```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include "data_structures.h"
#include "subroutines.h"
#include "InOut.h"

int main( void )
{

/*************************************************************************
**********************
 ***************************** User-defined input parameters
*****************************

*************************************************************************
**********************/

    int lambda = 1;     // Parameter describing geometry of system (=1
for planar, =3 for spherical)

    double s_0 = 1.;    // Initial interface position
    double R   = 5.;    // Position of far boundary

    int nAlpha = 100;           // Number of points in phase alpha
    double dAlpha = 1E-7;       // Diffusion coefficient of phase
alpha
    double initialAlpha = 0.8;  // Initial concentration in phase
alpha
    double interAlpha   = 0.6;  // Interfacial concentration in phase
alpha



    int nBeta    = 100;
    double dBeta = 1E-5;
    double initialBeta  = 0.4;
    double interBeta   = 0.0;

    double time_step=0.1;
    int n_time_steps=10;    // As written, output written to maximum
100000000 steps

    double tol = 1.E-8;     // Used to determine whether linearisation
converges at each timestep

/*************************************************************************
**********************
 ***************************************** ENDS
*****************************************
```

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



    FILE *fpt;
    int i, tmp;
    two_phase *whole_system;

    whole_system =(two_phase *) calloc (1, sizeof(two_phase));

    whole_system->s = s_0;
    whole_system->l = R;
    whole_system->old_s = whole_system->s;
    whole_system->future_s = whole_system->s;


    // Alpha is on left
    whole_system->left =(single_phase *) calloc (1,
sizeof(single_phase));

    whole_system->left->n = nAlpha;
    whole_system->left->d_coeff = dAlpha;
    whole_system->left->c_boundary = interAlpha;

    whole_system->left->u =(double *) calloc (whole_system->left->n,
sizeof(double));
    whole_system->left->c =(double *) calloc (whole_system->left->n,
sizeof(double));
    whole_system->left->future_c =(double *) calloc (whole_system-
>left->n, sizeof(double));



    for (i=0; i<whole_system->left->n; i++)
    {
    whole_system->left->u[i] = double(i)/double(whole_system->left->n
- 1);
    whole_system->left->c[i] = initialAlpha;
    whole_system->left->future_c[i] = whole_system->left->c[i];
    }
    whole_system->left->c[whole_system->left->n-1] = whole_system-
>left->c_boundary;
    whole_system->left->future_c[whole_system->left->n-1] =
whole_system->left->c_boundary;

    // Beta is on right
    whole_system->right =(single_phase *) calloc (1,
sizeof(single_phase));

    whole_system->right->n = nBeta;
    whole_system->right->d_coeff = dBeta;
    whole_system->right->c_boundary = interBeta;
```

```
    whole_system->right->u =(double *) calloc (whole_system->right->n,
sizeof(double));
    whole_system->right->c =(double *) calloc (whole_system->right->n,
sizeof(double));
    whole_system->right->future_c =(double *) calloc (whole_system-
>right->n, sizeof(double));

    for (i=0; i<whole_system->right->n; i++)
    {
    whole_system->right->u[i] = double(i)/double(whole_system->right-
>n - 1);
    whole_system->right->c[i] = initialBeta;
    whole_system->right->future_c[i] = whole_system->right->c[i];
    }
    whole_system->right->c[0] = whole_system->right->c_boundary;
    whole_system->right->future_c[0] = whole_system->right-
>c_boundary;



    fpt=fopen("results.txt","w");

    fprintf(fpt, "Time\tInterface Position\n");
    /***** Looping over time (for the given timestep) *****/
    for (i=0; i<n_time_steps+1; i++)
    {
        if(i<100)
            {out_interface(whole_system, double(i)*time_step, fpt);}
        else if((i<1000) && (i%10 == 0))
            {out_interface(whole_system, double(i)*time_step, fpt);}
        else if((i<10000) && (i%100 == 0))
            {out_interface(whole_system, double(i)*time_step, fpt);}
        else if((i<100000) && (i%1000 == 0))
            {out_interface(whole_system, double(i)*time_step, fpt);}
        else if((i<1000000) && (i%10000 == 0))
            {out_interface(whole_system, double(i)*time_step, fpt);}
        else if((i<10000000) && (i%100000 == 0))
            {out_interface(whole_system, double(i)*time_step, fpt);}
        else if((i<100000000) && (i%1000000 == 0))
            {out_interface(whole_system, double(i)*time_step, fpt);}

        if(lambda == 1)
            {tmp = take_step_planar(whole_system, time_step, tol);}
        else if(lambda == 3)
            {tmp = take_step_spherical(whole_system, time_step, tol);}
        else
            {printf("\nProblem with geometry\n\n");}


        if(tmp<0)    // Interface has moved beyond the end of the
system
            {printf("Finished at time = %lg", double(i)*time_step);
fclose(fpt); return 0;}
```

```
        if(i%1000 == 0)
            {printf("\nTimestep %d complete\n", i);}
        else if (i%10 == 0)
            {printf(".");}
    }

    free(whole_system->left->u);
    free(whole_system->left->c);
    free(whole_system->left->future_c);
    free(whole_system->right->u);
    free(whole_system->right->c);
    free(whole_system->right->future_c);
    free(whole_system->left);
    free(whole_system->right);

    fclose(fpt);

    printf("\n \n");
    return 1;

}
```

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "data_structures.h"
#include "InOut.h"


void out_profile(two_phase *tp, FILE *fp, double time)
{       // Prints:
            //  time
            //  x[0],   c(x[0]),
            //  x[1],   c(x[1]),
            //  ....    ....
            //  x[n],   c(x[n]),
            //  x[n+1], c(x[n+1]),
            //  x[x+2], c(x[n+2]),
            //  ...     ...
            //  x[n+N], c(x[n+N]),
            //

    int i;

    fprintf(fp, "%lg,\n", time);
    for(i=0; i<(tp->left->n); i++)
```

```
    {
        fprintf(fp, "%lg,\t", tp->s*tp->left->u[i]);
        fprintf(fp, "%lg,\n", tp->left->c[i]);
    }

    for(i=0; i<(tp->right->n); i++)
    {
        fprintf(fp, "%lg,\t", tp->s + (tp->l-tp->s)*tp->right->u[i]);
        fprintf(fp, "%lg,\n", tp->right->c[i]);
    }
    fprintf(fp, "\n");
}

void out_interface(two_phase *tp, double time, FILE *fp)
{
    fprintf(fp, "%lg\t%lg\n", time, tp->s);
}
```

```
#include <stdlib.h>
#include <stdio.h>
#include "trimatrix.h"
#include "data_structures.h"
#include "subroutines.h"


// #########  This file uses 1st order conservative scheme to generate
a solution on an irregular mesh ######### //

/*******************************************
    Given the concentration profiles and interface position at
timestep j and estimates for the values at j+1,
    updates the estimate for interface position at timestep j+1
*******************************************/
int new_interface_planar(two_phase *tp, double time_step, double
pass_nmb)
{
```

```
    //note: For first (implicit step), we require some first estimates
for solution variables
            // 1) the elements in future_c to take the values of
'current_' c
            // 2) the value of future_s to be the same as s

    double v;
    double diff_l, diff_r;
    double rhs, lhs;

    if(pass_nmb==0)             // FIRST PASS -
    {v=tp->s-tp->old_s;}            // Use old interface position to
determine sign of interface velocity
    else
    {v=tp->future_s-tp->s;}     // SECOND (or higher) PASS -
                                    // Use estimate of future position
to determine sign of interface velocity

    diff_l=(tp->left->c_boundary - tp->left->future_c[tp->left->n-
2])/(1. - tp->left->u[tp->left->n-2]);
    diff_l= diff_l*tp->left->d_coeff / tp->future_s;

    diff_r=(tp->right->future_c[1] - tp->right->c_boundary)/(tp-
>right->u[1]);
    diff_r= diff_r*tp->right->d_coeff / (tp->l-tp->future_s);

    rhs = (diff_r - diff_l)*time_step;


    if(v>=0)            //POSITIVE VELOCITY - Use points to the right:
    {
        lhs = tp->left->c_boundary;
        lhs = lhs - tp->right->future_c[1]*(1-tp->right->u[1]/2.);
        lhs = lhs - tp->right->c_boundary*tp->right->u[1]/2.;
    }
    else                //NEGATIVE VELOCITY - Use points to the left:
    {
        lhs = tp->left->future_c[tp->left->n-2] * (0.5+tp->left->u[tp-
>left->n-2]/2.);
        lhs = lhs + tp->left->c_boundary * (0.5-tp->left->u[tp->left-
>n-2]/2.);
        lhs = lhs - tp->right->c_boundary;
    }

    tp->future_s=tp->s + rhs/lhs;

    return 0;

}
```

```
/*******************************************
    Given the concentration profiles and interface position at
timestep j and estimates for the interface at j+1,
    finds concentrations at j+1 in phase to the left of the boundary
*******************************************/
void new_concentration_left_planar(two_phase *tp, double time_step)
{
    int i;
    double tmpA, tmpB, left_diff, right_diff, left_sum, right_sum;
    trimatrix_system tms;


    tms.dim = tp->left->n;

// RESERVE MEMORY FOR tms
    tms.lo   = (double *) calloc(tms.dim, sizeof(double));
    tms.diag = (double *) calloc(tms.dim, sizeof(double));
    tms.up   = (double *) calloc(tms.dim, sizeof(double));
    tms.rhs  = (double *) calloc(tms.dim, sizeof(double));
    tms.c    = tp->left->future_c;



// FILL tms
    tmpA=tp->left->d_coeff*time_step / tp->future_s;
    tmpB=(tp->future_s-tp->s);

    if(tp->future_s >= tp->s)               // POSITIVE VELOCITY - use
points to the right:
    {
        tms.lo[0]  =0.;
        tms.diag[0]=-tmpA/tp->left->u[1] - tp->future_s*tp->left-
>u[1]/2.;
        tms.up[0]  = tmpA/tp->left->u[1] + tmpB*tp->left->u[1]/2.;
        tms.rhs[0] =-tp->left->c[0]*tp->s*tp->left->u[1] / 2.;

        for(i=1; i<(tms.dim-1); i++)
        {
            left_diff = tp->left->u[i] - tp->left->u[i-1];
            left_sum  = tp->left->u[i] + tp->left->u[i-1];
            right_diff= tp->left->u[i+1] - tp->left->u[i];
            right_sum = tp->left->u[i+1] + tp->left->u[i];

            tms.lo[i]  = tmpA/left_diff;
            tms.diag[i]=-tmpA*(1/left_diff + 1/right_diff)  -
tmpB*left_sum/2.
                        - tp->future_s*(right_sum-left_sum)/2.;
            tms.up[i]  = tmpA/right_diff + tmpB*right_sum/2.;
            tms.rhs[i] =-tp->s*tp->left->c[i]*(right_sum-left_sum)/2.;
        }

    }
```

```
    else                                        // NEGATIVE VELOCITY - use
points to the left:
    {
        tms.lo[0]  =0.;
        tms.diag[0]=-tmpA/tp->left->u[1] + tmpB*tp->left->u[1]/2. -
tp->future_s*tp->left->u[1]/2.;
        tms.up[0]  = tmpA/tp->left->u[1];
        tms.rhs[0] =-tp->left->c[0]*tp->s*tp->left->u[1] / 2.;

        for(i=1; i<(tms.dim-1); i++)
        {

            left_diff = tp->left->u[i] - tp->left->u[i-1];
            left_sum  = tp->left->u[i] + tp->left->u[i-1];
            right_diff= tp->left->u[i+1] - tp->left->u[i];
            right_sum = tp->left->u[i+1] + tp->left->u[i];

            tms.lo[i]  = tmpA/left_diff  - tmpB*left_sum/2;
            tms.diag[i]=-tmpA*(1/left_diff + 1/right_diff) +
tmpB*right_sum/2.
                        - tp->future_s*(right_sum-left_sum)/2.;
            tms.up[i]  = tmpA/right_diff;
            tms.rhs[i] =-tp->s*tp->left->c[i]*(right_sum-left_sum)/2.;
        }
    }

    tms.lo[tms.dim-1]  = 0.;
    tms.diag[tms.dim-1]=-1.;
    tms.up[tms.dim-1]  = 0.;
    tms.rhs[tms.dim-1] =-tp->left->c_boundary;


// SOLVE tms
    solve_trimatrix_system(tms);


// FREE MEMORY RESERVED FOR tms
    free(tms.lo);
    free(tms.diag);
    free(tms.up);
    free(tms.rhs);
}



/*****************************************
    Given the concentration profiles and interface position at
timestep j and estimates for the interface at j+1,
    finds concentrations at j+1 in phase to the right of the boundary
*****************************************/
void new_concentration_right_planar(two_phase *tp, double time_step)
{
```

```
    int i;
    double tmp, tmpA, tmpB, left_diff, right_diff, left_sum,
right_sum;
    trimatrix_system tms;

    tms.dim = tp->right->n;

// RESERVE MEMORY FOR tms
    tms.lo   = (double *) calloc(tms.dim, sizeof(double));
    tms.diag = (double *) calloc(tms.dim, sizeof(double));
    tms.up   = (double *) calloc(tms.dim, sizeof(double));
    tms.rhs  = (double *) calloc(tms.dim, sizeof(double));
    tms.c    = tp->right->future_c;



// FILL tms
    tmpA=tp->l-tp->future_s;
    tmpA=tp->right->d_coeff*time_step/tmpA;
    tmpB=(tp->future_s-tp->s);



    tms.lo[0]  = 0.;
    tms.diag[0]=-1.;
    tms.up[0]  = 0.;
    tms.rhs[0] =-tp->right->c_boundary;

    if(tp->future_s >= tp->s)               // POSITIVE VELOCITY - use
points to the right:
    {
        for(i=1; i<(tms.dim-1); i++)
        {
            left_diff = tp->right->u[i] - tp->right->u[i-1];
            left_sum  = tp->right->u[i] + tp->right->u[i-1];
            right_diff= tp->right->u[i+1] - tp->right->u[i];
            right_sum = tp->right->u[i+1] + tp->right->u[i];

            tms.lo[i]  = tmpA/left_diff;
            tms.diag[i]=-tmpA*(1/right_diff + 1/left_diff) - tmpB*(1.-
left_sum/2.)
                        - (tp->l-tp->future_s)*(right_sum-left_sum)/
2.;
            tms.up[i]  = tmpA/right_diff + tmpB*(1. - right_sum/2.);
            tms.rhs[i] =-(tp->l-tp->s)*tp->right->c[i]*(right_sum-
left_sum) / 2.;
        }

        tmp=tp->right->u[tp->right->n-2];

        tms.lo[tms.dim-1]  = tmpA/(1.-tmp);
        tms.diag[tms.dim-1]=-tmpA/(1.-tmp) - tmpB*(1. - (1.+tmp)/2.) -
(tp->l-tp->future_s)*(1.-tmp)/2.;
        tms.up[tms.dim-1]  = 0;
```

```
        tms.rhs[tms.dim-1] =-tp->right->c[tp->right->n-1]*(tp->l-tp-
>s)*(1.-tmp) / 2.;
    }

    else                                    // NEGATIVE VELOCITY - use
points to the left:
    {
        for(i=1; i<(tms.dim-1); i++)
        {
            left_diff = tp->right->u[i] - tp->right->u[i-1];
            left_sum  = tp->right->u[i] + tp->right->u[i-1];
            right_diff= tp->right->u[i+1] - tp->right->u[i];
            right_sum = tp->right->u[i+1] + tp->right->u[i];

            tms.lo[i]  = tmpA/left_diff - tmpB*(1.- left_sum/2.);
            tms.diag[i]=-tmpA*(1/right_diff + 1/left_diff) + tmpB*(1.
- right_sum/2.)
                        - (tp->l-tp->future_s)*(right_sum - left_sum)
/ 2.;
            tms.up[i]  = tmpA/right_diff;
            tms.rhs[i] =-(tp->l-tp->s)*tp->right->c[i]*(right_sum-
left_sum) / 2.;
        }

        tmp=tp->right->u[tp->right->n-2];

        tms.lo[tms.dim-1]  = tmpA/(1.-tmp) - tmpB*(1. - (1.+tmp)/2.);
        tms.diag[tms.dim-1]=-tmpA/(1.-tmp) - (tp->l-tp->future_s)*(1.-
tmp)/2.;
        tms.up[tms.dim-1]  =0;
        tms.rhs[tms.dim-1] =-tp->right->c[tp->right->n-1]*(tp->l-tp-
>s)*(1.-tmp) / 2.;
    }


// SOLVE tms
    solve_trimatrix_system(tms);


// FREE MEMORY RESERVED FOR tms
    free(tms.lo);
    free(tms.diag);
    free(tms.up);
    free(tms.rhs);
}




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

```
    Takes system from time step j to timestep j+1
*******************************************/
int take_step_planar(two_phase *tp, double dt, double tolerance)
{
    int i, count;
    double error;
    double *tmp_dbl_ptr;


    /************************* BOOKKEEPING
*************************/
    // First prediction of interface position at j+1 is position at j
    tp->future_s = tp->s;
    for(i=0; i<(tp->left->n); i++)
        {tp->left->future_c[i]  = tp->left->c[i];}
    for(i=0; i<(tp->right->n); i++)
        {tp->right->future_c[i] = tp->right->c[i];}



    /************************* TAKE STEP *************************/
    error=tolerance+1;
    count=0;
    while((error>tolerance) || (count==1)) // need at least one
implicit loop
    {

        error=tp->future_s;

        // find new interface position
        new_interface_planar(tp, dt, count);

        // solve for concentration to left of interface:
        new_concentration_left_planar(tp, dt);

        // solve for concentration to right of interface:
        new_concentration_right_planar(tp, dt);

        if(error>tp->future_s)              // (old_estimate >
new_estimate)
            {error = error-tp->future_s;}
        else                                // (new_estimate <=
old_estimate)
            {error = tp->future_s-error;}

        count++;
    }


    /************************* BOOKKEEPING
*************************/
    // Update s:
    tp->old_s = tp->s;
```

```
    tp->s     = tp->future_s;

    if((tp->s<0.) || (tp->s>tp->l))
    {// Interface has moved beyond the ends of the system
        printf("\n\nInterface has moved beyond the end of the
system\n");
        return -1;
    }
    else
    {// Update c:
        tmp_dbl_ptr = tp->left->c;
        tp->left->c        = tp->left->future_c;
        tp->left->future_c = tmp_dbl_ptr;
        tmp_dbl_ptr = tp->right->c;
        tp->right->c        = tp->right->future_c;
        tp->right->future_c = tmp_dbl_ptr;
        return count;
    }
}
```

```
#include <stdlib.h>
#include "trimatrix.h"

/*************************************
 algorithm for solving system of linear equations with tridiagonal
matrix
*************************************/
void solve_trimatrix_system (trimatrix_system tms)
{
    double *alf, *bet;
    int n;
    int i;

    n=tms.dim;
    alf = (double *) calloc(n+1, sizeof(double));
    bet = (double *) calloc(n+1, sizeof(double));

    alf[0]=bet[0]=0.0;
    for( i=0; i<=n-1; i++)
    {
        alf[i+1] = tms.up[i]/(-tms.diag[i] - tms.lo[i]*alf[i] );
```

```
        bet[i+1]=( tms.lo[i]*bet[i] - tms.rhs[i] )/(-tms.diag[i] -
tms.lo[i]*alf[i] );
    }

    tms.c[n-1] = bet[n];
    for( i=n-2; i>=0; i--)
    {
        tms.c[i] = alf[i+1]*tms.c[i+1]+bet[i+1];
    }

    free(alf);
    free(bet);
}
```