Modelización computacional de nanoestructuras: introducción a la espectroscopia teórica

ANEXOS DEL TFG

De

Andrés María Belaza
Promoción 2010-2014

20 de junio de 2014

Director:
Dr. Alberto Castro

Ponente:
Dr. Jose Luis Alonso

Universidad de Zaragoza
Facultad de Ciencias
Departamento de Física Teórica

Anexo A. Formula de Rabi para interacciones armónicas.

0.1. Imagen de interacción

Para poder demostrar fácilmente la expresión de la formula de Rabi introducimos la imagen de interacción. Supongamos que tenemos un Hamiltoniano de la forma siguiente:

$$\mathcal{H} = \mathcal{H}_0 + \mathcal{V}(t) \tag{1}$$

En una base de autoestados del Hamiltoniano sin perturbar, $\mathcal{H}_0$. Y definimos la función de ondas $|\alpha\rangle_T$ en la imagen de interacción a partir de la función de ondas en la imagen de Schröndinger $|\alpha\rangle_S$ de la siguiente manera, manteniendo unidades atómicas, $\hbar = 1$:

$$|\alpha\rangle_T = e^{iH_0 t} |\alpha\rangle_S \tag{2}$$

La interpretación es que la base de la interacción es la base que evoluciona en el tiempo de la forma que evolucionaría la base si no hubiera perturbación. De esta forma, podemos reescribir la ecuación de Schöndinger en la imagen de interacción:

$$i\frac{\partial}{\partial t} |\alpha\rangle_T = -H_0 \cdot e^{iH_0 t} |\alpha\rangle_S + e^{iH_0 t} \left(i\frac{\partial}{\partial t} |\alpha\rangle_S \right) = ... \tag{3}$$
$$... = -H_0 \cdot e^{iH_0 t} |\alpha\rangle_S + e^{iH_0 t} H_0 |\alpha\rangle_S + e^{iH_0 t} V |\alpha\rangle_S$$

Pero el Hamiltoniano sin perturbar conmuta con su exponencial así que:

$$i\frac{\partial}{\partial t} |\alpha\rangle_T = e^{iH_0 t} V e^{-iH_0 t} |\alpha\rangle_I$$

$$i\frac{\partial}{\partial t} |\alpha\rangle_T = V' |\alpha\rangle_I \tag{4}$$

Con $V' = e^{iH_0 t} V e^{-iH_0 t}$ el potencial en la imagen de interacción. Con esto podemos ahora calcular como varian las distintas componentes de una base a lo largo del tiempo.En general, cualquier función de Ondas puede ser desarrollada en función de elementos de la base:

$$|\alpha\rangle_T = \sum_n c_n(t) |n\rangle \tag{5}$$

Y si aplicamos la ecuación de Schöndinger a estos coeficientes

$$i\frac{\partial}{\partial t} \sum_n c_n(t) |n\rangle = e^{iH_0 t} V e^{-iH_0 t} \sum_m c_m(t) |m\rangle$$

$$i\sum_n \dot{c}_n(t) |n\rangle = e^{iH_0 t} V \left(\sum_m e^{-iE_m t} |m\rangle c_m(t) \right)$$

Si multiplico por la izquierda el autovector $\langle n|$

$$i\dot{c}_n = \langle n| e^{iH_0 t} V \left(\sum_m e^{-iE_m t} |m\rangle c_m(t) \right) = e^{-iE_n t} \langle n| V \left(\sum_m e^{-iE_m t} |m\rangle c_m(t) \right)$$

Obteniendo asi una expresión para la evolución de los coeficientes

$$\dot{c}_n = -i \sum_m e^{i(E_n - E_m)t} \langle n| V |m\rangle c_m(t) \tag{6}$$

0.2. Expresión de Rabi

Ahora estamos preparados para plantear nuestro problema de dos niveles. Vamos a resolverlo para $c_1(t=0) = 1$, $c_2(t=0) = 0$ y para una interacción del tipo. $\langle 1| V |1\rangle = \langle 2| V |2\rangle = 0$, $\langle 1| V |2\rangle = W_{12} e^{i\omega t}$ y $\langle 2| V |1\rangle = (\langle 1| V |2\rangle)^*$. A partir de (6) obtenemos:

$$\begin{aligned} \dot{c}_1 &= -ie^{i(E_1 - E_2)t} W_{12} e^{i\omega t} c_2 = -iW_{12} e^{i\Delta t} \\ \dot{c}_2 &= -iW_{12} e^{-i\Delta t} \end{aligned} \tag{7}$$

Con $\Delta \equiv \omega - (E_2 - E_1)$. Si volvemos a derivar para $\dot{c}_1$

$$\ddot{c}_1 = \Delta W_{12} e^{i\Delta t} c_2 + -iW_{12} e^{i\Delta t} \dot{c}_2 = \Delta \frac{1}{-i} \left(-iW_{12} e^{i\Delta t} c_2 \right) + -iW_{12} e^{i\Delta t} \left(-iW_{12} e^{-i\Delta t} \right) = i\Delta \dot{c}_1 - W_{12}^2 c_1$$

$$\ddot{c}_1 - i\Delta \dot{c}_1 + W_{12}^2 c_1 = 0$$

Que presenta soluciones del tipo $c_1 = Ae^{\lambda_+ t} + Be^{\lambda_- t}$ donde λ_+ y λ_- son las soluciones del polinomio característico:

$$\lambda_\pm = \frac{i}{2}(\Delta \pm \sqrt{\Delta^2 + 4W_{12}^2}) = i(\frac{\Delta}{2} \pm \alpha)$$

Donde, para evitar arrastrar la raíz, hemos definido $\alpha \equiv \sqrt{\frac{\Delta^2}{4} + W_{12}^2}$. Para calcular c_2, basta con aplicar (7), y derivando c_1.

$$\begin{aligned} c_1 &= e^{i\frac{\Delta}{2}t} \left(Ae^{i\alpha t} + Be^{-i\alpha t} \right) \\ c_2 &= \frac{1}{W_{12}} e^{i\frac{\Delta}{2}t} \left(-A(\alpha + \frac{\Delta}{2}) e^{i\alpha t} + B(\alpha - \frac{\Delta}{2}) e^{-i\alpha t} \right) \end{aligned}$$

Si aplicamos las condiciones iniciales $c_1 = 1$, $c_2 = 0$, podemos obtener$A = \frac{\alpha - \frac{\Delta}{2}}{2\alpha}$ y $B = \frac{\alpha + \frac{\Delta}{2}}{2\alpha}$

Nuestro interés cae en la probabilidad de encontrar en el autoestado $|2\rangle$ a un tiempo T, es decir $|c_2(T)|^2$, por lo que primero calculamos $c_2(T)$

$$c_2 = \frac{1}{W_{12}} e^{i\frac{\Delta}{2}t} \left(-\frac{\alpha - \frac{\Delta}{2}}{2\alpha} (\alpha + \frac{\Delta}{2}) e^{i\alpha t} + \frac{\alpha + \frac{\Delta}{2}}{2\alpha} (\alpha - \frac{\Delta}{2}) e^{-i\alpha t} \right) = \frac{-i}{W_{12}\alpha} \left(\alpha^2 - \frac{\Delta^2}{4} \right) \left(\frac{e^{i\alpha t} - e^{i\alpha t}}{2i} \right)$$

Insertando la definición de α y sustituyendo la definición de la funcion seno a partir de la exponencial compleja:

$$c_2 = \frac{-iW_{12}}{\sqrt{\frac{\Delta^2}{4} + W_{12}^2}} e^{i\frac{\Delta}{2}t} \sin\left(\sqrt{\frac{\Delta^2}{4} + W_{12}^2}t\right) \tag{8}$$

Calculando su cuadrado, obtenemos la expresión de la formula de Rabi para este tipo de oscilación.

$$P_{12} = |c_2|^2 = \frac{W_{12}^2}{\frac{\Delta^2}{4} + W_{12}^2} \sin^2\left(\sqrt{\frac{\Delta^2}{4} + W_{12}^2}t\right) \tag{9}$$

Anexo B. Implementación computacional en codigo C.

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <gsl/gsl_multimin.h>

#define SIZE 2
#define DT 0.00001
#define TMAX 20
#define PI 3.1415
#define CONTROL 1000

//La estructura numero complejo
struct complex
{   double re;
    double im;
} typedef complex;


complex H[SIZE][SIZE];
complex dHomega[SIZE][SIZE]; //Derivadas respecto a las
    variables
complex dHlam[SIZE][SIZE];
complex H_0[SIZE][SIZE];//Hamiltoniano Molecular
complex W[SIZE][SIZE];//Perturbacion
complex U[SIZE][SIZE];//Operador evolucion
complex Op[SIZE][SIZE];//Operador a medir
complex PHICERO[SIZE];

complex I;

double MiF(double *parametros);
void MiDF(double *parametros, double *gradiente);
void actualiza(double *parametros, double tiempo);
void actualizaevolucion(void);
void actualizaevolucion2(void);

complex multiplica (complex, complex);
complex conjugado(complex);

//----------ESTRUCTURAS PARA LA GSL. ---------
//--NOTA: Las cambio de signo porque el algoritmo busca un
    minimo, y nosotros buscamos un maximo.
double my_f (const gsl_vector *v, void *params)
{
  double *p;
```

```
44
45    p = (double *)malloc(2*sizeof(double));
46
47  p[0] = gsl_vector_get(v, 0);
48  p[1] = gsl_vector_get(v, 1);
49   return -MiF(p);
50  }
51  // The gradient of f, df = (df/dx, df/dy).
52  void my_df (const gsl_vector *v, void *params, gsl_vector *df)
53  {
54    double *p;
55    double *grad;
56  p = (double *)malloc(2*sizeof(double));
57  grad = (double *)malloc(2*sizeof(double));
58
59  p[0] = gsl_vector_get(v, 0);
60  p[1] = gsl_vector_get(v, 1);
61
62   MiDF(p,grad);
63  grad[0]=-grad[0];
64  grad[1]=-grad[1];
65
66  gsl_vector_set(df, 0, grad[0]);
67  gsl_vector_set(df, 1, grad[1]);
68  }
69
70
71  /* Compute both f and df together. */
72  void my_fdf (const gsl_vector *x, void *params, double *f,
        gsl_vector *df)
73  {
74  *f = my_f(x, params);
75  my_df(x, params, df);
76  }
77
78
79  //
        ----------------------------------------------------------------------------------------
80
81
82  int main()
83  {
84    int i,j;
85
86     I.re=0;
87     I.im=1;
88
89     //Inicializaciones de las matrices.
90     for(i=0;i<SIZE;i++)
```

```
91          {for(j=0;j<SIZE;j++)
92              {H[i][j].re=0;
93              H[i][j].im=0;
94              W[i][j].re=H_0[i][j].re=0;
95              W[i][j].im=H_0[i][j].im=0;
96              Op[i][j].re=Op[i][j].im=0;
97              dHomega[i][j].re=dHlam[i][j].re=0;
98              dHomega[i][j].im=dHlam[i][j].im=0;
99              }
100         }
101     H_0[0][0].re=0;
102     H_0[1][1].re=1;
103     Op[1][1].re=1;
104
105     PHICERO[0].re=1;
106     PHICERO[0].im=0;
107     PHICERO[1].re=0;
108     PHICERO[1].im=0;
109
110     //.....................
111     printf("EMPEZAMOS\n");
112
113     printf("Empezamos con el calculo del maximo\n");
114
115     gsl_multimin_function_fdf my_func;
116
117     double p[1] = {0};
118     my_func.n = 2; /* number of function components */
119     my_func.f = &my_f;
120     my_func.df = &my_df;
121     my_func.fdf = &my_fdf;
122     my_func.params = (void *)p;
123
124     int iter = 0;
125     int status;
126     const gsl_multimin_fdfminimizer_type *T;
127     gsl_multimin_fdfminimizer *s;
128     gsl_vector *x;
129
130
131     /* Starting point, x = (0.001,0.9) */
132     x = gsl_vector_alloc (2);
133     gsl_vector_set (x, 0, 0.01);
134     gsl_vector_set (x, 1, 0.8);
135
136
137     //Para escoger el Algoritmo de minimizacion entre los
            multiples que permite GSL
138
139     T = gsl_multimin_fdfminimizer_conjugate_fr;
```

```
140       //T = gsl_multimin_fdfminimizer_vector_bfgs2;
141       // T = gsl_multimin_fdfminimizer_conjugate_pr;
142       // T = gsl_multimin_fdfminimizer_steepest_descent;
143
144       s = gsl_multimin_fdfminimizer_alloc (T, 2);
145
146       gsl_multimin_fdfminimizer_set (s, &my_func, x, 0.001, 1e-3)
              ;
147
148       do
149         {
150           iter++;
151           status = gsl_multimin_fdfminimizer_iterate (s);
152           printf ("\ %5d \ %.5f \ %.5f \ %10.5f\n", iter,
                  gsl_vector_get (s->x, 0), gsl_vector_get (s->x, 1)
                  ,-(s->f));
153           if (status)
154             break;
155           status = gsl_multimin_test_gradient (s->gradient, 1e-3)
                  ;
156           if (status == GSL_SUCCESS)
157             { printf ("Minimum found at:\n");
158               printf ("\ %5d \ %.5f \ %.5f \ %10.5f\n", iter,
                      gsl_vector_get (s->x, 0), gsl_vector_get (s->x,
                      1),s->f);
159             }
160         }
161       while (status == GSL_CONTINUE && iter < 100);
162
163       gsl_multimin_fdfminimizer_free (s);
164       gsl_vector_free (x);
165       printf("\n \t FIN\n");
166
167
168     return 0;
169   }
170
171   //...................FUNCIONES AUXILIARES
          ........................
172   complex multiplica(complex a, complex b)
173   {complex tem;
174   tem.re=a.re*b.re-a.im*b.im;
175   tem.im=a.re*b.im+a.im*b.re;
176   return tem;
177   }
178
179   complex conjugado(complex a)
180   {complex tem;
181     tem.im=-a.im;
182     tem.re=a.re;
```

```
183     return tem;
184   }
185
186   //--- FUNCION QUE DETERMINA EL VALOR DEL OPERADOR A TRAVES DE
          LA EVOLUCION
187   double MiF(double *parametros)
188   { double solucion=0;
189     int i, j, k;
190     complex tem,tem2;
191     complex phi[SIZE], phi2[SIZE];
192     double t;
193
194
195     for(i=0;i<SIZE;i++)
196       {
197         phi[i].re=PHICERO[i].re;
198         phi[i].im=PHICERO[i].im;
199       }
200
201
202
203     for(t=0;t<TMAX;t+=DT)
204         {
205            //Calculo el Hamiltoniano en el Instante t
206           actualiza(parametros,t+DT/2);
207
208           //Inicializo phi2 que son las cordenadas de la funcion
                de Ondas, en el paso siguiente.
209           for(i=0;i<SIZE;i++)
210               {
211               phi2[i].re=0;
212               phi2[i].im=0;
213               }
214
215           //Calculo el operador evolucion
216
217           actualizaevolucion();
218
219           //Y lo aplico
220           for(i=0;i<SIZE;i++)
221             {tem.re=0;
222              tem.im=0;
223               for(j=0;j<SIZE;j++)
224                 {tem.re+=(multiplica(U[i][j],phi[j])).re;
225                 tem.im+=(multiplica(U[i][j],phi[j])).im;
226                 }
227              phi2[i].re=phi[i].re+tem.re;
228              phi2[i].im=phi[i].im+tem.im;
229             }
230
```

```
231          //Colocamos la funcion de Ondas al principio del bucle
232          for(i=0;i<SIZE;i++)
233              {
234              phi[i].re=phi2[i].re;
235              phi[i].im=phi2[i].im;
236              }
237
238          }
239
240         solucion=0;
241          for(i=0;i<SIZE;i++)
242            {
243              tem.re=0;
244              tem.im=0;
245            for(j=0;j<SIZE;j++)
246              {
247                tem.re+=multiplica(Op[i][j],phi[j]).re;
248                tem.im+=multiplica(Op[i][j],phi[j]).im;
249              }
250            solucion+=multiplica(conjugado(phi[i]),tem).re;
251            }
252    return solucion;
253  }
254
255  //............FUNCION QUE CALCULA EL GRADIENTE
       ....................................................
256
257
258  void MiDF(double *parametros, double *gradiente)
259  { int i, j, k;
260    complex tem,tem2;
261    complex phi[SIZE], phi2[SIZE];
262    complex xi[SIZE], xi2[SIZE];
263    double t,acumulador;
264
265    for(i=0;i<SIZE;i++)
266      gradiente[i]=0;
267
268
269    for(i=0;i<SIZE;i++)
270      {
271        phi[i].re=PHICERO[i].re;
272        phi[i].im=PHICERO[i].im;
273      }
274
275
276
277    for(t=0;t<TMAX;t+=DT)
278        {
279           //Calculo el Hamiltoniano en el Instante t
```

```
                                                actualiza(parametros,t+
                                                    DT/2);

        //Inicializo phi2 que son las cordenadas de la funcion
            de Ondas, en el paso siguiente.
        for(i=0;i<SIZE;i++)
            {
            phi2[i].re=0;
            phi2[i].im=0;
            }

        //Calculo el operador evolucion

        actualizaevolucion();

        //Y lo aplico
        for(i=0;i<SIZE;i++)
          {tem.re=0;
           tem.im=0;
            for(j=0;j<SIZE;j++)
              {tem.re+=(multiplica(U[i][j],phi[j])).re;
              tem.im+=(multiplica(U[i][j],phi[j])).im;
              }
           phi2[i].re=phi[i].re+tem.re;
           phi2[i].im=phi[i].im+tem.im;
          }

        //Colocamos la funcion de Ondas al principio del bucle
        for(i=0;i<SIZE;i++)
            {
            phi[i].re=phi2[i].re;
            phi[i].im=phi2[i].im;
            }

        }


//---- Calculo Xi en en instante T
        for(i=0;i<SIZE;i++)
          {tem.re=0;
           tem.im=0;
           for(j=0;j<SIZE;j++)
             {tem.re+=multiplica(Op[i][j],phi[j]).re;
              tem.im+=multiplica(Op[i][j],phi[j]).im;
             }
           xi[i].re=tem.re;
           xi[i].im=tem.im;
          }
```

```
328     for(t=TMAX;t>0;t-=DT)
329         {
330            //Calculo el Hamiltoniano en el Instante t
331                                                 actualiza(parametros,t-
                                                        DT/2);
332
333           //Inicializo phi2 que son las cordenadas de la funcion
                 de Ondas, en el paso siguiente.
334           for(i=0;i<SIZE;i++)
335               {
336               phi2[i].re=0;
337               phi2[i].im=0;
338                                                        xi2[i].re=0;
339                                                        xi2[i].im=0;
340               }
341
342                                     //Calculo el operador evolucion
343
344                                     actualizaevolucion2();
345
346                                     //Y lo aplico
347                                     for(i=0;i<SIZE;i++)
348                               {tem.re=0;
349              tem.im=0;
350               for(j=0;j<SIZE;j++)
351                 {tem.re+=(multiplica(U[i][j],phi[j])).re;
352                 tem.im+=(multiplica(U[i][j],phi[j])).im;
353                 }
354              phi2[i].re=phi[i].re+tem.re;
355              phi2[i].im=phi[i].im+tem.im;
356             }
357
358           //Lo mismo para xi
359
360           for(i=0;i<SIZE;i++)
361             {tem.re=0;
362              tem.im=0;
363               for(j=0;j<SIZE;j++)
364                 {tem.re+=(multiplica(U[i][j],xi[j])).re;
365                 tem.im+=(multiplica(U[i][j],xi[j])).im;
366                 }
367              xi2[i].re=xi[i].re+tem.re;
368              xi2[i].im=xi[i].im+tem.im;
369             }
370
371           //Colocamos la funcion de Ondas al principio del bucle
372           for(i=0;i<SIZE;i++)
373               {
374               phi[i].re=phi2[i].re;
375               phi[i].im=phi2[i].im;
```

```
376               xi[i].re=xi2[i].re;
377               xi[i].im=xi2[i].im;
378               }
379
380
381           //----ACUMULAMOS PARA CALCULAR LA INTEGRAL
382           acumulador=0;
383           for(i=0;i<SIZE;i++)
384             {
385               tem.re=0;
386               tem.im=0;
387               for(j=0;j<SIZE;j++)
388                 {tem.re+=multiplica(dHlam[i][j],phi[j]).re;
389                  tem.im+=multiplica(dHlam[i][j],phi[j]).im;
390                 }
391               acumulador+=multiplica(conjugado(xi[i]),tem).im;
392             }
393
394           gradiente[0]+=acumulador;
395
396           acumulador=0;
397           for(i=0;i<SIZE;i++)
398             {
399               tem.re=0;
400               tem.im=0;
401               for(j=0;j<SIZE;j++)
402                 {tem.re+=multiplica(dHomega[i][j],phi[j]).re;
403                  tem.im+=multiplica(dHomega[i][j],phi[j]).im;
404                 }
405               acumulador+=multiplica(conjugado(xi[i]),tem).im;
406             }
407
408           gradiente[1]+=acumulador;
409
410
411           }
412
413         for(i=0;i<2;i++)
414            {gradiente[i]=-2*DT*gradiente[i];}
415
416
417   }
418
419   //.........ACTUALIZADOR DE LOS OPERADORES AL INSTANTE T
420
421   void actualiza(double *parametros, double t)
422   {int i,j;
423     double LAM=parametros[0];
424     double OMEGA=parametros[1];
425     W[0][1].re=cos(OMEGA*t)*LAM;
```

```
426     W[0][1].im=sin(OMEGA*t)*LAM;
427     W[1][0].re=cos(OMEGA*t)*LAM;
428     W[1][0].im=-sin(OMEGA*t)*LAM;
429
430     //Actualizo las derivadas para el gradiente
431
432     dHomega[0][1].re=-sin(OMEGA*t)*LAM*t;
433     dHomega[0][1].im=cos(OMEGA*t)*LAM*t;
434     dHomega[1][0].re=-sin(OMEGA*t)*LAM*t;
435     dHomega[1][0].im=-cos(OMEGA*t)*LAM*t;
436
437     dHlam[0][1].re=cos(OMEGA*t);
438     dHlam[0][1].im=sin(OMEGA*t);
439     dHlam[1][0].re=cos(OMEGA*t);
440     dHlam[1][0].im=-sin(OMEGA*t);
441
442
443   for(i=0;i<SIZE;i++)
444               {for(j=0;j<SIZE;j++)
445                   {
446                     H[i][j].re=H_0[i][j].re+W[i][j].re;
447                     H[i][j].im=H_0[i][j].im+W[i][j].im;
448                   }
449               }
450
451
452   }
453
454
455   //.......ACTUALIZA EL OPERADOR EVOLUCION
456   void actualizaevolucion(void)
457   {int i, j,k;
458     complex tem[SIZE][SIZE],tem2[SIZE][SIZE];
459     complex sum;
460    for(i=0;i<SIZE;i++)
461       for(j=0;j<SIZE;j++)
462         {
463           U[i][j].re=0;
464          U[i][j].im=0;
465          tem[i][j].re=0;
466          tem[i][j].im=0;
467          tem[i][j].re=0;
468          tem[i][j].im=0;
469         }
470
471
472
473    //Calculo el termino lineal y se lo a ado al operador
          evolucion
474    for(i=0;i<SIZE;i++)
```

```
475       for(j=0;j<SIZE;j++)
476         {
477          tem[i][j].re=-(multiplica(H[i][j],I)).re*DT;
478          tem[i][j].im=-(multiplica(H[i][j],I)).im*DT;
479         }
480
481    for(i=0;i<SIZE;i++)
482      for(j=0;j<SIZE;j++)
483           U[i][j]=tem[i][j];
484
485    //Calculo el termino de orden 2 y se lo a ado al operadoe
         Evolucion:
486
487    for(i=0;i<SIZE;i++)
488       for(j=0;j<SIZE;j++)
489         {
490           sum.re=0;
491           sum.im=0;
492           for(k=0;k<SIZE;k++)
493             {
494               sum.re+=tem[i][k].re*H[k][j].re*DT;
495               sum.im+=tem[i][k].im*H[k][j].im*DT;
496             }
497           tem2[i][j].re=-(multiplica(sum,I)).re*0.5;
498           tem2[i][j].im=-(multiplica(sum,I)).im*0.5;
499         }
500
501    for(i=0;i<SIZE;i++)
502      for(j=0;j<SIZE;j++)
503        {
504          U[i][j].re+=tem2[i][j].re;
505          U[i][j].im+=tem2[i][j].im;
506        }
507
508    //Orden 3
509
510    for(i=0;i<SIZE;i++)
511       for(j=0;j<SIZE;j++)
512         {
513           sum.re=0;
514           sum.im=0;
515           for(k=0;k<SIZE;k++)
516             {
517               sum.re+=tem2[i][k].re*H[k][j].re*DT;
518               sum.im+=tem2[i][k].im*H[k][j].im*DT;
519             }
520           tem[i][j].re=-(multiplica(sum,I)).re/3;
521           tem[i][j].im=-(multiplica(sum,I)).im/3;
522         }
523
```

```
524    for(i=0;i<SIZE;i++)
525      for(j=0;j<SIZE;j++)
526        {
527          U[i][j].re+=tem[i][j].re;
528          U[i][j].im+=tem[i][j].im;
529        }
530
531    //Orden 4
532
533   for(i=0;i<SIZE;i++)
534       for(j=0;j<SIZE;j++)
535         {
536           sum.re=0;
537           sum.im=0;
538           for(k=0;k<SIZE;k++)
539             {
540               sum.re+=tem[i][k].re*H[k][j].re*DT;
541               sum.im+=tem[i][k].im*H[k][j].im*DT;
542             }
543           tem2[i][j].re=-(multiplica(sum,I)).re/4;
544           tem2[i][j].im=-(multiplica(sum,I)).im/4;
545         }
546
547    for(i=0;i<SIZE;i++)
548      for(j=0;j<SIZE;j++)
549        {
550          U[i][j].re+=tem2[i][j].re;
551          U[i][j].im+=tem2[i][j].im;
552        }
553   }
554
555
556   //..........ACTUALIZA EL OPERADOR DE EVOLUCION TEMPORAL HACIA
         ATRAS.......
557
558   void actualizaevolucion2(void)
559   {int i, j,k;
560     complex tem[SIZE][SIZE],tem2[SIZE][SIZE];
561     complex sum;
562    for(i=0;i<SIZE;i++)
563       for(j=0;j<SIZE;j++)
564         {
565           U[i][j].re=0;
566          U[i][j].im=0;
567          tem[i][j].re=0;
568          tem[i][j].im=0;
569          tem[i][j].re=0;
570          tem[i][j].im=0;
571         }
572
```

```
573
574
575     //Calculo el termino lineal y se lo a ado al operador
            evolucion
576     for(i=0;i<SIZE;i++)
577        for(j=0;j<SIZE;j++)
578          {
579            tem[i][j].re=-(multiplica(H[i][j],I)).re*(-DT);
580            tem[i][j].im=-(multiplica(H[i][j],I)).im*(-DT);
581          }
582
583     for(i=0;i<SIZE;i++)
584       for(j=0;j<SIZE;j++)
585            U[i][j]=tem[i][j];
586
587     //Calculo el termino de orden 2 y se lo a ado al operadoe
            Evolucion:
588
589     for(i=0;i<SIZE;i++)
590        for(j=0;j<SIZE;j++)
591          {
592            sum.re=0;
593            sum.im=0;
594            for(k=0;k<SIZE;k++)
595              {
596                sum.re+=tem[i][k].re*H[k][j].re*(-DT);
597                sum.im+=tem[i][k].im*H[k][j].im*(-DT);
598              }
599            tem2[i][j].re=-(multiplica(sum,I)).re*0.5;
600            tem2[i][j].im=-(multiplica(sum,I)).im*0.5;
601          }
602
603     for(i=0;i<SIZE;i++)
604       for(j=0;j<SIZE;j++)
605         {
606           U[i][j].re+=tem2[i][j].re;
607           U[i][j].im+=tem2[i][j].im;
608         }
609
610     //Orden 3
611
612     for(i=0;i<SIZE;i++)
613        for(j=0;j<SIZE;j++)
614          {
615            sum.re=0;
616            sum.im=0;
617            for(k=0;k<SIZE;k++)
618              {
619                sum.re+=tem2[i][k].re*H[k][j].re*(-DT);
620                sum.im+=tem2[i][k].im*H[k][j].im*(-DT);
```

```
            }
          tem[i][j].re=-(multiplica(sum,I)).re/3;
          tem[i][j].im=-(multiplica(sum,I)).im/3;
        }

   for(i=0;i<SIZE;i++)
     for(j=0;j<SIZE;j++)
       {
         U[i][j].re+=tem[i][j].re;
         U[i][j].im+=tem[i][j].im;
       }

   //Orden 4

  for(i=0;i<SIZE;i++)
      for(j=0;j<SIZE;j++)
        {
          sum.re=0;
          sum.im=0;
          for(k=0;k<SIZE;k++)
            {
              sum.re+=tem[i][k].re*H[k][j].re*(-DT);
              sum.im+=tem[i][k].im*H[k][j].im*(-DT);
            }
          tem2[i][j].re=-(multiplica(sum,I)).re/4;
          tem2[i][j].im=-(multiplica(sum,I)).im/4;
        }

   for(i=0;i<SIZE;i++)
     for(j=0;j<SIZE;j++)
       {
         U[i][j].re+=tem2[i][j].re;
         U[i][j].im+=tem2[i][j].im;
       }
  }
```