Program for Derivation of the Shape of Raindrops

Brian Lim*
School of Applied and Engineering Physics, Cornell University
Ithaca, NY 14853
(Dated: May 19, 2006)

This document contains a sample output and all the source code for the program used to calculate raindrop shapes.

I. SAMPLE PROGRAM OUTPUT

The following is a sample output of the summary (Stage 5) at the end of the run for $d = 4.3755$mm, $U = 8.3$m/s.

```
==================================
 Raindrop Calculation
 -------------------------------
 Run at: Wed May 17 14:19:31 2006
==================================

--------------------------------------
Input parameters
----------------

d = 4.38 mm
U = 8.30 m/s


--------------------------------------
Processed parameters
--------------------

Vs = 43.8613189169097810 mm^3
Re = 2266.86
Cdp_emp = 3.5600068037701188e-001


--------------------------------------
Critical adjustable parameters
------------------------------

Rt = 2.8200520621871144e-003
c = 8.7872412806065103e-001

A = 1.7683852433685803e-005
ax = 6.8177556817222018e-001

Ga = 1.4149331104496898e+000
Gd = 1.3549313015188618e+000

Lam = 5.4400000000000004e-001

--------------------------------------
Error constraints
-----------------

RES_DEG = 1
BRACKET_ERR = 1.0e-006
NEWTON_ERR = 1.0e-005
INTEGRATION_ERR = 1.0e-005
ERR_PHI = 1.0e-004
ERR_PHI_FACTOR = 10

--------------------------------------
Stage 5
-------

see Results

--------------------------------------
Results
-------

sh.closed = yes

Rt = 2.8200520621871144e-003
sh.V  = 4.6977752379960239e-008
sh.A  = 1.8328743969958099e-005

Lam  = 5.4400000000000004e-001
Gd  = 1.3549313015188618e+000
sh.ax  = 7.8812618371459087e-001

sh.SA  = 6.3675843635142203e-005

D  = 2.6094077115144178e-004
W  = 4.5934454085708263e-004


--------------------------------------
Time elapsed: 0.67 seconds
--------------------------------------
```

*E-mail: bl223@cornell.edu

II. C PROGRAM CODE

A C program was written to do the number crunching to calculate the coordinates for the raindrop shape.

A. main.c

```c
/*============================================================================*
 * main.c *
 * Main file that executes the program in stages.
 *============================================================================*/

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include <stdbool.h>

#include <gsl/gsl_errno.h>
#include <gsl/gsl_math.h>
#include <gsl/gsl_spline.h>

#include "array.h"
#include "project.h"
#include "nummethods.h"
#include "shape.h"


#include <gsl/gsl_spline.h>

/*============================================================================*/
/* Internally used structs */

struct shape_vol_params {
    double U;
    double c;
    PRES_DIST * pdist;
    PHYS_PROPS phys;

    double Vs; // volume of sphere, i.e. desired volume

    // used only in balanceWeight
    double Rt; // radius of curvature
    double D; // drag force
};
typedef struct shape_vol_params SHAPE_VOL_PARAMS;

struct dCdp_integ_params {
    PRES_DIST * pdist;
    double Cdp_emp;
};
typedef struct dCdp_integ_params DCDP_INTEG_PARAMS;

// used to mark a horizontal bar for printing
char* hr = "\n-----------------------------------------------------------\n";

```

```
50  /*=========================================================================*/
51
52  void printResults(SHAPE sh, double Rt, PRES_DIST pdist,
53                    double U, double Cdp_emp, PHYS_PROPS phys);
54
55  double balanceVolume(double Rt_lo, double Rt_hi,
56                       double Vs,
57                       double U, double c, double d,
58                       PRES_DIST * pdist, PHYS_PROPS phys);
59  double balanceWeight(double Rt_lo, double Rt_hi,
60                       double Vs, double Rt, double D,
61                       double U, double c, double d,
62                       PRES_DIST * pdist, PHYS_PROPS phys);
63  double balanceDrag(double Gd_lo, double Gd_hi,
64                     double ax,
65                     PRES_DIST * pdist, double Cdp_emp);
66
67  double bracket_ShapeVolume(double Rt, void * params);
68  double bracket_ShapeWeight(double Lam, void * params);
69
70  double bracket_Drag(double Gd, void * params);
71  inline double dCdp_integ(double ti, void * params);
72  inline double Cdp_empirical(double Re);
73  inline double cp(double Re);
74
75  /*=========================================================================*/
76
77  int main(int argc, char *argv[]) {
78
79      /*-------------------------------------------------------------------*/
80      /* Set up physical conditions */
81
82      PHYS_PROPS phys;
83      phys.st = WATER_AIR_SURFACE_TENSION;
84      phys.rho_w = WATER_DENSITY;
85      phys.rho_a = AIR_DENSITY;
86      phys.vis_a = AIR_VISCOSITY;
87      phys.g = GRAVITY_SEA_LEVEL;
88
89      /*-------------------------------------------------------------------*/
90
91      // note that U cannot be too big, or else the shape is invalid
92      // should test under poorer error first
93      double d = 5e-3; // diameter of equivalent sized sphere
94      double U = 9.65;
95
96      /*-------------------------------------------------------------------*/
97
98      double Re = phys.rho_a*U*d/phys.vis_a;
99      double Cdp_emp = Cdp_empirical(Re);
100
101     /*-------------------------------------------------------------------*/
102     /* Calculate volume of sphere with diameter d, which is also the required
103      * volume for the raindrop.
104      */
105     double q = d/2.0;
106     double Vs = 4.0/3.0*M_PI*gsl_pow_3(q);
107
```

```
108         double W = Vs * (phys.rho_w - phys.rho_a) * phys.g;
109
110         /*-------------------------------------------------------------------------*/
111         /* Print intro */
112
113         // calculate time
114         time_t rawtime;
115         struct tm * timeinfo;
116         time ( &rawtime );
117         timeinfo = localtime ( &rawtime );
118
119         system("CLS"); // clear screen
120
121         printf("\n");
122         printf("==================================\n");
123         printf(" Raindrop Calculation\n");
124         printf(" --------------------------------\n");
125         printf(" Run at: %s", asctime(timeinfo));
126         printf("==================================\n");
127
128         /*-------------------------------------------------------------------------*/
129         /* Print program parameters */
130
131         printf("%s", hr);
132         printf("Input parameters\n");
133         printf("----------------\n\n");
134
135         printf("d = %.2f mm\n", d *1e3);
136         printf("U = %.2f m/s\n", U);
137         printf("\n");
138
139         printf("%s", hr);
140         printf("Processed parameters\n");
141         printf("--------------------\n\n");
142         printf("Vs = %.16f mm^3\n", Vs *1e9);
143         printf("Re = %.2f\n", Re);
144         printf("Cdp_emp = %.16e\n", Cdp_emp);
145         printf("\n");
146
147         /*-------------------------------------------------------------------------*/
148         /* Read data from file data */
149
150         /* Chose Re=157200 from Fage 1937 because it is the lowest Re with
151          * most points
152          */
153         char * pdistfilename = "../data/fage1937_R157200.txt";
154         DAT dat = {pdistfilename}; // don't need to set the rest
155         loadArrays(&dat);
156         scaleArray(dat.x,M_PI/180.0,dat.len); // scale from degrees to radians
157
158         /*-------------------------------------------------------------------------*/
159         /* */
160         /*
161          * Parameters in this section are to be set prior to each run.
162          */
163
164         // choose stage
165         int stage = 2;
```

```
166
167         // taken from earlier result
168         double Rt = 5.0750733878580055e-004;
169         double ax = 6.9999999999999996e-001;
170
171         double lam = ax/sqrt(1-gsl_pow_2(ax));
172         double Ga = 4.0/9.0 / gsl_pow_2(lam)
173                     / gsl_pow_2((gsl_pow_2(lam)+1)*atan(1/lam)-lam);
174                     // note that acot(t) = atan(t)
175         double Gd = 1;
176         double A = 7.9342085371085720e-007;
177
178         double Lam = .8; // start with 0 and raise via Stage 4
179         double Lam_inc = 1; // to use to help search for highest Lam
180
181         double c = (1 + cp(Re))/2; // for the b factor
182
183         printf("%s", hr);
184         printf("Critical adjustable parameters\n");
185         printf("------------------------------\n\n");
186         printf("Rt = %.16e\n", Rt);
187         printf("c = %.16e\n", c);
188         printf("\n");
189         printf("A = %.16e\n", A);
190         printf("ax = %.16e\n", ax);
191         printf("\n");
192         printf("Ga = %.16e\n", Ga);
193         printf("Gd = %.16e\n", Gd);
194         printf("\n");
195         printf("Lam = %.16e\n", Lam);
196
197         PRES_DIST pdist = {dat.x,dat.y, dat.len,
198                            Ga,Gd, Lam};
199
200         SHAPE sh;
201
202         /*-------------------------------------------------------------------------*/
203         /* Print error constraints */
204
205         printf("%s", hr);
206         printf("Error constraints\n");
207         printf("-----------------\n\n");
208         printf("RES_DEG = %d\n", RES_DEG);
209         printf("BRACKET_ERR = %.1e\n", BRACKET_ERR);
210         printf("NEWTON_ERR = %.1e\n", NEWTON_ERR);
211         printf("INTEGRATION_ERR = %.1e\n", INTEGRATION_ERR);
212         printf("ERR_PHI = %.1e\n", ERR_PHI);
213         printf("ERR_PHI_FACTOR = %d\n", ERR_PHI_FACTOR);
214
215         /*=========================================================================*/
216
217         printf("%s", hr);
218         printf("Stage %d\n", stage);
219         printf("-------\n\n", stage);
220
221         // set start time
222         time_t start = clock();
223
```

```
224         switch (stage) {
225
226         /*-------------------------------------------------------------------------*/
227         /* Balance volume (V = Vs), by Rt iteration */
228
229         case 1: {
230
231             double Rt_lo = .5e-3; // this needs to be callibrated
232             double Rt_hi = 1.0e-3;
233
234             double Rt = balanceVolume(Rt_lo, Rt_hi, Vs, U, c, d, &pdist, phys);
235
236             SHAPE sh = shape(U, Rt, c, pdist, phys);
237
238             printResults(sh, Rt, pdist,  U, Cdp_emp, phys);
239
240             cleanup(sh);
241
242         } break;
243
244         /*-------------------------------------------------------------------------*/
245         /* Balance drag, by Gd iteration */
246
247         case 2: {
248
249             double Gd_lo = 0.1; // this needs to be callibrated
250             double Gd_hi = 5.3;
251
252             Gd = balanceDrag(Gd_lo, Gd_hi, ax, &pdist, Cdp_emp);
253             pdist.Gd = Gd;
254
255             sh = shape(U, Rt, c, pdist, phys);
256
257             printResults(sh, Rt, pdist,  U, Cdp_emp, phys);
258
259             printf("sh.SA  = %.16e\n", sh.SA);
260             printf("\n");
261
262             cleanup(sh);
263
264         } break;
265
266         /*-------------------------------------------------------------------------*/
267         /* Balance weight, by Lam iteration */
268         /* NOT WORKING */
269
270         case 3: {
271
272             printf("DO NOT USE\n");
273
274     //         double Lam_lo = .5; // this needs to be callibrated
275     //         double Lam_hi = 20;
276     //
277     //         // empirical pressure drag multiplied by Lam
278     //         double D = Cdp_emp * phys.rho_a*gsl_pow_2(U)/2 * A; // D = D(U,A)
279     //
280     //         Lam = W / D;
281     //
```

```
282 //        Lam = balanceWeight(Lam_lo, Lam_hi, Vs, Rt, W, U, c, d, &pdist, phys);
283 //        pdist.Lam = Lam;
284 //
285 //        sh = shape(U, Rt, c, pdist, phys);
286 //
287 //        printResults(sh, Rt, pdist,  U, Cdp_emp, phys);
288 //
289 //        cleanup(sh);
290
291     } break;
292
293     /*-------------------------------------------------------------------------*/
294     /* Increase Lam until just before shape is not closed */
295
296     case 4: {
297
298         bool closed;
299
300         Lam_inc = 1;
301
302         while (Lam_inc > 1e-6) {
303             printf("Lam = %.5f\tLam_inc = %.5f\n", pdist.Lam, Lam_inc);
304             do {
305                 sh = shape(U, Rt, c, pdist, phys);
306                 closed = sh.closed;
307                 cleanup(sh);
308                 pdist.Lam += Lam_inc;
309             }
310             while (sh.closed);
311
312             pdist.Lam -= 2*Lam_inc;
313
314             Lam_inc /= 10;
315         }
316         printf("\nLam = %.5f\n\n", pdist.Lam);
317
318     } break;
319
320     /*-------------------------------------------------------------------------*/
321     /* Run the shape once and read results */
322
323     case 5: {
324
325          printf("see Results\n");
326
327         sh = shape(U, Rt, c, pdist, phys);
328
329         printResults(sh, Rt, pdist,  U, Cdp_emp, phys);
330
331         cleanup(sh);
332
333     } break;
334
335     /*-------------------------------------------------------------------------*/
336     /* Print SHAPE coordinates in table form */
337
338     case 6: {
339
```

```
340             sh = shape(U, Rt, c, pdist, phys);
341
342             int i;
343             for (i = 0; i < sh.len; i++) {
344                 printf("%.5e\t%.5e\t%.5e\t%.5e\t%.5e\n",
345                        sh.phi[i], sh.s[i], sh.x[i], sh.z[i], sh.Vi[i]);
346             }
347
348             cleanup(sh);
349
350         } break;
351
352         /*-------------------------------------------------------------------------*/
353
354         } // end switch
355
356         // set end time
357         time_t end = clock();
358
359         printf("%s", hr);
360         printf("Time elapsed: %.2f seconds", difftime(end,start) / CLK_TCK);
361         printf("%s", hr);
362
363         /*=========================================================================*/
364         /* Clean up, deallocate, etc */
365
366         // deallocation for loaded data
367         free(dat.x);
368         free(dat.y);
369
370         /*-------------------------------------------------------------------------*/
371         /* Quit */
372
373         //system("PAUSE");
374         return 0;
375     }
376
377     /*=========================================================================*/
378
379     /**
380      * Prints results summary after each run.
381      */
382     void printResults(SHAPE sh, double Rt, PRES_DIST pdist,
383                       double U, double Cdp_emp, PHYS_PROPS phys) {
384         printf("%s", hr);
385         printf("Results\n");
386         printf("-------\n\n");
387
388         printf("sh.closed = %s", sh.closed?"yes\n":"NO");
389         if (!sh.closed) {printf("\t(at phi = %.10f, x = %.6e)\n",
390                                 sh.phi[sh.len-1], sh.x[sh.len-1]);}
391         printf("\n");
392
393         printf("Rt = %.16e\n", Rt);
394         printf("sh.V  = %.16e\n", sh.V);
395         printf("sh.A  = %.16e\n", sh.A);
396         printf("\n");
397
```

```
398         printf("Lam  = %.16e\n", pdist.Lam);
399         printf("Gd  = %.16e\n", pdist.Gd);
400         printf("sh.ax  = %.16e\n", sh.ax);
401         printf("\n");
402 
403         printf("sh.SA  = %.16e\n", sh.SA);
404         printf("\n");
405 
406         double D = Cdp_emp * phys.rho_a*gsl_pow_2(U)/2 * sh.A;
407         double W = sh.V * phys.rho_w * phys.g;
408         printf("D  = %.16e\n", D);
409         printf("W  = %.16e\n", W);
410         printf("\n");
411     }
412 
413     /*============================================================================*/
414 
415     /**
416      * Returns Rt that balanced volume.
417      */
418     double balanceVolume(double Rt_lo, double Rt_hi,
419                          double Vs,
420                          double U, double c, double d,
421                          PRES_DIST * pdist, PHYS_PROPS phys) {
422         double Rt = Rt_lo;
423 
424         SHAPE_VOL_PARAMS params = {U, c, pdist, phys, Vs};
425 
426         Rt = bracket_root(Rt_lo, Rt_hi, &bracket_ShapeVolume, &params);
427         return Rt;
428     }
429 
430     /**
431      * Returns Lam that balanced weight.
432      */
433     double balanceWeight(double Lam_lo, double Lam_hi,
434                          double Vs, double Rt, double D,
435                          double U, double c, double d,
436                          PRES_DIST * pdist, PHYS_PROPS phys) {
437         double Lam = Lam_lo;
438 
439         SHAPE_VOL_PARAMS params = {U, c, pdist, phys, Vs, Rt, D};
440 
441         Lam = bracket_root(Lam_lo, Lam_hi, &bracket_ShapeWeight, &params);
442         return Lam;
443     }
444 
445     /**
446      * Returns Gd that balanced drag.
447      */
448     double balanceDrag(double Gd_lo, double Gd_hi,
449                        double ax,
450                        PRES_DIST * pdist, double Cdp_emp) {
451         double Gd = 0;
452 
453         /* Don't need prototypical bounds checking
454          */
455 
```

```
456         // set to parameters
457         DCDP_INTEG_PARAMS params2 = {pdist, Cdp_emp};
458 
459         Gd = bracket_root(Gd_lo, Gd_hi, &bracket_Drag, &params2);
460         return Gd;
461     }
462 
463     /*=========================================================================*/
464     /* The following functions are used for bracket root solving for the desired
465      * drag.
466      */
467 
468     /**
469      * Returns the difference between integrated pressure drag coefficient,
470      * Cdp_integ, and empirical coefficient, Cdp_emp, to allow the bracketing root
471      * finder to converge to 0.
472      */
473     double bracket_Drag(double Gd, void * params) {
474         DCDP_INTEG_PARAMS * par_ptr = (DCDP_INTEG_PARAMS *)params;
475         DCDP_INTEG_PARAMS par = *par_ptr;
476 
477         PRES_DIST * pdist = par_ptr->pdist;
478         pdist->Gd = Gd; // set Gd value
479 
480         double Cdp_emp = par.Cdp_emp;
481         double Cdp_integ = integrate(&dCdp_integ, params, 0, M_PI);
482 
483         double diff = Cdp_emp - Cdp_integ;
484         return diff;
485     }
486 
487     /**
488      * Function prepared for integrate(...) in nummethods.c
489      */
490     inline double dCdp_integ(double ti, void * params) {
491         DCDP_INTEG_PARAMS par = *(DCDP_INTEG_PARAMS *)params;
492         PRES_DIST pdist = *(par.pdist);
493         double Lam = pdist.Lam;
494 
495         double dCdp = 2*cpres(pdist, ti)*cos(ti)*sin(ti);
496         return dCdp * Lam;
497     }
498 
499     /**
500      * Cdp = Cd * cp, where cp is from cp(Re)
501      * Cd depends on Re, according to Hughes and Brighton 1991.
502      */
503     double Cdp_empirical(double Re) {
504         // Estimates of drag coefficient for a sphere from Hughes and Brighton 1991
505         double Cd;
506         if (Re < 1) { // 0 <= Re < 1
507             Cd = 24*sqrt(Re);
508         }
509         else if (Re < 3e5) { // 1 <= Re < 3e5
510             Cd = .47;
511         }
512         else { // Re >= 3e5
513             Cd = .2;
```

```
514         }
515
516         double Cdp = Cd * cp(Re);
517         return Cdp;
518     }
519
520     /**
521      * Empirically determined cp = Cdp/Cd from a quartic interpolation for the Re
522      * region between data from LeClair 1970 and Achenbach 1972,
523      * c = exp(P0 + P1*ln(Re) + P2*(ln(Re))^2 + P3*(ln(Re))^3 + P4*(ln(Re))^4).
524      * See Matlab code.
525      */
526     inline double cp(double Re) {
527         double P0 = 5.70988944547302;
528         double P1 = -3.71461895736931;
529         double P2 = 0.81511929394895;
530         double P3 = -0.07827242925479;
531         double P4 = 0.00253003887432;
532
533         // linear fit
534     //    double P0 = 2.51342550695548;
535     //    double P1 = -0.56417527884695;
536     //    double P2 = 0;
537     //    double P3 = 0;
538     //    double P4 = 0;
539
540         // exp( P0 + P1*ln(Re) + P2*(ln(Re))^2 + P3*(ln(Re))^3 + P4*(ln(Re))^4 )
541         double cf = exp( P0 + P1*log(Re) + P2*gsl_pow_2(log(Re)) +
542                          P3*gsl_pow_3(log(Re)) + P4*gsl_pow_4(log(Re)) );
543         double cp = 1 - cf;
544         return cp;
545     }
546
547     /*============================================================================*/
548     /* The following functions are used for bracket root solving for the desired
549      * volume.
550      */
551
552     /**
553      * Returns the difference between sphere volume, Vs, and raindrop volume, V, to
554      * allow the bracketing root finder to converge to 0.
555      */
556     double bracket_ShapeVolume(double Rt, void * params) {
557         // Extract parameters
558         SHAPE_VOL_PARAMS par = *(SHAPE_VOL_PARAMS *)params;
559         double U = par.U;
560         double c = par.c;
561         PRES_DIST pdist = *par.pdist;
562         PHYS_PROPS phys = par.phys;
563
564         SHAPE sh = shape(U, Rt, c, pdist, phys);
565
566         double diff = sh.V - par.Vs;
567         cleanup(sh);
568
569         return diff;
570     }
571
```

```
572 /*=========================================================================*/
573 /* The following functions are used for bracket root solving for the desired
574  * volume.
575  */
576
577 /**
578  * Returns the difference between drag, D, and weight, W, to allow the
579  * bracketing root finder to converge to 0.
580  */
581 double bracket_ShapeWeight(double Lam, void * params) {
582     // Extract parameters
583     SHAPE_VOL_PARAMS par = *(SHAPE_VOL_PARAMS *)params;
584     double U = par.U;
585     double Rt = par.Rt;
586     double c = par.c;
587     PRES_DIST * pdist = par.pdist;
588     PHYS_PROPS phys = par.phys;
589
590     double Vs = par.Vs;
591
592     // set variable
593     pdist->Lam = Lam;
594
595     SHAPE sh = shape(U, Rt, c, *pdist, phys);
596
597 //    double W = sh.V * phys.rho_w * phys.g; // weight = V*rho*g
598     double W = Vs * phys.rho_w * phys.g; // weight = V*rho*g
599     double D = Lam * par.D;
600
601     cleanup(sh);
602
603     double diff = D - W;
604     return diff;
605 }
606
607 /*=========================================================================*/
```

B. project.h

```
/*=========================================================================*
 * project.h *
 * Header file containing global constants for the program.
 *=========================================================================*/

#ifndef FILE_project_SEEN
#define FILE_project_SEEN

/*=========================================================================*/
/* Global constants */

#define PRINT_TABLE false

/*-------------------------------------------------------------------------*/
/* Physical constants */

/* source: http://www.hbcpnetbase.com
 * primary source: CRC Handbook of Physics and Chemistry, 86th ed
 */
/*
 * surface tension (20degC), sigma = 72.75 dynes/cm = 72.75e-3 J/m^2
 *                 (30degC), sigma = 71.20 dynes/cm = 71.20e-3 J/m^2
 */
#define WATER_AIR_SURFACE_TENSION 72.75e-3
/*
 * water density (20degC), rho_w = 0.99821 gm/cm^3 = 997.07 kg/m^3
 *               (30degC), rho_w = 0.99565 gm/cm^3 = 995.65 kg/m^3
 */
#define WATER_DENSITY 997.07
/*
 * air density (300K, 1bar), rho_a = 1.161 kg/m^3
 */
#define AIR_DENSITY 1.161
/*
 * air viscosity (300K, 1bar), vis_a = 18.6e-6 Pa-s = 18.6e-6 N-s/m^2
 */
#define AIR_VISCOSITY 18.6e-6
/*
 * standard gravity, g = 9.80665 m/s^2
 */
#define GRAVITY_SEA_LEVEL 9.80665

/*-------------------------------------------------------------------------*/
/* Error constraints */

#define RES_DEG 1 // degree resolution to print points

#define BRACKET_ERR 1e-6 // error bound for bracket root finding
#define NEWTON_ERR 1e-5 // error bound for Newton root finding
                        // used to converge to critical angles
#define INTEGRATION_ERR 1e-5 // error bound for integration (quadrature)

// Error bound in angles (all fractional)
#define ERR_PHI 1e-4 // don't go less than 1e-5
```

```
55  #define ERR_PHI_FACTOR 10
56
57  /*============================================================================*/
58  /* Structs */
59
60  struct phys_properties {
61      double st;
62      double rho_w;
63      double rho_a;
64      double vis_a;
65      double g;
66  };
67  typedef struct phys_properties PHYS_PROPS;
68
69  #endif /* FILE_project_SEEN */
70
71  /*============================================================================*/
```

C. shape.h

```c
/*=========================================================================*
 * shape.h *
 * Header for module that calculates the coordinates of the raindrop shape.
 *=========================================================================*/

#ifndef FILE_shape_SEEN
#define FILE_shape_SEEN

#include "project.h"
#include "cpres.h"

/*-------------------------------------------------------------------------*/
/* Structs */

struct shape {
    long len;
    int iter; // total iterations to calculate

    /* Arrays */
    double * phi; // tangent angle
    double * s; // arc length
    double * DphiDs; // curvature
    double * x; // x
    double * z; // z
    double * Vi; // incremental V

    /* Final values */
    double V; // total volume
    double A; // cross-sectional area
    double SA; // surface area
    double ax; // axis ratio

    /* Indicates if shape is closed */
    bool closed;
};
typedef struct shape SHAPE;

struct raindrop_params {
    double C;
    double WeA;
};
typedef struct raindrop_params RAINDROP;

/*-------------------------------------------------------------------------*/
/* Function prototypes */

SHAPE shape(double U, double Rt, double c, PRES_DIST pdist, PHYS_PROPS phys);
void cleanup(SHAPE sh);

#endif /* FILE_shape_SEEN */

/*=========================================================================*/
```

D. shape.c

```c
/*============================================================================*
 * shape.c *
 * Module that calculates the coordinates of the raindrop shape.
 *============================================================================*/

#include <stdbool.h>
#include <gsl/gsl_errno.h>
#include <gsl/gsl_math.h>
#include <gsl/gsl_roots.h>
#include "project.h"
#include "nummethods.h"
#include "rk4applied.h"

/*============================================================================*/
/* Structs */

struct dimless_shape_pos {
    /* All parameters dimensionless form */
    double phi; // tangent angle
    double S; // arc length
    double DphiDS; // curvature
    double X; // x
    double Z; // z
    double V; // volume
    double SA; // surface area
};
typedef struct dimless_shape_pos DIMLESS_POS;

struct shape_deltas {
    /* All parameters dimensionless form */
    double Dphi;
    double DS;
    double DX;
    double DZ;
    double DV;
    double DSA;
};
typedef struct shape_deltas DELTAS;

struct nr_Deltas_params {
    double C;
    double WeA;
    double cpres0;
    PRES_DIST pdist;

    // pointers
    DIMLESS_POS * pos;
    DELTAS * d;

    // array
    double * phic;

    int c; // counter for critical angle
};
```

```
55  typedef struct nr_Deltas_params DELTAS_PARAMS;
56
57  /*--------------------------------------------------------------------------*/
58  /* Function prototypes */
59
60  void cleanupParams(DELTAS_PARAMS * par_ptr);
61
62  void DeltasMinErr(double DS, DELTAS_PARAMS * par);
63
64  double nr_Deltas(double DS, void * params);
65  double nr_Deltas_deriv(double DS, void * params);
66  void nr_Deltas_fdf(double DS, void * params, double * nrD, double * dnrD);
67
68  void Deltas(double DS, DELTAS_PARAMS * par);
69
70  /*==========================================================================*/
71  /* Function definitions */
72
73  /**
74   * Returns the shape coordinates of the raindrop given the input parameters.
75   */
76  SHAPE shape(double U, double Rt, double c, PRES_DIST pdist, PHYS_PROPS phys) {
77      // declare
78      SHAPE sh;
79
80      //DIMLESS_POS pos;
81      DELTAS_PARAMS par;// = {C,WeA,cpres0,pdist,pos,&d,phic[0]};
82
83      /*--------------------------------------------------------------------------*/
84      /* Initialize SHAPE */
85      sh.iter = 0;
86      sh.len = 1;
87      sh.phi = (double *)malloc(sizeof(double));
88      sh.s = (double *)malloc(sizeof(double));
89      sh.DphiDs = (double *)malloc(sizeof(double));
90      sh.x = (double *)malloc(sizeof(double));
91      sh.z = (double *)malloc(sizeof(double));
92      sh.Vi = (double *)malloc(sizeof(double));
93
94      /*--------------------------------------------------------------------------*/
95      /* Initialize parameters */
96
97      double b = sqrt(c * (phys.rho_w - phys.rho_a)*phys.g / phys.st);
98      par.C = b*Rt;
99
100     par.WeA = gsl_pow_2(phys.rho_a*U)/(2.0*phys.st*b); // We/A
101     par.cpres0 = cpres(pdist, M_PI);
102
103     par.pdist = pdist;
104
105     DIMLESS_POS pos;
106     par.pos = &pos; // needs to point to something
107
108     DELTAS d = {0.0,0.0,0.0,0.0,0.0,0.0};
109     par.d = &d;
110
111     // critical angles (targets)
112     double resdeg = RES_DEG; // degree resolution of critical angles
```

```
113         double Dphic = resdeg *(M_PI/180.0);
114         int NUM_ANG = 180/resdeg + 1; // number of angles
115         par.phic = (double *)createFilledArray(NUM_ANG, 0.0, M_PI);
116 
117         /*-------------------------------------------------------------------------*/
118 
119         // assign first values
120         sh.iter = 1;
121         sh.phi[0] = pos.phi = 0;
122         sh.s[0] = pos.S = 0;
123         sh.DphiDs[0] = 1/Rt; pos.DphiDS = 1/par.C;
124         sh.x[0] = pos.X = 0;
125         sh.z[0] = pos.Z = 0;
126         pos.SA = 0;
127         pos.V = 0;
128 
129         // some default value
130         double DS = 1;
131         double e = ERR_PHI;
132 
133         /*-------------------------------------------------------------------------*/
134 
135         par.c = 1; // skip trivial zeroth point
136         bool crit = false;
137 
138         // print zeroth point
139         if (PRINT_TABLE) {
140             printf("%.5e\t%.5e\t%.5e\t%.5e\t%.5e\n",
141                    sh.phi[0], sh.s[0], sh.x[0], sh.z[0], sh.Vi[0]);
142         }
143 
144         /*-------------------------------------------------------------------------*/
145 
146         // to store maximu, X value to help determine axis ratio
147         double Xmax = 0;
148 
149         /*-------------------------------------------------------------------------*/
150 
151         while (pos.phi <= M_PI && d.Dphi >= 0) {
152             if (sh.iter > 1) {
153                 pos.DphiDS = d.Dphi/d.DS; // curvature
154             }
155 
156             // get Deltas within error bounds
157             DeltasMinErr(DS, &par);
158 
159             /*-------------------------------------------------------------------------*/
160             /* If near critical values, then converge towards it */
161 
162             double phi = pos.phi + d.Dphi;
163             double dphic = phi - par.phic[par.c];
164             if (phi > par.phic[par.c]) { // catch before exceeding
165                 /* Passed critical value, so backtrack */
166                 // Determine DS from Dphi search
167                 DS = (par.phic[par.c] - pos.phi) / pos.DphiDS;
168 
169                 // Newton-Raphson convergence
170                 newton_root(DS, &nr_Deltas, &nr_Deltas_deriv, &nr_Deltas_fdf, &par);
```

```
171
172             crit = true;
173         }
174
175         // grow DS by 10%
176         DS = d.DS * 1.1;
177
178         /*-------------------------------------------------------------------------*/
179         // assign values to the current point
180         pos.phi += d.Dphi;
181         pos.S   += d.DS;
182         pos.X   += d.DX;
183         pos.Z   += d.DZ;
184         pos.SA   += d.DSA;
185         pos.V   += d.DV;
186
187         /*-------------------------------------------------------------------------*/
188         /* Obtain largest X value to help find axis ratio */
189
190         if (pos.X > Xmax) {
191             Xmax = pos.X;
192         }
193
194         /*-------------------------------------------------------------------------*/
195
196         // print only critical values
197         if (crit) {
198             // increase size of array
199             sh.len++;
200             sh.phi = (double *)realloc(sh.phi, sh.len*sizeof(double));
201             sh.s = (double *)realloc(sh.s, sh.len*sizeof(double));
202             sh.DphiDs = (double *)realloc(sh.DphiDs, sh.len*sizeof(double));
203             sh.x = (double *)realloc(sh.x, sh.len*sizeof(double));
204             sh.z = (double *)realloc(sh.z, sh.len*sizeof(double));
205             sh.Vi = (double *)realloc(sh.Vi, sh.len*sizeof(double));
206
207             // copy values
208             int end = sh.len - 1;
209             sh.phi[end] = pos.phi;
210             sh.s[end] = pos.S / b;
211             sh.DphiDs[end] = pos.DphiDS * b;
212             sh.x[end] = pos.X / b;
213             sh.z[end] = pos.Z / b;
214             sh.Vi[end] = pos.V / gsl_pow_3(b);
215
216             if (PRINT_TABLE) {
217                printf("%.5e\t%.5e\t%.5e\t%.5e\t%.5e\n",
218                       sh.phi[end], sh.s[end], sh.x[end], sh.z[end], sh.Vi[end]);
219             }
220
221             // set volume and surface area
222             // put this here, since pos gets corrupted at end of loop
223             sh.SA = pos.SA / gsl_pow_2(b);
224             sh.V = pos.V / gsl_pow_3(b);
225
226             // not sure why this is needed, but it is
227             if (par.c == NUM_ANG-1) { // last point
228                 break;
```

```
229                 }
230
231                 crit = false; // reset for non-critical angles
232                 par.c++; // ready for next critical value
233             }
234
235             sh.iter++;
236         }
237
238         /*------------------------------------------------------------------------*/
239         /* Note if (d.Dphi <) 0 => open bottom */
240
241         if (d.Dphi < 0) {
242             sh.closed = false;
243         }
244         else {
245             sh.closed = true;
246         }
247
248         /*------------------------------------------------------------------------*/
249         /* Calculate final values */
250
251         double xmax = Xmax / b; // scale to dimensioned form
252
253         // cross-sectional area
254         double A = M_PI*gsl_pow_2(xmax);
255         sh.A = A;
256
257         // axis ratio
258         double zmax = sh.z[sh.len-1]; // last z
259         sh.ax = (zmax/2.0) / xmax;
260
261         /*------------------------------------------------------------------------*/
262         /* Clean up */
263
264         cleanupParams(&par);
265
266         return sh;
267     }
268
269     /**
270      * Invoke this to deallocate all pointers in SHAPE
271      */
272     void cleanup(SHAPE sh) {
273          free(sh.phi);
274          free(sh.s);
275          free(sh.DphiDs);
276          free(sh.x);
277          free(sh.z);
278     }
279
280     /**
281      * Invoked internally to deallocate all pointers in DELTAS_PARAMS
282      */
283     void cleanupParams(DELTAS_PARAMS * par_ptr) {
284         free(par_ptr->phic);
285     }
286
```

```
287
288
289 /**
290  * Ensures that error is small enough
291  */
292 void DeltasMinErr(double DS, DELTAS_PARAMS * par_ptr) {
293     DELTAS * d = par_ptr->d;
294
295     double Dphi_1, Dphi_2;
296     double a = ERR_PHI_FACTOR;;
297     double e = ERR_PHI;
298     double pdiff;
299
300     int count = 0;
301
302     do {
303         // double-step
304         Deltas(DS/2, par_ptr);
305         Dphi_2 = d->Dphi;
306         Deltas(DS/2, par_ptr);
307         Dphi_2 += d->Dphi;
308
309         // single-step
310         Deltas(DS, par_ptr);
311         Dphi_1 = d->Dphi;
312
313         pdiff = fabs(Dphi_2-Dphi_1)/fabs(Dphi_2);
314
315         // For next iteration, error too big, so shrink by half
316         DS /= 2;
317
318         count++;
319     } while (pdiff >= a*e
320             && Dphi_1 >= e*1e-16); // prevent shrinking too small
321 }
322
323 /**
324  * Calculates the results of taking the next step
325  */
326 void Deltas(double DS, DELTAS_PARAMS * par_ptr) {
327     DELTAS_PARAMS par = *par_ptr;
328
329     // Extract parameters
330     double C = par.C;
331     double WeA = par.WeA;
332     double cpres0 = par.cpres0;
333     PRES_DIST pdist = par.pdist;
334     DIMLESS_POS pos = *par.pos;
335     double phic = par.phic[par.c];
336     DELTAS * d = par_ptr->d;
337
338     double phi = pos.phi;
339     double X = pos.X;
340     double Z = pos.Z;
341     double V = pos.V;
342     double SA = pos.SA;
343
344     /*-------------------------------------------------------------------------*/
```

```
345
346         double Dphi1;
347         if (phi == 0) {Dphi1 = (2.0/C + Z - 1.0/C) * DS;}
348         else {Dphi1 = (2.0/C + Z - sin(phi)/X -
349               WeA*(cpres(pdist, M_PI-phi)-cpres0)) * DS;}
350         double DX1 = cos(phi) * DS;
351         double DZ1 = sin(phi) * DS;
352         double DV1 = M_PI*gsl_pow_2(X)*sin(phi) * DS;
353         double DSA1 = 2*M_PI*X * DS;
354
355         double Dphi2;
356         if (phi == 0) {Dphi2 = (2.0/C + (Z+DZ1/2) - 1.0/C) * DS;}
357         else {Dphi2 = (2.0/C + (Z+DZ1/2) - sin(phi+Dphi1/2)/X -
358               WeA*(cpres(pdist, M_PI-phi-Dphi1/2)-cpres0)) * DS;}
359         double DX2 = cos(phi+Dphi1/2) * DS;
360         double DZ2 = sin(phi+Dphi1/2) * DS;
361         double DV2 = M_PI*gsl_pow_2(X+DX1/2)*sin(phi+Dphi1/2) * DS;
362         double DSA2 = 2.0*M_PI*(X+DX1/2) * DS;
363
364         double Dphi3;
365         if (phi == 0) {Dphi3 = (2.0/C + (Z+DZ2/2) - 1.0/C) * DS;}
366         else {Dphi3 = (2.0/C + (Z+DZ2/2) - sin(phi+Dphi2/2)/X -
367               WeA*(cpres(pdist, M_PI-phi-Dphi2/2)-cpres0)) * DS;}
368         double DX3 = cos(phi+Dphi2/2) * DS;
369         double DZ3 = sin(phi+Dphi2/2) * DS;
370         double DV3 = M_PI*gsl_pow_2(X+DX2/2)*sin(phi+Dphi2/2) * DS;
371         double DSA3 = 2.0*M_PI*(X+DX2/2) * DS;
372
373         double Dphi4;
374         if (phi == 0) {Dphi4 = (2.0/C + (Z+DZ3) - 1.0/C) * DS;}
375         else {Dphi4 = (2.0/C + (Z+DZ3) - sin(phi+Dphi3)/X -
376               WeA*(cpres(pdist, M_PI-phi-Dphi3)-cpres0)) * DS;}
377         double DX4 = cos(phi+Dphi3) * DS;
378         double DZ4 = sin(phi+Dphi3) * DS;
379         double DV4 = M_PI*gsl_pow_2(X+DX3)*sin(phi+Dphi3) * DS;
380         double DSA4 = 2.0*M_PI*(X+DX3) * DS;
381
382         /*-------------------------------------------------------------------------*/
383
384         d->DS = DS;
385         d->Dphi = (Dphi1 + 2*Dphi2 + 2*Dphi3 + Dphi4) / 6;
386         d->DX = (DX1 + 2*DX2 + 2*DX3 + DX4) / 6;
387         d->DZ = (DZ1 + 2*DZ2 + 2*DZ3 + DZ4) / 6;
388         d->DV = (DV1 + 2*DV2 + 2*DV3 + DV4) / 6;
389         d->DSA = (DSA1 + 2*DSA2 + 2*DSA3 + DSA4) / 6;
390     }
391
392     /*=========================================================================*/
393     /* The following functions are used for Newton-Raphson convergence to the
394      * critical phi angles
395      */
396
397     /**
398      * Returns the difference between the returned angle and the current critical
399      * phi angle to allow the Newton-Raphson root finder to converge it to 0.
400      * f function needed by the Newton-Raphson root finder.
401      */
402     double nr_Deltas(double DS, void * params) {
```

```
403         // Extract parameters
404         DELTAS_PARAMS * par_ptr = (DELTAS_PARAMS *)params;
405         DELTAS_PARAMS par = *par_ptr;
406 
407         // rename
408         DIMLESS_POS pos = *par.pos;
409         double phic = par.phic[par.c];
410         DELTAS * d = par_ptr->d;
411 
412         Deltas(DS, &par);
413 
414         double phi = pos.phi + (*d).Dphi;
415         double diff = phi - phic;
416         return diff;
417     }
418 
419     /**
420      * Returns the derivative at the phi angle to allow the Newton-Raphson root
421      * finder to converge the angle to the current critical phi angle.
422      * df function needed by the Newton-Raphson root finder.
423      */
424     double nr_Deltas_deriv(double DS, void * params) {
425         // Extract parameters
426         DELTAS_PARAMS * par_ptr = (DELTAS_PARAMS *)params;
427         DELTAS_PARAMS par = *par_ptr;
428         DELTAS * d = par_ptr->d;
429 
430         Deltas(DS, &par);
431 
432         double DphiDS = (*d).Dphi / (*d).DS;
433         return DphiDS;
434     }
435 
436     /**
437      * fdf function needed by the Newton-Raphson root finder.
438      */
439     void nr_Deltas_fdf(double DS, void * params, double * nrD, double * dnrD) {
440         // Extract parameters
441         DELTAS_PARAMS * par_ptr = (DELTAS_PARAMS *)params;
442         DELTAS_PARAMS par = *par_ptr;
443 
444         // rename
445         DIMLESS_POS pos = *par.pos;
446         double phic = par.phic[par.c];
447         DELTAS * d = par_ptr->d;
448 
449         Deltas(DS, par_ptr);
450 
451         // function value
452         double phi = pos.phi + (*d).Dphi;
453         double diff = phi - phic;
454         *nrD = diff;
455 
456         // derivative
457         double DphiDS = (*d).Dphi / (*d).DS;
458         *dnrD = DphiDS;
459     }
460 
```

```
461     /*============================================================================*/
```

E. array.h

```c
/*============================================================================*
 * array.h *
 * Header file of module to provide utility functions for arrays.
 *============================================================================*/

#ifndef FILE_array_SEEN
#define FILE_array_SEEN

/*----------------------------------------------------------------------------*/
/* Structs */

struct datafile {
    char* filePath;
    double* x;
    double* y;
    int len;
};
typedef struct datafile DAT;

/*----------------------------------------------------------------------------*/
/* Function prototypes */

void scaleArray(double*,double,int);
void scaleArrayCopy(double* orig, double* copy, double scale, int len);

// need to free()
void fillArray(double * array, int len, double a, double b);
double * createFilledArray(int len, double a, double b);

void copyArray(double* orig, double* copy, int len);
double * cloneArray(double * orig, int len);

void loadArrays(DAT * dat);

double linear(double x0, double y0, double x1, double y1, double x);


#endif /* FILE_array_SEEN */

/*============================================================================*/
```

F. array.c

```c
/*============================================================================*
 * array.c *
 * Module to provide utility functions for arrays.
 *============================================================================*/

#include <stdio.h>
#include <stdlib.h>
#include <gsl/gsl_spline.h>
#include "array.h"

/*----------------------------------------------------------------------------*/

/**
 * Scales the array.
 */
void scaleArray(double* array, double scale, int len) {
    // note that len needs to be explicit here, cannot be calculated
    int i;
    for (i = 0; i < len; i++) {
        array[i] *= scale;
    }
}

/**
 * Creates a copy of orig and scales it.
 */
void scaleArrayCopy(double* orig, double* copy, double scale, int len) {
    copyArray(orig, copy, len);
    scaleArray(copy, scale, len);
}

/**
 * Creates a clone of orig and scales it.
 */
double * scaleArrayClone(double* orig, double scale, int len) {
    double * copy = cloneArray(orig, len);
//    scaleArray(copy, scale, len);
}

/*----------------------------------------------------------------------------*/

/**
 * Fills an array with values between a and b, inclusive, for
 * length len.
 */
void fillArray(double * array, int len, double a, double b) {
    double step = (b-a)/(double)(len-1); // include end points
    int i;
    for (i = 0; i < len; i++, a+=step) {
        array[i] = a;
    }
}

/**
```

```
55   * Creates an array and fills it with values between a and b, inclusive, for
56   * length len.
57   */
58  double * createFilledArray(int len, double a, double b) {
59      double * array = (double *)malloc(len*sizeof(double));
60      double step = (b-a)/(double)(len-1); // include end points
61      int i;
62      for (i = 0; i < len; i++, a+=step) {
63          array[i] = a;
64      }
65      return array;
66  }
67
68  /*---------------------------------------------------------------------------*/
69
70  /**
71   * Makes a copy of the orig array into copy.
72   */
73  void copyArray(double* orig, double* copy, int len) {
74      // note that len needs to be explicit here, cannot be calculated
75      int i;
76      for (i = 0; i < len; i++) {
77          copy[i] = orig[i];
78      }
79  }
80
81  /**
82   * Clone an array.
83   */
84  double * cloneArray(double * orig, int len) {
85      double * copy = (double *)malloc(len*sizeof(double));
86      printf("len = %d\n", len);
87      int i;
88      for (i = 0; i < len; i++) {
89          copy[i] = orig[i];
90      }
91      return copy;
92  }
93
94  /*---------------------------------------------------------------------------*/
95
96  /**
97   * Loads into DAT two columns of data from a file.
98   */
99  void loadArrays(DAT * dat) {
100     FILE* fp;
101     fp = fopen(dat->filePath, "r");
102
103     double xi, yi;
104     int i = 0;
105
106     double * x = (double *)malloc(i*sizeof(double));
107     double * y = (double *)malloc(i*sizeof(double));
108
109     while (!feof(fp)) {
110         fscanf(fp, "%lf\t%lf", &xi, &yi); // %lf for long float == double
111
112         x = (double*)realloc(x, (i+1)*sizeof(double));
```

```
113             y = (double*)realloc(y, (i+1)*sizeof(double));
114             x[i] = (double)xi;
115             y[i] = (double)yi;
116 
117             i++;
118         }
119 
120         fclose(fp);
121 
122         dat->x = x;
123         dat->y = y;
124 
125         dat->len = i;
126     }
127 
128     /*------------------------------------------------------------------------*/
129 
130     /**
131      * Linear interpolation. NOT USED?
132      */
133     double linear(double x0, double y0, double x1, double y1, double x) {
134         // s = (y-y0)/(y1-y0) = (x-x0)/(x1-x0)
135         double s = (x-x0)/(x1-x0);
136         double y = y0 + s*(y1-y0);
137         return y;
138     }
139 
140     /*========================================================================*/
```

G. nummethods.h

```
/*=========================================================================*
 * nummethods.h *
 * Header file of module to provide wrapper functions to numerical methods.
 *=========================================================================*/

#ifndef FILE_raindrop_SEEN
#define FILE_raindrop_SEEN

/*-------------------------------------------------------------------------*/

int newton_root(double x0, void * f, void * df, void * fdf, void * params);
double bracket_root(double x_lo, double x_hi, void * f, void * params);

double spline(double* x, double* y, int len, double xi);

double integrate(void * df, void * params, double a, double b);

#endif

/*=========================================================================*/
```

H. nummethods.c

```c
/*============================================================================*
 * nummethods.c *
 * Module to provide wrapper functions to numerical methods.
 *============================================================================*/

#include <gsl/gsl_errno.h>
#include <gsl/gsl_math.h>
#include <gsl/gsl_roots.h>
#include <gsl/gsl_integration.h>
#include <gsl/gsl_spline.h>
#include "project.h"
#include "nummethods.h"

/*----------------------------------------------------------------------------*/
/* Internally used structs */

struct da_params {
    gsl_interp_accel * acc;
    gsl_spline * spline;
    double L;
    double n;
};
typedef struct da_params DA_PARAMS;

double da(double,void *);
double da1(double,void *);

double sum_a = 0;

/*----------------------------------------------------------------------------*/

/**
 * Returns the root, via the Newton-Raphson method.
 */
int newton_root(double x0, void * f, void * df, void * fdf, void * params) {
    // note that x0 cannot be 0

    int status;
    int iter = 0, max_iter = 100;

    gsl_function_fdf FDF;

    FDF.f = f;
    FDF.df = df;
    FDF.fdf = fdf;
    FDF.params = params;

    const gsl_root_fdfsolver_type *T = gsl_root_fdfsolver_newton;
    gsl_root_fdfsolver *s = gsl_root_fdfsolver_alloc (T);

    gsl_root_fdfsolver_set(s, &FDF, x0);
//    printf ("using %s method\n", gsl_root_fdfsolver_name (s));
//    printf ("%-5s %12s %12s\n", "iter", "root", "err(est)");

```

```
55          double x = x0;
56          do {
57              iter++;
58              status = gsl_root_fdfsolver_iterate(s);
59
60              x0 = x;
61              x = gsl_root_fdfsolver_root(s);
62
63              status = gsl_root_test_delta(x, x0, 0, NEWTON_ERR);
64      //        if (status == GSL_SUCCESS)
65      //            printf ("Converged:\n");
66      //        printf ("%5d %12.12f %12.15f\n", iter, x, x - x0);
67          }
68          while (status == GSL_CONTINUE && iter < max_iter);
69
70          return status;
71      }
72
73      /*--------------------------------------------------------------------------*/
74
75      /**
76       * Returns the root, via bracketing.
77       */
78      double bracket_root(double x_lo, double x_hi, void * f, void * params) {
79          int status;
80          int iter = 0, max_iter = 100;
81
82          double r = 0;
83
84          gsl_function F;
85          F.function = f;
86          F.params = params;
87
88          /* Cannot use gsl_root_fsolver_brent because function not strictly monotonic
89           * at small scale.
90           */
91          const gsl_root_fsolver_type *T = gsl_root_fsolver_falsepos;
92          gsl_root_fsolver *s = gsl_root_fsolver_alloc (T);
93
94          gsl_root_fsolver_set (s, &F, x_lo, x_hi);
95          printf ("Root finding: using %s method\n", gsl_root_fsolver_name (s));
96          printf ("%5s [%12s, %12s] %12s %12s\n",
97                  "iter", "lower", "upper", "root", "err(est)");
98
99          do {
100             iter++;
101             status = gsl_root_fsolver_iterate (s);
102             r = gsl_root_fsolver_root (s);
103
104             x_lo = gsl_root_fsolver_x_lower (s);
105             x_hi = gsl_root_fsolver_x_upper (s);
106
107             status = gsl_root_test_interval (x_lo, x_hi, 0, BRACKET_ERR);
108
109             if (status == GSL_SUCCESS)
110                 printf ("Converged:\n");
111             printf ("%5d [%.5e, %.5e] %.5e %.5e\n",
112                     iter, x_lo, x_hi, r, x_hi - x_lo);
```

```
113         }
114         while (status == GSL_CONTINUE && iter < max_iter);
115 
116         printf("\n");
117         printf("Root = %.16e\n", r);
118         printf("\n");
119 
120         return r;
121     }
122 
123     /*-------------------------------------------------------------------------*/
124 
125     /**
126      * Integration
127      */
128     double integrate(void * df, void * params, double a, double b) {
129         // prepare integration
130         gsl_integration_workspace* w = gsl_integration_workspace_alloc(1000);
131         double result, error;
132 
133         gsl_function Fn;
134         Fn.function = df;
135         Fn.params = params;
136 
137         gsl_integration_qags(&Fn, a, b, 0, INTEGRATION_ERR, 1000, w,
138                              &result, &error);
139 
140         // deallocate
141         gsl_integration_workspace_free(w);
142 
143         return result;
144     }
145 
146     /*-------------------------------------------------------------------------*/
147 
148     /**
149      * Returns a value on the spline curve from the input data points.
150      */
151     double spline(double * x, double * y, int len, double xi) {
152         // allocate
153         gsl_interp_accel * acc = gsl_interp_accel_alloc();
154         gsl_spline * spline = gsl_spline_alloc(gsl_interp_cspline, len); // cubic spline
155         // initialize spline
156         gsl_spline_init(spline, x, y, len);
157 
158         // evaluate spline
159         double yi = gsl_spline_eval(spline, xi, acc);
160 
161         // spline cleanup
162         gsl_spline_free(spline);
163         gsl_interp_accel_free(acc);
164 
165         return yi;
166     }
167 
168     /*=========================================================================*/
```

I. cpres.h

```c
/*=========================================================================*
 * cpres.h *
 * Header file of module to calculate the corrected dimensionless pressure
 * distribution, K.
 *=========================================================================*/

#ifndef FILE_cpres_SEEN
#define FILE_cpres_SEEN

/*-------------------------------------------------------------------------*/
/* Structs */

struct pressure_dist {
    double* t; // theta
    double* k; // kappa
    int len; // length of arrays

    /* Correction parameters */
    double Ga; // if > 1, then apply distortion corrections (see equation/graph)
    double Gd;

    /* Amplitude factor */
    double Lam;
};
typedef struct pressure_dist PRES_DIST;

/*-------------------------------------------------------------------------*/
/* Function prototypes */

double cpres(PRES_DIST pd, double tt);

#endif

/*=========================================================================*/
```

J. cpres.c

```c
/*============================================================================*
 * cpres.c *
 * Module to calculate the corrected dimensionless pressure distribution, K.
 *============================================================================*/

#include <stdbool.h>
#include <gsl/gsl_math.h>
#include <gsl/gsl_spline.h>
#include "nummethods.h"
#include "cpres.h"

/*----------------------------------------------------------------------------*/
/* Function prototypes */

double pres(PRES_DIST,double);

/*----------------------------------------------------------------------------*/
/* Function definitions */

/**
 * Corrected pressure distribution.
 */
double cpres(PRES_DIST pd, double ti) {
    double ki;

    if (pd.Ga > 1) {
        double t1 = 72 *M_PI/180; // 72^o
        double t2 = 88 *M_PI/180; // 88^o

        double Ga;
        if (ti < t1) { // 0 <= ti < t1
            Ga = pd.Ga;
        }
        else if (ti < t2) { // t1 <= ti < t2
            // linear transition
            Ga = (ti-t1)/(t2-t1)*(pd.Gd-pd.Ga) + pd.Ga;
        }
        else { // t2 <= ti
            Ga = pd.Gd;
        }

        ki = 1 - Ga*(1-pres(pd,ti));
    }
    else { // should be Ga == 1
        ki = pres(pd,ti);
    }

    ki *= pd.Lam; // scale with amplitude
    return ki;
}

/**
 * Original pressure distribution.
 */
```

```
55  double pres(PRES_DIST pd, double ti) {
56      double ki = spline(pd.t,pd.k,pd.len, ti);
57      return ki;
58  }
59
60  /*============================================================================*/
```

III. MATLAB PROGRAM CODE

Matlab was used for data analysis and graphing of the results.

A. raindrop.m

```
%=========================================================================%
% raindrop.m %
% To render graphs based on coordinates input of the raindrop shapes.
%=========================================================================%

d = 4.3755; % specify diameter of equivalent volume sphere, in mm

%-------------------------------------------------------------------------%

% load raindrop shape
shape = load(['../results/run_' num2str(d) '/table.txt']);
phi = shape(:,1);
s = shape(:,2);
x = shape(:,3);
z = shape(:,4);
Vi = shape(:,5);

%-------------------------------------------------------------------------%

d = d * 1e-3;% scale to m
q = d/2; % radius

%-------------------------------------------------------------------------%
% Center graph

% flip z-axis
z = -z;

% get middle height
i = find(Vi < Vi(end)/2); % median volume increment
midz = z(i(end));
% translate z-axis
z = z - midz;

%-------------------------------------------------------------------------%
% Get polar coordinates

r = sqrt(x.^2 + z.^2);
t = atan(x./-z); % azimuthal angle, from the bottom pole

% since atan(neg)<0, add pi to make in range of pi/2 and pi
index = find(t<0);
t(index) = t(index) + pi;

% cap off with a point at the bottom
r(end+1) = abs(z(end));
t(end+1) = pi;

%-------------------------------------------------------------------------%
```

```
% Fourier cosine series

dr = r - q;
c = fouriercosine(t,dr/q,pi,10);

%=========================================================================%
% Plot graphs

% reflect x and add
x = [x fliplr(-x)]; % do not closing with horizontal line,
                    % show hole if present
z = [z fliplr(z)];

% scale to mm
scale = 1./1e-3;

%-------------------------------------------------------------------------%

% x-z graph
figure;
hold on; axis equal;
plot([0 0].*scale, [-q q].*scale, 'k:');
plot([-q q].*scale, [0 0].*scale, 'k:');
plot([-q/sqrt(2) q/sqrt(2)].*scale, [-q/sqrt(2) q/sqrt(2)].*scale, 'k:');
plot([-q/sqrt(2) q/sqrt(2)].*scale, [q/sqrt(2) -q/sqrt(2)].*scale, 'k:');
% circle of sphere of equivalent volume
Xs = []; Ys = [];
for theta = -pi/2:.01:3*pi/2
    Xs = [Xs q*cos(theta)];
    Ys = [Ys q*sin(theta)];
end
plot(Xs.*scale,Ys.*scale, 'k:');

plot(x.*scale,z.*scale, 'b-');
xlabel('x[mm]');
ylabel('z[mm]');

%-------------------------------------------------------------------------%

% x-y-z 3D graph
figure; hold on; axis equal;
[x,y,z] = revolve3D(x,z, pi/72);
camlight left;
lighting gouraud; material shiny;
surf(x,y,z, 'FaceAlpha', .4, 'FaceColor', [.7 .88 1], ...
            'EdgeColor','none', 'FaceLighting','gouraud');
% sphere of equivalent volume
[xs,ys,zs] = sphere;
xs = xs .*q;
ys = ys .*q;
zs = zs .*q;
mesh(xs,ys,zs, 'EdgeAlpha', .4, 'FaceColor', 'none', ...
               'EdgeColor',[0 0 0], 'LineStyle',':');
xlabel('x[mm]');
ylabel('z[mm]');
zlabel('y[mm]');

%-------------------------------------------------------------------------%
```

```
108
109 % r-t graph
110 figure;
111 hold on;
112 plot([0 180],[q q].*scale, 'k:'); % graph for circle
113 plot(t*180/pi,r.*scale);
114 xlabel('\theta[^o]');
115 ylabel('r[mm]');
116
117 %-------------------------------------------------------------------------%
118
119 % c_n bar chart
120 figure; hold on; grid on;
121 bar(0:length(c)-1, c);
122 xlabel('n');
123 ylabel('c_n');
124
125 %-------------------------------------------------------------------------%
126
127 % phi-s graph
128 figure;
129 hold on;
130 plot([0 180], [0 pi*q].*scale, 'k:'); % graph for circle
131 plot(phi.*180/pi, s.*scale);
132 xlabel('\phi[^o]');
133 ylabel('s[mm]');
134
135 %=========================================================================%
```

B. fouriercosine.m

```
%=========================================================================%
% fouriercosine.m %
% Fourier cosine decomposition of a function.
%=========================================================================%

function a = fouriercosine(t,f, L, N)

    fhandle = @calc_da;

    for n = 0:N
        a(n+1) = quad(fhandle, 0,L); % quadrature integration

        if (n == 0)
            a(n+1) = a(n+1)/2;
        end
    end

    function da = calc_da(tt)
        ff = spline(t,f, tt);
        da = 2/L .* ff .* cos(n*pi.*tt/L);
    end

end

%=========================================================================%
```

C. revolve3D.m

```
%=========================================================================%
% revolve3D.m %
% Utility function to revolve a 2D shape to 3D.
%=========================================================================%

% revolve around x=0, i.e. z-axis
function [X,Y,Z] = revolve3D(x,z, step)

    steps = 0:step:2*pi;
    X = zeros(length(x),length(steps));
    Y = zeros(length(x),length(steps));
    Z = zeros(length(x),length(steps));

    for i = 1:length(z);
        for j = 1:length(steps)
            t = steps(j);
            X(i,j) = x(i)*cos(t);
            Y(i,j) = x(i)*sin(t);
            Z(i,j) = z(i);
        end
    end

end
%=========================================================================%
```

D. trends.m

```
%=========================================================================%
% trends.m %
% To render graphs based to show the trends depending on d.
%=========================================================================%

% load results data
data = load('../results/results.txt');
d = data(:,1);
ax = data(:,2);
Rt = data(:,6);
V = data(:,3);
A = data(:,4);
SA = data(:,5);
Gd = data(:,7);
Lam = data(:,8);
D = data(:,9);
W = data(:,10);

% load input and derived input data
data = load('../results/input.txt');
U = data(:,2);
Vs = data(:,6);
Re = data(:,7);
Cdp = data(:,8);

dd = d(1):.2:d(end);

%=========================================================================%
% (1) Reynolds Number, Re

p = polyfit(d,Re, 1); % linear
Ree = polyval(p,dd);

figure; hold on;
plot(dd,Ree, 'b-');
plot(d,Re, 'ro');
xlabel('d (mm)');
ylabel('Re');

%-------------------------------------------------------------------------%
% (2) Pressure drag coefficient, Cdp

% don't know fit

figure; hold on;
plot(d,Cdp, 'ro');
xlabel('d (mm)');
ylabel('Cdp');

%-------------------------------------------------------------------------%
% (3) Axis ratio, ax

p = polyfit(d,ax, 1); % linear
axx = polyval(p,dd);
```

```
55
56  figure; hold on;
57  plot(dd,axx, 'b-');
58  plot(d,ax, 'ro');
59  xlabel('d (mm)');
60  ylabel('\alpha');
61
62  %-------------------------------------------------------------------------%
63  % (4) Radius of Curvature, Rt
64
65  Rt = Rt * 1e3; % scale
66
67  p = polyfit(d,Rt, 2); % quadratic
68  Rtt = polyval(p,dd);
69
70  figure; hold on;
71  plot(dd,Rtt, 'b-');
72  plot(d,Rt, 'ro');
73  xlabel('d (mm)');
74  ylabel('R_t (mm)');
75
76  %-------------------------------------------------------------------------%
77  % (5) Sphere and Raindrop Volumes, Vs and V
78
79  V = V * 1e9; % scale
80
81  p = polyfit(d,Vs, 3); % cubic
82  Vss = polyval(p,dd);
83  p = polyfit(d,V, 3); % cubic
84  VV = polyval(p,dd);
85
86  figure; hold on;
87  plot(dd,Vss, 'b--');
88  plot(dd,VV, 'b-');
89  plot(d,Vs, 'rx');
90  plot(d,V, 'ro');
91  xlabel('d (mm)');
92  ylabel('V (mm^3)');
93  legend('Sphere volume fit', 'Raindrop volume fit', ...
94         'Sphere volume data', 'Raindrop volume data');
95
96  %-------------------------------------------------------------------------%
97  % (6) Cross-sectional Area, A
98
99  A = A * 1e6; % scale
100
101 p = polyfit(d,A, 2); % quadratic
102 AA = polyval(p,dd);
103
104 figure; hold on;
105 plot(dd,AA, 'b-');
106 plot(d,A, 'ro');
107 xlabel('d (mm)');
108 ylabel('A (mm^2)');
109
110 %-------------------------------------------------------------------------%
111 % (7) Surface Area, SA
112
```

```
113   SA = SA * 1e6; % scale
114
115   p = polyfit(d,SA, 2); % quadratic
116   SAA = polyval(p,dd);
117
118   figure; hold on;
119   plot(dd,SAA, 'b-');
120   plot(d,SA, 'ro');
121   xlabel('d (mm)');
122   ylabel('Surface area (mm^2)');
123
124   %-------------------------------------------------------------------------%
125   % (8) Pressure Wake Constant Factor, Gd
126
127   p = polyfit(d,Gd, 1); % linear
128   Gdd = polyval(p,dd);
129
130   figure; hold on;
131   plot(dd,Gdd, 'b-');
132   plot(d,Gd, 'ro');
133   xlabel('d (mm)');
134   ylabel('G_d');
135
136   %-------------------------------------------------------------------------%
137   % (9) Pressure Distribution Amplitude, Lam
138
139   p = polyfit(d,Lam, 6); % 6th order
140   Lamm = polyval(p,dd);
141
142   figure; hold on;
143   plot(dd,Lamm, 'b-');
144   plot(d,Lam, 'ro');
145   xlabel('d (mm)');
146   ylabel('\Lambda');
147
148   %-------------------------------------------------------------------------%
149   % (10) Drag and Weight, D, W
150
151   p = polyfit(d,D, 3); % cubic
152   DD = polyval(p,dd);
153   p = polyfit(d,W, 3); % cubic
154   WW = polyval(p,dd);
155
156   figure; hold on;
157   plot(dd,DD, 'b-');
158   plot(dd,WW, 'b--');
159   plot(d,D, 'ro');
160   plot(d,W, 'rx');
161   xlabel('d (mm)');
162   ylabel('Force (N)');
163   legend('Drag fit', 'Weight fit', ...
164          'Drag data', 'Weight data');
165
166   %=========================================================================%
```

E. comparephoto.m

```
%=========================================================================%
% comparephoto.m %
% Edge detection of photos from Magono 1954, and
% Comparison with the model
%=========================================================================%

d = [4.8 6.5];

n = 1;
I = imread(['../graphics/mangono1954_' num2str(d(n)) 'mm.jpg']);

I = imresize(I,2);
% imshow(I);

BW = im2bw(I,.41);
% figure;
% imshow(BW);

dim = size(BW);
col = round(dim(1)/2);
row = min(find(BW(:,col)));
boundary = bwtraceboundary(BW,[row, col],'N');

figure; imshow(I);

b2 = smooth(boundary(:,2));
b1 = smooth(boundary(:,1));
for i = 1:30
    b2 = smooth(b2);
    b1 = smooth(b1);
end

hold on;
% plot(col,row, 'ro');
plot(boundary(:,2),boundary(:,1),'g-','LineWidth',10);
plot(b2,b1,'r','LineWidth',5);

figure; hold on; axis equal;
x = b2;
y = b1;
cx = (min(x)+max(x))/2;
cy = (min(y)+max(y))/2;
x = x - cx; % center
y = y - cy;
y = -y; % flip from image coords to cartesian

%-------------------------------------------------------------------------%

% scale to mm
xspan = max(x)-min(x);
x = x .* d(n)/xspan;
y = y .* d(n)/xspan;

% shift bottom to y=0
```

```
ybot = min(y);
y = y - ybot;

%-------------------------------------------------------------------------%

% estimate volume elements
vol_i = find(x>0);
vol_x = x(vol_i);
vol_y = y(vol_i);
Vi(1) = 0;
for i = 2:length(vol_i)
    dvol_y = -(vol_y(i) - vol_y(i-1));
    Vi(i) = Vi(i-1) + pi*vol_x(i)^2 * dvol_y;
end

% get middle height
i = find(Vi < Vi(end)/2) + vol_i(1); % median volume increment
mid = y(i(end));

% translate y-axis
y = y - mid;

plot(x,y,'r-');

%=========================================================================%

% axis ratio
ax = (max(y)-min(y))/(max(x)-min(x))

%-------------------------------------------------------------------------%

% estimate diameter of equivalent-sized sphere
% approximate raindrop to an oblate spheroid
% V = pi/6*d^3 = ax * pi/6*xspan^3
d = (max(x)-min(x)) * ax^(1/3)

%=========================================================================%

data = load('../results/run_4.3755/table.txt');
x = data(:,3);
y = data(:,4);
Vi = data(:,5);

% scale to mm
x = x .* 1e3;
y = y .* 1e3;

% reflect x and add
x = [x; flipud(-x)]; % handles closing with horizontal line
y = [y; flipud(y)];

% flip y-axis
y = -y;
% get middle height
i = find(Vi < Vi(end)/2); % median volume increment
mid = y(i(end));
% translate y-axis
y = y - mid;
```

```
113
114   plot(x,y, 'b--');
115
116   xlabel('x[mm]');
117   ylabel('z[mm]');
118   legend('Magono 1954 (photo)', 'Current model');
119
120   %=========================================================================%
```