Appendix D

Particle swarm optimization software

D.1 Users guide

D.2 OEPSA

In this appendix the OEPSA (Optimization Environment for Particle Swarm Algorithms) software is presented.

D.2.1 Overview

During the investigation into the particle swarm paradigm, an environment was developed which had to comply with the following requirements:

(a) Portability - The software had to be developed with multiple platforms and operating systems as target environments in mind. ANSI C was the language of choice because of it's common usage and ease of portability.

(b) Visualization - In order to be able to study particle behavior during a search, some means of realtime visualization of the swarm was required. The standard Linux graphics library, svgalib, was used for this purpose.

(c) Batch processing - For the benchmarking of multiple problems with varying algorithm parameters an efficient batch processor was needed.

(d) Postprocessing of results - In order for the results of test runs to be reported (with minimal extra effort) in tables and graphs, the results had to be processed and stored in a structured manner.

(e) Ability to interface with external programs - The software also had to be able to interact with extenal programs such as the finite element solver used in optimizing the truss structures. This interaction typically entails the passing of design variables to the solver and receiving a fitness value and constraint values from the solver after analysis.

(f) Script file - In order to change the algorithm parameters without recompiling code, and to ease batch processing of large number of problems with varying parameters, a script file reader which accepts values specified in an external human readable text file was implemented.

All of the numerous algorithm modifications to the particle swarm detailed in this thesis have been incorporated into the program, and can be selected by changing the appropriate setting in the script file (see Section D.3.1).

D.3 Visual interface

The visual interface is one of the most important aspects of the software since it allows the user to observe particle behavior. During parameter sensitivity studies it is also useful to observe any behavioral changes brought about by parameter variations.

The visual interface only allows two user defined dimensions to be displayed at any time.

The screen is divided into two sections, the left section displays the particle swarm in real-time moving through the problem space. The solution point is indicated by a light blue circle and particles are indicated as white dots. Individual particle's best remembered positions $\boldsymbol{p}_k^i$ are represented by yellow dots and the overall swarm best coordinates $\boldsymbol{p}_g^i$ by a red circle with the corresponding fitness value displayed next to it.

The right section of the screen display continually updates information in the best fitness coodinates and varying parameter values. The top right section displays the current coordinates of the swarm best value $\boldsymbol{p}_g^i$ together with the swarm best fitness value g_{best}. The center right section displays the solution coordinates and optimim fitness value, and the inertia weight w and maximum allowable velocity v^{max}. The bottom section is taken up by a graph which displays the fitness history as the search progresses.

D.3.1 Input script file

An example script file with typical values for parameters is presented below, followed by explanations of the various applicable settings. On/off implies that either a 1 or 0 should be used respectively.

`first_problem = 1`	First problem in batch set.
`last_problem = 16`	Last problem in batch set.
`problem_repeats = 50`	Number of problem repetitions.
`use_display = 0`	Realtime visualization (on/off).

`display_refresh = 20`	Number of function evaluations between display updates.
`history_output = 0`	Write fitness history to file (on/off).
`output_filename = result`	Batch result output filename.
`respawn_in_feasable = 0`	Re-initialize infeasable particle until feasable (on/off).
`modification = 3`	Type of PSOA to use: 0 = Standard pso (w = 1), 1 = Linearly decreasing momentum (w), 2 = Constriction factor (K), 3 = Dynamic inertia + velocity.
`bounds_method = 3`	Type of bounds to enforce: 1 = No bounds, 2 = Respawn, 3 = Bounce.
`stopping_method = 1`	Type of stopping criterion: 1 = Error minimization, 2 = Inertia stopping method, 3 = No-improve stopping method, 4 = Maximum velocity.
`tolerance_method = 1`	If `stopping_method = 1`, specify tolerance method to be: 1 = stop when value comes within absolute tolerance, 2 = stop when value comes within percentage value of solution,
`tolerance = 0.001`	Specify either an absolute or percentile tolerance (see above).
`inertia_stop = 0.01`	If `stopping_method = 2`, the value of inertia weight to be stopped upon.
`no_improvement_stop = 1000`	If `stopping_method = 3`, the number of Nfe with no improvement in $f(\boldsymbol{p}_k^g)$,
`no_improvement_tolerance = 0.01`	within `no_improvement_tolerance` before algorithm is stopped.
`velocity_stop = 0.01`	Absolute value to be used as stopping criteria if `stopping_method = 4`.
`limit_max_velocity = 1`	Maximum velocity enforcement on/off.
`amount_of_particles = 20`	Swarm population.
`max_function_evaluations = 30000`	Maximum allowed function evaluations.
`allowable_infeasability = 0.02`	Normalized infeasability value which is acceptable to user.
`velocity_fraction = 1`	Initial fraction of bounds to be taken

	as maximum step size or "velocity".
`initial_inertia = 1.0`	Linear inertia reduction initial value.
`final_inertia = 0.5`	Linear inertia reduction final value.
`inertia_change_end = 4000`	N_{fe} at which to stop linear inertia reduction.
`wait_for_improvement = 10`	Number of function evaluations to wait before reducing velocity and inertia.
`inertia_reduction_fraction = 0.01`	Absolute increment by which to reduce inertia.
`velocity_reduction_fraction = 0.01`	Absolute increment by which to reduce velocity.
`initial_lambda = 1000`	λ penalty initial value
`final_lambda = 1000000`	λ penalty final value
`lambda_change_end = 4000`	Number of function evaluations N_{fe} where `final_lambda` is reached
`personal_scaler = 2.0`	Cognitive value c_1
`group_scaler = 2.0`	Social value c_2

D.4 Source code

In this section selected fragments of the software developed are detailed and discussed. For the sake of brevity only sections directly responsible for the PSOA's workings are presented, and the remainder (visual interface, batch processor, output postprocessor etc.) are ignored.

D.4.1 Particle swarm initialization

The following functions are responsible for initializing swarm positions $\boldsymbol{x}_0^i$, initial velocities $\boldsymbol{v}_0^i$, and inertia values w_0. Function values are calculated for particle positions $\boldsymbol{x}_0^i$. The best swarm value f_{best}^g is selected from these, with $\boldsymbol{p}_0^i$ set equal to the appropriate coordinate. The maximum allowable velocity $\boldsymbol{v}^{max}$ is also calculated.

```
void initialize_particle_positions(void)
   {

   /* this function initializes all the particle positions */
   int i,j,test;
   int randominteger;
   float randomfloat;
```

```
    float range[MAXDIMENSIONS];

    for (i = 0; i < problem_data.problem_dimensions; ++i)
        range[i] = problem_data.position_upperbound[i] -
problem_data.position_lowerbound[i];

    if (problem_data.respawn_in_feasable == 0)
        {
        for (i = 0; i < problem_data.amount_of_particles; ++i)
            {
            for (j = 0; j < problem_data.problem_dimensions; ++j)
                {
                randominteger = rand();
                randomfloat = (float) randominteger/RAND_MAX;
                particle[i].coordinates[j] = randomfloat*range[j] +
problem_data.position_lowerbound[j];
                } /* for j */
            } /* for i */
        }
    else
        {
        printf(''\nRespawning particles into feasable region'');
        for (i = 0; i < problem_data.amount_of_particles; ++i)
            {
            do
                {
                test = 0;

                for (j = 0; j < problem_data.problem_dimensions; ++j)
                    {
                    randominteger = rand();
                    randomfloat = (float) randominteger/RAND_MAX;
                    particle[i].coordinates[j] = randomfloat*range[j] +
problem_data.position_lowerbound[j];
                    }

                /* test if initialized position is within constraints */
                function_result =
evaluate_problem(problem_data.problem_number,problem_data.problem_dimensions,
particle[i].coordinates);

                if (function_result.normalized_infeasability >
problem_data.allowable_infeasability)
                    {
                    test = 1;
                    printf(''\nRe-initializing particle %i which violates constraints by %f'', i,
function_result.normalized_infeasability);
```

```
                }

            } while (test == 1);
            printf(''\nParticle %i normalized infeasability = %f'', i,
function_result.normalized_infeasability);
            particle[i].constraints_violated = 0;
            } /* for i */
        } /* if else */
    } /* initialize_particle_positions */


void initialize_particle_velocities(void)
    {

    /* this function initializes all the particle velocities */
    int i,j;
    int randominteger;
    float randomfloat;
    float velocity_range[MAXDIMENSIONS];

    /* Init velocity bounds */
    for (j = 0; j < problem_data.problem_dimensions; ++j)
        {
        velocity_range[j] = velocity_fraction *
(problem_data.position_upperbound[j]-problem_data.position_lowerbound[j]);
        }

    for (i = 0; i < problem_data.amount_of_particles; i++)
        {
        for (j = 0; j < problem_data.problem_dimensions; ++j)
            {
            randominteger = rand();
            randomfloat = (float)randominteger / RAND_MAX;
            particle[i].velocity[j] = randomfloat * velocity_range[j] -
(velocity_range[j] * 0.5);
            } /* for j */
        } /* for i */
    } /* initialize_particle velocities */


void initialize_particle_inertia_values(void)
    {
    /* this function sets up all the particle inertia values */
    int i,j;
    int randominteger;
    float randomfloat;
    float range;

    /* Init all at specified initial_inertia value */
    for (i = 0; i < problem_data.amount_of_particles;i++)
        particle[i].inertia = problem_data.initial_inertia;
    } /* initialize_particle_inertia_values */


void initialize_particle_function_values(void)
    {

    /* This function initializes all the particle function values */
    int i,j;
    int infeasable_particles = 0;

    for (i = 0; i < problem_data.amount_of_particles; ++i)
        {
        function_result =
evaluate_problem(problem_data.problem_number,problem_data.problem_dimensions,
particle[i].coordinates);

        particle[i].current_value = function_result.value;
        particle[i].best_value = function_result.value;
        particle[i].normalized_infeasability = function_result.normalized_infeasability;

        for (j = 0; j < problem_data.problem_dimensions; ++j)
            {
            particle[i].best_value_coordinates[j] = particle[i].coordinates[j];
            }

        /* feasability check */
        if (particle[i].normalized_infeasability >= problem_data.allowable_infeasability)
            {
            particle[i].constraints_violated = 1;
            printf(''\nConstraints violated =
%f'',function_result.normalized_infeasability);
            ++infeasable_particles;
            }
        else
            {
            particle[i].constraints_violated = 0;
            }

        /* Initialize best swarm value (get lowest value of the particle[].best.value
        array and store it and its coordinates in the comm_data struct)         */

        /* initialize swarm best if NI < NI_tol, and search for better one */
        if (particle[i].constraints_violated == 0)
           {
           comm_data.swarm_best_value = particle[i].best_value;  /* for temporary
comparison */
           comm_data.swarm_best_normalized_infeasability =
particle[i].normalized_infeasability;
           }
        } /* for particle i */

        printf(''\nParticles outside feasable region =
%i/%i'',infeasable_particles,problem_data.amount_of_particles);

    /* search for best particle fitness value */
    for (i = 0; i < problem_data.amount_of_particles; ++i)
       {
       if ( (particle[i].best_value < comm_data.swarm_best_value) &&
(particle[i].constraints_violated == 0) )
          {
          comm_data.swarm_best_value = particle[i].best_value;
          comm_data.swarm_best_normalized_infeasability =
particle[i].normalized_infeasability;

          for (j = 0; j < problem_data.problem_dimensions; ++j)
             {
             comm_data.swarm_best_coordinates[j] = particle[i].best_value_coordinates[j];
             } /* for j */
          } /* for i */
       }  /* if */
    }  /* initialize_particle_function_values */


void initialize_swarm_max_velocity(void)
   {

   /* This function initializes the swarm maximum velocity */
   double range;
   int j;

   for (j = 0; j < problem_data.problem_dimensions; ++j)
      {
      comm_data.swarm_max_velocity[j] = velocity_fraction *
(problem_data.position_upperbound[j]-problem_data.position_lowerbound[j]);
      }
   } /* initialize_swarm
```

D.4.2 Search

The following code implements the velocity (2.2) and position (2.1) rules. Allowance for the limitation of maximum velocity in the velocity update function and bounds implementation in the position update function are also implemented. Additional functions required during the search are detailed, which update particle inertia values, maximum allowed velocities and fitness values.

```
void update_particle_velocity(int i)
    {

    /* This function is used to update a single particle velocity during the search */
    int j;
    int randominteger1;
    double randomdouble1;
    int randominteger2;
    double randomdouble2;

    double range;
    double personal_scaler    = problem_data.personal_scaler;
    double group_scaler       = problem_data.group_scaler;
    double old;
    double personal;
    double group;
    double varphi;

    for (j = 0; j < problem_data.problem_dimensions; ++j)
        {
        randominteger1 = rand();
        randomdouble1 = (double) randominteger1/RAND_MAX;
        randominteger2 = rand();
        randomdouble2 = (double) randominteger2/RAND_MAX;

        old = particle[i].velocity[j];

        /* social pressure */
        if ( (particle[i].constraints_violated == 1) &&
 (problem_data.use_social_pressure == 1) )
            {
            personal = 0;
            }
        else
            {
            personal =
randomdouble1*(particle[i].best_value_coordinates[j]-particle[i].coordinates[j]);
            }

        group =
randomdouble2*(comm_data.swarm_best_coordinates[j]-particle[i].coordinates[j]);
        if (problem_data.modification != 2)
            {
            particle[i].velocity[j] = particle[i].inertia*old + personal_scaler*personal
+ group_scaler*group;
            }
        else /* constriction factor */
            {
            varphi = personal_scaler + group_scaler;
            constriction_factor = 2/(fabs(2 - varphi - sqrt(SQUARE(varphi) - 4*varphi)
));
            particle[i].velocity[j] = constriction_factor*(old + personal_scaler*personal
+ group_scaler*group);
            }

        /* max speed check */
        if (problem_data.limit_max_velocity == 1)
            {
            if (particle[i].velocity[j] > comm_data.swarm_max_velocity[j])
                particle[i].velocity[j] = comm_data.swarm_max_velocity[j];
            if (particle[i].velocity[j] < -comm_data.swarm_max_velocity[j])
                particle[i].velocity[j] = -comm_data.swarm_max_velocity[j];
            } /* if */
        } /* for j */
    } /* update_particle_velocity */


void update_particle_position(int i)
   {

   /* This function is used to update a particle position during the search */
   int j,k;
   int randominteger;
   float randomfloat;
   double range[MAXDIMENSIONS];
   double temp;
   double temp2;
   double upperbound, lowerbound;
   switch(problem_data.bounds_method)
      {
      case 1:
      /* no bounds method */

      for (j = 0; j < MAXDIMENSIONS; ++j)
         {
         particle[i].coordinates[j] = particle[i].coordinates[j] +
particle[i].velocity[j];
         }

      break;


      case 2:
      /* random respawn position if out of bounds method */

      for (j = 0; j < problem_data.problem_dimensions; ++j)
         {
         particle[i].coordinates[j] = particle[i].coordinates[j] +
particle[i].velocity[j];

         if ((particle[i].coordinates[j] < problem_data.position_lowerbound[j]) ||
(particle[i].coordinates[j]>problem_data.position_upperbound[j]))
            {
            randominteger = rand();
            randomfloat = (float) randominteger/RAND_MAX;
            particle[i].coordinates[j] = randomfloat*range[j] +
problem_data.position_lowerbound[j];
            } /* if */
         } /*for j */
      break;
      case 3:
      /* bounce off boundaries method (velocity reversal/sign changeover)*/

      for (j = 0; j < problem_data.problem_dimensions; ++j)
         {
         particle[i].coordinates[j] = particle[i].coordinates[j] +
particle[i].velocity[j];
         if (particle[i].coordinates[j] < problem_data.position_lowerbound[j])
            {
            particle[i].velocity[j] = fabs(particle[i].velocity[j]);
            particle[i].coordinates[j] = particle[i].coordinates[j] +
particle[i].velocity[j];
            } /* if */
```

```
        else if (particle[i].coordinates[j] > problem_data.position_upperbound[j])
            {
            particle[i].velocity[j] = -fabs(particle[i].velocity[j]);
            particle[i].coordinates[j] = particle[i].coordinates[j] +
particle[i].velocity[j];
            } /* else */
        } /* for j */
      break;
      } /* case */
  } /* update_particle_position */


void update_particle_function_value(int i)
  {

   /* This function updates a particle function value during search */
   function_result =
evaluate_problem(problem_data.problem_number,problem_data.problem_dimensions,
particle[i].coordinates);
   particle[i].current_value = function_result.value;
   particle[i].normalized_infeasability = function_result.normalized_infeasability;


   /* Infeasability check */
   if (particle[i].normalized_infeasability > problem_data.allowable_infeasability)
      {
      particle[i].constraints_violated = 1;
      printf(''\nconstraints violated = %f'',function_result.normalized_infeasability);
      } /* if */
   else
      {
      particle[i].constraints_violated = 0;
      } /* else */
  } /* update_particle_function_value */


void update_inertia_max_velocity(void)
  {

   /* Function to alter modification parameters, eg. article inertia, velocity etc */
   double inertia_range;
   double velocity_range;
   double temp;
   double fraction;
   int i,j;

   inertia_range = problem_data.initial_inertia - problem_data.final_inertia;
```

```
    velocity_range = problem_data.initial_velocity - problem_data.final_velocity;

    switch(problem_data.modification)
        {
        case 0:
        /*standard pso */

        for (i = 0; i < problem_data.amount_of_particles; ++i)
            {
            particle[i].inertia = 1;
            }
        break;

        case 1:
        /* linearly decreasing inertia */

        if (comm_data.function_evaluations < problem_data.inertia_change_end)
            {
            temp = (double)(problem_data.inertia_change_end -
comm_data.function_evaluations);
            temp = (double)(temp / problem_data.inertia_change_end);
            for (i = 0; i < problem_data.amount_of_particles; ++i)
                {
                particle[i].inertia = problem_data.final_inertia + inertia_range*temp;
                } /* for i */
            } /* if */
         else
            {
            for (i = 0; i < problem_data.amount_of_particles; ++i)
                {
                particle[i].inertia = problem_data.final_inertia;
                } /* for i */
            } /* else */

        break;

        case 2:
        /* constriction factor */

        /* nothing done to either inertia or velocity */

        break;

        case 3:
        /* Dynamic inertia and velocity reduction modification */

        if (no_improvement_iteration >= problem_data.wait_for_improvement)
```

```
            {
            no_improvement_iteration = 0;

            for (i = 0; i < problem_data.amount_of_particles; ++i)
                {
                particle[i].inertia = particle[i].inertia * (1 -
problem_data.inertia_reduction_fraction);
                } /* for */

            for (j = 0; j < problem_data.problem_dimensions; ++j)
                {
                comm_data.swarm_max_velocity[j] = comm_data.swarm_max_velocity[j] * (1 -
problem_data.velocity_reduction_fraction);
                } /* for */
            } /* if */

        break;

        } /* end switch */
    } /* end update_inertia_max_velocity */

void update_best_function_value(int i)
    {

    /* Updates particle_best_value and swarm_best_value and their coordinates */
    int j;

    if ((particle[i].current_value < particle[i].best_value) &&
(particle[i].constraints_violated != 1))
        {
        particle[i].best_value = particle[i].current_value;

        for (j = 0; j < problem_data.problem_dimensions; ++j)
            {
            particle[i].best_value_coordinates[j] = particle[i].coordinates[j];
            } /* for j */
        } /* if */


    if ((particle[i].current_value < comm_data.swarm_best_value) &&
(particle[i].constraints_violated != 1))
        {

        if ( (particle[i].current_value + problem_data.no_improvement_tolerance) <
comm_data.swarm_best_value)
```

```
            {
            no_improvement = 0;
            }

        comm_data.swarm_best_value = particle[i].current_value;
        comm_data.swarm_best_normalized_infeasability =
particle[i].normalized_infeasability;
        no_improvement_iteration = 0;

        for (j = 0; j < problem_data.problem_dimensions; ++j)
            {
            comm_data.swarm_best_coordinates[j] = particle[i].coordinates[j];
            } /* for j */
        } /* if */
    } /* update_best_function_value */
```

D.4.3 Termination

The following functions implement several stopping methods, among them the logical and *a priori* stopping criteria.

```
void check_for_stop(FILE *outputfilepointer, int stopping_method)
    {

    /* Function to check if stopping condition is satisfied */
    int j;
    double temp_max_velocity;

    switch(stopping_method)
        {

        case 1:
        /* Absolute (error minimization) */

        if ((comm_data.swarm_best_value <= (problem_data.solution_func_value +
comm_data.tolerance))&&(found_solution_flag == 0))
            {

            found_solution_flag = 1;

            result[problem_iteration].converged_flag = 1;
            result[problem_iteration].function_evaluations =
comm_data.function_evaluations;
```

```
                result[problem_iteration].normalized_infeasability =
comm_data.swarm_best_normalized_infeasability;
                result[problem_iteration].swarm_best_value = comm_data.swarm_best_value;

                for (j = 0; j < problem_data.problem_dimensions; ++j )
                    {
                    result[problem_iteration].swarm_best_coordinates[j] =
comm_data.swarm_best_coordinates[j];
                    }

                ++comm_data.times_converged;

                display_results();

                write_outputfile(outputfilepointer,problem_iteration);

                } /* end if */

            break;

            case 2:
            /* Inertia */
            if (particle[0].inertia <= problem_data.inertia_stop)
                {
                found_solution_flag = 1;

                result[problem_iteration].converged_flag = 1;
                result[problem_iteration].function_evaluations =
comm_data.function_evaluations;
                result[problem_iteration].normalized_infeasability =
comm_data.swarm_best_normalized_infeasability;
                result[problem_iteration].swarm_best_value = comm_data.swarm_best_value;

                for (j = 0; j < problem_data.problem_dimensions; ++j )
                    {
                    result[problem_iteration].swarm_best_coordinates[j] =
comm_data.swarm_best_coordinates[j];
                    }

                ++comm_data.times_converged;

                display_results();

                write_outputfile(outputfilepointer,problem_iteration);
            } /* end if */

        break;

        case 3:
        /* No improvement */

        if (no_improvement >= problem_data.no_improvement_stop)
            {

            found_solution_flag = 1;

            result[problem_iteration].converged_flag = 1;
            result[problem_iteration].function_evaluations =
comm_data.function_evaluations;
            result[problem_iteration].normalized_infeasability =
comm_data.swarm_best_normalized_infeasability;
            result[problem_iteration].swarm_best_value = comm_data.swarm_best_value;

            for (j = 0; j < problem_data.problem_dimensions; ++j )
                {
                result[problem_iteration].swarm_best_coordinates[j] =
comm_data.swarm_best_coordinates[j];
                }

            ++comm_data.times_converged;

            display_results();

            write_outputfile(outputfilepointer,problem_iteration);

            } /* end if */

        break;

        case 4:
        /* Velocity */

        temp_max_velocity = 0;

        /* check if velocity vector magnitude is smaller than velocity_stop */
        for (j = 0; j < problem_data.problem_dimensions; ++j)
            {
            temp_max_velocity = SQUARE(comm_data.swarm_max_velocity[j]);
            } /* for j */

            if ( (sqrt(temp_max_velocity)) <= problem_data.velocity_stop)
```

```
                {
                found_solution_flag = 1;

                result[problem_iteration].converged_flag = 1;
                result[problem_iteration].function_evaluations =
comm_data.function_evaluations;
                result[problem_iteration].normalized_infeasability =
comm_data.swarm_best_normalized_infeasability;
                result[problem_iteration].swarm_best_value = comm_data.swarm_best_value;

                for (j = 0; j < problem_data.problem_dimensions; ++j )
                    {
                    result[problem_iteration].swarm_best_coordinates[j] =
comm_data.swarm_best_coordinates[j];
                    } /* j */

                ++comm_data.times_converged;

                display_results();

                write_outputfile(outputfilepointer,problem_iteration);

                } /* if */

            break;
        } /* switch */
    } /* check_for_stop */
```

D.4.4 Main program

The initial part of the main program is presented below (with the postprocessing section removed). The sequential algorithm is implemented herein, but this can easily be changed to an asynchronous implementation by updating on a per swarm basis.

```
 int main(void)
  {
    FILE *inputfilepointer, *outputfilepointer, *historyfilepointer, *bestfilepointer,
*parameterfilepointer;
    int stopping_condition_flag = 0;
    int counter = 0;
    int iteration = 0;
    int pcle;
    int repeat;
    int k,z,t,j;
    int updatecount;
    int random_init;
    int temp_converged;


 /* Seed random number generator using current time */
    time_t curr_time;
    srand( (unsigned) time(&curr_time));

 comm_data.times_converged = 0;

 if ( (inputfilepointer = fopen(''script'',''r'') ) == NULL)
    {
       printf(''\a\nCould not open input script file \n'');
       exit(1);
       }

    read_inputfile(inputfilepointer);


 /*  Open file for problem data and solutions found output */

   if ( (outputfilepointer = fopen(output_filename,''w'') ) == NULL)
    {
       printf(''\a\nCould not open problem data output file \n'');
       exit(1);
       }


 /*  Open file for problem data parameter variation output */

   if ( (parameterfilepointer = fopen(''parameter'',''w'') ) == NULL)
    {
       printf(''\a\nCould not open parameter results output file \n'');
       exit(1);
       }


 /*  Open file for function value history output */

   if (history_output == 1)
    {
       if ( (historyfilepointer = fopen(''history'',''w+'') ) == NULL)
        {
          printf(''\a\nCould not open history file for reading/writing\n'');
          exit(1);
          }
        if ( (bestfilepointer = fopen(''best_history'',''w'') ) == NULL)
         {
           printf(''\a\nCould not open best_history file for writing\n'');
           exit(1);
           }
        }

    for (test_problem = first_problem; test_problem <= last_problem; ++test_problem)
      {
        problem_data.problem_number = test_problem;
        initialize_problem_set_variables(problem_data.problem_number);
        set_tolerance();

    write_outputfile_header(outputfilepointer);

      for (t = 0; t < problem_data.problem_dimensions; ++t)
       {
         total_length[t] = (problem_data.position_upperbound[t]-
problem_data.position_lowerbound[t]);
        }

    print_problemdata_screen();

    /* start problem iterations */

      for (problem_iteration = 0; problem_iteration < problem_data.problem_repeats;
++problem_iteration)
        {
      found_solution_flag = 0;
         comm_data.function_evaluations = 0;
           iteration = 0;
         random_init = 0;
         no_improvement_iteration = 0;
         no_improvement = 0;

      /* print history start indicator*/
         if (history_output == 1)
           {
            fprintf(historyfilepointer,''\nstart#%i&%i'',problem_data.problem_number,
problem_iteration+1);
           } /* end if */

      /* initialize display */
    if (use_display == 1)
      {
      vga_init();    /* initialize graphics mode */
      cleardisplay();

      vga_setmode(VGAMODE);  /* set to appropriate mode */
      gl_setcontextvga(VGAMODE);
      physicalscreen = gl_allocatecontext();
      gl_getcontext(physicalscreen);

   setcustompalette();     /* initialize custom palette */
      /* initfont() here caused trouble with planar 256 color modes. */

      gl_setcontextvgavirtual(VGAMODE);
      backscreen = gl_allocatecontext();
      gl_getcontext(backscreen);

   initfont();

      gl_setcontextvgavirtual(VGAMODE);
      background = gl_allocatecontext();
      gl_getcontext(background);

      drawbackground();

      framerate = 0;
      framecount = 0;
      frameclock = clock();
      } /* if usedisplay */


    /* swarm initialization */

     initialize_particle_positions();
     initialize_particle_velocities();
     initialize_particle_inertia_values();
     initialize_particle_function_values();
     initialize_swarm_max_velocity();

    /* start sequential swarm iteration loop */

    pcle = 0;

    do
      {

      /* run through all the particles in sequence */
```

```
        if (use_display == 1)
           {
           if (updatecount  > display_refresh)
              {
                 drawscreen();
                 drawgraph();
               updatecount = 0;
              }
           }

    if (use_craziness == 1)
           {
           if (random_init  > craziness_period)
              {
              initialize_particle_velocities();
              random_init = 0;
              }
           }

        ++random_init;    /* craziness counter */
        ++no_improvement_iteration;
        ++no_improvement;

        update_particle_velocity(pcle);
        update_particle_position(pcle);
        update_particle_function_value(pcle);
          update_best_function_value(pcle);
        update_inertia_max_velocity();

        if (history_output == 1)
           {
           fprintf(historyfilepointer,''\n%f'',comm_data.swarm_best_value);
           }

    /* Stopping condition */
    check_for_stop(outputfilepointer,problem_data.stopping_method);


    if (use_display == 1)
              {
              /* Update frame rate every 3 seconds. */
              framecount++;
              if (clock() - frameclock >= CLOCKS_PER_SEC)
                 {
                 framerate = framecount * CLOCKS_PER_SEC / (clock() - frameclock);
                 framecount = 0;
```

```
                    frameclock = clock();
                    } /* end if */
                 }

            ++updatecount;

            if (pcle == (problem_data.amount_of_particles - 1))
              {
              pcle = 0;
              }
              else
              {
              ++pcle;
           }
            } /* while pcle < p - 1 */
         while ((comm_data.function_evaluations <=
problem_data.max_function_evaluations)&&(found_solution_flag == 0));


         if (comm_data.function_evaluations == problem_data.max_function_evaluations)
            {
            result[problem_iteration].swarm_best_value = 0;
            result[problem_iteration].function_evaluations = 0;
            result[problem_iteration].converged_flag = 0;
            }

      /* print history end indicator*/
          if (history_output == 1)
            {
             fprintf(historyfilepointer,''\nend#%i&%i'',problem_data.problem_number,
problem_iteration+1);
             } /* end if */


      found_solution_flag = 0;
         comm_data.function_evaluations = 0;

      } /* end problem_iteration */
```