

Help

```
#include      <stdio.h>
#include      <stdlib.h>
#include      "cdo_math.h"
#include      "structs.h"

/**
 * Initialisation of a grid of size n
 *
 * @param n : size of the grid
 * @return a grid ptr
 */

grid          *create_grid(int  n)
{
    grid          *gd;

    gd = malloc(sizeof(grid));
    gd->size = n;
    gd->data = malloc(n * sizeof(double));
    gd->delta = malloc(n * sizeof(double));

    return (gd);
}

/**
 * Initialisation of a grid of size n and data=x
 *
 * @param n : size of the grid
 * @param x : array of datas
 * @return a grid ptr
 */

grid          *init_grid_cdo(int      n,
                               const double  *x)
{
    grid          *gd;
    int jn;
```

```

    gd = malloc(sizeof(grid));
    gd->size = n;
    gd->data = malloc(n * sizeof(double));
    gd->delta = malloc(n * sizeof(double));
    gd->data[0] = x[0];
    for (jn = 1; jn < n; jn++)
    {
        gd->data[jn] = x[jn];
        gd->delta[jn] = x[jn] - x[jn - 1];
    }
    gd->delta[0] = gd->delta[1];

    return (gd);
}

/**
 * Initialisation of an homogene grid
 *
 * @param x0 : first value of gd->data
 * @param xn : last value of gd->data
 * @param delta : constant step size
 * @return a grid ptr
 */

grid *init_hom_grid(double    x0,
                    double    xn,
                    double    delta)
{
    grid    *gd;
    double    x;
    int jn;

    gd = malloc(sizeof(grid));
    gd->size = (int)ceil((xn - x0) / delta + delta / 2.);
    gd->data = malloc(gd->size * sizeof(double));
    gd->delta = malloc(gd->size * sizeof(double));
    for (jn = 0, x = x0; jn < gd->size; jn++, x += delta)
    {
        gd->data[jn] = x;
        gd->delta[jn] = delta;
    }
}

```

```

    }

    return (gd);
}

/**
 * Initialisation of a fine grid from an other one
 *
 * @param gd_init : a grid ptr
 * @param n : number of subdivisions of one interval
 * @return a grid ptr constructed with gd_init:
 * each interval [gd_init->data(i),gd_init->data(i+1)] is subdivided in n parts
 * same size new_delta and gd->data(j+1)=gd_data(j)+new_delta
 */

grid          *init_fine_grid(const grid      *gd_init,
                              int              n)
{
    grid      *gd;
    double     new_delta;
    int        jn_init;
    int        jn;
    int        j;

    gd = malloc(sizeof(grid));
    gd->size = n * (gd_init->size - 1) + 1;
    gd->data = malloc(gd->size * sizeof(double));
    gd->delta = malloc(gd->size * sizeof(double));
    jn = 1;
    gd->data[0] = gd_init->data[0];
    gd->delta[0] = gd_init->delta[0] / (double) n;
    for (jn_init = 0; jn_init < gd_init->size - 1; jn_init++)
    {
        new_delta = gd_init->delta[jn_init] / (double) n;
        for (j = 1; j < n; j++)
        {
            gd->data[jn] = gd->data[jn - 1] + new_delta;
            gd->delta[jn] = new_delta;
            jn++;
        }
    }
}

```

```
        gd->data[jn] = gd_init->data[jn_init + 1];
        gd->delta[jn] = new_delta;
        jn++;
    }
    return (gd);
}

/**
 * prints a Grid.
 *
 * @param gd : a Grid ptr.
 */

void          print_grid(grid          *gd)
{
    int jn;

    printf("Size:  %d\ n", gd->size);
    printf("Data:  ");
    for (jn = 0; jn < gd->size; jn++)
        printf("%g\ t", gd->data[jn]);
    printf("\ nDelta: ");
    for (jn = 0; jn < gd->size; jn++)
        printf("%g\ t", gd->delta[jn]);
    printf("\ n");
}

/**
 * free a grid pointer.
 *
 * @param gd : pointer to free
 */

void          free_grid(grid          *gd)
{
    free(gd->data);
    free(gd->delta);
    free(gd);
}

/**
```

```

* copy a step_fun.
*
* @param sf : a step_fun ptr
* @return a step_fun ptr initialised with sf
*/

step_fun      *copy_sf(const step_fun      *sf)
{
    step_fun      *nsf = malloc(sizeof(step_fun));
    int            jx, n;

    nsf->size = sf->size;
    nsf->data = malloc(sf->size * sizeof(step_element));
    for (jx = 0; jx < nsf->size; jx++)
    {
        nsf->data[jx].x1 = sf->data[jx].x1;
        nsf->data[jx].x2 = sf->data[jx].x2;
        nsf->data[jx].degree = sf->data[jx].degree;
        nsf->data[jx].y1 = sf->data[jx].y1;
        nsf->data[jx].y2 = sf->data[jx].y2;
        nsf->data[jx].a = malloc(sf->data[jx].degree * sizeof(double));
        for (n = 0; n < nsf->data[jx].degree; n++)
            nsf->data[jx].a[n] = sf->data[jx].a[n];
    }

    return (nsf);
}

/**
* creates a new piecewise constant step_fun.
*
* @param size : number of step_element
* @param x : array containing x1 and x2 for each step element
* @param y : array containing y1 and y2 for each step element
* @return a step_fun ptr
*/

step_fun      *init_constant_sf(int size, const double *x, const double *y)
{
    step_fun      *sf = malloc(sizeof(step_fun));

```

```

    int          jx;
    sf->size = size;
    sf->data = malloc(size * sizeof(step_element));
    for (jx = 0; jx < sf->size; jx++)
    {
        sf->data[jx].x1 = x[jx];
        sf->data[jx].x2 = x[jx + 1];
        sf->data[jx].degree = 0;
        sf->data[jx].y1 = y[jx];
        sf->data[jx].y2 = y[jx + 1];
        sf->data[jx].a = NULL;
    }

    return (sf);
}

/**
 * creates a new linear continuous step_fun.
 *
 * @param size : number of step_element
 * @param x : array containing x1 and x2 for each step element
 * @param y : array containing y1 and y2 for each step element
 * @return a linear continuous step_fun ptr with slope (y2-y1)/(x2-x1)
 */

step_fun          *init_cont_linear_sf(int size, const double  *x, const double *
{
    step_fun *sf = malloc(sizeof(step_fun));
    int jx;

    sf->size = size;
    sf->data = malloc(size * sizeof(step_element));
    for (jx = 0; jx < sf->size; jx++)
    {
        sf->data[jx].x1 = x[jx];
        sf->data[jx].x2 = x[jx + 1];
        sf->data[jx].degree = 1;
        sf->data[jx].y1 = y[jx];
        sf->data[jx].y2 = y[jx + 1];
        sf->data[jx].a = malloc(sizeof(double));
        sf->data[jx].a[0] = (y[jx + 1] - y[jx]) / (x[jx + 1] - x[jx]);
    }
}

```

```

    }

    return (sf);
}

/**
 * integrates a step_fun.
 *
 * @param sf : a step_fun ptr
 * @return a step_fun which represents the integration of sf
 */

step_fun *integrate_sf(const step_fun *sf)
{
    step_fun      *I_sf = malloc(sizeof(step_fun));
    double        y2;
    int           n;
    int           jx;

    I_sf->size = sf->size;
    I_sf->data = malloc(sf->size * sizeof(step_element));
    y2 = 0.;
    for (jx = 0; jx < sf->size; jx++)
    {
        I_sf->data[jx].x1 = sf->data[jx].x1;
        I_sf->data[jx].x2 = sf->data[jx].x2;
        I_sf->data[jx].y1 = y2;
        I_sf->data[jx].degree = sf->data[jx].degree + 1;
        I_sf->data[jx].a = malloc(I_sf->data[jx].degree * sizeof(double));
        I_sf->data[jx].a[0] = sf->data[jx].y1;
        for (n = 1; n < I_sf->data[jx].degree; n++)
            I_sf->data[jx].a[n] = sf->data[jx].a[n - 1] / ((double)n + 1.);
        y2 += evaluate_poly(I_sf->data[jx].degree, I_sf->data[jx].a, I_sf->data[jx].x2);
        I_sf->data[jx].y2 = y2;
    }

    return (I_sf);
}

```

```

/*
  Comparison between the 'x' values of two step elements a
  and b
*/

static int      compare_se_x(const void      *a,
                             const void      *b)
{
  step_element  *ea = (step_element *) a;
  step_element  *eb = (step_element *) b;

  if (ea->x1 < eb->x1 - MINDOUBLE) return (-1);
  if (ea->x1 > eb->x2 + MINDOUBLE) return (1);
  return (0);
}

/* Comparison between the 'y' values of two step elements a
  and b
*/

static int      compare_se_y(const void      *a,
                             const void      *b)
{
  step_element  *ea = (step_element *) a;
  step_element  *eb = (step_element *) b;

  if (ea->y1 < eb->y1 - MINDOUBLE) return (-1);
  if (ea->y1 > eb->y2 + MINDOUBLE) return (1);
  return (0);
}

/*
  A voir
*/
double          compute_sf(const step_fun    *sf,
                           double          x)
{
  step_element  a;
  step_element  *r;
  double        result;

```



```

    a.x1 = x;
    r = bsearch(&a, sf->data, sf->size, sizeof(step_element), compare_se_x);
    if (r == NULL) printf("ERROR: %g,%g,%g\ n", x, sf->data[0].x1, sf->data[0].x2);
    result = evaluate_poly(r->degree, r->a, (x - r->x1));
    return (r->y1 + result);
}

/* A voir
*/

double          inverse_sf(const step_fun    *sf,
                           double          y)
{
    step_element    a;
    step_element    *r;
    double          xn;
    double          result;
    double          tmp1;
    double          tmp2;

    a.y1 = y;
    r = bsearch(&a, sf->data, sf->size, sizeof(step_element), compare_se_y);
    if (r == NULL) printf("ERROR: %g,%g,%g\ n", y, sf->data[0].y1, sf->data[0].y2);
    switch (r->degree)
    {
        case 0 :
            return (r->x1);
        case 1 :
            return (r->x1 + (y - r->y1) / r->a[0]);
        default:
            result = 0.5 * (r->x1 + r->x2);
            do
            {
                xn = result;
                tmp1 = evaluate_poly(r->degree, r->a, xn);
                tmp2 = evaluate_dpoly(r->degree, r->a, xn);
                result = xn - (tmp1 + r->y1 - y) / (tmp2 + r->a[0]);
            }
            while (fabs(result - xn) > 1e-7);
            return (result);
    }
}

```

```
    }  
}  
  
void          free_step_fun(step_fun          **sf)  
{  
    int i;  
    if (*sf == NULL) return;  
    for (i = 0; i < (*sf)->size; i++)  
    {  
        if ((*sf)->data[i].a != NULL) free((*sf)->data[i].a);  
    }  
    free((*sf)->data);  
    free(*sf);  
    *sf = NULL;  
  
    return;  
}
```