UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE CIÊNCIAS EXATAS E DA NATUREZA
PROGRAMA DE PÓS-GRADUAÇÃO EM FÍSICA

HUGO ESTEVAO GOMES DE BRITTO RAMOS

Otimização de erros numéricos gerados em simulações computacionais de espectroscopia direta com pentes de frequência com átomos de Rubídio

Recife

2023

HUGO ESTEVAO GOMES DE BRITTO RAMOS

Otimização de erros numéricos gerados em simulações computacionais de espectroscopia direta com pentes de frequência com átomos de Rubídio

Trabalho apresentado ao Programa de Pós-graduação em Física do Centro de Ciências Exatas e da Natureza da Universidade Federal de Pernambuco, como requisito parcial para obtenção do grau de Mestre em Física

Área de Concentração: ÓPTICA

Orientador (a): MARCIO HERACLYTO GONÇALVES DE MIRANDA

Coorientador (a): LUCIO HORA ACIOLI

Recife

2023

Catalogação na fonte
Bibliotecária Mônica Uchôa, CRB4-1010

R173o Ramos, Hugo Estevao Gomes de Britto
Otimização de erros numéricos gerados em simulações computacionais de espectroscopia direta com pentes de frequência com átomos de rubídio / Hugo Estevao Gomes de Britto Ramos. – 2023.
64 f.: il., fig.

Orientador: Marcio Heraclyto Gonçalves de Miranda.
Dissertação (Mestrado) – Universidade Federal de Pernambuco. Centro de Ciências Exatas e da Natureza. Programa de Pós-graduação em Física. Recife, 2023.
Inclui referências e apêndices.

1.Óptica. 2. Pentes de frequência. 3. Espectroscopia. 4. Métodos numéricos. I. Miranda, Marcio Heraclyto Gonçalves de (Orientador). II. Título.

530 CDD (23. ed.) UFPE - CCEN 2024 – 003

HUGO ESTEVAO GOMES DE BRITTO RAMOS

OTIMIZAÇÃO DE ERROS NUMÉRICOS GERADOS EM SIMULAÇÕES COMPUTACIONAIS DE ESPECTROSCOPIA DIRETA COM PENTES DE FREQUÊNCIA COM ÁTOMOS DE RUBÍDIO

Dissertação apresentada ao Programa de Pós-Graduação em Física da Universidade Federal de Pernambuco, como requisito parcial para a obtenção do título de Mestre em Física.

Aprovada em: 07/12/2023.

BANCA EXAMINADORA

_____
Prof. Dr. Marcio Heraclyto Gonçalves de Miranda
Orientador
Universidade Federal de Pernambuco

_____
Prof. Dr. Daniel Felinto Pires Barbosa
Examinador Interno
Universidade Federal de Pernambuco

_____
Prof. Dr. Sérgio Ricardo Muniz
Examinador Externo
Universidade de São Paulo

AGRADECIMENTOS

Várias pessoas foram muito bondosas e pacientes em me ajudar com esta dissertação. Agradeço minha namorada, Raira, que fez muito(e um pouco mais) por mim. Aos meus pais Edilene e Estevão, que sempre tiveram a melhor das intenções e procuraram fazer o melhor por mim. Ao professor Marcio Heraclyto, meu orientador, por sua infinita paciência e compreensão, e por sempre avaliar meus textos e estudos, e sempre estar presente para guiar meus próximos passos quando eu precisava. A Lucas Melo e Cecília Veras, que com muita gentileza me apresentaram a parte computacional e experimental do laboratório, respectivamente. Agradeço em especial ao Lucas, que me ajudou várias vezes, e de todas as formas com dicas de programação e analisando meus códigos. Ao professor Daniel Felinto, que escreveu o artigo no qual este trabalho é baseado e que propôs ideias para este estudo, e sempre foi muito solícito em fazer reuniões para discutir o projeto. Agradeço também aos funcionários do Departamento de Física e da UFPE, que fazem funcionar um lugar de importância fundamental ao país. Agradeço os órgãos de fomento à ciência Capes e CNPQ. Em especial à CNPQ.

Agradeço aos colegas e amigos do departamento, com os quais tive frequências variadas de contato, espero ter ajudado-os com suas graduações e mestrados em pelo menos 10 % do que me ajudaram com dicas, aulas, boas conversas, risadas e humanidade. Sinto muita saudade de todos.

Metade de mim não acreditava que este trabalho seria concluído, e a outra metade tinha certeza, mas aqui estou finalmente prestes a apresentá-lo. Apesar de ser simples, significa demais para mim tê-lo escrito do início ao fim. Muito obrigado a todos.

RESUMO

O presente trabalho visa otimizar um problema de espectroscopia utilizando átomos de rubídio com pentes de frequência. É utilizado um método computacional para que sejam feitos cálculos de espectroscopia direta com pentes de frequência baseados em um desenvolvimento teórico que considera átomos de vários níveis e ordens de perturbação arbitrárias. Foram realizadas modificações em programas previamente desenvolvidos visando obter resultados mais precisos. É apresentado um panorama de como os programas foram implementados com novas instruções e dos métodos numéricos envolvidos, bem como sua forma de otimização. Os programas originais foram desenvolvidos por Felinto e López para o artigo Theory for direct frequency-comb spectroscopy(2009) no qual foi proposta a teoria base para este trabalho. Neste trabalho foram derivadas expressões gerais para a dinâmica de sistemas atômicos. Por fim são apresentados os resultados comparando-os com os do artigo de 2009 nos quais serão demonstrados ganhos de precisão.

Palavras-chave: óptica; pentes de frequência; espectroscopia; métodos numéricos.

ABSTRACT

This work goes towards optimizing a spectroscopy problem using Rubidium atoms with frequency combs. By employing a computational method to do direct frequency comb spectroscopy, based on a theoretical development that considers atoms of an arbitrary number of levels, and arbitrary perturbations orders. Modifications were made on C codes previously developed in order to obtain more precise results. A landscape of how the codes were modificated and implemented is presented, and also how the numeric methods work. The original codes were developed by Felinto and López for the article Theory for Direct Frequency-Comb Spectroscopy(2009) in which the base theory and codes for this work are presented. In this article general expressions for the dynamics of atomic systems are derived. At last, results are presented and compared to the ones in the 2009 article in which we demonstrate precision gains.

Keywords: Optics. Frequency Combs. Spectroscopy. Numerical Methods.

LISTA DE FIGURAS

SUMÁRIO

1 INTRODUÇÃO

Criados a mais de 20 anos para contar os ciclos de relógios atômicos (FORTIER; BAUMANN, 2019), os pentes de frequência são gerados a partir de lasers no regime de modos travados, e possuem uma série de características que os fazem capazes de conectar medidas de precisão espectroscópica e dinâmicas ultra rápidas. Por exemplo, esses lasers geram pulsos ultracurtos (tipicamente da ordem de fs) com uma relação fixa de fase entre si (YE; CUNDIFF, 2005). Chamamos esta estrutura de um "trem de pulsos". No domínio das frequências, isso se traduz em uma série de linhas de frequência discretas (geralmente da ordem de 10^6 linhas), uniformemente espaçadas (que lembram a aparência de um pente) e com grande largura espectral.

Pentes de frequência podem ser utilizados para realizar espectroscopia da dinâmica atômica, em um processo chamado Espectroscopia Direta por Pente de Frequência. Com um controle preciso do domínio temporal e espectral de um trem de pulsos, é possível se fazer estudos espectroscópicos com um pente de frequêcias (STOWE et al., 2008), e neste trabalho estaremos analisando este processo do ponto de vista teórico e computacional.

O presente trabalho foi realizado para otimizar uma solução de um problema teórico e computacional envolvendo átomos de rubídio através de espectroscopia direta com pentes de frequência. O problema físico consiste em calcular as populações em uma amostra de ^{87}Rb, que ao ser interrogada por um pente de frequências, sofre uma série de transições de dois fótons do estado fundamental $5S_{1/2}$ para $5P_{3/2}$ e $5D_{5/2}$.

Técnicas computacionais utilizando programas em C foram utilizadas para obter resultados numéricos para este problema (FELINTO; LÓPEZ, 2009). A partir disso, percebemos a possibilidade de melhorar a eficácia dos programas utilizados, diminuindo os erros que ocorrem no processo de cálculo numérico. Com isto, conseguimos obter resultados mais precisos com erros na ordem de 10^{-15}.

No capítulo 2, é detalhado um desenvolvimento teórico da ref. (FELINTO; LÓPEZ, 2009) que considera átomos de múltiplos níveis e ordens de perturbação arbitrárias. No capítulo 3, são detalhados os métodos numéricos e como foram realizadas modificações em programas previamente desenvolvidos visando obter resultados mais precisos. Os programas originais foram desenvolvidos por Felinto e López para o artigo Theory for direct frequency-comb spectroscopy (2009) (FELINTO; LÓPEZ, 2009) no qual foi também proposta a teoria base para este trabalho. Por fim, no capítulo 4 são apresentados os resultados comparando-os com os do trabalho

original nos quais serão demonstrados se existiram ou não ganhos de precisão, i.e, diminuição de erros. E no capítulo 5 é exposta a conclusão do trabalho, e também são discutidas alguns caminhos que podem ser seguidos a partir dos resultados deste trabalho.

2 DESENVOLVIMENTO TEÓRICO

2.1 INTRODUÇÃO

Neste capítulo será apresentado o ferramental teórico utilizado ao longo do trabalho. Serão comentados temas de Equações de Bloch para átomos de múltiplos níveis através do formalismo da matriz densidade, medidas de frequências com Pentes de Frequência e uma teoria que descreve espectroscopia com pentes de frequência(FELINTO; LÓPEZ, 2009) que será fundamental para as soluções numéricas dos próximos capítulos.

2.2 EQUAÇÕES ÓPTICAS DE BLOCH

As equações ópticas de Bloch são usadas para descrever a dinâmica temporal de populações atômicas entre diferentes níveis de energia quando há interação do átomo com a radiação. Elas são importantes para estudar o comportamento de sistemas atômicos. Aqui essas equações são apresentadas e cada um dos seus termos é explicado. Elas são a base do formalismo teórico que será desenvolvido a seguir e suas propriedades serão amplamente usadas para discutir os resultados dos próximos capítulos.

Usaremos então o formalismo da matriz de densidade para apresentar as equações:

$$\frac{\partial \rho_{ij}}{\partial t} = -i/\hbar \langle i|[H,\rho]|j\rangle - \Gamma_{ij}\rho_{ij} + \delta_{ij}\Sigma_r \gamma_{ir}\rho_{rr}. \tag{2.1}$$

Na equação acima, Γ_{ij} é a taxa de relaxação do componente ij da matriz densidade, δ_{ij} é o delta de Kronecker e o somatório representa a acumulação incoerente do i-ésimo nível pela população dos r níveis superiores. Como estamos estudando um sistema de múltiplos níveis, essas equações neste formato são fundamentais para o desenvolvimento da teoria. Poderemos com elas descrever efeitos coerentes e decoerentes, relaxação, decaimento, entre outros.

2.3 INFORMAÇÕES PERTINENTES SOBRE O ^{87}Rb

Considerando que o Rubídio 87 foi a espécie atômica utilizada neste trabalho, é conveniente traçar um panorama sobre suas características básicas. O átomo de Rubídio 87, possui 37 elétrons com apenas um na camada mais externa.

Sendo a transição $5S_{1/2} \rightarrow 5P_{3/2}$ cíclica, ela é frequentemente utilizada para o resfriamento de armadilhas magneto-ópticas.

As transições $5S_{1/2} \rightarrow 5P_{3/2}$ e $5S_{1/2} \rightarrow 5P_{1/2}$ possuem separação hiperfina de níveis, resultantes do acoplamento entre o momento angular de spin do elétron, $\vec{S}$, o momento angular orbital, $\vec{L}$ e o momento angular de spin do núcleo, $\vec{I}$. Temos

$$\vec{J} = \vec{L} + \vec{S}, \tag{2.2}$$

a partir desta equação acoplamos o momento angular de spin $\vec{S}$ e orbital $\vec{L}$, o que resulta no momento angular total do elétron $\vec{J}$. Então acoplamos $\vec{J}$ ao momento angular do núcleo:

$$\vec{F} = \vec{J} + \vec{I}, \tag{2.3}$$

onde o módulo de $\vec{F}$ deve satisfazer

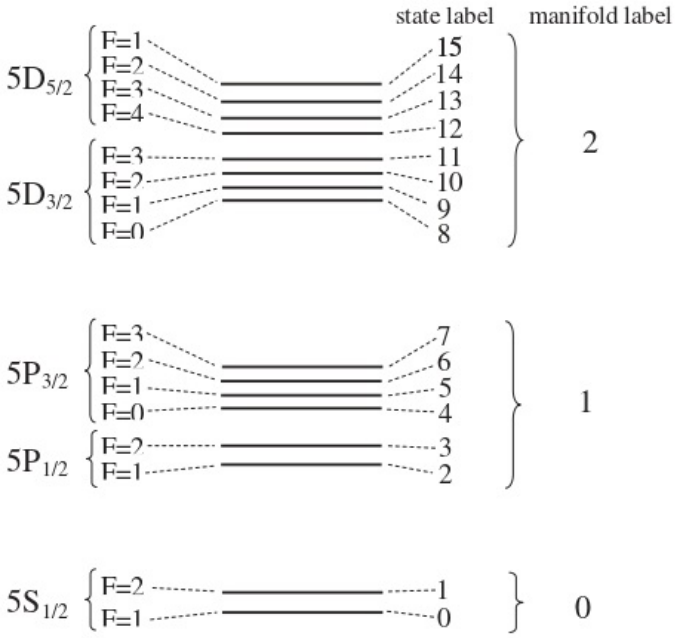
$$|J - I| \leq F \leq |J + I|, \tag{2.4}$$

com $\vec{F}$ o momento angular total. Com isso, podemos analisar os possíveis valores de F e seu efeito nos níveis de energia. Por exemplo, para o estado $5S_{1/2}$ temos F=1 ou F=2, para $5P_{1/2}$ F pode ser 1 ou 2 e o estado $5P_{3/2}$ pode ter F = 0, 1, 2 ou 3.

Abaixo, na figura 1 estão esquematizados os níveis de energia separados em 3 grupos (ou variedades):

Temos um total de 16 estados hiperfinos que englobam a transição $5S \rightarrow 5P \rightarrow 5D$. Os 3 grupos são enumerados em 0, 1 e 2 e servem para o cálculo de matrizes Φ_n que serão apresentadas adiante neste capítulo na seção 2.5 no termo M dentro da exponencial da equação (2.22).

Figura 1 – Níveis de energia do Rb^{87}

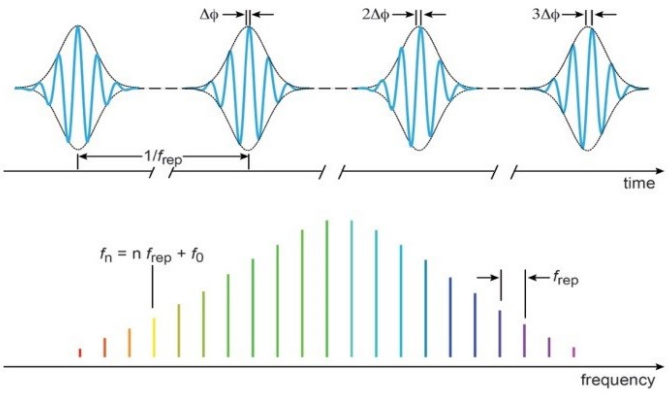


Fonte: (FELINTO; LÓPEZ, 2009)

2.4 PENTES DE FREQUÊNCIA

Um pente de frequência é formado quando um laser entra em um regime de operação chamado de modos travados, este tipo de laser possui um espectro de frequência com múltiplos modos longitudinais equidistantes de acordo com a taxa de repetição do laser (STOWE et al., 2008). Neste regime, o laser emite um trem de pulsos ultracurtos, tipicamente da ordem de centenas de fs, o que resulta na formação do espectro de frequências em formato de "pente".

Figura 2 – Definição esquemática de um Pente de Frequências no domínio do tempo(acima) e da frequência(abaixo)



Fonte: (PICQUÉ; HÄNSCH, 2019)

Quando temos centenas de milhares de modos longitudinais na cavidade óptica do laser sendo somados de maneira coerente, um pente de frequência é formado e uma das suas mais importantes características é que a mudança de fase de um pulso para outro é constante, ou seja, os modos ópticos são correlacionados harmonicamente (FORTIER; BAUMANN, 2019). Estas propriedades conferem aos pentes de frequência uma alta precisão e um grande intervalo de frequências, isso permite a medição simultânea de múltiplas frequências. Os principais resultados práticos disso estão na área da espectroscopia e metrologia óptica, onde os pentes são utilizados como um medidor de frequência, ou uma "régua". As linhas do espectro do pente são estabilizadas com algum padrão de frequência, como por exemplo, a frequência de transição de um átomo de Cs, o que permite medidas de alta precisão de frequências, com largas faixas espectrais (PICQUÉ; HÄNSCH, 2019). Isso é realizado, por exemplo, travando as frequências de offset e repetição com um relógio de Césio, o que resulta nos modos ópticos serem referenciados à unidade de segundo do SI, tornando possível determinar frequências ópticas de transição.

As frequências ópticas dos modos longitudinais do pente são escritas como:

$$\nu_n = nf_{rep} + f_0, \tag{2.5}$$

onde é um número inteiro da ordem de 10^6 (tipicamente o número de linhas de frequências), f_0 é a frequência de offset e f_{rep} é a frequência de repetição. Esta equação representa a que talvez seja a mais importante característica dos pentes de frequência: a uniformidade em que os picos de frequência aparecem no formato de "pente". Até o momento, não foi detectado nenhum desvio nesta expressão (UDEM; HOLZWARTH; HÄNSCH, 2009). Temos também a equação que conecta a frequência de offset e a mudança de fase entre pulsos, nela ϕ é a diferença que existe entre a velocidade de grupo e a velocidade de fase ou entre a envoltória e a portadora,

$$f_0 = (\frac{1}{2\pi})\frac{d\phi}{dt}. \tag{2.6}$$

Podemos integrar em t para obter a variação na fase dos pulsos subsequentes.Para isso, basta integrar no intervalo 0 a T_R, onde $T_R = \frac{1}{f_rep}$ é o tempo de repetição:

$$\int_0^{T_R} dt 2\pi f_0 = \int_0^{T_R} dt\phi(t), \tag{2.7}$$

$$2\pi\frac{f_0}{f_{rep}} = \Delta\phi, \tag{2.8}$$

(CUNDIFF; YE, 2003). Com isso, temos uma conexão clara entre a diferença de fase dos pulsos e a frequência de offset do pente.

2.5 UMA TEORIA PARA A ESPECTROSCOPIA DIRETA COM PENTE DE FREQUÊNCIA

O tratamento da interação de um átomo com um pente de frequ encias começa com o Hamiltoniano do sistema

$$H = H_0 + V(t), \tag{2.9}$$

onde H_0 é o Hamiltoniano do átomo livre e $V(t)$ o potencial de interação com o campo

Iremos substituir este Hamiltoniano nas equações de Bloch, apresentadas na seção 2.1 para chegar às seguintes equações:

$$\frac{\partial\rho_{ij}}{\partial t} = -(i\omega_{ij} + \Gamma_{ij})\rho_{ij} - \frac{i}{\hbar} < i|[V,\rho]|j > +\delta_{ij}\Sigma_r\gamma_{ir}\rho_{rr}, \tag{2.10}$$

onde definimos ω_{ij} como a frequência de transição entre os níveis i e j.

$$\omega_{ij} = (E_i - E_j)/\hbar. \tag{2.11}$$

Iremos agora integrar a equação no tempo para obter uma expressão de $\rho_{ij}(t)$

$$\rho_{ij}(t)e^{(i\omega_{ij}+\Gamma_{ij})t} = \rho_{ij}^0 - i/\hbar\int_0^t\langle i|[V,\rho]|j\rangle dt' + \delta_{ij}\Sigma_r\gamma_{ir}\int_0^t\rho_{rr}(t')dt', \tag{2.12}$$

onde ρ_{ij}^0 é ρ_{ij} em t=0.

Faremos algumas aproximações provenientes do fato de que queremos entender a interação do átomo com cada pulso individualmente. Como estamos lidando com um pulso ultracurto (tipicamente da ordem de fs), a interação entre o pulso e o átomo é mais rápida do que o tempo de relaxação(FELINTO; LÓPEZ, 2009), o que nos permite ignorar sua dependência temporal com a relaxação.

Além disso, queremos entender a correlação entre um estado qualquer que interagiu com n pulsos após a interação dele com o próximo pulso n+1. Podemos fazer isso ao igualar $t = T_R$, o que significa fazer com que o intervalo de integração seja o próprio tempo de repetição do laser. Como dito anteriormente, a interação é muito rápida, então o tempo de repetição

é extremamente longo em relação ao potencial de interação dependente do tempo V, o que equivale ao limite t$\rightarrow \infty$.

Dessa forma temos

$$\rho_{ij}(t)e^{(i\omega_{ij}+\Gamma_{ij})t} = \rho_{ij}^0 - i/\hbar \int_0^t e^{i\omega_{ij}t'}\langle i|[V,\rho]|j\rangle dt' + \delta_{ij}\Sigma_r\gamma_{ir}\int_0^t e^{\Gamma_{ij}t'}\rho_{rr}(t')dt'. \tag{2.13}$$

Definimos ainda mais duas quantidades

$$\rho_{ij}^c = \rho_{ij}^n - i/\hbar \int_0^\infty e^{i\omega_{ij}t'}\langle i|[V^n(t),\rho^c]|j\rangle dt', \tag{2.14}$$

$$I_i = \sum_r \gamma_{ir}\int_0^{T_R} e^{\Gamma_{ii}t'}\rho_{rr}(t'), \tag{2.15}$$

o que nos permite escrever a equação na seguinte forma

$$\rho_{ij}^{n+1} = e^{-(i\omega_{ij}+\Gamma_{ij})T_R}(\rho_{ij}^c + \delta_{ij}I_i). \tag{2.16}$$

É conveniente deixar as equações neste formato, pois ficam em evidência os dois fatores que regem a evolução das populações atômicas. O primeiro é ρ_{ij}^c, a excitação do estado associado à interação com o pulso n+1 (pulso posterior) a partir do estado associado ao pulso anterior n (n-ésimo pulso). O outro é I_i, a redistribuição incoerente que acontece pela emissão espontânea das populações em vários estados. A seguir, fazemos uma explicação mais detalhada de cada efeito separadamente.

A excitação coerente pode ser calculada escrevendo os termos ρ_{ij}^c a partir do operador de evolução temporal U_I^n, que iremos expandir em uma série de Dyson (SAKURAI; COMMINS, 1995):

$$U_I^n = 1 + (-i/\hbar)\int_0^\infty dt' V_I^n(t') + (-i/\hbar)^2\int_0^\infty dt'\int_0^{t'} dt'' V_I^n(t')V_I^n(t'') + ..., \tag{2.17}$$

onde V_I^n é o potencial de interação na representação de Heisenberg (chamada no habitualmente no inglês de "interaction picture") dado por

$$V_I^n(t) = e^{\frac{iH_0t}{\hbar}}V^n(t)e^{\frac{-iH_0t}{\hbar}}. \tag{2.18}$$

Reorganizando os termos da equação para ρ_{ij}^c podemos escrevê-los da seguinte forma:

$$\rho_{ij}^c = < i|U_I^n\rho^n U_I^{n\dagger}|j > = \sum_{k,l}\rho_{k,l}^n < i|U_I^n|k >< l|U_I^{n\dagger}|j >, \tag{2.19}$$

Esta expressão é pertinente, pois conseguimos obter a partir dela os ρ_{ij}^{c}, para isso usamos os elementos da matriz do estado "anterior"ρ_{ij}^{n} e também os elementos $< i|U_I^n|j >= U_{ij}^{n}$ da matriz de evolução temporal, que podem ser todos calculados a partir da série de Dyson apresentada acima.

A partir de agora iremos finalmente incluir um pente de frequência na teoria, e a partir disso obteremos simplificações no cálculo do operador de evolução temporal. Ao analisarmos o potencial de interação de dipolo do n-ésimo pulso com o átomo iremos obter uma expressão bastante razoável para U_I^n. Começamos com o campo elétrico de um pente de frequências:

$$E_n(t) = E_0(t - nT_R)e^{i\omega_0 nT_R}, \tag{2.20}$$

(YE; CUNDIFF, 2005),

onde $\omega_0 = 2\pi f_0$ e n=0 representa o primeiro pulso e assim por diante.

Com isso chegamos a

$$V^n(t) = \sum \mu_{ij} E_0(t - nT_R)e^{i\omega_0 n}|i >< j|. \tag{2.21}$$

Sempre mantendo-se atentos ao fato de que o i no argumento da exponencial é a unidade imaginária e o i no resto das expressões é o índice. Esta equação representa a interação do n-ésimo pulso com o átomo, onde μ_{ij} é o momento de dipolo entre os estados da transição do estado i para o estado j. Como existe um somatório com os índices i e j, a equação engloba todos os estados e transições do sistema e sempre com j>i. Notemos que a única contribuição do índice n é uma mudança de fator de fase de ordem $e^{2\pi nif_0}$. Dessa forma, podemos escrever o operador de evolução temporal do n-ésimo pulso em termos do operador do primeiro pulso com a seguinte equação:

$$U_I^n = \Phi_n U_I^0 \Phi_n^{\dagger}, \tag{2.22}$$

Aqui, Φ_n é uma matriz diagonal cujos elementos são dados da seguinte forma:

a) Para estados j na variedade de menor energia, o primeiro, temos δ_{jk},

b) Para estados j da segunda variedade temos $e^{2\pi nif_0 T_R \delta_{jk}}$,

c) E por fim, estados j da variedade M+1, ou seja, que tenham energia maior do que estados da variedade M temos $e^{2\pi nMif_0 T_R \delta_{jk}}$.

Com isso, temos uma conexão direta entre o operador U de qualquer pulso e o operador do primeiro pulso.

Isso conclui a primeira parte do cálculo da equação 2.16. Discutiremos agora o termo à direita da soma, que trata da distribuição incoerente gerada pela emissão espontânea entre vários estados diferentes. Para calcularmos a expressão 2.15, usamos o método de iterações, onde substituímos parte da equação de I com a equação 2.16 por ela mesma, prestando atenção nos índices. Colocaremos ρ_{ij} no termo $\rho_{rr}(t')$ de 2.15. Com isso, teremos:

$$I_i = \sum_r \gamma_{ir} \int_0^{T_R} dt' e^{\Gamma_{ii} t'} e^{-(iw_{rr}+\Gamma_{rr})T_R} (\rho_{rr}^c + \delta_{rr} I_i). \tag{2.23}$$

Percebamos que esta equação se apresenta de maneira recursiva, ou seja, a substituímos dentro de si mesma. No formato acima, temos o I_i do lado direito da equação com variável independente t'', um rótulo necessário para que haja coerência em diferenciar a variável da integral e a variável do I_i sendo integrado. Além disso temos $\omega_{rr} = (E_r - E_r)/\hbar = 0$ e $\delta_{rr} = 1$.

A redistribuição incoerente ocorre em uma escala de tempo diferente da excitação coerente pelo pulso ultracurto(FELINTO; LÓPEZ, 2009), o que nos permite retirar o termo ρ_{ij}^c da integral para obter um resultado aproximado. Reorganizando os termos e expandindo $I_i(t'')$ chegamos na expressão:

$$\begin{aligned} I_i \approx \sum_r \gamma_{ir} \rho_{rr}^c \int_0^{T_R} dt' e^{(\Gamma_{ii}-\Gamma_{rr})t'} + \\ \sum_{r,s} \gamma_{ir}\gamma_{rs} \int_0^{T_R} dt' e^{(\Gamma_{ii}-\Gamma_{rr})t'} \int_0^{t'} dt'' e^{(\Gamma_{rr}-\Gamma_{ss})t''} + ..., \end{aligned} \tag{2.24}$$

O processo se repete indefinidamente. Cada termo da soma descreve transições advindas de emissões espontâneas diferentes. Por exemplo, no primeiro termo temos γ_{ir}, que está relacionada ao preenchimento do nível i pelo nível r. Já no segundo termo, $\gamma_{ir}\gamma_{rs}$ faz esse termo descrever uma emissão em duas etapas, do nível s para o r e, depois, do r para o i. Os termos continuam na soma até se chegar no nível com maior número de emissões em sequência, ou seja, o de maior energia.

Por fim, usaremos a aproximação impulsiva para reescrever o operador U e deixá-lo em um formato ainda mais simples e manejável. Começamos com a expressão 2.21, considerando que um pulso é emitido em escalas de tempo muito menores do que as das transições envolvidas, isso nos permite escrever

$$1/\hbar \int_{-\infty}^{\infty} V_I^0(t) dt \approx e a_0 \int_{-\infty}^{\infty} \epsilon(t) M' dt, \tag{2.25}$$

onde o raio de Bohr e a carga do elétron foram colocados por conveniência (isso irá servir para deixarmos uma expressão em termos da área do pulso em unidades atômicas), ϵ é a envoltória

do pulso. $M^{'}$ aqui é a matriz que contém os momentos de dipolo normalizados das transições de maneira que $M^{'}_{ij} = \mu_{i,j}/ea_0$.

Definimos a área do pulso como

$$\theta_a = 2ea_0/\hbar \int_{-\infty}^{\infty} \epsilon(t)dt, \tag{2.26}$$

que nos permite escrever de maneira compacta a expressão:

$$1/\hbar \int_{-\infty}^{\infty} V_I^0(t)dt \approx \frac{\theta_a}{2} M^{'}. \tag{2.27}$$

Neste momento iremos lembrar da equação 2.17 para o operador de evolução temporal, substituímos a equação acima na série de Dyson para resultar em

$$U_I^n = 1 + (-i/\hbar) \int_0^{\infty} dt' V_I^0(t') + (-i/\hbar)^2 \int_0^{\infty} dt' \int_0^{t'} dt'' V_I^0(t') V_I^0(t'') + ..., \tag{2.28}$$

Finalmente o operador U pode ser aproximado para

$$U_I^0 \approx 1 + (-\frac{i\theta_a}{2}) M^{'} + (-\frac{i\theta_a}{2})^2 \frac{(M^{'2})}{2!} + ... \tag{2.29}$$

$$U_I^0 \approx e^{-\frac{i\theta_a M^{'}}{2}}, \tag{2.30}$$

o que nos fornece mais uma forma de representar o operador. Com estes resultados podemos prosseguir e começar a discutir os métodos numéricos.

3 ANÁLISE NUMÉRICA DO SISTEMA

3.1 O MÉTODO COMPUTACIONAL

Com a teoria mostrada no capítulo 2, podemos discutir como obter resultados numéricos. A série de Dyson para o operador de evolução temporal, equação (2.17) possui um formato que fica mais complicado à medida que aumentamos a ordem de perturbação, pois as integrais ficam cada vez mais complexas. No entanto, podemos contornar o problema usando a regra retangular de integração numérica (DAVIS; RABINOWITZ, 2007) e definindo matrizes P_j dependentes do tempo, que serão mostradas logo a seguir, e será mostrado que elas deixam a expressão de U_I^0 como uma soma linear das matrizes P_j. Estas matrizes possuem dimensão de V_I^0, o índice j varia de 0 a m, com m sendo a ordem máxima de perturbação considerada. Para explicar a relevância destas matrizes, começamos considerando um tempo inicial t_0 muito anterior à interação com o primeiro pulso. Neste caso temos

$$\begin{aligned} P_0(t_0) &= 1, \\ P_1(t_0) &= 0, \\ &\vdots \\ P_m(t_0) &= 0, \end{aligned} \tag{3.1}$$

Onde 1 e 0 representam a matriz identidade e a matriz nula respectivamente. Evoluindo o tempo de t_0 para um t_i posterior, o formato das matrizes ficará da seguinte maneira:

$$\begin{aligned} P_0(t_i) &= 1, \\ P_1(t_i) &= P_1(t_{i-1}) - i\Delta t/\hbar V_I^0(t_i)P_0(t_i), \\ P_2(t_i) &= P_2(t_{i-1}) - i\Delta t/\hbar V_I^0(t_i)P_1(t_i), \\ &\vdots \\ P_m(t_i) &= P_m(t_{i-1}) - i\Delta t/\hbar V_I^0(t_i)P_{m-1}(t_i), \end{aligned} \tag{3.2}$$

onde

$\Delta t = t_i - t_{i-1}$ é o passo de integração numérica, que como sabemos, é sempre discreta.

Notemos que, comparando estas expressões com a equação para U_I^0 teremos

$$U_I^0 = 1 + (-i/\hbar)\int_0^\infty dt' V_I^0(t') + (-i/\hbar)^2 \int_0^\infty dt' \int_0^{t'} dt'' V_I^0(t'') + ..., \tag{3.3}$$

e, considerando Δt muito pequeno, é possível escrever

$$U_I^0 \approx 1 + (-i/\hbar)\Delta t' V_I^0(t') + (-i/\hbar)^2 \Delta t' \Delta t'' V_I^0(t'') + ..., \tag{3.4}$$

fazendo $t_{i-1} = t_0$ e $t_i = t_N$ vemos que

$$U_I^0 \approx P_0(t_N) + P_1(t_N) + P_2(t_N) + ... + P_m(t_N). \tag{3.5}$$

Isso permite o cálculo numérico de U para qualquer ordem de perturbação apenas adicionando mais termos no somatório, sem aumento de complexidade dos cálculos.

Tendo em mente que a regra do trapezóide é mais precisa (PRESS et al., 2007), todos os cálculos realizados com o programa foram feitos utilizando-a para descrever as matrizes P:

$$\begin{aligned}
P_1(t_i) &= P_1(t_{i-1}) - i\Delta t/2\hbar[V_I^0(t_i)P_0(t_i) + V_I^0(t_{i-1})P_0(t_{i-1})], \\
P_2(t_i) &= P_2(t_{i-1}) - i\Delta t/2\hbar[V_I^0(t_i)P_1(t_i) + V_I^0(t_{i-1})P_1(t_{i-1})], \\
&\vdots \\
P_m(t_i) &= P_m(t_{i-1}) - i\Delta t/2\hbar[V_I^0(t_i)P_{m-1}(t_i) + V_I^0(t_{i-1})P_{m-1}(t_{i-1})],
\end{aligned} \tag{3.6}$$

Vemos aqui que a cada aumento na ordem de perturbação, temos uma nova equação, mas todas com o mesmo formato. Embora isso aumente levemente o tempo de execução de programa, não acrescenta nenhuma outra dificuldade aos cálculos, e é este o ponto central do método.

3.2 MEDINDO POPULAÇÕES NO RUBÍDIO 87

Aliando a teoria e o método de cálculo numérico apresentados anteriormente, temos a possibilidade de analisar interações atômicas com pulsos de qualquer formato, e em qualquer ordem de perturbação (esta última apenas aumenta o tempo de execução dos programas). O escopo deste trabalho consiste em estudar o Rubídio^{87}. Iremos ver que tipo de resultados obtemos para a transição de dois fótons 5S-5P-5D, sendo excitada por um trem de pulsos ultracurtos.

Antes de mostrar os resultados finais, é importante fazer algumas observações e escolher mais alguns parâmetros, além de discutir sobre os erros numéricos. Iremos calcular a população nos estados 5D sem expor de qual estado hiperfino específico veio a população. Vamos também definir a "área atômica do pulso", que vai ser útil posteriormente, pois os cálculos foram feitos com diferentes áreas e fizemos um comparativo entre os resultados, que serão expostos no capítulo 4.

Definimos agora a "área atômica do pulso", que é a área do pulso com alguns fatores multiplicados, para que fique em unidades atômicas, sendo eles constantes da natureza, como e, a carga elétrica, e a_0, o raio de Bohr:

$$\theta = 2ea_0/\hbar \int_{-\infty}^{\infty} \varepsilon(t)dt, \tag{3.7}$$

e com isso podemos prosseguir para falar sobre os erros numéricos.

3.3 ERROS NUMÉRICOS

Para finalizar este capítulo, precisamos entender como lidar com os erros numéricos, que são inevitáveis. Sabemos que um sistema quântico precisa ter a matriz densidade normalizada, e qualquer erro sistemático computacional se reflete em um afastamento da normalização. A ideia é saber o quanto a matriz densidade deixa de estar normalizada enquanto o sistema evolui no tempo. Então introduzimos a seguinte quantidade para representar este erro:

$$\varepsilon = |1 - \sum_i \rho_{ii}|, \tag{3.8}$$

(FELINTO; LÓPEZ, 2009), onde o índice i rotula todos os estados possíveis do sistema. Esta equação diz que o erro é a diferença entre população total e a unidade, 1. É esperado que este erro aumente para valores crescentes de área, e diminua para ordens maiores de perturbação que sejam consideradas nos cálculos realizados.

Tendo em mente que todos os métodos numéricos foram realizados usando a linguagem de programação C, notamos que na referência (FELINTO; LÓPEZ, 2009), os erros sempre estiveram entre 10^{-14} e 10^{-16}. Isso ocorre, entre outros motivos possíveis, por limitações das funções printf/scanf. Estas funções fazem, respectivamente, a escrita e a leitura dos arquivos com que os programas trabalham. Estas limitações ocorrem pois qualquer computador possui capacidade finita de armazenamento, além de poderem acontecer erros de arrendondamento (BENZ; HILDEBRANDT; HACK, 2012). Na rotina de execução original dos códigos, tínhamos 3

programas executados em sequência. O primeiro gera o vetor que representa o campo, armazenando em um arquivo os seus dados. O segundo, a partir da leitura do arquivo com o campo, calcula o operador de evolução temporal U, e cria um outro arquivo para armazená-lo. O terceiro calcula as populações e o erro associado, sendo necessária a leitura do arquivo com U, e a criação de um último arquivo que possui as populações e os erros. omo podemos ver no esquema abaixo. Esta rotina exige que o arquivo gerado por um programa por meio da função fprintf seja lido na execução do próximo com a função fscanf.

Este é um ponto crucial deste trabalho, pois com modificações nos programas originais, é possível contornar limitações presentes neles e diminuir o erro, que foi definido na seção anterior, equação 3.8. Estes três códigos foram unificados em uma única rotina de execução, amenizando os efeitos da fonte de erro discutida acima ao diminuir a quantidade de vezes que os arquivos precisam ser escritos e lidos. Posteriormente neste trabalho, observaremos se houve algum ganho numérico, i.e, diminuição dos erros. Testamos o programa modificado para alguns parâmetros presentes na referência (FELINTO; LÓPEZ, 2009) e para ordens de perturbação maiores do que 12.

3.4 O PROGRAMA MODIFICADO - UMA COMPARAÇÃO

Fazemos agora uma mostra das principais partes do código, destacando as principais mudanças realizadas, lembrando que este não é o código inteiro, apenas alguns trechos pertinentes de serem comentados.

Começamos com a geração do vetor que armazena as informações do campo elétrico:

```
1
2  #define                N        256*2       /* numero de elementos da grade */
3  S1 = (doispi*Tp)/(2*1.763);   /* pulse area with normalized maximum */
4
5          for(c1=0;c1<N;c1+=1){
6                  x[c1]=(c1-N/2)*dt;                  /* escala temporal */
7                  fx[2*c1]=1/(S1*cosh(1.763*x[c1]/Tp)); /* parte real do campo inicial */
8                  fx[2*c1+1]=0;                    /* parte imaginaria do campo inicial */
9          }
10
11 arq1=fopen("campoNorm_sech150fs_dt10fs.dat","a");
12         for(c1=0;c1<N;c1+=1){
13                 fprintf(arq1,"%g\t%g\t%g\n",x[c1],fx[2*c1],fx[2*c1+1]);
14         }
```

```
15          fclose(arq1);
```

A segunda parte deste código (a partir da linha 10) é abandonada pois o programa novo não precisa ler este arquivo. Ele já está com estes dados na memória enquanto as rotinas são executadas. Abaixo segue o código para a leitura deste arquivo, que também pôde ser abandonado:

```
1  void LoadField()
2  {
3       int       c21;
4       double    r1,r2,r3;
5       FILE      *ler;
6
7       ler=fopen("campoNorm_sech150fs_dt10fs.dat","r");
8       for(c21=0;c21<N;c21+=1){
9                    fscanf(ler,"%lf\t%lf\t%lf\n",&r1,&r2,&r3);
10                   x[c21]=r1;
11                   fx[2*c21]=r2;
12                   fx[2*c21+1]=r3;
13      }
14      fclose(ler);
15 }
```

Temos também a geração das matrizes P e da matriz U, que depois que são geradas, não precisam ser armazenadas em arquivos e lidos no próximo programa.

```
1  for(c22=0;c22<Nstates;++c22){
2              for(c23=0;c23<Nstates;++c23){
3                  U0[c22][c23].r = U0[c22][c23].i = 0.0;
4                  for(c21=0;c21<=Porder;++c21){
5                      P[c21][c22][c23].r = P[c21][c22][c23].i = 0.0;
6                      PA[c21][c22][c23].r = P[c21][c22][c23].i = 0.0;
7                      PB[c21][c22][c23].r = P[c21][c22][c23].i = 0.0;
8                  }
9              }
10             P[0][c22][c22].r = 1.0; //Matriz inicial igual à 'identidade', o resto, acima, é zero
11      }
```

Agora, o cálculo da matriz U, com as partes que foram retiradas comentadas:

```
1
2       //matriz U cálculo
3       //arq1=fopen("Umatrix_sech150fs_P10.dat","a");
```

```
4       for(c22=0;c22<Nstates;++c22){
5             c23=0;
6             for(c21=0;c21<=Porder;++c21){
7                 U0[c22][c23].r += P[c21][c22][c23].r;
8                 U0[c22][c23].i += P[c21][c22][c23].i;
9
10            }
11
12          //queremos nos livrar da próxima linha no programa final
13            //fprintf(arq1,"%.18e\t%.18e",U[c22][c23].r,U[c22][c23].i);
14            for(c23=1;c23<Nstates;++c23){
15               for(c21=0;c21<=Porder;++c21){
16                  U0[c22][c23].r += P[c21][c22][c23].r;
17                  U0[c22][c23].i += P[c21][c22][c23].i;
18
19            }
20               //Linha abaixo também se torna obsoleta
21               //fprintf(arq1,"\t%.18e\t%.18e",U[c22][c23].r,U[c22][c23].i);
22
23            }
24            //fprintf(arq1,"\n");
25      }
```

A seguir temos a função que lê o arquivo que foi dispensado, com os dados armazenados da matriz U.

```
1  void LoadU0Matrix() /*Não queremos fazer uso desta função, esta matriz precisa
2  se manter na memória a partir do momento que é calculada*/
3  {
4       int       c31,c32;
5       FILE      *arq1;
6
7       arq1=fopen("Umatrix_sech150fs_P10.dat","r");
8       for(c31=0;c31<Nstates;++c31){
9        fscanf(arq1,"%lf\t%lf",&U0[c31][0].r,&U0[c31][0].i);
10       for(c32=1;c32<Nstates;++c32){
11            fscanf(arq1,"\t%lf\t%lf",&U0[c31][c32].r,&U0[c31][c32].i);
12       }
13       fscanf(arq1,"\n");
14      }
15      fclose(arq1);
16 }
```

Por fim, temos o cálculo das populações e o armazenamento em arquivo de seus valores:

```
for(n=0;n<200;++n){
            /* phase for the n-th pulse in the train */
            phase = n*f0;
            LoadPhin(phase,frb);
            LoadUnMatrix();
            /* Coherent interaction with the ultrashort pulse */
            Calculate_rhoc();
            /* incoherent redistribution */
            for(c31=0;c31<Nstates;++c31){
                pc[c31] = rhoc[c31][c31].r;

            }
            for(c31=0;c31<Nstates;++c31){
               if(S[c31].man == 1){
                  for(c32=0;c32<Nstates;++c32){
                      rhoc[c31][c31].r += gamma1[c31][c32]*pc[c32]*D32;

                  }
               }
               if(S[c31].man == 0){
                  for(c32=0;c32<Nstates;++c32){
                      rhoc[c31][c31].r += gamma1[c31][c32]*pc[c32]*D21 + gamma2[c31][c32]*pc[c32]*D321;

                  }
               }
            }
            /* free induction decay */
            for(c31=0;c31<Nstates;++c31){
                for(c32=0;c32<Nstates;++c32){
                    rho[c31][c32].r = rhoc[c31][c32].r*RE[c31][c32].r - rhoc[c31][c32].i*RE[c31][c32].i;
                    rho[c31][c32].i = rhoc[c31][c32].r*RE[c31][c32].i + rhoc[c31][c32].i*RE[c31][c32].r;

                }
            }
         }

         r11 = r22 = r33 = 0.0;
         for(c31=0;c31<Nstates;++c31){
             if(S[c31].man == 2)
                r33 += rho[c31][c31].r;

             if(S[c31].man == 1)
                r22 += rho[c31][c31].r;

             if(S[c31].man == 0)
                r11 += rho[c31][c31].r;

         }
```

```
49
50          //queremos printar este arquivo apenas no final, ele nos dá as populações
51
52
53          printf("%lf\t%.8e\t%.8e\t%.8e\t%.8e\n",(frb-fr)*(1e12),r11,r22,r33,r11+r22+r33-1.0);
54          arq1=fopen("sech20fs_P12_001.dat","a");
55          fprintf(arq1,"%lf\t%.8e\t%.8e\t%.8e\t%.8e\n",(frb-fr)*(1e12),r11,r22,r33,r11+r22+r33-1.0);
56          fclose(arq1);
57      }
```

Notemos que, em vários passos, funções printf e scanf dos códigos antigos foram descartadas com sucesso, ou seja, precisam ser utilizadas em menos pontos do código, o que, como já antes explicado, tem como objetivo gerar aumento na precisão. A seguir, aferimos de maneira objetiva se houve algum ganho desta natureza.

4 ANÁLISE DOS RESULTADOS NUMÉRICOS

4.1 EXECUÇÃO DO PROGRAMA

O programa possui uma rotina de execução que ocorre em sequência. Em sua versão original, haviam 3 programas que eram executados um depois do outro. O primeiro gerava um campo elétrico com todos os parâmetros necessários para representar o laser. O segundo programa lia os dados do primeiro e calculava o operador de evolução temporal, ou seja, os elementos da matriz U. E no terceiro, eram lidos os resultados gerados pelo segundo programa e calculadas as populações. Estes programas foram modificados de forma a se unirem em um único código que realiza todas as operações em apenas uma execução.

Além de aumentar a praticidade, isso exclui a necessidade de se utilizar as funções scan e print múltiplas vezes. Sabemos que essas funções, como já discutido na seção 3.3 possuem limitações, então, assim como qualquer resultado numérico, os cálculos nos fornecem apenas aproximações. A cada print e scan utilizados, uma nova aproximação acontece, além disso, como os programas utilizam operações iterativas e os cálculos são acumulativos existe uma perda de dados quando são transferidos de um programa para o outro. Dessa forma, a ideia é que seja executado o programa com apenas um print para escrever os resultados, e ao final dos cálculos, a precisão aumente.

4.2 RESULTADOS

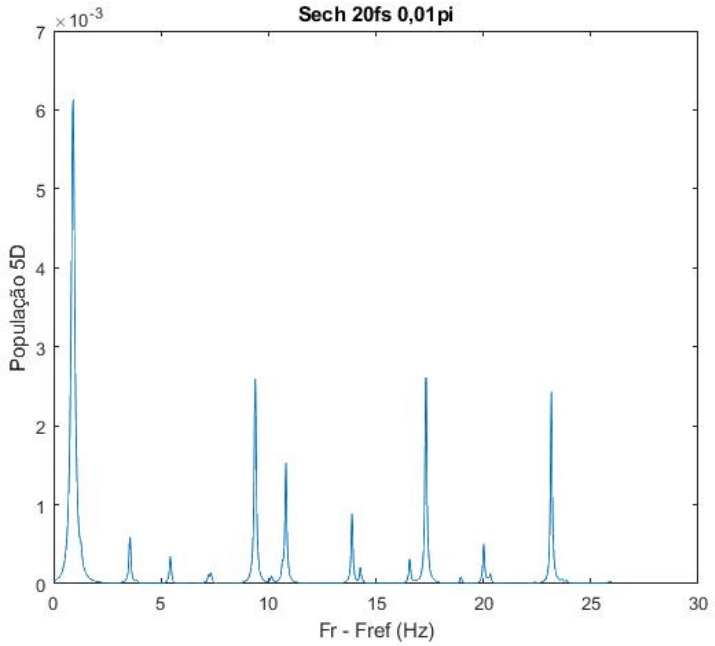
Iremos analisar a utilização de pulsos ultracurtos do formato secante hiperbólica

$$\varepsilon(t) = \varepsilon_0 sech(1.763t/T_p), \tag{4.1}$$

com os parâmetros relevantes a seguir: largura temporal T_p= 20 fs, $f_{rep} = 100,12$ MHz, frequência de offset f_o =10 MHz, cálculos na ordem $m = 12$, $\theta = 0,01\pi$ e considerando a observação do espectro após a interação com 200 pulsos.

A partir desses dados, geramos os seguintes espectros:

Figura 3 – Espectro com as populações do nível 5D para um pulso de largura de 20fs

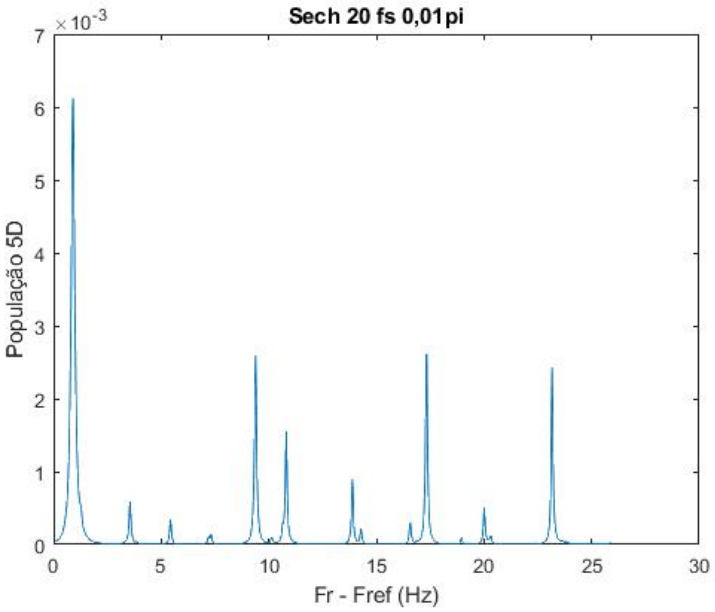


Fonte: O autor(2023)

A figura 3 é a equivalente da figura 6(a), curva vermelha, da referência (FELINTO; LÓPEZ, 2009).

Aumentando a ordem para $m = 20$ e mantendo os outros parâmetros inalterados, teremos o seguinte espectro:

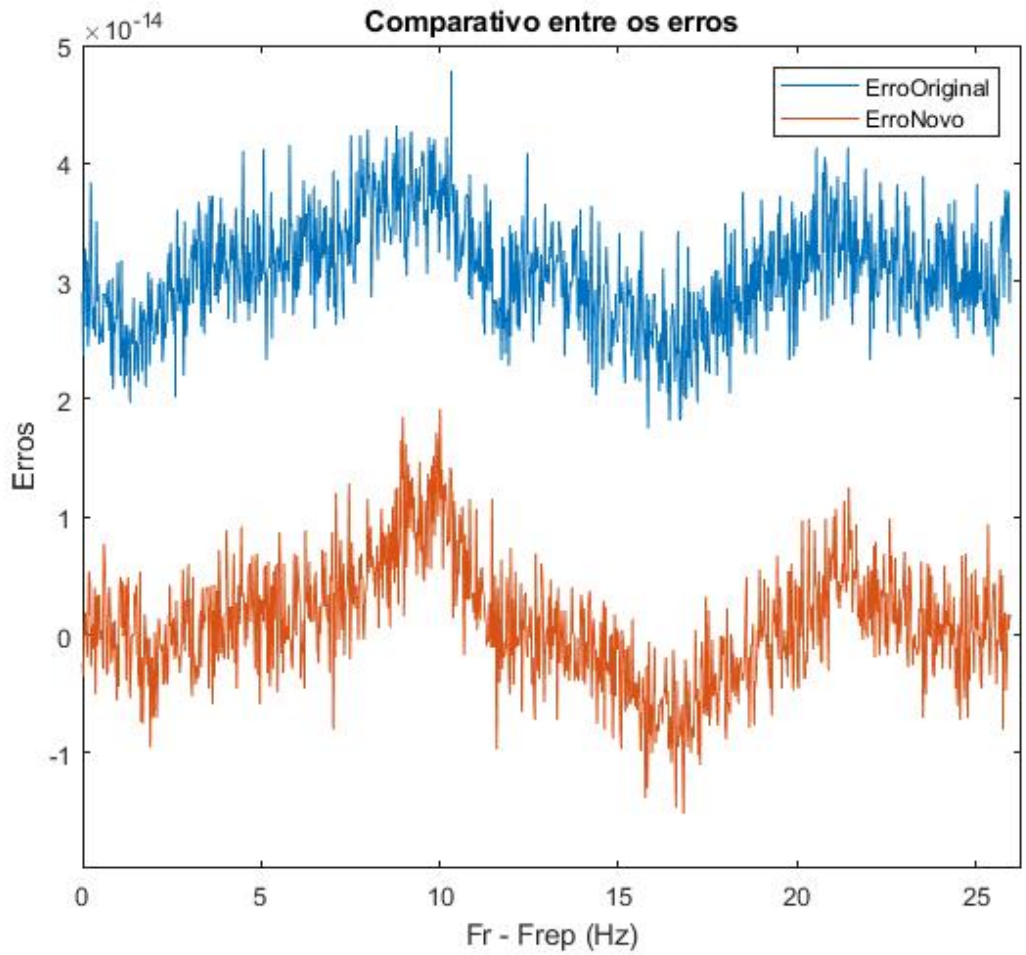
Figura 4 – Espectro de ordem 20 para sech de 20fs



Fonte: O autor(2023)

A figura 4apresenta mudanças significativas em relação à figura 3, mas demonstra o poder da teoria, ao evidenciar que mesmo havendo aumento na ordem, a utilização do programa não apresenta aumentos perceptíveis em seu tempo de execução. Ao comparar as imagens 3 e 4 com as de (FELINTO; LÓPEZ, 2009), fica clara a similaridade entre elas, o que demonstra uma consistência entre os resultados do programa antigo e o programa novo modificado. Fazemos agora a comparação, central para este trabalho, que é entre os erros obtidos pelo programa novo e o programa antigo, levando em consideração espectros com parâmetros idênticos. Gerando gráficos com os erros 3.8 calculados no programa, averiguamos se de fato é possível observar alguma diminuição nestas quantidades.

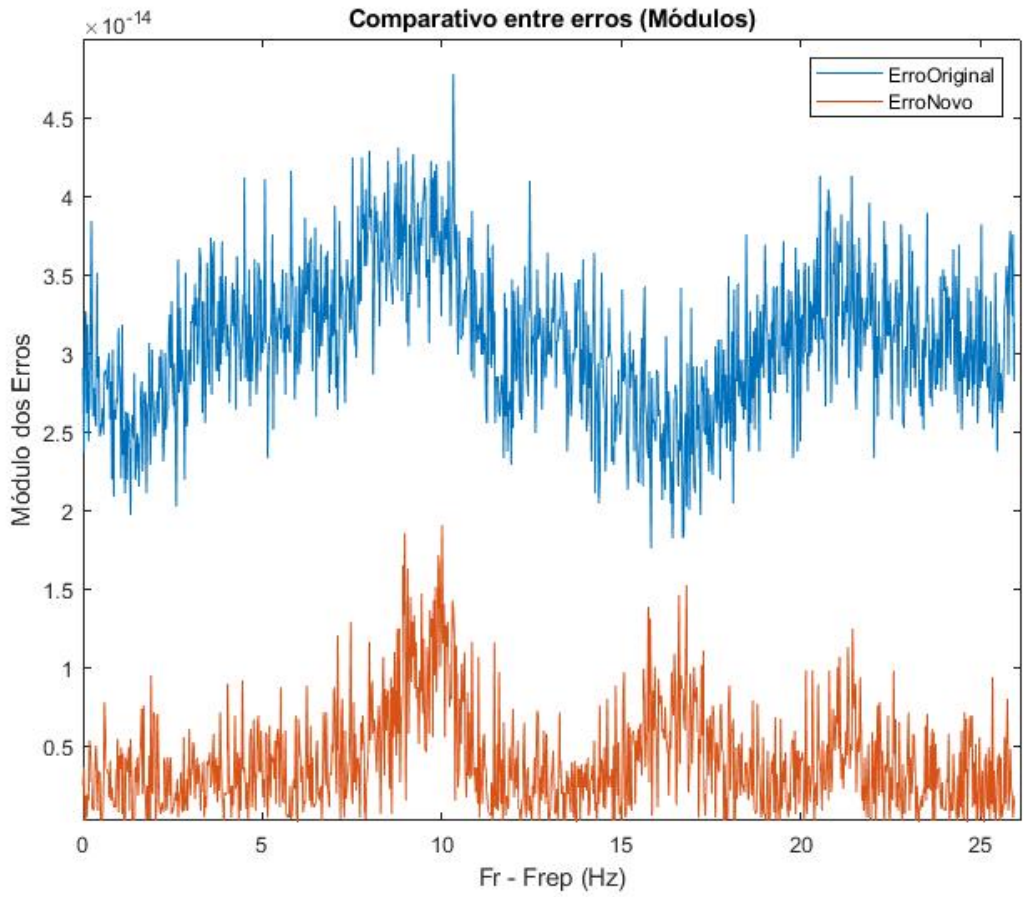
Figura 5 – Justaposição dos erros entre o programa atualizado e o programa original demonstrando a diferença entre os erros



Fonte: O autor(2023)

A seguir, geramos o mesmo gráfico mas com os módulos dos erros segundo a definição 3.8.

Figura 6 – Comparação dos erros entre o programa atualizado e o programa original demonstrando a diferença entre os erros



Fonte: O autor(2023)

Fica evidente a partir das figura 5 e 6 que o erro do programa atual está em uma ordem de grandeza menor do que o anterior na maioria dos pontos. O programa original possui erros que tendem a oscilar em volta de $2,0\text{x}10^{-14}$ e $4,5\text{x}10^{-14}$ enquanto que na linha marrom, vemos que os novos cálculos geram erros que tendem a oscilar entre $2\text{x}10^{-14}$ e $0,1\text{x}10^{-14}$, que equivalem a oscilar entre o menor erro da linha azul e $1\text{x}10^{-15}$, que é de uma ordem de grandeza menor do que a ordem dos programas antigos.

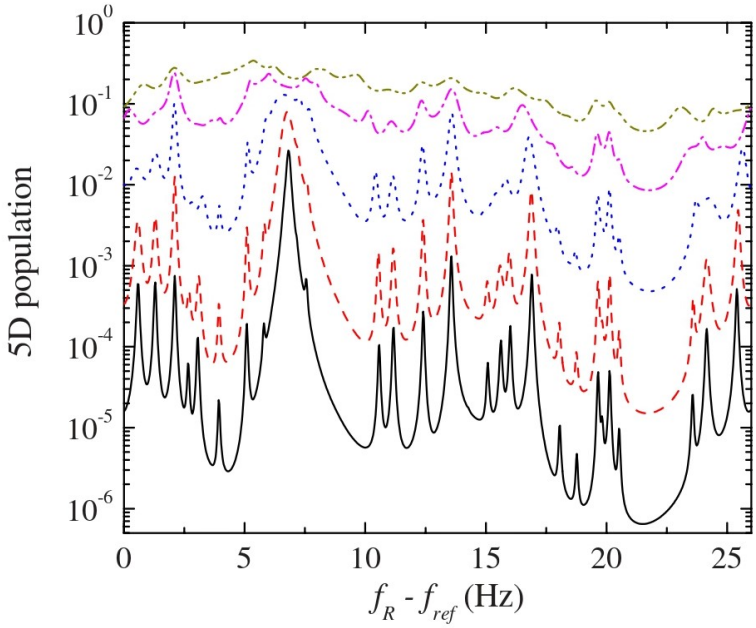
5 CONCLUSÃO

Foi demonstrado, a partir da comparação entre os resultados dos programas originais e dos cálculos do programa modificado, que existiu uma diminuição dos erros de, ao menos, uma ordem de grandeza. Isso ocorreu devido através de uma reformulação dos códigos, unificado-os em apenas um programa, que foi capaz de gerar erros mais satisfatórios, como mostrado nas figuras 5 e 6. É importante ressaltar o amplo conjunto de possibilidades a serem exploradas a partir deste resultado. A começar pelos parâmetros, neste trabalho obtemos uma otimização para pulsos de secante hiperbólica e separação temporal 20fs, além de ordem de perturbação 20 e $\theta_a = 0,01\pi$. A perspectiva de vários testes envolvendo outros formatos de pulso, outras larguras temporais, além de ordens maiores de perturbação e áreas de pulso pode vir a trazer um entendimento mais detalhado dos espectros do Rb^{87}.

É possível também fazermos tentativas de amenizar ainda mais os erros que surgem nativamente do cálculo numérico realizado pelo programa com buscas bibliográficas, para entendimento e proveito de técnicas não utilizadas neste trabalho, como em (DAMOUCHE; MARTEL; CHAPOUTOT, 2018) e (FU; BAI; SU, 2015).

O principal potencial para futuras publicações e descobertas está relacionado à figura 5 de (FELINTO; LÓPEZ, 2009), reproduzida abaixo. Temos as populações do nível 5D em função de Fr para vários valores de θ_a, onde de baixo para cima o valor vai aumentando. A curva preta abaixo é de $\theta_a = 0,01\pi$ e a curva no topo é de $\theta_a = 0,1\pi$. Vemos que o aumento da área atômica de pulso causa uma perda de resolução, chegando a uma situação de saturação onde as características do espectro se perdem. Com cálculos numéricos mais precisos, é imediato que o próximo passo a se tomar é investigar mais a fundo os espectros com valores mais altos de θ_a.

Figura 7 – População no nível 5D com valores crescentes, de baixo para cima, de θ_a



Fonte: (FELINTO; LÓPEZ, 2009)

Caso haja obtenção de espectros detalhados, mesmo com valores que podem ir de $\theta_a = 0, 4\pi$, ou $\theta_a = 0, 1\pi$, ou até maiores, estaremos com acesso a regiões inexploradas do espectro.

REFERÊNCIAS

BENZ, F.; HILDEBRANDT, A.; HACK, S. A dynamic program analysis to find floating-point accuracy problems. *ACM SIGPLAN Notices*, ACM New York, NY, USA, v. 47, n. 6, p. 453–462, 2012.

CUNDIFF, S. T.; YE, J. Colloquium: Femtosecond optical frequency combs. *Reviews of Modern Physics*, APS, v. 75, n. 1, p. 325, 2003.

DAMOUCHE, N.; MARTEL, M.; CHAPOUTOT, A. Numerical program optimisation by automatic improvement of the accuracy of computations. *International Journal of Intelligent Engineering Informatics*, Inderscience Publishers (IEL), v. 6, n. 1-2, p. 115–145, 2018.

DAVIS, P. J.; RABINOWITZ, P. *Methods of numerical integration*. [S.l.]: Courier Corporation, 2007.

FELINTO, D.; LÓPEZ, C. E. Theory for direct frequency-comb spectroscopy. *Physical Review A*, APS, v. 80, n. 1, p. 013419, 2009.

FORTIER, T.; BAUMANN, E. 20 years of developments in optical frequency comb technology and applications. *Communications Physics*, Nature Publishing Group UK London, v. 2, n. 1, p. 153, 2019.

FU, Z.; BAI, Z.; SU, Z. Automated backward error analysis for numerical code. *ACM SIGPLAN Notices*, ACM New York, NY, USA, v. 50, n. 10, p. 639–654, 2015.

PICQUÉ, N.; HÄNSCH, T. W. Frequency comb spectroscopy. *Nature Photonics*, Nature Publishing Group UK London, v. 13, n. 3, p. 146–157, 2019.

PRESS, W. H.; TEUKOLSKY, S. A.; VETTERLING, W. T.; FLANNERY, B. P. Numerical recipes in c++. *The art of scientific computing*, v. 2, p. 1002, 2007.

SAKURAI, J. J.; COMMINS, E. D. *Modern quantum mechanics, revised edition*. [S.l.]: American Association of Physics Teachers, 1995.

STOWE, M. C.; THORPE, M. J.; PE'ER, A.; YE, J.; STALNAKER, J. E.; GERGINOV, V.; DIDDAMS, S. A. Direct frequency comb spectroscopy. *Advances in Atomic, Molecular, and Optical Physics*, Elsevier, v. 55, p. 1–60, 2008.

UDEM, T.; HOLZWARTH, R.; HÄNSCH, T. Femtosecond optical frequency combs. *The European Physical Journal Special Topics*, Springer, v. 172, p. 69–79, 2009.

YE, J.; CUNDIFF, S. T. *Femtosecond optical frequency comb: principle, operation and applications*. [S.l.]: Springer Science & Business Media, 2005.

APÊNDICES

APÊNDICE A - PROGRAMA PRINCIPAL MODIFICADO

```
//Programa preliminar para juntar campo_02, matrix_U3 e rho3

#include  <time.h>
#include  <stdio.h>
#include  <stdlib.h>
#include  <math.h>
#include  "daniel.h"   /* biblioteca com funcoes auxiliares (FFT,
   integracao,etc) */
#include  "Parameters.h"

//Programa Campo-02 definies

/*  unidade de tempo: ps  */
/* unidade de comprimento: cm  */

/*  parametros do meio  */
#define   wl  (doispi*385.286) /* frequencia do laser em rad*(ps)^-1 */
#define   LI  (doispi*0.001)    /* largura de linha inomogenea (Doppler)
   */
#define   LH2 (doispi*18.7E-6)  /* largura de linha homogenea (T2) */
#define   alfa  0.0   /* alfa/LI = o coeficiente de absorcao */

/*  parametros do laser  */
#define   Tp  0.02          /* largura temporal do pulso em ps*/

/*  parametros da observacao  */
#define   s 5.0   /* distancia percorrida na amostra */

/*  parametros da grade temporal  */
#define   N 256*2       /* numero de elementos da grade */
#define   dt  0.0025     /* diferencial de tempo em ps */

/*  constantes  */
#define   doispi  6.28318530717959
```

```
33 #define    c 0.03  /* velocidade da luz */

35

   // campo_02 variveis e funcoes

37

   void       prop_lin(double coord[],double data[],int nn,double dd);
39 void     GVD(double coord[],double data[],int nn,double dd,double L);
   double     funcao1(double x1);
41 double     funcao2(double x1);

43 double     w1, DD;
   double       wt[8],D[12];
45

   //Programa Matrix U3 definicoes

47

   #define      Porder 20

49

51 //MatrixU3 variveis e funcoes

53 typedef struct {
      double  r;
55    double  i;
   } complex;

57

   //complex  U[Nstates][Nstates];
59 complex  P[Porder+1][Nstates][Nstates];
   complex  PA[Porder+1][Nstates][Nstates];
61 complex  PB[Porder+1][Nstates][Nstates];
   double   I[Nstates][Nstates],d[Nstates][Nstates];
63 double   *x,*fx;

65 void     Detunings();
   void     LoadDipoleMomentsU3();
67 //void     LoadField();
   void     dyson(complex M1[][Nstates],int P2, int P3);

69

   //rho3 variaveis e funcoes

71
```

```
complex  Un[Nstates][Nstates];
complex  U0[Nstates][Nstates];
complex  Phin[Nstates][Nstates];
double   Mu[Nstates][Nstates];
double   gamma1[Nstates][Nstates];
double   gamma2[Nstates][Nstates];
complex  rho[Nstates][Nstates],rhoc[Nstates][Nstates];
complex  RE[Nstates][Nstates];

void     LoadDipoleMomentsrho3();
//void     LoadU0Matrix();
void     Load_gammas();
void     LoadPhin(double nf0, double frep);
void     LoadUnMatrix();
void     Calculate_rhoc();


int main(void)
{
  //main do campo_02
  int     c1,ini,fim,lim;
  //double  *x,*fx;
    double  C,S1;
  time_t  first,second;
  FILE  *arq1;

  x=dvector(0,N);
  fx=dvector(0,2*N);

  first=time(NULL);

  wt[0] = 0.0;
  wt[1] = doispi*0.0068346826;
  wt[2] = doispi*377.1112247286;
  wt[3] = doispi*377.1120413876;
  wt[4] = doispi*384.2344540709;
  wt[5] = doispi*384.2345262889;
  wt[6] = doispi*384.2346832359;
  wt[7] = doispi*(384.2281152033 + 0.0068346826);
```

```
111
    D[0] = wl - (wt[2]-wt[0]);
113 D[1] = wl - (wt[3]-wt[0]);
    D[2] = wl - (wt[4]-wt[0]);
115 D[3] = wl - (wt[5]-wt[0]);
    D[4] = wl - (wt[6]-wt[0]);
117 D[5] = wl - (wt[7]-wt[0]);
    D[6] = wl - (wt[2]-wt[1]);
119 D[7] = wl - (wt[3]-wt[1]);
    D[8] = wl - (wt[4]-wt[1]);
121 D[9] = wl - (wt[5]-wt[1]);
    D[10] = wl - (wt[6]-wt[1]);
123 D[11] = wl - (wt[7]-wt[1]);

125 S1 = (doispi*Tp)/(2*1.763);   /* pulse area with normalized maximum */

127 for(c1=0;c1<N;c1+=1){
      x[c1]=(c1-N/2)*dt;     /* escala temporal */
129   fx[2*c1]=1/(S1*cosh(1.763*x[c1]/Tp)); /* parte real do campo inicial
     */
      fx[2*c1+1]=0;                    /* parte imaginaria do campo inicial
     */
131 }


133
         C = 0.0;
135
      // GVD(x,fx,N,dt,C);          /* chirping */
137
    /* propagacao linear do campo */
139
    //prop_lin(x,fx,N,dt);
141
    //queremos nos livrar deste trecho abaixo no programa final
143
    /*arq1=fopen("campoNorm_sech150fs_dt10fs.dat","a");
145 for(c1=0;c1<N;c1+=1){
      fprintf(arq1,"%g\t%g\t%g\n",x[c1],fx[2*c1],fx[2*c1+1]);
147 }
```

```
fclose(arq1); */

second=time(NULL);
printf("t de  m quina : %f s\n",difftime(second,first));

  //Nao podemos esvaziar os vetores nesta parte pois seus dados ainda
 serao reaproveitados abaixo

  //free_dvector(x,0,N);
//free_dvector(fx,0,2*N);

//main do matrixU3

    int       c21,c22,c23,c24;
    double    r1,r2,r3;
    complex   VI[Nstates][Nstates];
    time_t    t1,t2;

    t1 = time (NULL);

    LoadStates();

    LoadDipoleMomentsU3();

    Detunings();

    /*y=dvector(0,N);
  fy=dvector(0,2*N);*/
  //LoadField(); o campo nao vai ser carregado pois ja temos informacao
   dele e nao zeramos os dvectors

 /* for(c21=0;c21<N;c21+=1){
       //aqui acontece o loadfield sem necessidade de ler o arquivo
 (???)
    x[c21]=r1;
    fx[2*c21]=r2;
    fx[2*c21+1]=r3;
  } */

```

```
185       for(c22=0;c22<Nstates;++c22){
                for(c23=0;c23<Nstates;++c23){
187                 U0[c22][c23].r = U0[c22][c23].i = 0.0;
                    for(c21=0;c21<=Porder;++c21){
189                     P[c21][c22][c23].r = P[c21][c22][c23].i = 0.0;
                        PA[c21][c22][c23].r = P[c21][c22][c23].i = 0.0;
191                     PB[c21][c22][c23].r = P[c21][c22][c23].i = 0.0;
                    }
193             }
                P[0][c22][c22].r = 1.0; //Matriz inicial igual a 'identidade'
     e o resto, acima, e zero
195       }

197       //VI  c lculo
          for(c21=0;c21<N;++c21){
199          for(c22=0;c22<Nstates;++c22){
                for(c23=c22;c23<Nstates;++c23){
201                VI[c22][c23].r = (theta/2.0)*(dt/2.0)*I[c22][c23]*(fx[2*
    c21+1]*cos(d[c22][c23]*x[c21])+fx[2*c21]*sin(d[c22][c23]*x[c21]));
                   VI[c22][c23].i = (theta/2.0)*(dt/2.0)*I[c22][c23]*(-fx[2*
    c21]*cos(d[c22][c23]*x[c21])+fx[2*c21+1]*sin(d[c22][c23]*x[c21]));
203                VI[c23][c22].r = -VI[c22][c23].r;
                   VI[c23][c22].i = VI[c22][c23].i;
205             }
             }
207

             for(c24=0;c24<Porder;++c24)
209              dyson(VI,c24,c24+1);
                 printf("%d\n",c21);
211       }

213       //matriz U  c lculo
          //arq1=fopen("Umatrix_sech150fs_P10.dat","a");
215       for(c22=0;c22<Nstates;++c22){
                c23=0;
                for(c21=0;c21<=Porder;++c21){
217                 U0[c22][c23].r += P[c21][c22][c23].r;
                    U0[c22][c23].i += P[c21][c22][c23].i;
219
```

```
221           }

223       //queremos nos livrar deste trecho no programa final
            //fprintf(arq1,"%.18e\t%.18e",U[c22][c23].r,U[c22][c23].i);
225           for(c23=1;c23<Nstates;++c23){
                 for(c21=0;c21<=Porder;++c21){
227                 U0[c22][c23].r += P[c21][c22][c23].r;
                    U0[c22][c23].i += P[c21][c22][c23].i;
229

              }
                 //fprintf(arq1,"\t%.18e\t%.18e",U[c22][c23].r,U[c22][c23].
231 i);

233           }
              //fprintf(arq1,"\n");
235     }
      /*fclose(arq1);
237

      free_dvector(x,0,N);
239  free_dvector(fx,0,2*N);*/


241

      //t2 = time (NULL);
243  //printf("%ld minutes\n\n",(t2-t1)/60);

245     //main do pho3

247    int        c31,c32,c33,c34,n,df;
       double     W,GAMMA,D32,D21,D31,D321,r33,r22,r11,frb,dfr,aW;
249    double     phase;
       double     pc[Nstates];
251


253    //Ja apareceu essa funcao ali em cima
       //LoadStates();
255

       LoadDipoleMomentsrho3();
257
```

```
 //LoadU0Matrix(); As informaoees nao precisam ser lidas no arquivo,
ja estao neste programa, foram calculadas nas linhas anteriores
259
  Load_gammas();
261
  /* Spectrum as a function of fr */
263
  for(df=0;df<1000;++df){
265   frb = fr + 0.000000000026*df/1000.0;
   /* free-induction-decay parameters */
267   for(c31=0;c31<Nstates;++c31){
        for(c32=0;c32<Nstates;++c32){
269           W = S[c31].w - S[c32].w;
              GAMMA = S[c31].Gamma + S[c32].Gamma;
271           aW = W/frb;
              RE[c31][c32].r = cos(aW)*exp(-GAMMA/frb);
273           RE[c31][c32].i = -sin(aW)*exp(-GAMMA/frb);
        }
275   }
      /* Decay factors */
277   D32 = (exp((Gamma22-Gamma33)/frb)-1)/(Gamma22-Gamma33);
      D21 = -(exp(-Gamma22/frb)-1)/Gamma22;
279   D31 = -(exp(-Gamma33/frb)-1)/Gamma33;
      D321 = (D31-D21)/(Gamma22-Gamma33);
281   //printf("%.4e\n%.4e\n%.4e\n",D32,D21,D321);
      /* Initial state */
283   for(c31=0;c31<Nstates;++c31){
          for(c32=0;c32<Nstates;++c32){
285           rho[c31][c32].r = 0.0;
              rho[c31][c32].i = 0.0;
287       }
      }
289   rho[0][0].r = 3.0/8.0;
      rho[1][1].r = 5.0/8.0;
291   /* Temporal evolution */

293   for(n=0;n<200;++n){
          /* phase for the n-th pulse in the train */
295       phase = n*f0;
```

```
          LoadPhin(phase,frb);
297       LoadUnMatrix();
          /* Coherent interaction with the ultrashort pulse */
299       Calculate_rhoc();
          /* incoherent redistribution */
301       for(c31=0;c31<Nstates;++c31){
              pc[c31] = rhoc[c31][c31].r;
303
          }
305       for(c31=0;c31<Nstates;++c31){
             if(S[c31].man == 1){
307             for(c32=0;c32<Nstates;++c32){
                    rhoc[c31][c31].r += gamma1[c31][c32]*pc[c32]*D32;
309
                }
311          }
             if(S[c31].man == 0){
313             for(c32=0;c32<Nstates;++c32){
                    rhoc[c31][c31].r += gamma1[c31][c32]*pc[c32]*D21 +
gamma2[c31][c32]*pc[c32]*D321;
315
                }
317          }
          }
319       /* free induction decay */
          for(c31=0;c31<Nstates;++c31){
321           for(c32=0;c32<Nstates;++c32){
                  rho[c31][c32].r = rhoc[c31][c32].r*RE[c31][c32].r -
rhoc[c31][c32].i*RE[c31][c32].i;
323               rho[c31][c32].i = rhoc[c31][c32].r*RE[c31][c32].i +
rhoc[c31][c32].i*RE[c31][c32].r;

325           }
          }
327    }

329    r11 = r22 = r33 = 0.0;
       for(c31=0;c31<Nstates;++c31){
331        if(S[c31].man == 2)
```

```
                r33 += rho[c31][c31].r;
333
            if(S[c31].man == 1)
                r22 += rho[c31][c31].r;
335
            if(S[c31].man == 0)
                r11 += rho[c31][c31].r;
337
        }
339
        //queremos printar este arquivo apenas no final, ele nos da as
    populacoes
341
        printf("%lf\t%.8e\t%.8e\t%.8e\t%.8e\n",(frb-fr)*(1e12),r11,r22,
343 r33,r11+r22+r33-1.0);
        arq1=fopen("sech20fs_P20_PulseArea0dot01_001.dat","a");
        fprintf(arq1,"%lf\t%.8e\t%.8e\t%.8e\t%.8e\n",(frb-fr)*(1e12),r11,
345 r22,r33,r11+r22+r33-1.0);
        fclose(arq1);
      }
347
  return(0);
}
349
// F u n e s  do campo_02
351
/*void prop_lin(double coord[],double data[],int nn,double dd)
{
353   int c1,c2;
  double  a1,a2,a3,a4,pgauss;
  FILE   *arq1;
355
  FFT(coord,data,nn,dd);
357
  pgauss=1/(sqrt(doispi)*LI);
359
  for(c1=0;c1<nn;++c1){
    w1=coord[c1];
361
```

```
    a1=a2=0.0;
    for(c2=0;c2<12;++c2){
            DD = D[c2];
        a1 += alfa*s*LH2*pgauss*qtrap(funcao1,-DD-0.5,-DD+0.5);      //
   perfil voight
        a2 += alfa*s*pgauss*qtrap(funcao2,-DD-0.5,-DD+0.5);
        }
    a3=data[2*c1];
    a4=data[2*c1+1];
    data[2*c1]=exp(-a1)*(a3*cos(a2)+a4*sin(a2));
    data[2*c1+1]=exp(-a1)*(-a3*sin(a2)+a4*cos(a2));
  }

  /*arq1=fopen("campo_05espec.dat","a");
  for(c1=0;c1<N;c1+=1){
    fprintf(arq1,"%g\t%g\n",coord[c1],sqrt(data[2*c1]*data[2*c1]+data[2*
   c1+1]*data[2*c1+1]));
  }
  fclose(arq1);*/

  //IFFT(coord,data,nn,1/(nn*dd));
//}

double funcao1(double x1) /* absorcao */
{
  double  a1,a2,a3;

  a1=(x1+w1)*(x1+w1)+LH2*LH2;
  a3=-DD-x1;
  a2=exp(-a3*a3/(2*LI*LI))/a1;
  return(a2);
}

double funcao2(double x1) /* dispersao */
{
  double  a1,a2,a3;

  a1=(x1+w1)*(x1+w1)+LH2*LH2;
  a3=-DD-x1;
```

```
  a2=(x1+w1)*exp(-a3*a3/(2*LI*LI))/a1;
  return(a2);
}

void GVD(double coord[],double data[],int nn,double dd,double L)
{
  /* propagacao atraves de SQ1 com lambda de 800 nm */

  int c1;
  double  a2,a3,a4,a5,n1,n2,k2,w1;

  FFT(coord,data,nn,dd);

        k2=L*Tp*Tp/2.0;  /* beta2 */

  for(c1=0;c1<nn;++c1){
    w1=coord[c1];
    a2=k2*(w1*w1/2.0);
    a4=data[2*c1];
    a5=data[2*c1+1];
    data[2*c1]=a4*cos(a2)+a5*sin(a2);
    data[2*c1+1]= -a4*sin(a2)+a5*cos(a2);
  }

  IFFT(coord,data,nn,1/(nn*dd));
}

// F u n e s  do MatrixU3

void dyson(complex M1[][Nstates],int P2,int P3)
{
     int    c21,c22,c23;

     for(c21=0;c21<Nstates;++c21){
            for(c22=0;c22<Nstates;++c22){
               PB[P3][c21][c22].r = PB[P3][c21][c22].i = 0.0;
               for(c23=0;c23<Nstates;++c23){
                   PB[P3][c21][c22].r += M1[c21][c23].r*P[P2][c23][c22].r
    - M1[c21][c23].i*P[P2][c23][c22].i;
```

```
443                     PB[P3][c21][c22].i += M1[c21][c23].r*P[P2][c23][c22].i
    + M1[c21][c23].i*P[P2][c23][c22].r;
                }
445             P[P3][c21][c22].r += PA[P3][c21][c22].r + PB[P3][c21][c22
    ].r;
                P[P3][c21][c22].i += PA[P3][c21][c22].i + PB[P3][c21][c22
    ].i;
447             PA[P3][c21][c22].r = PB[P3][c21][c22].r;
                PA[P3][c21][c22].i = PB[P3][c21][c22].i;
449         }
     }
451 }

453 void LoadDipoleMomentsU3()
{
455     int       c21,c22;
     double    r21;
457     FILE      *ler;

459     ler=fopen("DipoleMoments_01.dat","r");
    for(c21=0;c21<Nstates;c21+=1){
461         for(c22=0;c22<Nstates-1;c22+=1){
              fscanf(ler,"%lf\t",&r21);
463         I[c21][c22]= r21;
            }
465         fscanf(ler,"%lf\n",&r21);
            I[c21][Nstates-1]= r21;
467 }
    fclose(ler);
469 }

471 /*void LoadField()
{
473     int       c21;
     double    r1,r2,r3;
475     FILE      *ler;

477     ler=fopen("campoNorm_sech150fs_dt10fs.dat","r");
     for(c21=0;c21<N;c21+=1){
```

```
479       fscanf(ler,"%lf\t%lf\t%lf\n",&r1,&r2,&r3);
          x[c21]=r1;
481       fx[2*c21]=r2;
          fx[2*c21+1]=r3;
483      }
         fclose(ler);
485 }*/

487 void Detunings()
{
489       int      c21,c22;

491       for(c21=0;c21<Nstates;++c21){
              for(c22=c21;c22<Nstates;++c22){
493               if(I[c21][c22]!=0.0){
                     d[c21][c22] = (S[c21].w - S[c22].w + wl);
495               }
                  d[c22][c21] = d[c21][c22];
497           }
          }
499 }

501 // f u n e s  do pho3

503 void LoadDipoleMomentsrho3()
{
505      int      c31,c32;
         double   r31;
507      FILE     *ler;

509      ler=fopen("DipoleMoments_01.dat","r");
        for(c31=0;c31<Nstates;c31+=1){
511           for(c32=0;c32<Nstates-1;c32+=1){
                  fscanf(ler,"%lf\t",&r31);
513           Mu[c31][c32]= r31;
              }
515           fscanf(ler,"%lf\n",&r31);
              Mu[c31][Nstates-1]= r31;
517     }
```

```
    fclose(ler);
}

/*void LoadU0Matrix() //Nao queremos fazer uso desta funao, esta matriz precisa se manter na memoria a partir do momento em que e calculada
{
     int      c31,c32;
     FILE     *arq1;

     arq1=fopen("Umatrix_sech150fs_P10.dat","r");
     for(c31=0;c31<Nstates;++c31){
      fscanf(arq1,"%lf\t%lf",&U0[c31][0].r,&U0[c31][0].i);
      for(c32=1;c32<Nstates;++c32){
            fscanf(arq1,"\t%lf\t%lf",&U0[c31][c32].r,&U0[c31][c32].i);
      }
      fscanf(arq1,"\n");
     }
     fclose(arq1);
}*/

void Load_gammas()
{
     int     c31,c32,c33;
     double  sMu2;

     for(c31=0;c31<Nstates;++c31){
         for(c32=0;c32<Nstates;++c32){
             gamma1[c31][c32] = 0.0;
             sMu2 = 0.0;
             for(c33=0;c33<Nstates;++c33){
                 if(S[c32].man > S[c33].man)
                    sMu2 += Mu[c33][c32]*Mu[c33][c32];
             }
             if(S[c32].man > S[c31].man)
                gamma1[c31][c32] = (2*S[c32].Gamma)*Mu[c31][c32]*Mu[c31][c32]/sMu2;
         }
     }
     for(c31=0;c31<Nstates;++c31){
```

```
555          for(c32=0;c32<Nstates;++c32){
                 gamma2[c31][c32] = 0.0;
557              for(c33=0;c33<Nstates;++c33)
                     gamma2[c31][c32] += gamma1[c31][c33]*gamma1[c33][c32];
559          }
         }
561 }

563 void LoadPhin(double nf0, double frep)
    {
565      int      c31,c32;

567      for(c31=0;c31<Nstates;++c31){
             for(c32=0;c32<Nstates;++c32)
569          Phin[c31][c32].r = Phin[c31][c32].i = 0.0;
             Phin[c31][c31].r = cos(TwoPi*nf0*S[c31].man/frep);
571          Phin[c31][c31].i = sin(TwoPi*nf0*S[c31].man/frep);
         }
573 }

575 void LoadUnMatrix()
    {
577      int       c31,c32,c33;
         complex   MAux[Nstates][Nstates];
579

         for(c31=0;c31<Nstates;++c31){
581          for(c32=0;c32<Nstates;++c32){
                 MAux[c31][c32].r = MAux[c31][c32].i = 0.0; //zerando as
    linhas e colunas das partes real e imaginaria
583              for(c33=0;c33<Nstates;++c33){
                     MAux[c31][c32].r += U0[c31][c33].r*Phin[c32][c33].r + U0
    [c31][c33].i*Phin[c32][c33].i;
585                  MAux[c31][c32].i += U0[c31][c33].i*Phin[c32][c33].r - U0
    [c31][c33].r*Phin[c32][c33].i;
                 }
587          }
         }
589      for(c31=0;c31<Nstates;++c31){
             for(c32=0;c32<Nstates;++c32){
```

```
591             Un[c31][c32].r = Un[c31][c32].i = 0.0;
                for(c33=0;c33<Nstates;++c33){
593                 Un[c31][c32].r += Phin[c31][c33].r*MAux[c33][c32].r -
    Phin[c31][c33].i*MAux[c33][c32].i;
                    Un[c31][c32].i += Phin[c31][c33].i*MAux[c33][c32].r +
    Phin[c31][c33].r*MAux[c33][c32].i;
595             }
          }
597     }
}
599
void Calculate_rhoc()
601 {
      int       c31,c32,c33;
603   complex   MAux[Nstates][Nstates];

605   for(c31=0;c31<Nstates;++c31){
          for(c32=0;c32<Nstates;++c32){
607           MAux[c31][c32].r = MAux[c31][c32].i = 0.0;
              for(c33=0;c33<Nstates;++c33){
609               MAux[c31][c32].r += rho[c31][c33].r*Un[c32][c33].r + rho
    [c31][c33].i*Un[c32][c33].i;
                  MAux[c31][c32].i += rho[c31][c33].i*Un[c32][c33].r - rho
    [c31][c33].r*Un[c32][c33].i;
611           }
          }
613   }
      for(c31=0;c31<Nstates;++c31){
615       for(c32=0;c32<Nstates;++c32){
              rhoc[c31][c32].r = rhoc[c31][c32].i = 0.0;
617           for(c33=0;c33<Nstates;++c33){
                  rhoc[c31][c32].r += Un[c31][c33].r*MAux[c33][c32].r - Un
    [c31][c33].i*MAux[c33][c32].i;
619               rhoc[c31][c32].i += Un[c31][c33].i*MAux[c33][c32].r + Un
    [c31][c33].r*MAux[c33][c32].i;
              }
621       }
      }
623 }
```

APÊNDICE B - PARAMETERS.H: PROGRAMA SECUNDÁRIO COM OS PARÂMETROS

```
/* List of parameters for DFCS of rubidium 87 */

/*  time unit: ps  */
/*  length unit: cm  */

/* constants */
#define   TwoPi   6.28318530717959
#define   c 0.03          /* light velocity */

/* temporal grating parameters */
#define   N 256*2        /* number of grating elements */
#define   dt  0.0025         /* time step */

/* laser parameters */
#define     wl     (TwoPi*385.286)   /* central frequency */
#define   fr     (0.00010012)   /* repetition rate */
//#define   fr     (0.000100)   /* repetition rate for FIG5*/
#define     f0     (0.000010)          /* offset frequency */
#define     theta  (TwoPi*0.01/2.0) /* pulse-area parameter for a
    transition
                                       with dipole moment equal to the
    electric
                                       charge e times the Bohr radius a0
     */

/* rubidium 87 parameters */
#define     Nstates   16         /* number of atomic states */
#define     Rad5S5P   5.14       /* radial part of the dipole moment for
    the
                                    5S-5P transition of rubidium in units
                                    of the Bohr radius */
#define     Rad5P5D   1.88       /* radial part of the dipole moment for
    the
                                    5P-5D transition of rubidium in units
                                    of the Bohr radius */
#define     Gamma22    (TwoPi*0.00000596)      /* decay rate of
    population from level 2 */
```

```
#define     Gamma33    (TwoPi*0.00000066)      /* decay rate of
    population from level 3 */

/* structure for state parameters */
typedef struct {
   double  man;                  /* state manifold */
   double  l;                    /* quantum number l */
   double  j;                    /* quantum number j */
   double  F;                    /* quantum number F */
   double  w;                    /* transition frequency from lowest
    energy state */
   double  Gamma;                /* decay rate */
} state;

state    S[Nstates];

void     LoadStates();

void LoadStates()
{
      /* 5S(1/2)F(1) */
      S[0].man = 0.0;
      S[0].l = 0.0;
      S[0].j = 0.5;
      S[0].F = 1.0;
      S[0].w = 0.0;
      S[0].Gamma = 0.0;
      /* 5S(1/2)F(2) */
      S[1].man = 0.0;
      S[1].l = 0.0;
      S[1].j = 0.5;
      S[1].F = 2.0;
      S[1].w = TwoPi*0.0068346826;
      S[1].Gamma = 0.0;
      /* 5P(1/2)F(1) */
      S[2].man = 1.0;
      S[2].l = 1.0;
      S[2].j = 0.5;
      S[2].F = 1.0;
```

```
        S[2].w = TwoPi*377.1112247286;
        S[2].Gamma = Gamma22/2.0;
        /* 5P(1/2)F(2) */
        S[3].man = 1.0;
        S[3].l = 1.0;
        S[3].j = 0.5;
        S[3].F = 2.0;
        S[3].w = TwoPi*377.1120413876;
        S[3].Gamma = Gamma22/2.0;
        /* 5P(3/2)F(0) */
        S[4].man = 1.0;
        S[4].l = 1.0;
        S[4].j = 1.5;
        S[4].F = 0.0;
        S[4].w = TwoPi*384.2344540709;
        S[4].Gamma = Gamma22/2.0;
        /* 5P(3/2)F(1) */
        S[5].man = 1.0;
        S[5].l = 1.0;
        S[5].j = 1.5;
        S[5].F = 1.0;
        S[5].w = TwoPi*384.2345262889;
        S[5].Gamma = Gamma22/2.0;
        /* 5P(3/2)F(2) */
        S[6].man = 1.0;
        S[6].l = 1.0;
        S[6].j = 1.5;
        S[6].F = 2.0;
        S[6].w = TwoPi*384.2346832359;
        S[6].Gamma = Gamma22/2.0;
        /* 5P(3/2)F(3) */
        S[7].man = 1.0;
        S[7].l = 1.0;
        S[7].j = 1.5;
        S[7].F = 3.0;
        S[7].w = TwoPi*(384.2281152033 + 0.0068346826);
        S[7].Gamma = Gamma22/2.0;
        /* 5D(3/2)F(0) */
        S[8].man = 2.0;
```

```
      S[8].l = 2.0;
      S[8].j = 1.5;
      S[8].F = 0.0;
      S[8].w = TwoPi*770.487024638;
      S[8].Gamma = Gamma33/2.0;
      /* 5D(3/2)F(1) */
      S[9].man = 2.0;
      S[9].l = 2.0;
      S[9].j = 1.5;
      S[9].F = 1.0;
      S[9].w = TwoPi*770.487038137;
      S[9].Gamma = Gamma33/2.0;
      /* 5D(3/2)F(2) */
      S[10].man = 2.0;
      S[10].l = 2.0;
      S[10].j = 1.5;
      S[10].F = 2.0;
      S[10].w = TwoPi*770.4870660678;
      S[10].Gamma = Gamma33/2.0;
      /* 5D(3/2)F(3) */
      S[11].man = 2.0;
      S[11].l = 2.0;
      S[11].j = 1.5;
      S[11].F = 3.0;
      S[11].w = TwoPi*770.4871102902;
      S[11].Gamma = Gamma33/2.0;
      /* 5D(5/2)F(4) */
      S[12].man = 2.0;
      S[12].l = 2.0;
      S[12].j = 2.5;
      S[12].F = 4.0;
      S[12].w = TwoPi*(770.5691327326 + 0.0068346826);
      S[12].Gamma = Gamma33/2.0;
      /* 5D(5/2)F(3) */
      S[13].man = 2.0;
      S[13].l = 2.0;
      S[13].j = 2.5;
      S[13].F = 3.0;
      S[13].w = TwoPi*770.5759962382;
```

```
        S[13].Gamma = Gamma33/2.0;
148     /* 5D(5/2)F(2) */
        S[14].man = 2.0;
150     S[14].l = 2.0;
        S[14].j = 2.5;
152     S[14].F = 2.0;
        S[14].w = TwoPi*770.576019193;
154     S[14].Gamma = Gamma33/2.0;
        /* 5D(5/2)F(1) */
156     S[15].man = 2.0;
        S[15].l = 2.0;
158     S[15].j = 2.5;
        S[15].F = 1.0;
160     S[15].w = TwoPi*770.576035133;
        S[15].Gamma = Gamma33/2.0;
162
}
```

APÊNDICE C - DANIEL.H: FUNÇÕES AUXILIARES CRIADAS PELO DANIEL FELINTO

```
#include  <math.h>

#define   SWAP(a,b) tempr=(a);(a)=(b);(b)=tempr
#define   FUNC(x)   ((*func)(x))
#define   EPS     1.0e-5
#define   JMAX    40
#define   JMAXP     (JMAX+1)
#define   K     5

double  *dvector(int nl,int nh);
void  free_dvector(double *v,int nl,int nh);
void  nrerror(char error_text[]);
void  FFT(double coord[],double data[],int nn,double dd);
void  IFFT(double coord[],double data[],int nn,double dd);
void  four1(double data[],int nn,int isign);
double  trapzd(double (*func)(double), double a, double b, int n);
double  qtrap(double (*func)(double), double a, double b);

void nrerror(char error_text[])
{
  fprintf(stderr,"Numerical Recipes run-time error...\n");
  fprintf(stderr,"%s\n",error_text);
  fprintf(stderr,"...now exiting to system...\n");
  exit(1);
}

double *dvector(int nl,int nh)
/* Allocates a double vector with range [nl..nh] */
{
  double  *v;

  v=(double *)malloc((unsigned) (nh-nl+1)*sizeof(double));
  if (!v) nrerror("allocation failure in dvector()");
  return v-nl;
}

void free_dvector(double *v,int nl,int nh)
```

```
/* Frees a double vector allocated by dvector() */
{
  free((char*) (v+nl));
}

void FFT(double coord[],double data[],int nn,double dd)
{
  int c1;
  double  tempr;

  four1(data-1,nn,1);

  for(c1=0;c1<nn;c1+=2){
    SWAP(data[c1],data[c1+nn]);
    SWAP(data[c1+1],data[c1+1+nn]);
  }
  for(c1=0;c1<nn;c1+=1){
    coord[c1]=2*3.14159265*(c1-nn/2)/(nn*dd);
    data[2*c1] *= dd;
    data[2*c1+1] *= dd;
  }
}

void IFFT(double coord[],double data[],int nn,double dd)
{
  int c1;
  double  tempr;

  for(c1=0;c1<nn;c1+=2){
    SWAP(data[c1],data[c1+nn]);
    SWAP(data[c1+1],data[c1+1+nn]);
  }

  four1(data-1,nn,-1);

  for(c1=0;c1<nn;c1+=1){
    coord[c1]=(c1-nn/2)/(nn*dd);
    data[2*c1] *= dd;
    data[2*c1+1] *= dd;
```

```
77    }
  }
79
  void four1(double data[],int nn,int isign)
81{
    int n,mmax,m,j,istep,i;
83  double  wtemp,wr,wpr,wpi,wi,theta;
    double  tempr,tempi;
85
    n=nn<<1;
87  j=1;
    for(i=1;i<n;i+=2){
89    if(j>i){
        SWAP(data[j],data[i]);
91      SWAP(data[j+1],data[i+1]);
      }
93    m=n>>1;
      while(m >= 2 && j > m){
95      j -= m;
        m >>= 1;
97    }
      j += m;
99  }
    mmax=2;
101 while(n>mmax){
      istep=mmax << 1;
103   theta=isign*(6.28318530717959/mmax);
      wtemp=sin(0.5*theta);
105   wpr = -2.0*wtemp*wtemp;
      wpi=sin(theta);
107   wr=1.0;
      wi=0.0;
109   for(m=1;m<mmax;m+=2){
        for(i=m;i<=n;i+=istep){
111       j=i+mmax;
          tempr=wr*data[j]-wi*data[j+1];
113       tempi=wr*data[j+1]+wi*data[j];
          data[j]=data[i]-tempr;
115       data[j+1]=data[i+1]-tempi;
```

```
        data[i] += tempr;
117     data[i+1] += tempi;
      }
119   wr=(wtemp=wr)*wpr-wi*wpi+wr;
      wi=wi*wpr+wtemp*wpi+wi;
121   }
    mmax=istep;
123 }
}
125

double trapzd(double (*func)(double), double a, double b, int n)
127 /* This routine computes the n'th stage of refinement of an extended
    trapezoidal rule.
func is input as a pointer to the function to be integrated between
    limits a and b,
129 also input. When called with n=1, the routine returns the crudest
    estimate
of the integral of f(x) from a to b. Subsequent calls with n=2,3,...
131 (in that sequential order) will improve the accuracy by adding 2^(n-2)
    additional
interior points. */
133 {

135   double    x,tnm,sum,del;
  static double s;
137   int     it,j;

139   if (n == 1) {
    return (s=0.5*(b-a)*(FUNC(a)+FUNC(b)));
141   } else {
    for (it=1,j=1;j<n-1;j++) it <<= 1;
143   tnm=it;
    del=(b-a)/tnm;  /* This is the spacing of the points to be added. */
145   x=a+0.5*del;
    for (sum=0.0,j=1;j<=it;j++,x+=del) sum += FUNC(x);
147   s=0.5*(s+(b-a)*sum/tnm);  /*This replaces s by its refined value. */
    return s;
149 }
}
```

```
151
double qtrap(double (*func)(double), double a, double b)
153 /* Returns the integral of the function func from a to b. The parameter
       EPS can be
set to the desired fractional accuracy and JMAX so that 2^(JMAX-1) is the
155 maximum allowed number of steps. Integration is performed by the
       trapezoidal rule. */
{
157   double  trapzd(double (*func)(double), double a, double b, int n);
   void  nrerror(char error_text[]);
159   int   j;
   double  s,olds;
161
   olds = -1.0e30;     /* Any number that is unlikely to be the average of
      the */
163   for (j=1;j<=JMAX;j++) {  /* function at its endpoints will do here. */
     s=trapzd(func,a,b,j);
165     if (j > 5)    /* Avoid spurious early convergence. */
       if (fabs(s-olds) < EPS*fabs(olds) ||
167         (s == 0.0 && olds == 0.0)) return s;
       olds=s;
169     }
     nrerror("Too many steps in routine qtrap");
171     return 0.0;   /* Never get here. */
}
173
double qsimp(double (*func)(double), double a, double b)
175 /*Returns the integral of the function func from a to b. The parameters
       EPS can be
set to the desired fractional accuracy and JMAX so that 2^(JMAX-1) is the
      maximum
177 allowed number of steps. Integration is performed by Simpson's rule. */
{
179   double  trapzd(double (*func)(double), double a, double b, int n);
   void  nrerror(char error_text[]);
181   int   j;
   double  s,st,ost,os;
183
   ost = os = -1.0e30;
```

```
  for (j=1;j<=JMAX;j++) {
    st=trapzd(func,a,b,j);
    s=(4.0*st-ost)/3.0;  /* Compare equation (4.2.4), above. */
    if (j > 5)           /* Avoid spurious early convergence. */
      if (fabs(s-os) < EPS*fabs(os) ||
        (s == 0.0 && os == 0.0)) return s;
    os=s;
    ost=st;
  }
  nrerror("Too many steps in routine qsimp");
  return 0.0;   /* Never get here. */
}

/*Here EPS is the fractional accuracy desired, as determined by the
   extrapolation
error estimate; JMAX limits the total number of steps; K is the number of
    points
used in the extrapolation. */

double qromb(double (*func)(double), double a, double b)
/*Returns the integral of the function func from a to b. Integration is
   performed
by Romberg's method of order 2K, where, e.g., K=2 is Simpson's rule. */
{
  void  polint(double xa[], double ya[], int n, double x, double *y,
   double *dy);
  double  trapzd(double (*func)(double), double a, double b, int n);
  void  nrerror(char error_text[]);
  double  ss,dss;
  double  s[JMAXP],h[JMAXP+1]; /* These store the successive trapezoidal
   approxi-*/
  int   j;      /* mations and their relative stepsizes. */

  h[1]=1.0;
  for (j=1;j<=JMAX;j++) {
    s[j]=trapzd(func,a,b,j);
    if (j >= K) {
      polint(&h[j-K],&s[j-K],K,0.0,&ss,&dss);
      if (fabs(dss) <= EPS*fabs(ss)) return ss;
```

```
219     }
        h[j+1]=0.25*h[j];
221     /* This is a key step: The factor is 0.25 even though the stepsize */
        /* is decreased by only 0.5. This makes the extrapolation a
       polynomial */
223     /* in h 2 as allowed by equation (4.2.1), not just a polynomial in h.
        */
      }
225   nrerror("Too many steps in routine qromb");
      return 0.0; /* Never get here. */
227 }

229 void polint(double xa[], double ya[], int n, double x, double *y, double
       *dy)
    /* Given arrays xa[1..n] and ya[1..n], and given a value x, this routine
       returns a
231 value y, and an error estimate dy. If P(x) is the polynomial of degree N
       -1 such
    that P(xa_i) =ya_i ; i= 1,...,n, then the returned value y = P(x). */
233 {
      int   i,m,ns=1;
235   double  den,dif,dift,ho,hp,w;
      double  *c,*d;
237

      dif=fabs(x-xa[1]);
239   c=dvector(1,n);
      d=dvector(1,n);
241   for (i=1;i<=n;i++) {  /*Here we find the index ns of the closest table
       entry, */
        if ( (dift=fabs(x-xa[i])) < dif) {
243       ns=i;
          dif=dift;
245     }
        c[i]=ya[i];    /* and initialize the tableau of c's and d's. */
247     d[i]=ya[i];
      }
249   *y=ya[ns--];                    /* This is the initial approximation to
       y. */
      for (m=1;m<n;m++) {             /* For each column of the tableau, */
```

```
    for (i=1;i<=n-m;i++) { /* we loop over the current c's and d's and */
      ho=xa[i]-x;    /* update them. */
      hp=xa[i+m]-x;
      w=c[i+1]-d[i];
      if ( (den=ho-hp) == 0.0) nrerror("Error in routine polint");
      /* This error can occur only if two input xa's are
         (to within roundo ) identical. */
      den=w/den;
      d[i]=hp*den; /* Here the c's and d's are updated. */
      c[i]=ho*den;
    }
    *y += (*dy=(2*ns < (n-m) ? c[ns+1] : d[ns--]));
    /* After each column in the tableau is completed, we decide which
       correction, c or d, we want to add to our accumulating value of y,
       i.e., which path to take through the tableau-forking up or down.
       We do this in such a way as to take the most "straight line" route
       through the tableau to its apex, updating ns accordingly to keep
       track of where we are. This route keeps the partial approximations
       centered (insofar as possible) on the target x. The last dy added
       is thus the error indication. */
  }
  free_dvector(d,1,n);
  free_dvector(c,1,n);
}
```