_expected_err = g_u_expected_err * np.ones(len(g_u)) # create an array of
    this value
348.
349.        # Plot fitted u against theta*theta
350.        plt.errorbar(theta, g_u, yerr=u_err, marker='o')
351.        plt.ylabel('Contact Line Speed / um/s')
352.        plt.xlabel('Contact Angle / degree')
353.        plt.grid()
```

```
354.        plt.title('Drop 1 Side View [de Gennes Fit]\n Contact Line Speed vs Contact
    Angle')
355.        plt.show()
356.
357.        # Plot residuals with statistically expected errors
358.        plt.errorbar(theta, g_u-
    u, yerr=g_u_expected_err, marker='o', linestyle='None')
359.        plt.ylabel('Residuals / um/s')
360.        plt.xlabel('Contact Angle / degree')
361.        plt.grid()
362.        plt.title('Drop 1 Side View [de Gennes Fit]\n Residuals with Statistically E
    xpected Errors')
363.        plt.show()
364.
365.        # Plot residuals with actual errors
366.        plt.errorbar(theta, g_u-u, yerr=u_err, marker='o', linestyle='None')
367.        plt.ylabel('Residuals / um/s')
368.        plt.xlabel('Contact Angle / degree')
369.        plt.grid()
370.        plt.title('Drop 1 Side View [de Gennes Fit]\n Residuals with Actual Errors')

371.        plt.show()
372.
373.        # Calculate chi-square value
374.        g_chisq = np.sum(np.power(((u-g_u)/u_err), 2))
375.        g_reduced_chisq = g_chisq/ (42-3)
376.        print('[de Gennes Fit] Chi-Sq:', g_chisq)
377.        print('[de Gennes Fit] Reduced Chi-Sq:', g_reduced_chisq)
378.
379.        # Comment
380.        print('[de Gennes Fit] Comment: Ok Fit.')
381.
382.        print('[de Gennes Fit] Fitted Equation: ax + b')
383.
384.        # Show errors on coefficients
385.        print('[de Gennes Fit] a = %f +-
    %f' % (g_coeff[0], math.sqrt(g_covr[0][0])))
386.        print('[de Gennes Fit] b = %f +-
    %f' % (g_coeff[1], math.sqrt(g_covr[1][1])))
387.
388.        #=================================================
389.        #            COX VOINOV FIT AND ANALYSIS
390.        #=================================================
391.        # Calculate coefficients and error in coefficients
392.        (v_coeff, v_covr) = np.polyfit(theta*theta*theta, u, 1, cov=True)
393.
394.        # Calculate u values of linear fit
395.        v_u = np.polyval(v_coeff, theta*theta*theta)
396.
397.        # Calculate statistically expected errors for the fit
398.        v_u_expected_err = np.sqrt((1/(42-4))*(np.sum(np.power(u-
    v_u, 2)))) # this value is a constant
399.        v_u_expected_err = v_u_expected_err * np.ones(len(v_u)) # create an array of
    this value
400.
401.        # Plot fitted u against theta
402.        plt.errorbar(theta, v_u, yerr=u_err, marker='o')
403.        plt.ylabel('Contact Line Speed / um/s')
404.        plt.xlabel('Contact Angle / degree')
405.        plt.grid()
406.        plt.title('Drop 1 Side View [Cox Voinov Fit]\n Contact Line Speed vs Contact
    Angle')
407.        plt.show()
408.
409.        # Plot residuals with statistically expected errors
```

```
410.        plt.errorbar(theta, v_u-
    u, yerr=v_u_expected_err, marker='o', linestyle='None')
411.        plt.ylabel('Residuals / um/s')
412.        plt.xlabel('Contact Angle / degree')
413.        plt.grid()
414.        plt.title('Drop 1 Side View [Cox Voinov Fit]\n Residuals with Statistically
    Expected Errors')
415.        plt.show()
416.
417.        # Plot residuals with actual errors
418.        plt.errorbar(theta, v_u-u, yerr=u_err, marker='o', linestyle='None')
419.        plt.ylabel('Residuals / um/s')
420.        plt.xlabel('Contact Angle / degree')
421.        plt.grid()
422.        plt.title('Drop 1 Side View [Cox Voinov Fit]\n Residuals with Actual Errors'
    )
423.        plt.show()
424.
425.        # Calculate chi-square value
426.        v_chisq = np.sum(np.power(((u-v_u)/u_err), 2))
427.        v_reduced_chisq = v_chisq/ (42-4)
428.        print('[Cox Voinov Fit] Chi-Sq:', v_chisq)
429.        print('[Cox Voinov Fit] Reduced Chi-Sq:', v_reduced_chisq)
430.
431.        # Comment
432.        print('[Cox Voinov Fit] Comment: Ok Fit.')
433.
434.        print('[Cox Voinov Fit] Fitted Equation: ax^3 + b')
435.
436.        # Show errors on coefficients
437.        print('[Cox Voinov Fit] a = %f +-
    %f' % (v_coeff[0], math.sqrt(v_covr[0][0])))
438.        print('[Cox Voinov Fit] b = %f +-
    %f' % (v_coeff[1], math.sqrt(v_covr[1][1])))
439.
440.
441.        #================================================
442.        #       COMPARE ALL CHI-SQ VALUES
443.        #================================================
444.        print('\n[Quadratic Fit] Reduced Chi-Sq:', q_reduced_chisq)
445.        print('[Cubic Fit] Reduced Chi-Sq:', c_reduced_chisq)
446.        print('[Linear Fit] Reduced Chi-Sq:', l_reduced_chisq)
447.        print('[de Gennes Fit] Reduced Chi-Sq:', g_reduced_chisq)
448.        print('[Cox Voinov Fit] Reduced Chi-Sq:', v_reduced_chisq)
449.
450.
451.
452.
453.
454.
455.
456.
457.
458.
459.
460.
461.
462.
463.
464.
465.
466.
467.
468.
```

Appendix 2 – Python Script for Analysis of Top View of Droplet 1

```
# -*- coding: utf-8 -*-
"""
Created on Mon Mar  5 13:07:55 2018

@author: mbcxajg2
**************************************
    DATA ANALYSIS: DROP 1 TOP VIEW
**************************************

"""

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

#====================================
#           FUNCTION 1
#====================================
# Returns (contact angle/degree, contact angle error/degree) from inputs: mean radius, mean radius error
def get_theta(mean_radius, mean_radius_error):
    """
    float, float -> tuple
    Returns (contact_angle, contact_angle_error) from mean radius
    """
    r = mean_radius * 1e-06                 # convert into SI units
    r_error = mean_radius_error * 1e-06     # convert into SI units

    # solve for roots of H
    h_roots = np.roots([math.pi/6, 0, math.pi*3*(math.pow(r, 2))/6, (-7.6*1e-15)])
    # only want real root of H_roots
    h = h_roots.real[abs(h_roots.imag)<1e-5]
    theta = (math.pi/2) - math.atan(((r*r)-(h*h))/(2*r*h))

    # calculate h_error by varing r by +- r_error and recalculating h
    # compute h with r = r + r_error
    h_roots_positve_r_error = np.roots([math.pi/6, 0, math.pi*3*(math.pow((r + r_error), 2))/6, (-7.6*1e-15)])
    h_positive_r_error = h_roots.real[abs(h_roots_positve_r_error.imag)<1e-5]
    # compute h with r = r - r_error
    h_roots_negative_r_error = np.roots([math.pi/6, 0, math.pi*3*(math.pow((r - r_error), 2))/6, (-7.6*1e-15)])
    h_negative_r_error = h_roots.real[abs(h_roots_negative_r_error.imag)<1e-5]
    # take the larget h_error obtained
    if abs(h_positive_r_error - h) > abs(h_negative_r_error - h):
        h_error = abs(h_positive_r_error - h)
    else:
        h_error = abs(h_negative_r_error - h)

    # compute dtheta/dr
    dtheta_dr = -(1/(1+math.pow(((math.pow(r,2)-math.pow(h,2))/(2*r*h)),2)))*((1/(2*h))-(h/(2*r*r)))
    # compute dtheta/dh
    dtheta_dh = -(1/(1+math.pow((((r*r)-(h*h))/(2*r*h)),2)))*(-(r/(2*h*h))-(1/(2*r)))
    # compute theta_error
    theta_error = math.sqrt( (dtheta_dr*dtheta_dr*r_error*r_error) + (dtheta_dh*dtheta_dh*h_error*h_error) )

    return (theta*180/math.pi, theta_error*180/math.pi)

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

```
#           FUNCTION 2
#====================================
# Returns (speed/um/s, speed_error/um/s) from inputs:
#   mean radius, previous mean radius, next mean radius,
#   time at mean radius, time at previous mean radius, time at next mean radius,
#   mean radius error, previous mean radius error, next mean radius error
def get_speed(r, prev_r, next_r, t, prev_t, next_t, r_err, prev_r_err, next_r_err):

    """
    float * 9 -> tuple
    Calculates contact line speed for every mean radius value except first and last mean radius values
    Returns (speed/, speed_error)
    """
    v1 = (r - prev_r) / (t - prev_t)  # speed at an instance before r
    v2 = (next_r - r) / (next_t - t)  # speed at an instance after r
    v = (v1 + v2) / 2  # speed at r
    v1_err = math.sqrt( ( (1 / (t - prev_t))**2 ) * (r_err**2 + prev_r_err**2) )
    v2_err = math.sqrt( ( (1 / (next_t - t))**2 ) * (next_r_err**2 + r_err**2) )
    v_err = math.sqrt( ((1/4) * (v1_err**2)) + ((1/4) * (v2_err**2)) )
    return (v, v_err)


#=========================================
#                LOAD DATA
#=========================================
# rx_data[:, 0] -> time/s | rx_data[:, 1] -> radius/um
r1_data = np.loadtxt('Top_view_drop_1_data_run1.txt')
r2_data = np.loadtxt('Top_view_drop_1_data_run2.txt')
r3_data = np.loadtxt('Top_view_drop_1_data_run3.txt')
data_len = len(r1_data)

#=========================================
#                  TIME
#=========================================
time = r1_data[:, 0]        # time values are same for r1_data to r3_data

#=========================================
#     MEAN RADIUS (UM) AND ERROR
#=========================================
# Calculate mean radius values
mean_r = (r1_data[:, 1] + r2_data[:, 1] + r3_data[:, 1]) / 3

# Array to store mean radius error values
mean_r_err = np.zeros(data_len)

       # Calculate mean radius error values
       for i in range(data_len):
           mean_r_err[i] = np.std([r1_data[i][1], r2_data[i][1], r3_data[i][1]]) / math.sqrt(2)

       #=============================================
       #     CONTACT ANGLE (degree) AND ERROR
       #=============================================
       # Arrays to store contact angle and error
       theta = np.zeros(data_len)
       theta_err = np.zeros(data_len)

       # Calculate contact angle and error
       for i in range(data_len):
           theta[i], theta_err[i] = get_theta(mean_r[i], mean_r_err[i])

       #================================================
       #     CONTACT LINE SPEED (um/s) AND ERROR
       #================================================
       # Arrays to store contact line speed and  error
```

```
        u = np.zeros(data_len)
        u_err = np.zeros(data_len)

        # Calculate contact line speed and error
        for i in range(data_len):
            if i == 0:  # no speed for first data point
                continue
            elif i == data_len - 1: # no speed for last data point
                continue
            else:
                u[i], u_err[i] = get_speed(mean_r[i], mean_r[i-1], mean_r[i+1], time[i], time[i-1], time[i+1], mean_r_err[i], mean_r_err[i-1], mean_r_err[i+1])

        # Only want arrays with valid speeds
        u = u[1:-1]
        u_err = u_err[1:-1]

        #================================================
        #         PLOT: MEAN RADIUS (UM)
        #                   VS
        #                TIME (S)
        #================================================
        # Plot
        plt.errorbar(time, mean_r, yerr=mean_r_err, marker='o', markersize=2, linestyle='None')
        plt.ylabel('Mean Radius / um')
        plt.xlabel('Time / s')
        plt.grid()
        plt.title('Drop 1 Top View\n Mean Radius vs Time')
        plt.show()

        #================================================
        #           ADJUST CONTACT ANGLE VALUES
        #================================================
        # Only want contact angles with valid contact line speeds for graph of
        # contact lines speed against contact angle
        theta = theta[1:-1]
        theta_err = theta_err[1:-1]

        #================================================
        #     PLOT: CONTACT LINE SPEED (UM/S)
        #                   VS
        #           CONTACT ANGLE (DEGREE)
        #================================================
        # Plot
        plt.errorbar(theta, u, xerr=theta_err, yerr=u_err, marker='o', markersize=2, linestyle='None')
        plt.ylabel('Contact Line Speed / um/s')
        plt.xlabel('Contact Angle / degree')
        plt.grid()
        plt.title('Drop 1 Top View\n Contact Line Speed vs Contact Angle')
        plt.show()

        #================================================
        #          QUADRATIC FIT AND ANALYSIS
        #================================================
        # Calculate coefficients and error in coefficients
        (q_coeff, q_covr) = np.polyfit(theta, u, 2, cov=True)

        # Calculate u values of quadratic fit
        q_u = np.polyval(q_coeff, theta)

        # Calculate statistically expected errors for the fit
        q_u_expected_err = np.sqrt((1/(1744-3))*(np.sum(np.power(u-q_u, 2)))) # this value is a constant
```

```
        q_u_expected_err = q_u_expected_err * np.ones(len(q_u)) # create an array of this value

        # Plot fitted u against theta
        plt.errorbar(theta, q_u, yerr=u_err, marker='o', markersize=3)
        plt.ylabel('Contact Line Speed / um/s')
        plt.xlabel('Contact Angle / degree')
        plt.grid()
        plt.title('Drop 1 Top View [Quadratic Fit]\n Contact Line Speed vs Contact Angle')
        plt.show()

        # Plot residuals with statistically expected errors
        plt.errorbar(theta, q_u-u, yerr=q_u_expected_err, marker='o', markersize=2,  linestyle='None')
        plt.ylabel('Residuals / um/s')
        plt.xlabel('Contact Angle / degree')
        plt.grid()
        plt.title('Drop 1 Top View [Quadratic Fit]\n Residuals with Statistically Expected Errors')
        plt.show()

        # Plot residuals with actual errors
        plt.errorbar(theta, q_u-u, yerr=u_err, marker='o', markersize=2, linestyle='None')
        plt.ylabel('Residuals / um/s')
        plt.xlabel('Contact Angle / degree')
        plt.grid()
        plt.title('Drop 1 Top View [Quadratic Fit]\n Residuals with Actual Errors')

        plt.show()

        # Calculate chi-square value
        q_chisq = np.sum(np.power(((u-q_u)/u_err), 2))
        q_reduced_chisq = q_chisq/ (1744-3)
        print('[Quadratic Fit] Chi-Sq:', q_chisq)
        print('[Quadratic Fit] Reduced Chi-Sq:', q_reduced_chisq)

        # Comment on Fit
        print('[Quadratic Fit] Comment: Reduced Chi-Sq values within acceptable range. Good fit.')

        # Show quadratic equation
        print('[Quadratic Fit] Fitted Equation: ax^2 + bx +c')

        # Show errors on coefficients
        print('[Quadratic Fit] a = %f +- %f' % (q_coeff[0], math.sqrt(q_covr[0][0])))
        print('[Quadratic Fit] b = %f +- %f' % (q_coeff[1], math.sqrt(q_covr[1][1])))
        print('[Quadratic Fit] c = %f +- %f' % (q_coeff[2], math.sqrt(q_covr[2][2])))

        #================================================
        #            CUBIC FIT AND ANALYSIS
        #================================================
        # Calculate coefficients and error in coefficients
        (c_coeff, c_covr) = np.polyfit(theta, u, 3, cov=True)

        # Calculate u values of cubic fit
        c_u = np.polyval(c_coeff, theta)

        # Calculate statistically expected errors for the fit
        c_u_expected_err = np.sqrt((1/(1744-4))*(np.sum(np.power(u-c_u, 2)))) # this value is a constant
```

```
235.        c_u_expected_err = c_u_expected_err * np.ones(len(c_u)) # create an array of
    this value
236.
237.        # Plot fitted u against theta
238.        plt.errorbar(theta, c_u, yerr=u_err, marker='o')
239.        plt.ylabel('Contact Line Speed / um/s')
240.        plt.xlabel('Contact Angle / degree')
241.        plt.grid()
242.        plt.title('Drop 1 Top View [Cubic Fit]\n Contact Line Speed vs Contact Angle
    ')
243.        plt.show()
244.
245.        # Plot residuals with statistically expected errors
246.        plt.errorbar(theta, c_u-
    u, yerr=c_u_expected_err, marker='o', linestyle='None')
247.        plt.ylabel('Residuals / um/s')
248.        plt.xlabel('Contact Angle / degree')
249.        plt.grid()
250.        plt.title('Drop 1 Top View [Cubic Fit]\n Residuals with Statistically Expect
    ed Errors')
251.        plt.show()
252.
253.        # Plot residuals with actual errors
254.        plt.errorbar(theta, c_u-u, yerr=u_err, marker='o', linestyle='None')
255.        plt.ylabel('Residuals / um/s')
256.        plt.xlabel('Contact Angle / degree')
257.        plt.grid()
258.        plt.title('Drop 1 Top View [Cubic Fit]\n Residuals with Actual Errors')
259.        plt.show()
260.
261.        # Calculate chi-square value
262.        c_chisq = np.sum(np.power(((u-c_u)/u_err), 2))
263.        c_reduced_chisq = c_chisq/ (1744-4)
264.        print('[Cubic Fit] Chi-Sq:', c_chisq)
265.        print('[Cubic Fit] Reduced Chi-Sq:', c_reduced_chisq)
266.
267.        # Comment
268.        print('[Cubic Fit] Comment: Not as good as quadratic fit.')
269.
270.        # Show cubic equation
271.        print('[Cubic Fit] Fitted Equation: ax^3 + bx^2 + cx + d')
272.
273.        # Show errors on coefficients
274.        print('[Cubic Fit] a = %f +- %f' % (c_coeff[0], math.sqrt(c_covr[0][0])))
275.        print('[Cubic Fit] b = %f +- %f' % (c_coeff[1], math.sqrt(c_covr[1][1])))
276.        print('[Cubic Fit] c = %f +- %f' % (c_coeff[2], math.sqrt(c_covr[2][2])))
277.        print('[Cubic Fit] d = %f +- %f' % (c_coeff[3], math.sqrt(c_covr[3][3])))
278.
279.        #================================================
280.        #           LINEAR FIT AND ANALYSIS
281.        #================================================
282.        # Calculate coefficients and error in coefficients
283.        (l_coeff, l_covr) = np.polyfit(theta, u, 1, cov=True)
284.
285.        # Calculate u values of linear fit
286.        l_u = np.polyval(l_coeff, theta)
287.
288.        # Calculate statistically expected errors for the fit
289.        l_u_expected_err = np.sqrt((1/(1744-2))*(np.sum(np.power(u-
    l_u, 2)))) # this value is a constant
290.        l_u_expected_err = l_u_expected_err * np.ones(len(l_u)) # create an array of
    this value
291.
292.        # Plot fitted u against theta
293.        plt.errorbar(theta, l_u, yerr=u_err, marker='o')
294.        plt.ylabel('Contact Line Speed / um/s')
```

```
295.        plt.xlabel('Contact Angle / degree')
296.        plt.grid()
297.        plt.title('Drop 1 Top View [Linear Fit]\n Contact Line Speed vs Contact Angl
    e')
298.        plt.show()
299.
300.        # Plot residuals with statistically expected errors
301.        plt.errorbar(theta, l_u-
    u, yerr=l_u_expected_err, marker='o', linestyle='None')
302.        plt.ylabel('Residuals / um/s')
303.        plt.xlabel('Contact Angle / degree')
304.        plt.grid()
305.        plt.title('Drop 1 Top View [Linear Fit]\n Residuals with Statistically Expec
    ted Errors')
306.        plt.show()
307.
308.        # Plot residuals with actual errors
309.        plt.errorbar(theta, l_u-u, yerr=u_err, marker='o', linestyle='None')
310.        plt.ylabel('Residuals / um/s')
311.        plt.xlabel('Contact Angle / degree')
312.        plt.grid()
313.        plt.title('Drop 1 Top View [Linear Fit]\n Residuals with Actual Errors')
314.        plt.show()
315.
316.        # Calculate chi-square value
317.        l_chisq = np.sum(np.power(((u-l_u)/u_err), 2))
318.        l_reduced_chisq = l_chisq/ (1744-2)
319.        print('[Linear Fit] Chi-Sq:', l_chisq)
320.        print('[Linear Fit] Reduced Chi-Sq:', l_reduced_chisq)
321.
322.        # Comment
323.        print('[Linear Fit] Comment: Not as good as quadratic fit.')
324.
325.        print('[Linear Fit] Fitted Equation: ax + b')
326.
327.        # Show errors on coefficients
328.        print('[Linear Fit] a = %f +- %f' % (l_coeff[0], math.sqrt(l_covr[0][0])))
329.        print('[Linear Fit] b = %f +- %f' % (l_coeff[1], math.sqrt(l_covr[1][1])))
330.
331.        #=================================================
332.        #            DE GENNES FIT AND ANALYSIS
333.        #=================================================
334.        # Calculate coefficients and error in coefficients
335.        (g_coeff, g_covr) = np.polyfit(theta*theta, u, 1, cov=True)
336.
337.        # Calculate u values of linear fit
338.        g_u = np.polyval(g_coeff, theta*theta)
339.
340.        # Calculate statistically expected errors for the fit
341.        g_u_expected_err = np.sqrt((1/(1744-3))*(np.sum(np.power(u-
    g_u, 2)))) # this value is a constant
342.        g_u_expected_err = g_u_expected_err * np.ones(len(g_u)) # create an array of
     this value
343.
344.        # Plot fitted u against theta*theta
345.        plt.errorbar(theta, g_u, yerr=u_err, marker='o')
346.        plt.ylabel('Contact Line Speed / um/s')
347.        plt.xlabel('Contact Angle / degree')
348.        plt.grid()
349.        plt.title('Drop 1 Top View [de Gennes Fit]\n Contact Line Speed vs Contact A
    ngle')
350.        plt.show()
351.
352.        # Plot residuals with statistically expected errors
353.        plt.errorbar(theta, g_u-
    u, yerr=g_u_expected_err, marker='o', linestyle='None')
```

```
354.        plt.ylabel('Residuals / um/s')
355.        plt.xlabel('Contact Angle / degree')
356.        plt.grid()
357.        plt.title('Drop 1 Top View [de Gennes Fit]\n Residuals with Statistically Ex
    pected Errors')
358.        plt.show()
359.
360.        # Plot residuals with actual errors
361.        plt.errorbar(theta, g_u-u, yerr=u_err, marker='o', linestyle='None')
362.        plt.ylabel('Residuals / um/s')
363.        plt.xlabel('Contact Angle / degree')
364.        plt.grid()
365.        plt.title('Drop 1 Top View [de Gennes Fit]\n Residuals with Actual Errors')

366.        plt.show()
367.
368.        # Calculate chi-square value
369.        g_chisq = np.sum(np.power(((u-g_u)/u_err), 2))
370.        g_reduced_chisq = g_chisq/ (1744-3)
371.        print('[de Gennes Fit] Chi-Sq:', g_chisq)
372.        print('[de Gennes Fit] Reduced Chi-Sq:', g_reduced_chisq)
373.
374.        # Comment
375.        print('[de Gennes Fit] Comment: Ok Fit.')
376.
377.        print('[de Gennes Fit] Fitted Equation: ax + b')
378.
379.        # Show errors on coefficients
380.        print('[de Gennes Fit] a = %f +-
    %f' % (g_coeff[0], math.sqrt(g_covr[0][0])))
381.        print('[de Gennes Fit] b = %f +-
    %f' % (g_coeff[1], math.sqrt(g_covr[1][1])))
382.
383.
384.        #=====================================================
385.        #            COX VOINOV FIT AND ANALYSIS
386.        #=====================================================
387.        # Calculate coefficients and error in coefficients
388.        (v_coeff, v_covr) = np.polyfit(theta*theta*theta, u, 1, cov=True)
389.
390.        # Calculate u values of linear fit
391.        v_u = np.polyval(v_coeff, theta*theta*theta)
392.
393.        # Calculate statistically expected errors for the fit
394.        v_u_expected_err = np.sqrt((1/(1744-4))*(np.sum(np.power(u-
    v_u, 2)))) # this value is a constant
395.        v_u_expected_err = v_u_expected_err * np.ones(len(v_u)) # create an array of
    this value
396.
397.        # Plot fitted u against theta
398.        plt.errorbar(theta, v_u, yerr=u_err, marker='o')
399.        plt.ylabel('Contact Line Speed / um/s')
400.        plt.xlabel('Contact Angle / degree')
401.        plt.grid()
402.        plt.title('Drop 1 Top View [Cox Voinov Fit]\n Contact Line Speed vs Contact
    Angle')
403.        plt.show()
404.
405.        # Plot residuals with statistically expected errors
406.        plt.errorbar(theta, v_u-
    u, yerr=v_u_expected_err, marker='o', linestyle='None')
407.        plt.ylabel('Residuals / um/s')
408.        plt.xlabel('Contact Angle / degree')
409.        plt.grid()
410.        plt.title('Drop 1 Top View [Cox Voinov Fit]\n Residuals with Statistically E
    xpected Errors')
```

```
411.        plt.show()
412.
413.        # Plot residuals with actual errors
414.        plt.errorbar(theta, v_u-u, yerr=u_err, marker='o', linestyle='None')
415.        plt.ylabel('Residuals / um/s')
416.        plt.xlabel('Contact Angle / degree')
417.        plt.grid()
418.        plt.title('Drop 1 Top View [Cox Voinov Fit]\n Residuals with Actual Errors')
419.        plt.show()
420.
421.        # Calculate chi-square value
422.        v_chisq = np.sum(np.power(((u-v_u)/u_err), 2))
423.        v_reduced_chisq = v_chisq/ (1744-4)
424.        print('[Cox Voinov Fit] Chi-Sq:', v_chisq)
425.        print('[Cox Voinov Fit] Reduced Chi-Sq:', v_reduced_chisq)
426.
427.        # Comment
428.        print('[Cox Voinov Fit] Comment: Ok Fit.')
429.
430.        print('[Cox Voinov Fit] Fitted Equation: ax^3 + b')
431.
432.        # Show errors on coefficients
433.        print('[Cox Voinov Fit] a = %f +-
    %f' % (v_coeff[0], math.sqrt(v_covr[0][0])))
434.        print('[Cox Voinov Fit] b = %f +-
    %f' % (v_coeff[1], math.sqrt(v_covr[1][1])))
435.
436.
437.        #================================================
438.        #       COMPARE ALL CHI-SQ VALUES
439.        #================================================
440.        print('\n[Quadratic Fit] Reduced Chi-Sq:', q_reduced_chisq)
441.        print('[Cubic Fit] Reduced Chi-Sq:', c_reduced_chisq)
442.        print('[Linear Fit] Reduced Chi-Sq:', l_reduced_chisq)
443.        print('[de Gennes Fit] Reduced Chi-Sq:', g_reduced_chisq)
444.        print('[Cox Voinov Fit] Reduced Chi-Sq:', v_reduced_chisq)
```

Appendix 3 – Python Script for Analysis of Top View of Droplet 2

```
# -*- coding: utf-8 -*-
"""
Created on Mon Mar  5 14:53:08 2018

@author: mbcxajg2
**************************************
    DATA ANALYSIS: DROP 2 TOP VIEW
**************************************
"""

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

#=====================================
#           FUNCTION 1
#=====================================
# Returns (contact angle/degree, contact angle error/degree) from inputs: mean radius, mean radius error
def get_theta(mean_radius, mean_radius_error):
    """
    float, float -> tuple
    Returns (contact_angle, contact_angle_error) from mean radius
    """
    r = mean_radius * 1e-06                 # convert into SI units
    r_error = mean_radius_error * 1e-06     # convert into SI units

    # solve for roots of H
    h_roots = np.roots([math.pi/6, 0, math.pi*3*(math.pow(r, 2))/6, (-7.6*1e-15)])
    # only want real root of H_roots
    h = h_roots.real[abs(h_roots.imag)<1e-5]
    theta = (math.pi/2) - math.atan(((r*r)-(h*h))/(2*r*h))

    # calculate h_error by varing r by +- r_error and recalculating h
    # compute h with r = r + r_error
    h_roots_positve_r_error = np.roots([math.pi/6, 0, math.pi*3*(math.pow((r + r_error), 2))/6, (-7.6*1e-15)])
    h_positive_r_error = h_roots.real[abs(h_roots_positve_r_error.imag)<1e-5]
    # compute h with r = r - r_error
    h_roots_negative_r_error = np.roots([math.pi/6, 0, math.pi*3*(math.pow((r - r_error), 2))/6, (-7.6*1e-15)])
    h_negative_r_error = h_roots.real[abs(h_roots_negative_r_error.imag)<1e-5]
    # take the larget h_error obtained
    if abs(h_positive_r_error - h) > abs(h_negative_r_error - h):
        h_error = abs(h_positive_r_error - h)
    else:
        h_error = abs(h_negative_r_error - h)

    # compute dtheta/dr
    dtheta_dr = -(1/(1+math.pow(((math.pow(r,2)-math.pow(h,2))/(2*r*h)),2)))*((1/(2*h))-(h/(2*r*r)))
    # compute dtheta/dh
    dtheta_dh = -(1/(1+math.pow((((r*r)-(h*h))/(2*r*h)),2)))*(-(r/(2*h*h))-(1/(2*r)))
    # compute theta_error
    theta_error = math.sqrt( (dtheta_dr*dtheta_dr*r_error*r_error) + (dtheta_dh*dtheta_dh*h_error*h_error) )

    return (theta*180/math.pi, theta_error*180/math.pi)

#=====================================
#           FUNCTION 2
```

```
#====================================
# Returns (speed/um/s, speed_error/um/s) from inputs:
#   mean radius, previous mean radius, next mean radius,
#   time at mean radius, time at previous mean radius, time at next mean radius,
#   mean radius error, previous mean radius error, next mean radius error
def get_speed(r, prev_r, next_r, t, prev_t, next_t, r_err, prev_r_err, next_r_err):

    """
    float * 9 -> tuple
    Calculates contact line speed for every mean radius value except first and last mean radius values
    Returns (speed/, speed_error)
    """
    v1 = (r - prev_r) / (t - prev_t)  # speed at an instance before r
    v2 = (next_r - r) / (next_t - t)  # speed at an instance after r
    v = (v1 + v2) / 2  # speed at r
    v1_err = math.sqrt( ( (1 / (t - prev_t))**2 ) * (r_err**2 + prev_r_err**2) )
    v2_err = math.sqrt( ( (1 / (next_t - t))**2 ) * (next_r_err**2 + r_err**2) )
    v_err = math.sqrt( ((1/4) * (v1_err**2)) + ((1/4) * (v2_err**2)) )
    return (v, v_err)


#========================================
#                LOAD DATA
#========================================
# rx_data[:, 0] -> time/s | rx_data[:, 1] -> radius/um
r1_data = np.loadtxt('Top_view_drop_2_data_run1.txt')
r2_data = np.loadtxt('Top_view_drop_2_data_run2.txt')
data_len = len(r1_data)

#========================================
#                  TIME
#========================================
time = r1_data[:, 0]        # time values are same for r1_data to r2_data

#========================================
#     MEAN RADIUS (UM) AND ERROR
#========================================
# Calculate mean radius values
mean_r = (r1_data[:, 1] + r2_data[:, 1]) / 2

# Array to store mean radius error values
mean_r_err = np.zeros(data_len)

# Calculate mean radius error values
        for i in range(data_len):
            mean_r_err[i] = np.std([r1_data[i][1], r2_data[i][1]]) / math.sqrt(1)

        #========================================
        #     CONTACT ANGLE (degree) AND ERROR
        #========================================
        # Arrays to store contact angle and error
        theta = np.zeros(data_len)
        theta_err = np.zeros(data_len)

        # Calculate contact angle and error
        for i in range(data_len):
            theta[i], theta_err[i] = get_theta(mean_r[i], mean_r_err[i])

        #================================================
        #     CONTACT LINE SPEED (um/s) AND ERROR
        #================================================
        # Arrays to store contact line speed and  error
        u = np.zeros(data_len)
        u_err = np.zeros(data_len)

```

```
121.        # Calculate contact line speed and error
122.        for i in range(data_len):
123.            if i == 0:  # no speed for first data point
124.                continue
125.            elif i == data_len - 1: # no speed for last data point
126.                continue
127.            else:
128.                u[i], u_err[i] = get_speed(mean_r[i], mean_r[i-
    1], mean_r[i+1], time[i], time[i-1], time[i+1], mean_r_err[i], mean_r_err[i-
    1], mean_r_err[i+1])
129.
130.        # Only want arrays with valid speeds
131.        u = u[1:-1]
132.        u_err = u_err[1:-1]
133.
134.        #================================================
135.        #         PLOT: MEAN RADIUS (UM)
136.        #                     VS
137.        #                  TIME (S)
138.        #================================================
139.        # Plot
140.        plt.errorbar(time, mean_r, yerr=mean_r_err, marker='o', markersize=2, linest
    yle='None')
141.        plt.ylabel('Mean Radius / um')
142.        plt.xlabel('Time / s')
143.        plt.grid()
144.        plt.title('Drop 2 Top View\n Mean Radius vs Time')
145.        plt.show()
146.
147.        #================================================
148.        #           ADJUST CONTACT ANGLE VALUES
149.        #================================================
150.        # Only want contact angles with valid contact line speeds for graph of
151.        # contact lines speed against contact angle
152.        theta = theta[1:-1]
153.        theta_err = theta_err[1:-1]
154.
155.        #================================================
156.        #     PLOT: CONTACT LINE SPEED (UM/S)
157.        #                     VS
158.        #           CONTACT ANGLE (DEGREE)
159.        #================================================
160.        # Plot
161.        plt.errorbar(theta, u, xerr=theta_err, yerr=u_err, marker='o', markersize=2,
     linestyle='None')
162.        plt.ylabel('Contact Line Speed / um/s')
163.        plt.xlabel('Contact Angle / degree')
164.        plt.grid()
165.        plt.title('Drop 2 Top View\n Contact Line Speed vs Contact Angle')
166.        plt.show()
167.
168.        #================================================
169.        #          QUADRATIC FIT AND ANALYSIS
170.        #================================================
171.        # Calculate coefficients and error in coefficients
172.        (q_coeff, q_covr) = np.polyfit(theta, u, 2, cov=True)
173.
174.        # Calculate u values of quadratic fit
175.        q_u = np.polyval(q_coeff, theta)
176.
177.        # Calculate statistically expected errors for the fit
178.        q_u_expected_err = np.sqrt((1/(458-3))*(np.sum(np.power(u-
    q_u, 2)))) # this value is a constant
179.        q_u_expected_err = q_u_expected_err * np.ones(len(q_u)) # create an array of
     this value
180.
```

```
        # Plot fitted u against theta
        plt.errorbar(theta, q_u, yerr=u_err, marker='o')
        plt.ylabel('Contact Line Speed / um/s')
        plt.xlabel('Contact Angle / degree')
        plt.grid()
        plt.title('Drop 2 Top View [Quadratic Fit]\n Contact Line Speed vs Contact Angle')
        plt.show()

        # Plot residuals with statistically expected errors
        plt.errorbar(theta, q_u-u, yerr=q_u_expected_err, marker='o',  linestyle='None')
        plt.ylabel('Residuals / um/s')
        plt.xlabel('Contact Angle / degree')
        plt.grid()
        plt.title('Drop 2 Top View [Quadratic Fit]\n Residuals with Statistically Expected Errors')
        plt.show()

        # Plot residuals with actual errors
        plt.errorbar(theta, q_u-u, yerr=u_err, marker='o', linestyle='None')
        plt.ylabel('Residuals / um/s')
        plt.xlabel('Contact Angle / degree')
        plt.grid()
        plt.title('Drop 2 Top View [Quadratic Fit]\n Residuals with Actual Errors')

        plt.show()

        # Calculate chi-square value
        q_chisq = np.sum(np.power(((u-q_u)/u_err), 2))
        q_reduced_chisq = q_chisq/ (458-3)
        print('[Quadratic Fit] Chi-Sq:', q_chisq)
        print('[Quadratic Fit] Reduced Chi-Sq:', q_reduced_chisq)

        # Show quadratic equation
        print('[Quadratic Fit] Fitted Equation: ax^2 + bx +c')

        # Show errors on coefficients
        print('[Quadratic Fit] a = %f +- %f' % (q_coeff[0], math.sqrt(q_covr[0][0])))
        print('[Quadratic Fit] b = %f +- %f' % (q_coeff[1], math.sqrt(q_covr[1][1])))
        print('[Quadratic Fit] c = %f +- %f' % (q_coeff[2], math.sqrt(q_covr[2][2])))

        #================================================
        #            CUBIC FIT AND ANALYSIS
        #================================================
        # Calculate coefficients and error in coefficients
        (c_coeff, c_covr) = np.polyfit(theta, u, 3, cov=True)

        # Calculate u values of cubic fit
        c_u = np.polyval(c_coeff, theta)

        # Calculate statistically expected errors for the fit
        c_u_expected_err = np.sqrt((1/(458-4))*(np.sum(np.power(u-c_u, 2)))) # this value is a constant
        c_u_expected_err = c_u_expected_err * np.ones(len(c_u)) # create an array of this value

        # Plot fitted u against theta
        plt.errorbar(theta, c_u, yerr=u_err, marker='o')
        plt.ylabel('Contact Line Speed / um/s')
        plt.xlabel('Contact Angle / degree')
        plt.grid()
```

```
237.        plt.title('Drop 2 Top View [Cubic Fit]\n Contact Line Speed vs Contact Angle
    ')
238.        plt.show()
239.
240.        # Plot residuals with statistically expected errors
241.        plt.errorbar(theta, c_u-
    u, yerr=c_u_expected_err, marker='o', linestyle='None')
242.        plt.ylabel('Residuals / um/s')
243.        plt.xlabel('Contact Angle / degree')
244.        plt.grid()
245.        plt.title('Drop 2 Top View [Cubic Fit]\n Residuals with Statistically Expect
    ed Errors')
246.        plt.show()
247.
248.        # Plot residuals with actual errors
249.        plt.errorbar(theta, c_u-u, yerr=u_err, marker='o', linestyle='None')
250.        plt.ylabel('Residuals / um/s')
251.        plt.xlabel('Contact Angle / degree')
252.        plt.grid()
253.        plt.title('Drop 2 Top View [Cubic Fit]\n Residuals with Actual Errors')
254.        plt.show()
255.
256.        # Calculate chi-square value
257.        c_chisq = np.sum(np.power(((u-c_u)/u_err), 2))
258.        c_reduced_chisq = c_chisq/ (458-4)
259.        print('[Cubic Fit] Chi-Sq:', c_chisq)
260.        print('[Cubic Fit] Reduced Chi-Sq:', c_reduced_chisq)
261.
262.        # Show cubic equation
263.        print('[Cubic Fit] Fitted Equation: ax^3 + bx^2 + cx + d')
264.
265.        # Show errors on coefficients
266.        print('[Cubic Fit] a = %f +- %f' % (c_coeff[0], math.sqrt(c_covr[0][0])))
267.        print('[Cubic Fit] b = %f +- %f' % (c_coeff[1], math.sqrt(c_covr[1][1])))
268.        print('[Cubic Fit] c = %f +- %f' % (c_coeff[2], math.sqrt(c_covr[2][2])))
269.        print('[Cubic Fit] d = %f +- %f' % (c_coeff[3], math.sqrt(c_covr[3][3])))
270.
271.        #================================================
272.        #            LINEAR FIT AND ANALYSIS
273.        #================================================
274.        # Calculate coefficients and error in coefficients
275.        (l_coeff, l_covr) = np.polyfit(theta, u, 1, cov=True)
276.
277.        # Calculate u values of linear fit
278.        l_u = np.polyval(l_coeff, theta)
279.
280.        # Calculate statistically expected errors for the fit
281.        l_u_expected_err = np.sqrt((1/(458-2))*(np.sum(np.power(u-
    l_u, 2)))) # this value is a constant
282.        l_u_expected_err = l_u_expected_err * np.ones(len(l_u)) # create an array of
     this value
283.
284.        # Plot fitted u against theta
285.        plt.errorbar(theta, l_u, yerr=u_err, marker='o')
286.        plt.ylabel('Contact Line Speed / um/s')
287.        plt.xlabel('Contact Angle / degree')
288.        plt.grid()
289.        plt.title('Drop 2 Top View [Linear Fit]\n Contact Line Speed vs Contact Angl
    e')
290.        plt.show()
291.
292.        # Plot residuals with statistically expected errors
293.        plt.errorbar(theta, l_u-
    u, yerr=l_u_expected_err, marker='o', linestyle='None')
294.        plt.ylabel('Residuals / um/s')
295.        plt.xlabel('Contact Angle / degree')
```

```
296.        plt.grid()
297.        plt.title('Drop 2 Top View [Linear Fit]\n Residuals with Statistically Expec
    ted Errors')
298.        plt.show()
299.
300.        # Plot residuals with actual errors
301.        plt.errorbar(theta, l_u-u, yerr=u_err, marker='o', linestyle='None')
302.        plt.ylabel('Residuals / um/s')
303.        plt.xlabel('Contact Angle / degree')
304.        plt.grid()
305.        plt.title('Drop 2 Top View [Linear Fit]\n Residuals with Actual Errors')
306.        plt.show()
307.
308.        # Calculate chi-square value
309.        l_chisq = np.sum(np.power(((u-l_u)/u_err), 2))
310.        l_reduced_chisq = l_chisq/ (458-2)
311.        print('[Linear Fit] Chi-Sq:', l_chisq)
312.        print('[Linear Fit] Reduced Chi-Sq:', l_reduced_chisq)
313.
314.        print('[Linear Fit] Fitted Equation: ax + b')
315.
316.        # Show errors on coefficients
317.        print('[Linear Fit] a = %f +- %f' % (l_coeff[0], math.sqrt(l_covr[0][0])))
318.        print('[Linear Fit] b = %f +- %f' % (l_coeff[1], math.sqrt(l_covr[1][1])))
319.
320.        #=================================================
321.        #           DE GENNES FIT AND ANALYSIS
322.        #=================================================
323.        # Calculate coefficients and error in coefficients
324.        (g_coeff, g_covr) = np.polyfit(theta*theta, u, 1, cov=True)
325.
326.        # Calculate u values of linear fit
327.        g_u = np.polyval(g_coeff, theta*theta)
328.
329.        # Calculate statistically expected errors for the fit
330.        g_u_expected_err = np.sqrt((1/(458-3))*(np.sum(np.power(u-
    g_u, 2)))) # this value is a constant
331.        g_u_expected_err = g_u_expected_err * np.ones(len(g_u)) # create an array of
     this value
332.
333.        # Plot fitted u against theta
334.        plt.errorbar(theta, g_u, yerr=u_err, marker='o')
335.        plt.ylabel('Contact Line Speed / um/s')
336.        plt.xlabel('Contact Angle / degree')
337.        plt.grid()
338.        plt.title('Drop 2 Top View [de Gennes Fit]\n Contact Line Speed vs Contact A
    ngle')
339.        plt.show()
340.
341.        # Plot residuals with statistically expected errors
342.        plt.errorbar(theta, g_u-
    u, yerr=g_u_expected_err, marker='o', linestyle='None')
343.        plt.ylabel('Residuals / um/s')
344.        plt.xlabel('Contact Angle / degree')
345.        plt.grid()
346.        plt.title('Drop 2 Top View [de Gennes Fit]\n Residuals with Statistically Ex
    pected Errors')
347.        plt.show()
348.
349.        # Plot residuals with actual errors
350.        plt.errorbar(theta, g_u-u, yerr=u_err, marker='o', linestyle='None')
351.        plt.ylabel('Residuals / um/s')
352.        plt.xlabel('Contact Angle / degree')
353.        plt.grid()
354.        plt.title('Drop 2 Top View [de Gennes Fit]\n Residuals with Actual Errors')
```

```
355.        plt.show()
356.
357.        # Calculate chi-square value
358.        g_chisq = np.sum(np.power(((u-g_u)/u_err), 2))
359.        g_reduced_chisq = g_chisq/ (458-3)
360.        print('[de Gennes Fit] Chi-Sq:', g_chisq)
361.        print('[de Gennes Fit] Reduced Chi-Sq:', g_reduced_chisq)
362.
363.        # Comment
364.        print('[de Gennes Fit] Comment: Ok Fit.')
365.
366.        print('[de Gennes Fit] Fitted Equation: ax^2 + b')
367.
368.        # Show errors on coefficients
369.        print('[de Gennes Fit] a = %f +-
    %f' % (g_coeff[0], math.sqrt(g_covr[0][0])))
370.        print('[de Gennes Fit] b = %f +-
    %f' % (g_coeff[1], math.sqrt(g_covr[1][1])))
371.
372.        #=================================================
373.        #           COX VOINOV FIT AND ANALYSIS
374.        #=================================================
375.        # Calculate coefficients and error in coefficients
376.        (v_coeff, v_covr) = np.polyfit(theta*theta*theta, u, 1, cov=True)
377.
378.        # Calculate u values of linear fit
379.        v_u = np.polyval(v_coeff, theta*theta*theta)
380.
381.        # Calculate statistically expected errors for the fit
382.        v_u_expected_err = np.sqrt((1/(458-4))*(np.sum(np.power(u-
    v_u, 2)))) # this value is a constant
383.        v_u_expected_err = v_u_expected_err * np.ones(len(v_u)) # create an array of
    this value
384.
385.        # Plot fitted u against theta
386.        plt.errorbar(theta, v_u, yerr=u_err, marker='o')
387.        plt.ylabel('Contact Line Speed / um/s')
388.        plt.xlabel('Contact Angle / degree')
389.        plt.grid()
390.        plt.title('Drop 2 Top View [Cox Voinov Fit]\n Contact Line Speed vs Contact
    Angle')
391.        plt.show()
392.
393.        # Plot residuals with statistically expected errors
394.        plt.errorbar(theta, v_u-
    u, yerr=v_u_expected_err, marker='o', linestyle='None')
395.        plt.ylabel('Residuals / um/s')
396.        plt.xlabel('Contact Angle / degree')
397.        plt.grid()
398.        plt.title('Drop 2 Top View [Cox Voinov Fit]\n Residuals with Statistically E
    xpected Errors')
399.        plt.show()
400.
401.        # Plot residuals with actual errors
402.        plt.errorbar(theta, v_u-u, yerr=u_err, marker='o', linestyle='None')
403.        plt.ylabel('Residuals / um/s')
404.        plt.xlabel('Contact Angle / degree')
405.        plt.grid()
406.        plt.title('Drop 2 Top View [Cox Voinov Fit]\n Residuals with Actual Errors')

407.        plt.show()
408.
409.        # Calculate chi-square value
410.        v_chisq = np.sum(np.power(((u-v_u)/u_err), 2))
411.        v_reduced_chisq = v_chisq/ (458-4)
412.        print('[Cox Voinov Fit] Chi-Sq:', v_chisq)
```

```
413.        print('[Cox Voinov Fit] Reduced Chi-Sq:', v_reduced_chisq)
414.
415.        # Comment
416.        print('[Cox Voinov Fit] Comment: Ok Fit.')
417.
418.        print('[Cox Voinov Fit] Fitted Equation: ax^3 + b')
419.
420.        # Show errors on coefficients
421.        print('[Cox Voinov Fit] a = %f +-
    %f' % (v_coeff[0], math.sqrt(v_covr[0][0])))
422.        print('[Cox Voinov Fit] b = %f +-
    %f' % (v_coeff[1], math.sqrt(v_covr[1][1])))
423.
424.
425.        #================================================
426.        #       COMPARE ALL CHI-SQ VALUES
427.        #================================================
428.        print('\n[Quadratic Fit] Reduced Chi-Sq:', q_reduced_chisq)
429.        print('[Cubic Fit] Reduced Chi-Sq:', c_reduced_chisq)
430.        print('[Linear Fit] Reduced Chi-Sq:', l_reduced_chisq)
431.        print('[de Gennes Fit] Reduced Chi-Sq:', g_reduced_chisq)
432.        print('[Cox Voinov Fit] Reduced Chi-Sq:', v_reduced_chisq)
```