

```

/* BH_v13-07.clean.c
 * Author: norman chonacky
 *
 * Created on: 13 October 2009
 * Derived from: BH_v13-06.c
 * Validated on: 19 October 2009 with output in <output.v13-07x.txt>
 * Modified last on: 19 October 2009
 *
 * Objective: Remove all of the diagnostic output but retain report data.
 * Goal: An instructional version of the validated 1-D BH program for
 *       N-body system according the Barnes-Hut method.
 *
 * Goal: A full calculation of the net force on each particle in an
 *       N-body system according the Barnes-Hut method.
 */

#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>
#include <math.h>
#define PARTICLE_NOS 5 //Cap on the number of particles in system
#define SEG_CNT 2 //Number of branches for 1-D system: tree is binary.
#define DATA_CNT 2 //Data items per particle for 1-D system {m,x}.
#define QUALITY_FACTOR 1.1 //specified value for alpha.
#define MINIMUM 0.0001 //Value to avoid overflow error.

typedef struct {
    int particleID;
    int mass;
    float *position;
} datablock;

struct node_struct {
    int level, sector;
    float center;
    datablock *data; // pointer to data for this node
    struct node_struct *seg_0; // pointer toward x< branch node
    struct node_struct *seg_1; // pointer toward x> branch node
};

static struct node_struct *root;// the top node of the tree
long int ptr_root;
static datablock *parray[PARTICLE_NOS];
float location[PARTICLE_NOS] = // position data initialized with ...
    {9.0, 1.0, 4.0, 8.0, 2.0}; // ... these values.
float net_force[PARTICLE_NOS]={0,0,0,0,0};
float domain = 10.0, x_min = 0;
float root_center;
float alpha = QUALITY_FACTOR; // specify tolerance parameter alpha.
static int pseudo_ID = 0;//(neg.)ID's distinguish pseudo from real particles.
int target_seg = 0;
int max_segs = SEG_CNT;//Max. node segments for 1-D system: tree is binary.
int i;
FILE *out_file;

/*****
 * memory_error -- write error and die
 *****/

```

```

void memory_error(void)
{
    fprintf(stderr, "Error:Out of memory\n");
    exit(8);
}

/*****
 * grab_particle -- place particle data on the heap      *
 *                                                       *
 * Parameters                                           *
 *     string -- string to save                         *
 * Returns                                              *
 *     pointer to malloc-ed section of memory with     *
 *     the data copied into it.                         *
 *****/
datablock *grab_particle (int particle_index)    //place particle on heap
{
    datablock *new_particle;// where on the heap we put this particle

    new_particle = malloc(sizeof(datablock));
    if (new_particle == NULL)
        memory_error();

    new_particle->particleID = particle_index+1;
    new_particle->position = &location[particle_index];
    new_particle->mass = 1;
    return (new_particle);
}

/*****
 * Determine in which subdomain sector *
 * the position of this particle lies. *
 *****/
int next_segment(float position, float node_center){
    int direction;
    if (position <= node_center) direction =0;
    else direction =1;
    return(direction);
}

int powerof2 (int degree) {
    int current = 1;
    int i;
    for (i=0; i<degree; ++i) current = current*2;
    return current;
}

/*****
 * print all node metadata *
 *****/
void print_node(struct node_struct *this_node) {
    fprintf(out_file, "    This node is located at %x {%d,%d}\n",
        (unsigned int)(this_node), this_node->level, this_node->sector);
    fprintf(out_file, "    It has center value = %f\n", this_node->center);
    if (this_node->data != NULL) { //There is a particle here.
        fprintf(out_file, "    Its data pointer = %x\n",
            (unsigned int)this_node->data);
        fprintf(out_file,
            "    Its data are at location: %x {#%d, x=%4.2f}\n",

```

```

        (unsigned int)this_node->data, this_node->data->particleID,
        *this_node->data->position);
    }
    else fprintf(out_file, "    Its data pointer = NULL\n");
if (this_node->seg_0 != NULL)
    fprintf(out_file, "    Its seg_0 pointer = %x\n",
            (unsigned int)this_node->seg_0);
    else fprintf(out_file, "    Its seg_0 pointer = NULL\n");
if (this_node->seg_1 != NULL)
    fprintf(out_file, "    Its seg_1 pointer = %x\n",
            (unsigned int)this_node->seg_1);
    else fprintf(out_file, "    Its seg_1 pointer = NULL\n");
}

/*****
 * displace -- displaces an occupant particle from its node to a child *
 * spawned in the appropriate sector at the next level. *
 * Parameters *
 * node -- current node we are looking at *
 * value -- particle data to enter *
 *****/
void displace(struct node_struct **node, datablock *values,
             int target_level, int target_sect, int target_seg,
             float parent_ctr) {
    float offset=0;
    //Prepare to move occupant from its former node (parent) to new leaf
    if ((*node) != NULL){ //Target node mustn't exist in order to...
        //...displace the current occupant there!
        printf("Error: occupied node was not a leaf. Exiting program.");
        exit(110);
    }
    (*node) = malloc(sizeof(struct node_struct)); // Allocate a new node...
        //... assigned to parent's branch pointer.
    if ((*node) == NULL) // Allocation failed...
        memory_error(); //...announce error and exit, otherwise...

    // Initialize new node components as leaf
    (*node)->seg_0 = NULL;
    (*node)->seg_1 = NULL;
    (*node)->level = target_level; //...assign target level...
    (*node)->sector = target_sect; //...and assign target sector.
    if (target_level != 0) { // This node not root so calculate offset.
        offset = domain/powerof2(target_level+1);
    }
    // Calculate center coordinate for new node depending upon its sector.
    switch (target_seg) {
        case 0: // If new node in seg_0 ...
            (*node)->center = parent_ctr - offset;
            break;
        case 1: //... new node in seg_1
            (*node)->center = parent_ctr + offset;
            break;
        default:
            printf("Error in sector %d ", target_seg);
            exit (100); //This is fatal error - terminate program.
    }
}
else { // Level=0, must be root; so must be an error.
    printf("Error: displacement can't be into root. Exit.");
}

```

```

        exit(111);
    }
// Now displace the occupant particle to its new node.
(*node)->data = values;
fprintf(out_file,
"\nThese are metadata of a node newly created for the occupant:\n");
print_node(*node);
return;    // Finished displacing this particle to a new leaf.
}

/*****
* enter -- enter a particle into the tree
* Parameters
*   node -- current node we are looking at
*   value -- particle data to enter
*****/
void enter(struct node_struct **target_node, datablock *entrant_values,
          int target_level, float parent_ctr, int current_sect,
          int target_seg) {
    int displace_seg=0;
    int new_level=0, new_sect=0;
    float offset=0;
    datablock *occupant_values;
    struct node_struct *ptr_parent_node;

    if ((*target_node) != NULL){// Node DOES exist at target, so then...
//...target becomes parent for subsequent (displacement of occupant
// and/or) entry of new entrant particle.
        (ptr_parent_node) = (*target_node);
// Entrant and occupant, if extant, will move to next level...
// ... so increment current target node's level.
        new_level = (target_level+1);

//Is the target node occupied by another particle?
        if ((*target_node)->data != NULL){ //Yes it is occupied so ...
// ... split target domain and move occupant to a subdomain.
// ; .
            occupant_values = (*target_node)->data;//Copy occupant from target
            (*target_node)->data = NULL;//NULLify target's data pointer.
// Prep to split:
// In which segment of new subdomains does occupant particle lie,
            displace_seg= next_segment(*(occupant_values->position),
                                      (ptr_parent_node)->center);
// ... and to which new sector is it assigned?
            new_sect = (2*(ptr_parent_node)->sector) + displace_seg;
//Create new node and assign its address as a parent's branch pointer.
            switch (displace_seg) {
                case 0:
                    displace (&ptr_parent_node->seg_0, occupant_values,
                              new_level, new_sect, displace_seg,
                              ptr_parent_node->center);
                    break;
                case 1:
                    displace (&ptr_parent_node->seg_1, occupant_values,
                              new_level, new_sect, displace_seg,
                              ptr_parent_node->center);
                    break;
                default:

```

```
        printf("Error in sector for resident at level %d\n",
               new_level);
        exit (101); //This is fatal error - terminate program.
    }
} //Done moving occupant to its sector branch from target node.

// Now advance entrant particle. In which new subdomain does it lie?
target_seg= next_segment(*((entrant_values)->position),
                        (ptr_parent_node)->center);
printf(" Its movement is to level: %d and segment %d \n",
       new_level, target_seg);
// Navigate to next level along link to the sector proper...
// ...for this particle.
new_sect = (2*(ptr_parent_node)->sector) + target_seg;
switch (target_seg) {
    case 0:
        enter(&(ptr_parent_node)->seg_0, entrant_values, new_level,
             (ptr_parent_node)->center, new_sect, target_seg);
        return;
    case 1:
        enter(&(ptr_parent_node)->seg_1, entrant_values, new_level,
             (ptr_parent_node)->center, new_sect, target_seg);
        return;
    default:
        printf("Error in sector for current particle@level %d\n",
               new_level);
        exit (101); //This is fatal error - terminate program.
}
}

else { //Target node DOESN'T exist, so can place particle here!
    printf("\nEntering function ENTER with target *node = NULL\n");
    new_level = target_level;
    (*target_node) = malloc(sizeof(struct node_struct)); // Allocate new node.
    if ((*target_node) == NULL) // Allocation failed...
        memory_error(); //...announce error and exit, otherwise...

    // Initialize new node components as leaf
    (*target_node)->seg_0 = NULL;
    (*target_node)->seg_1 = NULL;
    (*target_node)->level = target_level; //...assign target level.
    (*target_node)->sector = current_sect; //...and assign sector index

    // Calculate center coordinate for new node
    if (target_level == 0) { // This node is root so initialize...
        ptr_parent_node = (*target_node); //...parent as root
        (ptr_parent_node)->center = root_center; //...with this as its center.
    }
    else { // This node not root so calculate offset.
        offset = domain/powerof2(target_level+1);
        switch (target_seg) {
            case 0: // If new node in sector "lesser"...
                (*target_node)->center = parent_ctr - offset;
                ptr_parent_node->seg_0 = (*target_node);
                break;
            case 1: //... new node in sector "greater".
                (*target_node)->center = parent_ctr + offset;
                ptr_parent_node->seg_1 = (*target_node);
        }
    }
}
```

```

        break;
        default:
            printf("Error in sector %d ", target_seg);
            exit (100); //This is fatal error - terminate program.
        }
    }
    // Now place the particle on target, a new leaf node.
    (*target_node)->data = entrant_values;
    fprintf(out_file,"These are metadata for newly created leaf:\n");
    print_node(*target_node);
} // Finished entering this particle.
} // Go back and Grab next.

/*****
 * load particle data into variables on the heap
 * int index; element of particle array from which we
 * are drawing a position in this first test version.
 *****/
void build_tree(struct node_strct **top, datablock *particle_ptr_array[])
{
    datablock *grab_particle(int );//Install particle's data on the heap
    datablock *pdata; //Pointer to new particle data
    int index;

    root_center = x_min+(domain/2); //This is root center.
    for (index = 0; index < PARTICLE_NOS; ++index) {
        pdata = grab_particle(index);
        particle_ptr_array[index] = pdata;//Build array of particle adrs.
        enter(&(*top), pdata, 0, root_center, 0, target_seg);
    }
}

void report_tree(struct node_strct *target) {

    if (target == NULL) return; // revert to recursive parent
    report_tree(target->seg_0);
    fprintf(out_file,"\nReporting on node @ %x \n",(unsigned int)(target));
    print_node(target);
    report_tree(target->seg_1);
}

/*****
 * new_pseudo -- create pseudo-particle space ...
 * ...on the heap.
 * Returns
 * pointer to malloc-ed section of memory with
 * the initialized values of data copied into it.
 *****/
datablock *new_pseudo() { //make pseudo-particle space on heap
    datablock *new_particle;// where on the heap we put this particle

    new_particle = malloc(sizeof(datablock));
    new_particle->position = malloc(sizeof(float));
    if (new_particle == NULL)
        memory_error();

    --pseudo_ID; //decrement pseudo_ID number for this particle

```

```

new_particle->particleID = pseudo_ID;
*(new_particle->position) = 0;
new_particle->mass = 0;
printf("A newly created particle is at location: %x {#%d, ?}\n",
      (unsigned int)new_particle,new_particle->particleID);
return (new_particle);
}

/*****
*   Decorate branch node's pseudo-particle...
*   ... with data based on all inferior particles.
*****/
void decorate_node (struct node_struct *parent_node) {
    int mass_tot=0, this_seg, i;
    float M_centroid=0.0;
    int m=0, x=1;
    float buffer[DATA_CNT][SEG_CNT] = {{0,0},{0,0}};

    if (parent_node->data == NULL) {
        printf(
            "\nParent_node @ %x has *data==NULL-fatal error\n",
            (unsigned int) parent_node);
        exit(1000);
    }

    for (this_seg=0; this_seg<max_segs; ++this_seg) {
        switch (this_seg) {
            case 0:
                if (parent_node->seg_0 != NULL) {
                    printf("Decorate is in seg_0 located at %x {#%d,%d}\n",
                          (unsigned int)(parent_node->seg_0),
                          (parent_node->seg_0)->level,
                          (parent_node->seg_0)->sector);
                    buffer[m][this_seg] = (parent_node->seg_0->data->mass);
                    buffer[x][this_seg] = *(parent_node->seg_0->data->position);
                }
                else printf("Pointer to seg_0 == NULL; ignore it.");
                break;
            case 1:
                if (parent_node->seg_1 != NULL) {
                    printf("Decorate is in seg_1 located at %x {#%d,%d}\n",
                          (unsigned int)(parent_node->seg_1),
                          (parent_node->seg_1)->level,
                          (parent_node->seg_1)->sector);
                    buffer[m][this_seg] = parent_node->seg_1->data->mass;
                    buffer[x][this_seg] = *(parent_node->seg_1->data->position);
                }
                else printf("Pointer to seg_1 == NULL; ignore it.");
                break; ;
            default:
                printf("Err: Walk_tree encounters this_seg: %d",this_seg);
                exit(888);
        }
    }
    for (i=0; i<max_segs; ++i) {
        mass_tot = mass_tot + (int)buffer[m][i];
        M_centroid = M_centroid + (buffer[m][i] * buffer[x][i]);
    }
}

```

```

    (parent_node->data->mass) = mass_tot;
    *(parent_node->data->position) = (float)(M_centroid/mass_tot);
    printf("Branch node @ %x in sector {%d,%d} has been decorated\n\n",
           (unsigned int)(parent_node),
           (parent_node)->level,(parent_node)->sector);
}

/*****
 * This function directs a systematic walk through a tree structure *
 *****/
void walk_tree(struct node_struct *this_node) {
    int is_branch_node,this_seg;

    printf("\nWalking into node located at %x designated {%d,%d}",
           (unsigned int)(this_node),
           this_node->level, this_node->sector);

    if (this_node->data == NULL) this_node->data = new_pseudo();
    is_branch_node=0;
    for (this_seg=0; this_seg<max_segs; ++this_seg) {
        switch (this_seg) {
            case 0:
                if (this_node->seg_0 != NULL) {
                    is_branch_node=1;
                    walk_tree(this_node->seg_0);
                }
                break;
            case 1:
                if (this_node->seg_1 != NULL) {
                    is_branch_node=1;
                    walk_tree(this_node->seg_1);
                }
                break;
            default:
                printf("Err: Walk_tree encounters this_seg: %d",this_seg);
                exit(888);
        }
    }
    //All daughters of this_node examined.
    if (is_branch_node > 0) decorate_node(this_node);
return;
}

/*****
 * This function calculates the net force on one particle due to
 * all others in the system to an approximation controlled by a
 * specified quality factor - alpha.
 *****/
float calculated(datablock *data, float D) {
    float force;

    if (D < MINIMUM) {
        printf("Fatal error - divide by 'zero'");
        exit(-1000);
    }

    force = (data->mass)/D; // for 1-D system
return(force);
}

```

```

}

/*****
 * This function calculates the net force on one particle due to
 * all others in the system to an approximation controlled by a
 * specified quality factor - alpha.
 *****/

float force_by_pruning(struct node_struct *node, datablock *paddr,
                      float alpha) {
    float separation, sub_size, composite_force=0;
    int segment=0;

    for (segment=0; segment<SEG_CNT; ++segment)    {
        switch (segment)    {
            case 0:
                if (node->seg_0 == NULL) break; //Node NULL - try next daughter
                if ((node->seg_0->data->particleID) > 0 ) { //Node_seg is leaf.
                    if ((node->seg_0->data) == paddr) //Leaf is THE particle.
                        return(0);
                    else { //Not THE particle so calculate its force.
                        separation = fabs((*(node->seg_0->data->position)) -
                                           (*(paddr->position)));
                        sub_size = (domain/powerof2(node->seg_0->level));
                        composite_force =
                            composite_force +
                            calculated(node->seg_0->data, separation);
                        return(composite_force);
                    }
                } // End processing seg_0 as leaf.
            else { //Node_seg is branch.
                separation = fabs((*(node->seg_0->data->position)) -
                                   (*(paddr->position)));
                sub_size = (domain/powerof2(node->seg_0->level));
                if ((sub_size/separation) < alpha) { //Passes test ...
                    //... so use pseudo-force and prune this branch.
                    composite_force =
                        composite_force +
                        calculated(node->seg_0->data, separation);
                    break; // Try next daughter.
                }
                else { //Fails test, so initiate recursion for...
                    //... continuing descent into this branch.
                    composite_force = composite_force +
                                        force_by_pruning(node->seg_0, paddr, alpha);
                    break;
                } // End processing seg_0 as branch.
            } // End treating case seg_0 for all.
        }
    }
    case 1:
        if (node->seg_1 == NULL) break; //Node NULL - try next daughter
        if ((node->seg_1->data->particleID)>0 ) { //Node_seg is leaf.
            if ((node->seg_1->data) == paddr) //Leaf is THE particle.
                break; // Try next daughter to avoid self-force.
            else { //Not THE particle, so use real particle force.
                separation = fabs((*(node->seg_1->data->position)) -
                                   (*(paddr->position)));
                sub_size = (domain/powerof2(node->seg_1->level));
                composite_force =

```

```

        composite_force +
        calculated(node->seg_1->data, separation);
    break;    // Try next daughter.
}
} //End of leaf processing for seg_1
else { //Node_seg is branch.
    separation = fabs((* (node->seg_1->data->position)) -
        (*(paddr->position)));
    sub_size = (domain/powerof2(node->seg_1->level));
    if ((sub_size/separation) < alpha){//Passes alpha test,
        composite_force =
            composite_force +
            calculated(node->seg_1->data, separation);
        break;    // Try next daughter.
    }
    else { //Fails alpha test, so initiate recursive...
        //... descent into this branch.
        composite_force = composite_force +
            force_by_pruning(node->seg_1, paddr, alpha);
        break;    // Try next daughter.
    }
} // End of branch processing for seg_1
default:
    printf("Segment selection error"); exit(segment);
} //End of switches.
} // End of segment processing.
return (composite_force);
}

/*****
 * This function directs a systematic calculation of net force on
 * each particle, one at a time, through the entire system set.
 *****/
void treat_particles (datablock *paddrs[], struct node_struct *top,
    float net_force[], float alpha) {
    int index;
    float separation;

    fprintf(out_file, "\n\n"); printf("\n\n");
    for (index=0; index<PARTICLE_NOS; ++index) {
        //First, test if the pseudo-particle @ root satisfies alpha
        separation =
            fabs((* (paddrs[index]->position)) - (*(root->data->position)));
        if ((domain/separation) < alpha) //This is easy. Use root's...
            //...pseudo-force as contribution and we're done with this particle.
            net_force[index] = calculated(top->data, separation);
        else // If not, enter recursive exploration starting from root.
            net_force[index] = force_by_pruning(top, paddrs[index], alpha);
        fprintf(out_file, "Force on particle %d = %f\n", index+1, net_force[index]);
        printf("Force on particle %d = %f\n", index+1, net_force[index]);
    }
}

/*****
int main(void)    {

    out_file = fopen("output.txt", "w");
    root = NULL;

```

```
    build_tree(&root, parray);
    printf("\n\nReporting on the tree after building.\n");
    fprintf(out_file, "\n\nReporting on the tree after building.\n");
    for (i=0; i<PARTICLE_NOS; ++i)
        fprintf(out_file, "\nPointer to ID# %d: %x\n",
                    i+1, (unsigned int) parray[i]);
    if (root == NULL) return(999);
    else report_tree(root);
    walk_tree(root);
    printf("\n\nReporting on the tree after walking.\n");
    report_tree(root);
    treat_particles (parray, root, net_force, alpha);
    fflush(out_file);
    fclose(out_file);
    return 0;
}
```