Technische Universität Berlin
Fakultät III – Prozesswissenschaften
Institut für Technischen Umweltschutz
Umweltverfahrenstechnik

Anhang I zur Diplomarbeit Entwicklung und Anwendung eines rechnergestützten Modells zur Ursachenanalyse großräumiger Luftschadstofftransporte - trajectory.c

Diplomarbeit
im Studiengang Technischer Umweltschutz

vorgelegt von
Roman Finkelnburg

Wissenschaftliche Leitung: Prof. Dr.-Ing. Sven-Uwe Geißen
Wissenschaftliche Betreuung: Dr.-Ing. Markus Pesch

Berlin, November 2007

```
/* trajectory.c */

/* Copyright (c) 2007 Roman Finkelnburg
 *
 * Permission to use, copy, modify, and distribute this software for any
 * purpose with or without fee is hereby granted, provided that the above
 * copyright notice and this permission notice appear in all copies.
 *
 * THE SOFTWARE IS PROVIDED "AS IS" AND THE AUTHOR DISCLAIMS ALL WARRANTIES
 * WITH REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED WARRANTIES OF
 * MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR
 * ANY SPECIAL, DIRECT, INDIRECT, OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
 * WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN
 * ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF
 * OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
 */

/*
 * Dieses Programm errechnet mit Hilfe von Bodenwinddatensaetzen Vorwaerts-
 * und Rueckwaertstrajektorien.
 *
 * *******
 * *INPUT*
 * *******
 * Die Bodenwinddaten muessen tageweise als ASCII-Dateien (bYYMMTT.new)
 * vorliegen. Die Daten muessen stundenweise zeitlich rueckwaerts geordnet
 * (23, 22, ..., 1, 0) in Datenbloecken zusammengefasst in den Dateien
 * vorliegen.
 * Format:
 * 2001 03 14 23
 *  Stationsnummer Windrichtung Windgeschwindigkeit
 *  Stationsnummer Windrichtung Windgeschwindigkeit
 *  Stationsnummer Windrichtung Windgeschwindigkeit
 *  Stationsnummer Windrichtung Windgeschwindigkeit
 *  01205 145 5
 * *ENDBLOCK
 * 2001 03 14 22
 *  Stationsnummer Windrichtung Windgeschwindigkeit
 *  Stationsnummer Windrichtung Windgeschwindigkeit
 *  Stationsnummer Windrichtung Windgeschwindigkeit
```

```
*  01205 187 8
* *ENDBLOCK
* (Die Winddaten sind durch ein Leerzeichen eingerueckt!
* Blockanfang (YYYY MM DD HH) und Blockende (*ENDBLOCK) sind nicht
* eingerueckt!)
*
* Zusaetzlich muss eine Datei mit Stationsinformationen angegeben werden.
* Format:
*  Stationsnummer Breitengrad[DDMM] Laengengrad[DDDMMM] (Hoehe) (Messeinheit)
*  Stationsnummer Breitengrad[DDMM] Laengengrad[DDDMMM] (Hoehe) (Messeinheit)
*  Stationsnummer Breitengrad[DDMM] Laengengrad[DDDMMM] (Hoehe) (Messeinheit)
*  01205 +6220 +00516 0038 2
*
* Fuer eine ausfuehrlichere Beschreibung der Eingabedaten kann in der Arbeit
* "Analyse eines rechnergestuetzten Trajektorienmodells zur Darstellung von
*  regionalen Luftschadstofftransporten", R. Finkelnburg, Projektarbeit am
*  Fachgebiet UVT, Institut fuer Technischen Umweltschutz, TU-Berlin nachge-
* lesen werden.
*
* ********
* *OUTPUT*
* ********
* Die berechnete Trajektorie wird im gewaehlten Ausgabeverzeichnis in einer
* Datei der Form
* RYYYYMMDD_HH.trj  R:    "B" Rueckwaertstrajektorie
*                         "F" Vorwaertstrajektorie
*                   YYYY: Jahr
*                   MM:   Monat
*                   DD:   Tag
*                   HH:   Stunde
* ausgegeben.
*
* *******
* *START*
* *******
* Zum Programmstart koennen in der Programmumgebung (Shell, MS-DOS) spezielle
* Umgebungsvariablen gesetzt werden. Ueber diese Umgebungsvariablen werden dem
* Programm die gewuenschten Start- und Ausfuehrungsparameter uebergeben. Ist
* eine Umgebungsvariable nicht gesetzt, so wird fuer diesen Parameter der
* programminterne Standardwert verwendet.
```

```
 *
 * Dem Programm koennen die folgenden Parameter uebergeben werden:
 *
 * PARAMETER                                    UMGEBUNGSVARIABLE STANDARDWERT
 * Startposition Laengengrad (Grad)             LO                13.4167
 * Startposition Breitengrad (Grad)             LA                52.5167
 * Startzeit Jahr                               YYYY              2000
 * Startzeit Monat                              MM                01
 * Startzeit Tag                                DD                01
 * Startzeit Stunde                             HH                00
 * Verfolgungszeit (h)                          TRACE             -96
 * Geschwindigkeitskorrektur                    SPEED             2.0
 * Richtungskorrektur (Grad)                    ROT               10.0
 * Berechnungsgebietsradius (km)                MAXR              200
 * Mindestwichtungsdistanz (km)                 MINR              2
 * Iterationen pro Zeitstunde                   IPERH             20
 * Iterationen pro Aufpunkt                     IPERPOINT         20
 * Zeitzonendifferenz (Startzeit/Winddaten) (h) ZONEDIFF          -1
 * Zeitzonenname                                ZONENAME          MEZ
 * Stationsinformationsdatei                    STATION           wstation.dat
 * Winddatensatzverzeichnis                     METEO             meteo/
 * Ausgabeverzeichnis                           OUTPUT            traj/
 * zulaessige Standardabweichung (0: off)       STDDEVIATION      0.0
 * Windgeschwindigkeitseinheit
 * (0:kn, 1:m/s, 2:aus Stationsliste)           DATAUNIT          0
 * zeitliche Aufloesung der Winddatensaetze (h) RES               3
 */

/*
 * PROGRAMMSTRUKTUR
 *
 * main()
 * .      read_env()
 * .      init_values()
 * .      .      get_amount_of_stations()
 * .      .      normalize_coords()
 * .      calculate()
 * .      .      prepare_calculate()
 * .      .      .      convert_timezone()
 * .      .      .      read_station_list()
```

```
 * .       .       .       new_winddata()
 * .       .       .       read_wind_data()
 * .       .       .       .       time_step_forward()
 * .       .       .       .       time_step_backward()
 * .       .       .       .       read_file()
 * .       .       .       init_wind_data()
 * .       .       .       .       get_next_element()
 * .       .       .       .       get_prev_element()
 * .       .       .       check_resolution()
 * .       .       .       wind_of_next_hour()
 * .       .       iterate()
 * .       .       .       copy_wind_current()
 * .       .       .       get_next_wind_data()
 * .       .       .       .       get_next_element()
 * .       .       .       .       get_prev_element()
 * .       .       .       check_resolution()
 * .       .       .       wind_of_next_hour()
 * .       .       .       convert_geo_to_cartesian()
 * .       .       .       calculate_wind_vector()
 * .       .       .       .       check_station_weight()
 * .       .       .       .       .       distance_to_station_in_cos()
 * .       .       .       .       average_sum()
 * .       .       .       .       deviation_sum()
 * .       .       .       .       std_deviation()
 * .       .       .       .       z_transformation()
 * .       .       .       .       end_sum()
 * .       .       normalize_coords()
 * .       print_output_file()
 * .       .       generate_output_filename()
 * .       reset_state()
 */

#include <assert.h>
#include <ctype.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/***********
```

```
 * DEFINES *
 ***********/
#define MILE      1.8532 /* Meile in km */
#define RE        6370.0 /* Erdradius in km */
#define MAXLINE   256    /* maximale Zeichenanzahl pro Zeichenkette */
#define RESMAX    24     /* maximaler Zeitabstand der Winddaten in h */

/****************
 * DECLARATIONS *
 ****************/

/* Startparameter */
struct param {
        char* name;
        enum {
                TYP_INT,
                TYP_FLOAT,
                TYP_STRING
        } type;
        union {
                char* s;
                int i;
                double f;
        } u;
        char* desc;
} param[] = {
        {"LO",          TYP_FLOAT,  { "13.4167" },
         "longitude [degree]"},

        {"LA",          TYP_FLOAT,  { "52.5167" },
         "latitude [degree]"},

        {"YYYY",        TYP_INT,    { "2000" },
         "year"},

        {"MM",          TYP_INT,    { "1" },
         "month"},

        {"DD",          TYP_INT,    { "1" },
         "day"},
```

```
{"HH",           TYP_INT,    { "0" },
 "hour"},

{"TRACE",        TYP_INT,    { "-96" },
 "term of calculation [h]"},

{"SPEED",        TYP_FLOAT,  { "2.0" },
 "correction of windspeed [factor]"},

{"ROT",          TYP_FLOAT,  { "10.0" },
 "correction of winddirection [degree]"},

{"MAXR",         TYP_INT,    { "200" },
 "radius of interpolation [km]"},

{"MINR",         TYP_INT,    { "2" },
 "least distance for weightning [km]"},

{"IPERH",        TYP_INT,    { "20" },
 "interations per hour"},

{"IPERPOINT",    TYP_INT,    { "20" },
 "iterations per point"},

{"ZONEDIFF",     TYP_INT,    { "-1" },
 "timezone difference [h]"},

{"ZONENAME",     TYP_STRING, { "MEZ" },
 "name of timezone"},

{"STATION",      TYP_STRING, { "wstation.dat" },
 "file of stationinformations"},

{"METEO",        TYP_STRING, { "meteo/" },
 "directory of input data"},

{"OUTPUT",       TYP_STRING, { "traj/" },
 "directory of output data"},
```

```
        {"STDDEVIATION", TYP_FLOAT,  { "0.0" },
         "standard deviation (0.0: off)"},

        {"DATAUNIT",     TYP_INT,    { "0" },
         "modus of input data (0:kn, 1:m/s, 2:mixed)"},

        {"RES",          TYP_INT,    { "3" },
         "resolution of wind data (0:off)"},

        {NULL,           0,          { NULL }, NULL }
};

enum {
        LO           = 0,
        LA           = 1,
        YYYY         = 2,
        MM           = 3,
        DD           = 4,
        HH           = 5,
        TRACE        = 6,
        SPEED        = 7,
        ROT          = 8,
        MAXR         = 9,
        MINR         = 10,
        IPERH        = 11,
        IPERPOINT    = 12,
        ZONEDIFF     = 13,
        ZONENAME     = 14,
        STATION      = 15,
        METEO        = 16,
        OUTPUT       = 17,
        STDDEVIATION = 18,
        DATAUNIT     = 19,
        RES          = 20
};

/* Datenelement fuer eine Station (Stationsliste) */
struct station {

        /* Stationsnummer */
```

```
        int nr;

        /* Messeinheit in der die Station Windgeschwindigkeiten misst */
        int unit;

        /* Stationsposition in kartesischen Koordinaten */
        double X[3];
};

/* Datenstruktur zum Speichern eines Zeitpunkts */
struct date {
        int year;
        int month;
        int day;
        int hour;
};

/* Windvektor einer Station */
struct wind {
        double u; /* 1ste Dimension des Windvektors */
        double v; /* 2te Dimension des Windvektors */
        int    p; /* Present-Flag (0: present, 1: empty) */
};

/* Struktur zur Speicherung des momentanen Programmstatus */
struct state {

        /*
         * Array zum Speichern der Aufpunktlaengengrade der
         * Trajektorie
         */
        double* lo;

        /*
         * Array zum Speichern der Aufpunktbreitengrade der
         * Trajektorie
         */
        double* la;

        /* aktuelle Trajektorienpunktnummer */
```

```
int point;

/* maximale Anzahl der Trajektorienpunkte */
int point_max;

/* Anzahl der in der Stationsliste enthaltenen Stationen */
int station_max;

/* Struktur zum Speichern der Stationsinformationen */
struct station* station_list;

/*
 * Struktur zum Speichern der Daten-Windfeldern (wind_data[0] und
 * wind_data[1])
 */
struct wind* wind_data;

/*
 * Struktur zum Speichern der Stunden-Windfelder (wind_current[0]
 * und wind_current[1])
 */
struct wind* wind_current;

/* aktuelle interne Berechnungszeit */
struct date time;

/*
 * zeitliche Differenz (h) zwischen letztem Daten-Windfeld und
 * aktueller Berechnugnszeit
 */
double diff;

/* zeitliche Differenz (h) zwischen den Daten-Windfeldern */
double data_diff;

/*
 * Umrechnungsfaktor von Windgeschwindigkeit (m/s) in zurueckgelegter
 * Distanz (Rad) pro Itereation
 */
double distance_per_step;
```

```
        /*
         * Minimalradius des Berechnungsgebiets angegeben als Kosinus des
         * Winkels der Ortvektoren des Kreismittelpunkts zum Kreisrand
         */
        double cos_min_r;

        /*
         * Maximalradius des Berechnungsgebiets angegeben als Kosinus des
         * Winkels der Ortvektoren des Kreismittelpunkts zum Kreisrand
         */
        double cos_max_r;
};

/* Struktur eines Stundenwindfelds */
struct winddata {
        struct date time;
        struct wind* wind;
        struct winddata* next;
        struct winddata* prev;
};

/**********
 * MACROS *
 **********/
#define rad2deg(f)        ((f) * 180 / M_PI)
#define deg2rad(f)        ((f) * M_PI / 180)

/*
 * Markos zum Auslesen der verschiedenen Datentypen (int, float, string) aus
 * der Programmstartparameterstruktur (struct param)
 */
#define get_int(p)     (assert(param[(p)].type == TYP_INT), param[(p)].u.i)
#define get_float(p)   (assert(param[(p)].type == TYP_FLOAT), param[(p)].u.f)
#define get_string(p)  (assert(param[(p)].type == TYP_STRING), param[(p)].u.s)

/*
 * Makro zum kopieren der Werte der Zeitstruktur src in die Zeitstruktur
 * dst
 */
```

```
#define time_copy(src, dst) do { \
        (dst).year = (src).year; \
        (dst).month = (src).month; \
        (dst).day = (src).day; \
        (dst).hour = (src).hour; \
} while (0)

/*
 * Makro zum vergleichen zweier Zeitstrukutren. Wenn die Zeiten gleich sind,
 * ist -> time_equal == 0
 */
#define time_equal(t1, t2)  ((t1).year == (t2).year && (t1).month == \
                             (t2).month && (t1).day == (t2).day && \
                             (t1).hour == (t2).hour)

/**************
 * PROTOTYPES *
 **************/

void              average_sum(double, double*, double*);
void              calculate(struct state*);
int               calculate_wind_vector(double, double*, double*, double*,
                                        struct state*);
void              check_resolution(int, int);
void              check_station_weight(struct state, struct wind*, double*,
                                       int, double*, double*, double*,
                                       double*, int);
void              convert_geo_to_cartesian(double, double, double*);
void              convert_timezone(struct state*);
void              copy_wind_current(struct state*);
void              deviation_sum(struct wind*, double, double, double*,
                                double*, double*, int);
double            distance_to_station_in_cos(int, double*, struct state);
void              end_sum(struct wind*, double*, double*, double*,
                          double*, int);
int               generate_output_filename(char*, size_t);
int               get_amount_of_stations(struct state*);
struct winddata* get_next_element(struct winddata*);
struct winddata* get_next_wind_data(struct state*, struct winddata*);
struct winddata* get_prev_element(struct winddata*);
```

```
void             init_values(struct state*);
struct winddata* init_wind_data(struct state*, struct winddata*);
void             iterate(struct state*, struct winddata**);
struct winddata* new_winddata(void);
void             normalize_coords(struct state*);
void             prepare_calculate(struct state*, struct winddata**);
void             print_output_file(const struct state*);
void             read_env(struct param*);
struct winddata* read_file(struct state*, char*);
void             read_station_list(struct state*);
void             read_wind_data(struct state*, struct winddata*);
void             reset_state(struct state*);
void             std_deviation(double, double*, double*);
void             time_step_backward(struct date*);
void             time_step_forward(struct date*);
void             wind_of_next_hour(struct state*, struct winddata*);
void             z_transformation_check(struct wind*, double, double, double,
                                        double, double, int);

/****************
 * MAINFUNCTION *
 ****************/

int
main(void)
{
        struct state state;

        /* Einlesen der uebergebenen Argumente */
        read_env(param);

        /*
         * Initialisieren der Datenstruktur zum Abbilden des programminternen
         * Berechnungsstatus
         */
        init_values(&state);

        /* Berechnen der Trajektorie */
        calculate(&state);
```

```
        /* Ausgeben der Trajektorie */
        print_output_file(&state);

        /* Reservierte Speicherbereiche wieder freigeben */
        reset_state(&state);

        return (0);
}

/***************
 * SUBROUTINES *
 ***************/

/*
 * Mittelwerte bilden
 *
 * >> Mittelwert = Summe / Anzahl <<
 *
 */
void
average_sum(double amount, double* u_average, double* v_average)
{
        /* Wenn Stationen in Reichweite waren */
        if (amount > 0) {
                if (*u_average != 0) {
                        *u_average /=amount;
                }
                if (*v_average != 0) {
                        *v_average /=amount;
                }
        }
}

/* Berechnen der einzelnen Trajektorienaufpunkte */
void
calculate(struct state *state)
{
        struct winddata* winddata;
        /* Alle Daten fuer die Berechnung zusammensammeln */
        prepare_calculate(state, &winddata);
```

```
        /* Start der Aufpunktberechnung */
        for (state->point = 1; state->point <= state->point_max;
             state->point += 1) {

                /* Kopieren der Koordinaten der letzten Aufpunktposition fuer
                 * naechste Iteration */
                state->lo[state->point] = state->lo[state->point - 1];
                state->la[state->point] = state->la[state->point - 1];

                /* Bis zum naechsten Aufpunkt iterieren */
                iterate(state, &winddata);

                /* Umrechnen auf geographischen Groessenbereich */
                normalize_coords(state);
        }
}

/*
 * Zeitliche und raeumliche Interpolation der beiden in wind_current
 * gespeicherten Windfelder zu einem Windvektor (u,v) an der aktuelle
 * Berechnungsposition (X)
 *
 * Rueckgabewert is
 *    0, wenn Vektor berechenet werden konnte
 *    1, wenn Vektor nicht berechnet werden konnte
 */
int
calculate_wind_vector(double hour_diff, double* u, double* v, double X[],
    struct state* state)
{
        int i;
        double u_sum_wind1, v_sum_wind1, weight_sum_wind1;
        double u_sum_wind2, v_sum_wind2, weight_sum_wind2;
        double amount_wind1, amount_wind2;
        double u_average_wind1, v_average_wind1;
        double u_average_wind2, v_average_wind2;
        double u_stddev_wind1, v_stddev_wind1;
        double u_stddev_wind2, v_stddev_wind2;
```

```
double weight[state->station_max];

/* Speicher reservieren */
struct wind wind1_in_range[state->station_max];
struct wind wind2_in_range[state->station_max];

/* Initialisieren */
u_sum_wind1 = v_sum_wind1 = weight_sum_wind1 = 0;
u_sum_wind2 = v_sum_wind2 = weight_sum_wind2 = 0;
amount_wind1 = amount_wind2 = 0;
u_average_wind1 = v_average_wind1 = 0;
u_average_wind2 = v_average_wind2 = 0;

for (i = 0; i < state->station_max; i++) {

        wind1_in_range[i].u = 0;
        wind1_in_range[i].v = 0;
        wind1_in_range[i].p = 0;

        wind2_in_range[i].u = 0;
        wind2_in_range[i].v = 0;
        wind2_in_range[i].p = 0;

        weight[i] = 0;
}

/*
 * Speichern der Winddaten von Stationen in Reichweite und
 * aufsummieren der gewichteten Werte
 *
 * >> Summe = sum(xi) <<
 *
 */
for (i = 0; i < state->station_max; i++) {

        check_station_weight(*state, wind1_in_range, X, i, weight,
            &u_average_wind1, &v_average_wind1,
            &amount_wind1, 0);

        check_station_weight(*state, wind2_in_range, X, i, weight,
```

```
            &u_average_wind2, &v_average_wind2,
            &amount_wind2, state->station_max);
}

/*
 * Wenn maximale Standardabweichung angegeben ist, werden alle
 * Stationen, die in u oder v eine groessere Abweichung vom u-
 * bzw. v-Mittelwert besitzen, aus der Berechnung genommen
 */
if (get_float(STDDEVIATION) > 0.0) {

        /* Mittelwerte bilden
         *
         * >> Mittelwert = Summe / Anzahl <<
         *
         */
        average_sum(amount_wind1,
            &u_average_wind1, &v_average_wind1);

        average_sum(amount_wind2,
            &u_average_wind2, &v_average_wind2);

        /* Werte zuruecksetzen */
        u_stddev_wind1 = v_stddev_wind1 = 0;
        u_stddev_wind2 = v_stddev_wind2 = 0;
        amount_wind1 = amount_wind2 = 0;

        /*
         * Errechnen der Summe der Abweichungsquadrate
         *
         * >> QSumme = sum((xi - Mittelwert) *(xi - Mittelwert)) <<
         *
         */
        for (i = 0; i < state->station_max; i++) {

                deviation_sum(wind1_in_range,
                    u_average_wind1, v_average_wind1,
                    &u_stddev_wind1, &v_stddev_wind1,
                    &amount_wind1, i);
```

```
                deviation_sum(wind2_in_range,
                    u_average_wind2, v_average_wind2,
                    &u_stddev_wind2, &v_stddev_wind2,
                    &amount_wind2, i);
            }

            /*
             * Wenn Wetterstation in Reichweite sind, kann Standardab-
             * weichung berechnet werden.
             *
             * >> Standardabweichung = wurzel(QSumme / Anzahl) <<
             *
             */
            std_deviation(amount_wind1, &u_stddev_wind1,
                &v_stddev_wind1);

            std_deviation(amount_wind2, &u_stddev_wind2,
                &v_stddev_wind2);

            /* Werte mit zu grosser Abweichung rauswerfen */
            for (i = 0; i < state->station_max; i++) {

                z_transformation_check(wind1_in_range,
                    u_average_wind1, v_average_wind1,
                    u_stddev_wind1, v_stddev_wind1,
                    get_float(STDDEVIATION), i);

                z_transformation_check(wind2_in_range,
                    u_average_wind2, v_average_wind2,
                    u_stddev_wind2, v_stddev_wind2,
                    get_float(STDDEVIATION), i);
            }
        }

        /* Berechnen des gemittelten gewichteten Windes */
        for (i = 0; i < state->station_max; i++) {

            end_sum(wind1_in_range, &u_sum_wind1, &v_sum_wind1,
                weight, &weight_sum_wind1, i);
```

```
        end_sum(wind2_in_range, &u_sum_wind2, &v_sum_wind2,
            weight, &weight_sum_wind2, i);
}

/*
 * Zeitliches Wichten:
 * Je nach Iterationsfortschritt entfernt sich die momentane
 * Berechnungszeit vom zweiten Windfeld in wind_current hin zum
 * ersten Windfeld int wind_current. Zwischen den beiden Windfeldern
 * in wind_current besteht immer genau 1 Zeitstunde Unterschied.
 * hour_diff gibt den Anteil an, der waerend der Iterationen schon
 * vergangen ist. Die zeitliche Wichtung erfolgt ueber hour_diff
 *
 * wind_current[1]|-----------------------------|wind_current[0]
 *                0         |                   1
 *                       hour_diff=0.3
 */
if ((hour_diff == 0)) {
        if (get_int(TRACE) > 0) {
                if (weight_sum_wind1 != 0) {
                        *u = u_sum_wind1 / weight_sum_wind1;
                        *v = v_sum_wind1 / weight_sum_wind1;
                        return 0;
                }
        }
        else {
                if (weight_sum_wind2 != 0) {
                        *u = u_sum_wind2 / weight_sum_wind2;
                        *v = v_sum_wind2 / weight_sum_wind2;
                        return 0;
                }
        }
        return 1;
}
else {
        if ((weight_sum_wind1 != 0) &&
            (weight_sum_wind2 != 0)) {

                if (get_int(TRACE) > 0) {
                        *u = (1.0 - hour_diff) *
```

```
                                (u_sum_wind1 / weight_sum_wind1) +
                                hour_diff *
                                (u_sum_wind2 / weight_sum_wind2);

                        *v = (1.0 - hour_diff) *
                                (v_sum_wind1 / weight_sum_wind1) +
                                hour_diff *
                                (v_sum_wind2 / weight_sum_wind2);
                }
                else {
                        *u = hour_diff *
                                (u_sum_wind1 / weight_sum_wind1) +
                                (1.0 - hour_diff) *
                                (u_sum_wind2 / weight_sum_wind2);

                        *v = hour_diff *
                                (v_sum_wind1 / weight_sum_wind1) +
                                (1.0 - hour_diff) *
                                (v_sum_wind2 / weight_sum_wind2);
                }
                return 0;
        }
        else {

                return 1;
        }
    }
}

/* Ueberpruefen, ob angegebene zeitliche Aufloesung der Winddaten zutrifft */
void
check_resolution(int res, int DeltaT)
{
    if ((res != 0) && (res != DeltaT)) {
            printf("Error: resolution of wind data is (%i)\n", DeltaT);
            exit (1);
    }
    else if (DeltaT > 24){
            printf("Error: resolution of wind data is (%i)\n", DeltaT);
            exit (1);
```

```
        }
}

/*
 * Ueberpruefen, ob Station in Reichweite ist und Berechnung ihres
 * oertlichen Wichtungsfaktors
 */
void
check_station_weight(struct state state, struct wind* wind_in_range,
    double* X, int i, double* weight, double* u_sum, double* v_sum,
    double* amount, int offset)
{
        double val;

        /* Wenn keine Daten vorhanden sind */
        if (state.wind_current[offset+i].p == 0)
                return;

        /* Berechnen der Distanz von Berechnungspunkt und Station i */
        val = distance_to_station_in_cos(i, X, state);

        /* Wenn Station naeher als min_r */
        if (val > state.cos_min_r) {
                val = state.cos_min_r;
        }

        /* Wenn Station innerhalb von max_r */
        if (val > state.cos_max_r) {

                /* Wichtung mit 1 / r^2 */
                weight[i] = 1 / (acos(val) * acos(val));

                wind_in_range[i].p = 1;

                wind_in_range[i].u = state.wind_current[offset+i].u *
                    weight[i];

                wind_in_range[i].v = state.wind_current[offset+i].v *
                    weight[i];
```

```
                /*
                 * Aufsummieren der Winddaten fuer spaetere
                 * Mittelwertberechnung
                 */
                *u_sum += wind_in_range[i].u;
                *v_sum += wind_in_range[i].v;
                *amount += 1;
        }
}

/*
 * Umrechnen der geographischen Positionsangabe in Rad (longitude,
 * latitude) in einen katesischen Ortsvektor (X)
 */
void
convert_geo_to_cartesian(double longitude, double latitude, double X[])
{
        X[0] = cos(latitude) * cos(longitude);
        X[1] = cos(latitude) * sin(longitude);
        X[2] = sin(latitude);
}

/* Umrechnung von externer Zeitzone in interne Zeitzone (GMT) */
void
convert_timezone(struct state* state) {

        int i;

        if (get_int(ZONEDIFF) < 0) {
                for (i = 0; i > get_int(ZONEDIFF); i--) {
                        time_step_backward(&state->time);
                }
        }
        else {
                for (i = 0; i < get_int(ZONEDIFF); i++) {
                        time_step_forward(&state->time);
                }
        }
}
```

```
/*
 * Umkopieren des Stundenwindfeldes in wind_current (wind_current[0] ->
 * wind_current[1]).
 */
void
copy_wind_current(struct state* state)
{
        int j;

        for (j = 0; j < state->station_max; j++) {

                state->wind_current[state->station_max + j].u =
                    state->wind_current[j].u;

                state->wind_current[state->station_max + j].v =
                    state->wind_current[j].v;

                state->wind_current[state->station_max + j].p =
                    state->wind_current[j].p;
        }
}

/*
 * Errechnen der Summe der Abweichungsquadrate
 *
 * >> QSumme = sum((xi - Mittelwert) *(xi - Mittelwert)) <<
 *
 */
void
deviation_sum(struct wind* wind_in_range,
    double u_average, double v_average,
    double* u_stddev, double* v_stddev,
    double* amount, int i)
{
        /* Wenn Daten vorhanden sind */
        if (wind_in_range[i].p != 0) {

                *u_stddev += (wind_in_range[i].u - u_average) *
                    (wind_in_range[i].u - u_average);
```

```
                *v_stddev += (wind_in_range[i].v - v_average) *
                    (wind_in_range[i].v - v_average);

                *amount += 1;
        }
}

/*
 * Errechnen des Skalarprodukts von zwei Ortsvektoren. Fuer dieses
 * Programm liegen alle Punkte (Ortsvektoren) auf einer Kugel-
 * oberflaeche mit dem Radius 1 (programminterne Berechnungen
 * behandeln die Erdkugel als Einheitskugel)
 *
 * -> Der Betrag aller Ortsvekoren (Kugelmittelpunkt bis Kugel-
 *    oberflaeche) ist 1
 *
 * Rueckgabewert ist das Skalarprodukt bzw. der Kosinus des Winkels
 * zwischen den beiden Ortsvektoren.
 */
double
distance_to_station_in_cos(int j, double X[], struct state state)
{
        return (state.station_list[j].X[0] * X[0] +
            state.station_list[j].X[1] * X[1] +
            state.station_list[j].X[2] * X[2]);
}

/* Berechnen des gemittelten gewichteten Windes */
void
end_sum(struct wind* wind_in_range, double* u_sum, double* v_sum,
    double* weight, double* weight_sum, int i)
{
        /* Wenn Daten vorhanden sind */
        if (wind_in_range[i].p != 0) {

                /* Aufaddieren der Windvektoren */
                *u_sum      += wind_in_range[i].u;
                *v_sum      += wind_in_range[i].v;
                *weight_sum += weight[i];
        }
```

```
}

/*
 * Generieren eines Ausgabedateinamens aus den gesetzten Programmparametern
 *
 * Rueckgabewert ist der generierte Dateinamen
 */
int
generate_output_filename(char* filename, size_t size)
{
        /* Wenn Rueckwaertstrajektorie */
        if (get_int(TRACE) < 0) {
                return snprintf(filename, size,
                    "%sB%04i%02i%02i_%02i.trj",
                    get_string(OUTPUT), get_int(YYYY),
                    get_int(MM), get_int(DD),
                    get_int(HH));
        }

        /* Wenn Vorwaertstrajektorie */
        else {
                return snprintf(filename, size,
                    "%sF%04i%02i%02i_%02i.trj",
                    get_string(OUTPUT), get_int(YYYY),
                    get_int(MM), get_int(DD),
                    get_int(HH));
        }
}

/*
 * Zaehlen der in der Stationsinformationsdatei enthaltenen
 * Stationen und rueckgabe der Anzahl als Rueckgabewer
 *
 * Rueckgabewert ist
 *    0, wenn keine Stationen gefunden wurden
 *    Anzahl, wenn Stationen gefunden wurden
 */
int
get_amount_of_stations(struct state* state)
{
```

```
        int   i, c;
        FILE* fh;

        /* Ueberpruefen, ob Datei existiert */
        if (!(fh = fopen(get_string(STATION), "r"))) {
                printf("Couldn't open file %s!\n", get_string(STATION));
                exit(1);
        }

        /* Zaehlen der Stationseintraege in der Datei */
        i = 0;
        while ((c = getc(fh)) && (c != EOF)) {
                if (c == '\n')
                        ++i;
        }

        /* Datei schliessen */
        fclose(fh);

        /* Rueckgabe der Stationsanzahl */
        return i;
}

/* Zum naechsten Listenelement in der Kette springen
 * Rueckgabewert ist das naechste Listenelement
 */
struct winddata*
get_next_element(struct winddata* winddata)
{
        do {
                winddata = winddata->next;
                if (winddata == NULL) {
                        printf("get_next_element: end of list!\n");
                        exit(1);
                }
        } while (winddata->wind == NULL);

        return winddata;
}
```

```
/*
 * Verschieben des Winddatenfeldes wind_data um data_diff bis zum
 * naechstn gueltigen Datenblock
 */
struct winddata*
get_next_wind_data(struct state* state, struct winddata* winddata)
{
        struct date time_dummy;
        int   i;

        time_dummy.year = winddata->time.year;
        time_dummy.month = winddata->time.month;
        time_dummy.day = winddata->time.day;
        time_dummy.hour = winddata->time.hour;

        winddata = get_next_element(winddata);

        state->data_diff = 0;

        /* Wenn Vorwaertstrajektorie */
        if (get_int(TRACE) > 0) {
                while (!time_equal(time_dummy, winddata->time)) {
                        time_step_forward(&time_dummy);
                        state->data_diff += 1;
                }
        }

        /* Wenn Rueckwaertstrajektorie */
        else {
                while (!time_equal(time_dummy, winddata->time)) {
                        time_step_backward(&time_dummy);
                        state->data_diff += 1;
                }
        }

        for (i = 0; i < state->station_max; i++) {
                if (get_int(TRACE) > 0) {
                        state->wind_data[i].u =
                            state->wind_data[state->station_max + i].u;
```

```
                    state->wind_data[i].v =
                        state->wind_data[state->station_max + i].v;

                    state->wind_data[i].p =
                        state->wind_data[state->station_max + i].p;

                    state->wind_data[state->station_max + i].u =
                        winddata->wind[i].u;

                    state->wind_data[state->station_max + i].v =
                        winddata->wind[i].v;

                    state->wind_data[state->station_max + i].p =
                        winddata->wind[i].p;
                }
                else {

                    state->wind_data[state->station_max + i].u =
                        state->wind_data[i].u;

                    state->wind_data[state->station_max + i].v =
                        state->wind_data[i].v;

                    state->wind_data[state->station_max + i].p =
                        state->wind_data[i].p;

                    state->wind_data[i].u =
                        winddata->wind[i].u;

                    state->wind_data[i].v =
                        winddata->wind[i].v;

                    state->wind_data[i].p =
                        winddata->wind[i].p;
                }
        }

        return winddata;
}
```

```
/* Zum vorherigen Listenelement in der Kette springen
 * Rueckgabewert ist das vorherige Listenelement
 */
struct winddata*
get_prev_element(struct winddata* winddata)
{
        do {
                winddata = winddata->prev;
                if (winddata == NULL) {
                        printf("get_prev_element: end of list!\n");
                        exit(1);
                }
        } while (winddata->wind == NULL);

        return winddata;
}

/*
 * Initialisieren der Datenstruktur zum Abbilden des programminternen
 * Berechnungsstatus
 */
void
init_values(struct state* state)
{
        memset(state, 0, sizeof(struct state));

        state->distance_per_step = 3.6 / (get_int(IPERH) * RE);
        state->cos_max_r = cos(get_int(MAXR) / RE);
        state->cos_min_r = cos(get_int(MINR) / RE);
        state->station_max = get_amount_of_stations(state);
        state->point_max = (int)((double)get_int(IPERH) /
            (double)get_int(IPERPOINT) *
            (double)abs(get_int(TRACE)));
        state->point = 0;

        state->lo = calloc(state->point_max + 1, sizeof(double));
        state->la = calloc(state->point_max + 1, sizeof(double));

        state->wind_data = calloc(2 * state->station_max,
            sizeof(struct wind));
```

```
        state->wind_current = calloc(2 * state->station_max,
            sizeof(struct wind));
        state->station_list = calloc(state->station_max,
            sizeof(struct station));

        state->time.year = get_int(YYYY);
        state->time.month = get_int(MM);
        state->time.day = get_int(DD);
        state->time.hour = get_int(HH);

        state->lo[0] = deg2rad(get_float(LO));
        state->la[0] = deg2rad(get_float(LA));

        /* Umrechnen der Startposition auf geographischen Groessenbereich */
        normalize_coords(state);
}

/*
 * Erstellen der ersten beiden Daten-Windfelder fuer die zeitliche
 * Interpolation waehrend der Trajektorienberechnung
 */
struct winddata*
init_wind_data(struct state* state, struct winddata* winddata)
{
        struct date time_dummy; /* Letzte gueltige Zeitangabe */
        int i;

        /*
         * Einlesen des ersten Daten-Windfeldes.
         *
         * Bis Dateiende erreicht oder Windfeld eingelesen ist, das am
         * naechsten zum Berechunungsstartzeitpunkt lieg (relativ zur
         * Berechnugnsrichtung vorher oder gleich).
         */
        while (!(time_equal(state->time, winddata->time))) {
                winddata = winddata->next;

                if (winddata == NULL) {
                        printf("init_wind_data: end of list!\n");
                        exit(1);
```

```
        }
}

state->diff = state->data_diff = 0;

/* Wenn Daten zur Startzeit vorhanden sind */
if(winddata->wind == NULL) {

        time_dummy.year = winddata->time.year;
        time_dummy.month = winddata->time.month;
        time_dummy.day = winddata->time.day;
        time_dummy.hour = winddata->time.hour;

        if (get_int(TRACE) > 0)
                winddata = get_prev_element(winddata);
        else
                winddata = get_next_element(winddata);

        while (!time_equal(time_dummy, winddata->time)) {
                time_step_backward(&time_dummy);
                state->diff += 1;
        }
}

for (i = 0; i < state->station_max; i++) {
        state->wind_data[i].u =
            winddata->wind[i].u;

        state->wind_data[i].v =
            winddata->wind[i].v;

        state->wind_data[i].p =
            winddata->wind[i].p;
}

time_dummy.year = winddata->time.year;
time_dummy.month = winddata->time.month;
time_dummy.day = winddata->time.day;
time_dummy.hour = winddata->time.hour;
```

```
        if (get_int(TRACE) > 0)
                winddata = get_next_element(winddata);
        else
                winddata = get_prev_element(winddata);

        while (!time_equal(time_dummy, winddata->time)) {
                time_step_forward(&time_dummy);
                state->data_diff += 1;
        }

        for (i = 0; i < state->station_max; i++) {
                state->wind_data[state->station_max + i].u =
                    winddata->wind[i].u;

                state->wind_data[state->station_max + i].v =
                    winddata->wind[i].v;

                state->wind_data[state->station_max + i].p =
                    winddata->wind[i].p;
        }

        if (get_int(TRACE) < 0) {
                winddata = winddata->next;
                while (winddata->wind == NULL)
                        winddata = winddata->next;
        }

        return winddata;
}

/* Iterieren bis zum naechsten Trajektorienaufpunkt */
void
iterate(struct state *state, struct winddata** winddata)
{
        int    j;

        /*
         * Anzahl der durchlaufenen Iterationen seit der letzten vollen
         * Zeitstunde
         */
```

```
int    iteration;

/* Windvektor der momentanen Berechnungsposition */
double u, v;

/*
 * Zeitverzatz in Stundenanteilen, seit der letzten vollen
 * Zeitstunde
 */
double hour_diff;

/* Momentane Berechnungsposition (kartesisch 3D) */
double X[3];

/*
 * Iterieren bis Iterationszahl fuer naechsten Aufpunkt erreicht
 * ist
 */
for (j = 0; j < get_int(IPERPOINT); j++) {

        /*
         * Berechnung der Iterationsanzahl seit letzter vollen
         * Zeitstunde
         */
        iteration = ((state->point - 1) * get_int(IPERPOINT) + j) %
            get_int(IPERH);

        /* Wenn naechste volle Zeitstunde erreicht ist ... */
        if (iteration == 0) {

                if (get_int(TRACE) > 0)
                        state->diff += 1;
                else
                        state->diff -= 1;

                /*
                 * Umkopieren des vorher "zukuenftigen"
                 * Stunden-Windfeldes (wind_current[0]) als
                 * "momentanes/vergangenes" Stunden-Windfeld
                 * (wind_curren[1])
```

```
         */
        copy_wind_current(state);

        /* Wenn das naechste Daten-Windfeld erreicht ist */
        if ((state->diff == state->data_diff) ||
            (state->diff == -1)) {

                /*
                 * Verschieben des Zeitfensters (wind_data)
                 * um state->delta_diff in Berechnungsrich-
                 * tung (lese neues Daten-Windfeld ein)
                 */
                *winddata = get_next_wind_data(state, *winddata);

                /*
                 * Ueberpruefen, ob die angegeben zeitliche
                 * Datenaufloesung mit vorhandeneer Datenauf-
                 * loesung uebereinstimmt
                 */
                check_resolution(get_int(RES),
                    state->data_diff);

                /*
                 * Setze den Zeitunterschied (state->diff)
                 * zum Daten-Windfeld auf 0
                 */
                if (get_int(TRACE) > 0)
                        state->diff = 0;
                else
                        state->diff = state->data_diff - 1;

        }

        /*
         * Generieren des naechsten "zukuenftigen"
         * Stunden-Windfeldes (wind_current[0])
         */
        wind_of_next_hour(state, *winddata);

}
```

```
                /*
                 * Berechnen des Stundenanteils seit der letzten vollen
                 * Zeitstunde (0 <= hour_diff < 1)
                 */
                hour_diff = (double)iteration / (double)get_int(IPERH);

                /*
                 * Umrechnen der momentanen Berechnungskoordinaten
                 * (state->lo, state->la) in kartesische 3D-Koordinaten (X)
                 */
                convert_geo_to_cartesian(state->lo[state->point],
                    state->la[state->point], X);

                /*
                 * Berechnen des interpolierten Windvektors fuer momentane
                 * Position (X)
                 */
                if (calculate_wind_vector(hour_diff, &u, &v, X, state) == 1) {
                        /*
                         * Wenn kein Windvektor berechnet werden konnte,
                         * breche Berechnung ab
                         */
                        state->point_max = state->point;
                        j = get_int(IPERPOINT);
                }

                /*
                 * Berechnung des raeumlichen Versatzes fuer diesen
                 * Interpolationsschritt
                 */
                state->lo[state->point] = state->lo[state->point] +
                    state->distance_per_step * u /
                    cos(state->la[state->point]);

                state->la[state->point] = state->la[state->point] +
                    state->distance_per_step * v;
        }
}
```

```
/* Initialisieren der Datenstruktur zum Speichern von
 * Windfeldern
 */
struct winddata*
new_winddata(void)
{
        struct winddata* winddata;

        winddata = malloc(sizeof(struct winddata));
        winddata->time.year = 0;
        winddata->time.month = 0;
        winddata->time.day = 0;
        winddata->time.hour = 0;
        winddata->wind = NULL;
        winddata->next = NULL;
        winddata->prev = NULL;

        return winddata;
}

/* Umrechnen auf geographischen Groessenbereich */
void
normalize_coords(struct state* state)
{
        double lo = rad2deg(state->lo[state->point]);
        double la = rad2deg(state->la[state->point]);

        /* Umrechnen der geografischen Laenge auf Wertebereich
         * -180..180
         */
        lo = (lo - ((int)lo / 360) * 360) / 180;
        if (fabs(lo) > 1.0)
                lo = (((int)lo * -1) - ((lo - (int)lo) * -1)) * 180;
        else
                lo *= 180;

        /* Umrechnen der geografischen Breite auf Wertebereich
         * -90..90
         */
        la = (la - ((int)la / 360) * 360) / 180;
```

```
        if (fabs(la) > 1.0)
                la = (((int)la * -1) - ((la - (int)la) * -1)) * 180;
        else
                la *= 180;

        la = (la - ((int)la / 180) * 180) / 90;
        if (fabs(la) > 1.0)
                la = ((((int)la * -1) - ((la - (int)la) * -1)) * 90) * -1;
        else
                la *= 90;

        /* Zurueckschreiben der umgerechneten Werte */
        state->lo[state->point] = deg2rad(lo);
        state->la[state->point] = deg2rad(la);
}

/*
 * Initialisieren aller Werte, die fuer die Berechnung
 * benoetigt werden
 */
void
prepare_calculate(struct state* state, struct winddata** winddata) {

        /* Umrechnung von externer Zeitzone in interne Zeitzone (GMT) */
        convert_timezone(state);

        /*
         * Einlesen der Stationsinformationen in die Datenstruktur
         * state->station_list
         */
        read_station_list(state);

        /*
         * Distanzberechnungsfaktor (distance_per_step) fuer
         * Berechnungsrichtung anpassen (rueckwaerts => negativ)
         */
        if (get_int(TRACE) > 0) {
                state->distance_per_step *= -1;
        }
        if (get_int(TRACE) == 0) {
```

```
                printf("Error: TRACE = 0!\n");
                exit (1);
        }

        /*
         * Sammeln der fuer die Berechnung benoetigten Winddaten
         * (Daten-Windfelder) in der temporaeren Sammeldatei
         */
        *winddata = new_winddata();
        read_wind_data(state, *winddata);

        /*
         * Einlesen der ersten fuer die Berechnung benoetigten 2
         * Daten-Windfelder in die Datenstruktur state->wind_data
         */
        *winddata = init_wind_data(state, *winddata);

        /* Ueberpruefen, ob die angegebenen Datensatzaufloesung korrekt ist */
        check_resolution(get_int(RES), state->data_diff);

        /*
         * Erzeugung des ersten interpolierten Stunden-Windfeldes
         * (state->wind_current) zur momentanen Berechnunszeitstunde aus den
         * Daten-Windfeldern
         */
        wind_of_next_hour(state, *winddata);
}

/* Ausgabe der berechneten Trajektorie in einer Datei */
void
print_output_file(const struct state* state)
{
        int j;
        char* filename;
        FILE*  fh;      /* Filehandle fuer Ausgabedatei*/

        /* Generieren des Namens der Trajektorienausgabedatei */
        filename = malloc(MAXLINE);
        if (generate_output_filename(filename, MAXLINE) >= MAXLINE) {
                printf("Linebuffer too small!\n");
```

```
                exit(1);
        }


        /* Ueberpruefen, ob Datei angelegt werden kann */
        if (!(fh = fopen(filename, "w"))) {
                printf("Couldn't write in file %s!\n", filename);
                exit(1);
        }

        /* Schreiben der Ausgabedatei */
        fprintf(fh, "YYYY=%4i | MM=%2i | DD=%2i | HH=%2i | ",
            get_int(YYYY), get_int(MM), get_int(DD), get_int(HH));
        fprintf(fh, "ZONEDIFF=%i | ZONENAME=%s\n",
            get_int(ZONEDIFF), get_string(ZONENAME));

        fprintf(fh, "LO=%8.4f | LA=%8.4f | IPERH=%i | IPERPOINT=%i | ",
            get_float(LO), get_float(LA),
            get_int(IPERH), get_int(IPERPOINT));
        fprintf(fh, "TRACE=%i\n", get_int(TRACE));

        fprintf(fh, "MINR=%i | MAXR=%i | STDDEVIATION=%6.3f | RES=%i | ",
            get_int(MINR), get_int(MAXR), get_float(STDDEVIATION),
            get_int(RES));
        fprintf(fh, "DATAUNIT=%i\n", get_int(DATAUNIT));

        fprintf(fh, "SPEED=%4.2f | ROT=%5.2f\n\n",
            get_float(SPEED), get_float(ROT));

        fprintf(fh, "Trajektorienpunkte: %i\n\n", (state->point));

        for (j = 0; j < state->point; j++) {
                fprintf(fh, "%11.10f;%11.10f\n", state->lo[j], state->la[j]);
        }

        /* Datei schliessen */
        fclose(fh);
        free(filename);
}
```

```
/*
 * Initialisieren und einlesen der Standardwerte und der gesetzten
 * Programmargumente aus der Programmumgebung
 */
void
read_env(struct param *param)
{
        char *s;

        for (/* */; param->name != NULL; param++) {
                if ((s = getenv(param->name)) == NULL)
                        s = param->u.s;

                switch (param->type) {
                case TYP_INT:
                        param->u.i = atoi(s);
                        printf("%s %d (%s)\n", param->name,
                            param->u.i, param->desc);
                        break;

                case TYP_FLOAT:
                        param->u.f = atof(s);
                        printf("%s %6.2f (%s)\n", param->name,
                            param->u.f, param->desc);
                        break;

                case TYP_STRING:
                        param->u.s = s;
                        printf("%s %s (%s)\n", param->name,
                            param->u.s, param->desc);
                        break;

                default:
                        printf("Internal error; bad parameter table!\n");
                        exit(1);
                }
        }
}

/*
```

```
 * Einlesen der Winddaten
 */
struct winddata*
read_file(struct state* state, char* name)
{
        char  line[MAXLINE];
        int   i, c, A, B, C;
        char* tok;
        FILE* fh;
        struct winddata* winddata_pnt;
        double wind_speed, wind_direction;

        winddata_pnt = new_winddata();

        /* Ueberpruefen, ob Datei existiert */
        if (!(fh = fopen(name, "r"))) {
                printf("Couldn't open file %s!\n", name);
                exit(1);
        }

        /* Naechsten Zeitangabeblock suchen */
        while (1) {

                i = -1;

                /* Einlesen einer Zeile */
                do {

                        c = fgetc(fh);

                        /* Wenn Dateiende */
                        if (c == EOF)
                                break;

                        line[++i] = (char)c;

                        /* Wenn Datenbuffer zu klein ist */
                        if (i == (MAXLINE - 1)) {
                                printf("Linebuffer too small!\n");
                                exit(1);
```

```
                }

                /* Carriage Return ignorieren */
                if (line[i] == '\r') {
                        i--;
                }

        } while (line[i] != '\n');

        /* Wenn Dateiende */
        if (c == EOF)
                break;

        line[++i] = '\0';

        /* Wenn eingelesene Zeile nicht Winddatenzeile ist */
        if (line[0] != ' ') {

                if (line[0] != '*') {
                        /* Zeichenkette terminieren */

                        if (get_int(TRACE) < 0) {
                                winddata_pnt->next = new_winddata();
                                winddata_pnt->next->prev =
                                    winddata_pnt;
                                winddata_pnt = winddata_pnt->next;
                        }
                        else {
                                winddata_pnt->prev = new_winddata();
                                winddata_pnt->prev->next =
                                    winddata_pnt;
                                winddata_pnt = winddata_pnt->prev;
                        }

                        /* Zuweisen der Zeitangabe */
                        if ((tok = strtok(line, " ")) == NULL) {
                                printf("Syntax error in wind data\n");
                                exit(1);
                        }
```

```
                for (i = 0; i < 4; i++) {
                        switch(i) {
                        case 0:
                                winddata_pnt->time.year =
                                    atoi(tok);
                                break;
                        case 1:
                                winddata_pnt->time.month =
                                    atoi(tok);
                                break;
                        case 2:
                                winddata_pnt->time.day =
                                    atoi(tok);
                                break;
                        case 3:
                                winddata_pnt->time.hour =
                                    atoi(tok);
                                break;
                        }
                        if (((tok = strtok(NULL, " ")) ==
                                NULL) && (i < 3)) {
                        printf("Syntax error in winddata\n");
                                exit(1);
                        }
                }
        }
}
else {
        if (winddata_pnt->wind == NULL) {
                winddata_pnt->wind =
                    calloc(state->station_max,
                        sizeof(struct wind));
                for (i = 0; i < state->station_max; i++) {
                        winddata_pnt->wind[i].u = 0;
                        winddata_pnt->wind[i].v = 0;
                        winddata_pnt->wind[i].p = 0;
                }
        }

        /* Winddaten in aktuelles Element einlesen und
```

```
   mit Stationsliste vergleichen */
/* Einlesen der Winddaten */
if ((tok = strtok(line, " ")) == NULL) {
        printf("Syntax error in wind data\n");
        exit(1);
}

for (i = 0; i < 3; i++) {
        switch(i) {
        case 0:
                A = atoi(tok);
                break;
        case 1:
                B = atoi(tok);
                break;
        case 2:
                C = atoi(tok);
                break;
        }
        if (((tok = strtok(NULL, " ")) == NULL) &&
            (i < 2)) {
                printf("Syntax error in wind data\n");
                exit(1);
        }
}

for (i = 0; i < state->station_max; i++) {

        /*
         * Wenn Stationsnummer in der Stationsliste
         * enhalten ist -> Winddaten speichern
         */
        if (state->station_list[i].nr == A) {
                wind_direction = B;
                wind_speed = C;

                /*
                 * Wenn Windgeschwindigkeiten in
                 * Knoten angegeben sind -> muss in
                 * m\s umgerechnet werden
```

```
 */
if (state->station_list[i].unit
    == 2) {
        wind_speed = wind_speed *
            MILE / 3.6;
}

/*
 * Anpassung der Bodenwindge-
 * schwindigkeit auf mittlere
 * Transportgeschwindigkeit in der
 * Mischungsschicht durch Windge-
 * schwindigkeitsfaktor
 */
wind_speed = wind_speed *
    get_float(SPEED);

/*
 * Hier wird (wenn rotation_flag
 * == 1) die Drehung des Bodenwindes
 * dynamisches auf den Laengengrad
 * angepasst. Winddrehung is
 * abhaengig vom Laengengrad der
 * Wetterstation
 *
 * -> Umrechnen der Winddrehung der
 * Startposition (Startparameter) auf
 * Rotation an der Wetterstations-
 * position
 */

/*
 * Windrichtungskorrektur
 */
wind_direction += get_float(ROT);

/* Grad -> RAD */
wind_direction =
    deg2rad(wind_direction);
```

```
                                        /*
                                         * Speichern der eingelesenen
                                         * korrigierten Werte in der
                                         * Winddatenstruktur
                                         */
                                        winddata_pnt->wind[i].u =
                                            wind_speed * sin(wind_direction);

                                        winddata_pnt->wind[i].v =
                                            wind_speed * cos(wind_direction);

                                        winddata_pnt->wind[i].p =
                                            1;
                                }
                        }
                }
        }

        fclose(fh);
        while (winddata_pnt->prev != NULL) {
                winddata_pnt = winddata_pnt->prev;
        }

        return winddata_pnt;
}

/*
 * Speichern der Stationsinformationen aus der Stationsinformations-
 * datei in der internen Datenstruktur state->station_list
 */
void
read_station_list(struct state* state)
{
        int    i, j;
        char   line[MAXLINE];
        double la_tmp, lo_tmp;
        FILE*  fh;
        char*  tok;

        la_tmp = lo_tmp = 0.0;
```

```
/* Ueberpruefen, ob Datei existiert */
if (!(fh = fopen(get_string(STATION), "r"))) {
        printf("Couldn't open file %s!\n", get_string(STATION));
        exit(1);
}

/* Einlesen der benoetigten Stationsparameter */
for (i = 0; fgets(line, MAXLINE, fh); i++) {

        if ((tok = strtok(line, " ")) == NULL) {
                printf("Syntax error in file %s\n",
                    get_string(STATION));
                exit(1);
        }

        for (j = 0; j < 5; j++) {
                /*XXX Testen, ob parameter ein integer ist */
                switch(j) {
                case 0:
                        state->station_list[i].nr =
                            atoi(tok);
                        break;
                case 1:
                        la_tmp = atoi(tok);
                        break;
                case 2:
                        lo_tmp = atoi(tok);
                        break;
                case 4:
                        switch(get_int(DATAUNIT)) {
                        case 0:
                                state->station_list[i].unit =
                                    2;
                                break;

                        case 1:
                                state->station_list[i].unit =
                                    1;
                                break;
```

```
                case 2:
                        state->station_list[i].unit =
                            atoi(tok);
                        break;
                default:
                        state->station_list[i].unit =
                            2;
                        break;
                }
                break;
        }
        if (((tok = strtok(NULL, " ")) == NULL) &&
            (j < 4)){
                printf("Syntax error in file %s\n",
                    get_string(STATION));
                exit(1);
        }
}

/*
 * Wenn Windeinheit aus der Stationsliste eingelesen
 * wurde -> range checking
 */
if (get_int(DATAUNIT) == 2) {
        if ((state->station_list[i].unit < 1) ||
            (state->station_list[i].unit > 2)) {
                printf("Unknown value for unit!\n");
                exit(1);
        }
}

/* Umrechnen der Koordinaten (Grad, Minuten) in Rad */
la_tmp = deg2rad((int)(la_tmp / 100) +
    (la_tmp - 100 * (int)(la_tmp / 100)) / 60);
lo_tmp = deg2rad((int)(lo_tmp / 100) +
    (lo_tmp - 100 * (int)(lo_tmp / 100)) / 60);

/*
 * Umrechnen der geographischen Koordinaten in
```

```
                 * kartesische Koordinaten
                 */
                state->station_list[i].X[0] = cos(la_tmp) * cos(lo_tmp);
                state->station_list[i].X[1] = cos(la_tmp) * sin(lo_tmp);
                state->station_list[i].X[2] = sin(la_tmp);
        }

        /* Datei schliessen */
        fclose(fh);
}

/* Erstellen einer Liste von benoetigten Winddatensaetzen und Einlesen
 * der Windfelddaten
 */
void
read_wind_data(struct state* state, struct winddata* winddata)
{
        int           i, j, k;
        char          name[MAXLINE];
        struct date* time;
        int           res = get_int(RES);
        struct winddata* winddata_pnt;

        winddata_pnt = winddata;
        /*
         * Wenn Aufloesung der Wetterdaten (Zeitabstand) nicht festgelegt
         * wurde, nimm als Maximalabstand RESMAX an
         */
        if (res == 0) {
                res = RESMAX;
        }

        /*
         * Errechnen der Groesse der Zeitspeicherstruktur fuer alle
         * Zeitstunden des Berechnungszeitraums
         */
        k = abs(get_int(TRACE)) + 2 * res;

        /* Reservieren des benoetigten Speichers */
        time = (struct date *)calloc(k, sizeof(struct date));
```

```
j = 0;

/* Erstellen einer Liste von beoetigten Tagesdatensaetzen */
time_copy(state->time, time[j]);

/*
 * Je nach Berechnungsrichtung vor- bzw. zurueckgehen um die maximale
 * zeitliche Differenz zweier Datensaetze, um ein Fehlen von
 * Datensaetzen  bei der spaeteren zeitlichen Interpolation zu
 * verhindern
 */
for (i = 0; i < res; i++) {

        /* Wenn Rueckwaertstrajektorie */
        if (get_int(TRACE) < 0) {
                time_step_forward(&time[j]);
        }

        /* Wenn Vorwaertstrajektorie */
        else {
                time_step_backward(&time[j]);
        }
}

/*
 * Kopieren der "korrigierten" Anfangszeit in das zweite Zeitelement
 * fuer die weitere Namensgenerierung
 */
j++;
time_copy(time[j-1], time[j]);

/*
 * Stundenweises vorsetzen in Berechnungsrichtung, bis Verfolgungs-
 * dauer vollstaendig durchlaufen wurde. Immer wenn ein neuer Tag
 * erreicht wird, wird in ein neues Zeitelement gewechselt
 * -> benoetigte Tagesdaten der einzulesenden Datensaetze sind in
 * der Liste (time) gespeichert
 */
for (i = 0; i < k; i++) {
```

```
        /* Wenn Rueckwaertstrajektorie */
        if (get_int(TRACE) < 0) {
                time_step_backward(&time[j]);
        }

        /* Wenn Vorwaertstrajektorie */
        else {
                time_step_forward(&time[j]);
        }

        /* Wenn naechster Tag erreicht, dann in Liste speichern */
        if (time[j].day != time[j-1].day) {
                j++;
                time_copy(time[j-1], time[j]);
        }
}

/* j ist Anzahl der generierten Tagesangaben */
for (i = 0; i < j; i++) {

        /*
         * Namensgenerierung einzulesender Winddatendateien aus
         * generierter Zeitenliste
         */

        /* 1999 -> 99 */
        time[i].year = time[i].year - (time[i].year / 100 * 100);

        if (snprintf(name, MAXLINE, "%sb%02i%02i%02i.new",
                get_string(METEO), time[i].year, time[i].month,
                time[i].day) >= MAXLINE) {
                printf("Linebuffer too small!\n");
                exit(1);
        }

        /*Ausgabe des generierten Dateinamens */
        printf("%s\n", name);

        /* Daten aus naechster Datei einlesen und anhaengen */
```

```
                winddata_pnt->next = read_file(state, name);

                if (get_int(TRACE) < 0) {
                        winddata_pnt->next = winddata_pnt->next->next;
                        if (winddata_pnt->prev != NULL)
                                winddata_pnt->next->prev = winddata_pnt;
                }
                else {
                        winddata_pnt->next->prev = winddata_pnt->prev;
                        if (winddata_pnt->prev != NULL)
                                winddata_pnt->prev->next = winddata_pnt->next;
                }

                while (winddata_pnt->next != NULL) {
                        winddata_pnt = winddata_pnt->next;
                }
        }

        /* Speicher freigeben */
        free(time);
}

/*
 * Freigeben der reservierten Speicherbereiche und schliessen der Sammeldatei
 */
void
reset_state(struct state* state)
{
        free(state->lo);
        free(state->la);
        free(state->wind_data);
        free(state->wind_current);
        free(state->station_list);
}

/*
 * Wenn Wetterstation in Reichweite sind, kann Standardabweichung
 * berechnet werden.
 *
 * >> Standardabweichung = wurzel(QSumme / Anzahl) <<
```

```
 *
 */
void
std_deviation(double amount, double* u_stddev,
    double* v_stddev)
{
        if (amount > 0) {
                *u_stddev = sqrt(*u_stddev / amount);
                *v_stddev = sqrt(*v_stddev / amount);
        }
}

/* Zuruecksetzen der uebergebenen Zeitstruktur um eine Zeitstunde */
void
time_step_backward(struct date* time) {

        if (time->hour != 0) {
                time->hour = time->hour - 1;
        }
        else {

                if (time->day != 1) {
                        time->hour = 23;
                        time->day = time->day - 1;
                }
                else {
                        time->month = time->month - 1;
                        time->hour = 23;

                        if (time->month == 0) {

                                time->month = 12;
                                time->year = time->year - 1;
                        }

                        if ((time->month == 1) ||
                            (time->month == 3) ||
                            (time->month == 5) ||
                            (time->month == 7) ||
                            (time->month == 8) ||
```

```
                        (time->month == 10) ||
                        (time->month == 12)) {
                            time->day = 31;
                    }
                    else {
                            if ((time->month == 4) ||
                                (time->month == 6) ||
                                (time->month == 9) ||
                                (time->month == 11)) {
                                    time->day = 30;
                            }
                            else {

                                    if (((time->year % 4) == 0) &&
                                        (((time->year % 100) != 0) ||
                                         ((time->year % 400) == 0))) {
                                            time->day = 29;
                                    }
                                    else {
                                            time->day = 28;
                                    }
                            }
                    }
            }
    }
}

/* Vorsetzten der uebergebenen Zeitstruktur um eine Zeitstunde */
void
time_step_forward(struct date* time) {

    int MaxDay;

    if (time->hour != 23) {
            time->hour = time->hour + 1;
    }
    else {
            if ((time->month == 1) ||
                (time->month == 3) ||
                (time->month == 5) ||
```

```
    (time->month == 7) ||
    (time->month == 8) ||
    (time->month == 10) ||
    (time->month == 12)) {
        MaxDay = 31;
}
else {
        if ((time->month == 4) ||
            (time->month == 6) ||
            (time->month == 9) ||
            (time->month == 11)) {
                MaxDay = 30;
        }
        else {

                if (((time->year % 4) == 0) &&
                    (((time->year % 100) != 0) ||
                     ((time->year % 400) == 0))) {
                        MaxDay = 29;
                }
                else {
                        MaxDay = 28;
                }
        }
}

if (time->day < MaxDay) {
        time->day = time->day + 1;
        time->hour = 0;
}
else {
        time->month = time->month + 1;
        time->day = 1;
        time->hour = 0;

        if (time->month > 12) {
                time->month = 1;
                time->year = time->year + 1;
        }
}
```

```
        }
}

/*
 * Zeitliches Interpolieren eines neuen Stundenwindfeldes fuer wind_current
 * aus den beiden eingelesenen Daten-Windfeldern in wind_data. Wenn zur
 * betrachteten Zeitstunde keine Daten vorliegen, wird aus den Daten
 * vor und nach der betrachteten Zeitstunde zeitlich gewichtet interpoliert.
 *
 * wind_data[0]|-----------------------------|wind_data[1]
 *             0          |                  data_diff=3
 *                      diff=1.1
 */
void
wind_of_next_hour(struct state* state, struct winddata* winddata)
{
        int j;
        struct winddata* winddata_pnt;

        winddata_pnt = winddata;
        /*
         * Wenn zwischen eingelesenen Datenbloecken keine Zeitdifferenz
         * besteht
         */
        if (state->data_diff == 0) {
                printf("Error: no time difference between wind data!\n");
                exit (1);
        }

        for (j = 0; j < state->station_max; j++) {

                /*
                 * Wenn Berechnungszeit (diff) mit Datenblockzeit von
                 * wind_data zusammenfaellt
                 */
                if (state->diff == 0) {

                        /* Wenn Winddaten vorhanden sind */
                        if (state->wind_data[j].p != 0) {
```

```
                state->wind_current[j].u =
                    state->wind_data[j].u;

                state->wind_current[j].v =
                    state->wind_data[j].v;

                state->wind_current[j].p = 1;

        }

        /* Wenn keine Winddaten vorhanden sind */
        else {

                state->wind_current[j].u = 0;
                state->wind_current[j].v = 0;
                state->wind_current[j].p = 0;
        }
}

/*
 * Wenn Berechnungszeit (diff) zwischen den Zeiten der Daten-
 * bloecke von wind_data lieg  -> interpolieren
 */
else {
        /*
         * Wenn Daten in beiden Daten-Windfeldern vorhanden
         * sind
         */

        if ((state->wind_data[j].p != 0) &&
            (state->wind_data[state->station_max + j].p
                != 0)) {
                state->wind_current[j].u =
                    state->wind_data[j].u *
                    (double)(state->data_diff -
                        state->diff) /
                    (double)state->data_diff +
                    state->wind_data[state->
                        station_max + j].u *
                    (double)state->diff /
```

```
                                    (double)state->data_diff;

                                state->wind_current[j].v =
                                    state->wind_data[j].v *
                                    (double)(state->data_diff -
                                        state->diff) /
                                    (double)state->data_diff +
                                    state->wind_data[state->
                                        station_max + j].v *
                                    (double)state->diff /
                                    (double)state->data_diff;
                                state->wind_current[j].p = 1;
                        }

                        /*
                         * Wenn nicht in beiden Daten-Windfeldern Daten
                         * vorhanden sind
                         */
                        else {

                                state->wind_current[j].u = 0;
                                state->wind_current[j].v = 0;
                                state->wind_current[j].p = 0;
                        }
                }
        }
}

/* Werte mit zu grosser Abweichung rauswerfen */
void
z_transformation_check(struct wind* wind_in_range, double u_average,
    double v_average, double u_stddev, double v_stddev,
    double stddev, int i)
{
        /* Wenn Daten vorhanden sind */
        if (wind_in_range[i].p != 0) {

                /*
                 * Z-Transformation
                 *
```

```
                * >> z = abs(xi - Mittelwert) / Standardabweichung <<
                *
                */

               /* Wennn Abweichung groesser als gefordert */
               if (((fabs(wind_in_range[i].u - u_average)
                        / u_stddev) > stddev) ||
                   ((fabs(wind_in_range[i].v - v_average)
                       / v_stddev) > stddev)) {

                       /* Daten i aus der Berechnung nehmen */
                       wind_in_range[i].p = 0;
               }
       }
}
```