Aplicação de Python OpenCL na paralelização da simulação de um cubo de estados iniciais segundo as equações de Lorenz

Felipe O. Paulino[1]

[1]Departamento de Física – Universidade Federal de São Carlos (UFSCar)
Caixa Postal 676 – 13.565-905 – São Carlos – SP – Brazil

Abstract. *Lorenz's equations are a well-known system in the context of chaos theory and dynamical systems. In this work, we present a parallel application, based on Python OpenCL, to calculate the trajectories of a cube of initial conditions within the phase space of the system. A comparison with a sequential implementation is made, as well as a performance analysis for cases in which different amounts of time steps are calculated per work-item. The trajectories were animated in order to observe some important features of the system, such as the volumetric contraction in the phase space and the presence of a fractal attractor.*

Resumo. *As equações de Lorenz consituem um conhecido sistema no contexto de teoria do caos e sistemas dinâmicos. Neste trabalho, é proposta uma aplicação paralela, baseada em Python OpenCL, para calcular as trajetórias de um cubo de condições iniciais no espaço de fases do sistema. É feita uma comparação com a implementação sequencial, bem como uma análise de desempenho para casos em que diferentes quantidades de passos no tempo são calculadas por work-item. As trajetórias foram animadas de maneira a se observar algumas caraterísticas importantes do sistema, como a contração volumétrica no espaço de fases e a presença de um atrator fractal.*

1. Descrição do Problema

Em 1880, estudos referentes ao problema dos três corpos levaram a descobertas de órbitas não periódicas e que não tendiam a pontos fixos do espaço de estados possíveis do sistema (espaço de fases) [Poincaré and Popp 2017]. Desde então, diversos estudos referentes a problemas com características similares foram publicados, levando a descoberta dos sistemas caóticos, bem como de algumas de suas propriedades [Birkhoff 1927, Kolmogorov 1941, Cartwright and Littlewood 1945].

Caos pode ser definido, no contexto de sistemas dinâmicos, como um comportamento aperiódico a longo prazo que apresenta alta sensibilidade às condições iniciais [Strogatz 2007]. Em síntese, isso significa que existem trajetórias no espaço de fases desses sistemas que não tendem a pontos fixos, nem a órbitas periódicas ou quasiperiódicas, e que trajetórias próximas entre si se distanciam de maneira exponencial ao longo do tempo (coeficiente de Liapounov positivo). Esses sistemas são determinísticos no sentindo de que seu comportamento, apesar de difícil de prever no longo prazo, não é inflenciado por fenômenos aleatórios de qualquer natureza, como acontece em sistemas estocásticos.

Em 1963, no contexto do estudo de convecção hidrodinâmica, foi proposto, a partir de simplificações das equações de Saltzman, um conjunto de equações diferenciais

para descrever a dinâmica de convecções atmosféricas [Lorenz 1963]. As equações de movimento do sistema proposto por Lorenz se escrevem como

$$\dot{x} = -\sigma(y - x), \tag{1a}$$
$$\dot{y} = x(\rho - z) - y, \tag{1b}$$
$$\dot{z} = xy - \beta z, \tag{1c}$$

sendo que o ponto em cima das variáveis é utilizado para denotar derivada temporal.

As variáveis x, y e z são as variáveis que descrevem o sistema, que tem, portanto, um espaço de fases tridimensional. Em outras palavras, cada estado do sistema é descrito como um ponto (x, y, z) de tal espaço. Essas variáveis são dependentes do tempo e estão relacionadas às condições atmosféricas. A saber, x é proporcional à intensidade da convecção, y está relacionado à diferença de temperatura das correntes de convexão e z ao gradiente vertical de temperatura. Com o passar do tempo, um certo estado inicial (x_0, y_0, z_0) evolui segundo a Eq. (1), descrevendo uma trajetória $(x(t), y(t), z(t))$ no espaço de fases.

Já σ, ρ e β são parâmetros constantes, sendo que σ é o número de Prandtl, ρ é o número de Rayleigh e β é uma constante relacionada ao diâmetro dos rolos convectivos.

Esse conjunto de equações apresenta duas não linearidades, uma no termo $-xz$ da Eq. (1b) e outra no termo xy da Eq. (1c). Essas não linearidades fazem que o sistema apresente comportamento caótico para $\rho > 1$ [Lorenz 1963]. Além disso, é importante ressaltar que se trata de um sistema autônomo, na medida em que as equações de movimento não dependem explicitamente do tempo.

Uma propriedade bastante característica desse conjunto de equações é que, para certos valores de ρ, seu espaço de fases possui dois pontos fixos instáveis que criam uma região atratora de dimensão fractal. Os atratores são regiões no espaço de fases de um sistema dinâmico para as quais tendem as trajetórias que se aproximam o suficiente. Sistemas caóticos não raramente apresentam atratatores de dimensão não inteira (fractal), como é o caso dos sistema de Lorenz [Procaccia 1988]. Tal atrator tem formato que assemelha-se às asas de uma borboleta, de onde provém o termo "efeito borboleta", usado para referir-se informalmente à caoticidade de certos sistemas.

Outra característica interessante dessas equações, é que qualquer volume de estados no espaço de fases contrai-se exponencialmente na medida em que as trajetórias tendem ao atrator, que por sua vez tem volume nulo [Strogatz 2007].

O objetivo deste trabalho é propagar no tempo um cubo de condições iniciais segundo as equações de Lorenz, de maneira sequencial e paralelizada e comparar os desempenhos computacionais em cada caso. Espera-se que se que na simulação seja possível observar algumas das características citadas acima, como a contração volumétrica do cubo e a tendência das partículas à região atratora. Como método de propagação dos problemas de valor inicial será utilizado o Runge-Kutta de quarta ordem (RK4).

O método RK4 é definido a partir de uma expansão em Taylor e seu erro é da ordem $O(h^4)$, sendo h o passo de tempo adotado [R. Bulirsch 1996]. Como o sistema de interesse é um sistema autônomo, definindo $\vec{r} = (x, y, z)$ pode-se escrever vetorialmente

a equação de propagação do RK4 como

$$\vec{r}_{n+1} = \vec{r}_n + \frac{h}{6}(\vec{k}_1 + 2\vec{k}_2 + 2\vec{k}_3 + \vec{k}_4) \tag{2}$$

sendo

$$\begin{aligned}
\vec{k}_1 &= \frac{d\vec{r}}{dt}(\vec{r}) \\
\vec{k}_2 &= \frac{d\vec{r}}{dt}\left(\vec{r} + \frac{h}{2}\vec{k}_1\right) \\
\vec{k}_3 &= \frac{d\vec{r}}{dt}\left(\vec{r} + \frac{h}{2}\vec{k}_2\right) \\
\vec{k}_4 &= \frac{d\vec{r}}{dt}\left(\vec{r} + h\vec{k}_3\right).
\end{aligned} \tag{3}$$

Será considerado, neste trabalho, um cubo de $21 \times 21 \times 21$ estados iniciais, sendo que, como dito anteriormente, cada estado é definido por três variáveis (x, y e z) em um espaço de fases tridimensional. A propagação no tempo será feita com um passo de 0.01, desde um tempo inicial $t_i = 0$ até um tempo final $t_f = 1.0$. Tais parâmetros foram escolhidos por experimentação, isto é, com esse passo de tempo percebe-se que a propagação ocorre de maneira suave e neste tempo final o sistema se mostra suficiente bem desenvolvido.

Com tais parâmetros, ressalta-se que em cada instante de tempo os diversos estados são descritos por três tensores de $21 \times 21 \times 21 = 9261$ elementos, cada um referente a uma das variáveis de estado, x, y e z. Isso significa que a cada um dos 100 passos de tempo considerados é necessário aplicar o método de propagação (Runge-Kutta de quarta ordem) em cada uma das três variáveis de estado de cada um dos 9261 estados considerados.

Com relação às constantes, serão adotados os mesmos valores propostos por Lorenz em seu artigo de 1963, isto é, $\sigma = 10$, $\rho = 28$ e $b = \frac{8}{3}$.

Destarte, almeja-se que o código desenvolvido possa ser facilmente adaptado para outros sistemas autônomos tridimensionais, descritos por diferentes equações de movimento.

2. Estratégia de Paralelização

Como o objetivo é simular a propagação de diversos problemas de valor inicial independentes (já que se tratam de condições iniciais distintas no espaço de fases), então a propagação de cada estado pode ser realizada de maneira paralelizada sem grandes problemas, haja visto que um estado não afeta e nem é afetado por outros estados.

Não é eficiente, contudo, propagar cada uma das variáveis x, y e z de um estado de maneira paralela, pois a taxa de variação de cada uma delas depende das demais. Além disso, não é possível calcular diferentes instantes de tempo de maneira paralela, pois, evidentemente, a propagação em instantes de tempos futuros depente da evolução do sistema em instantes de tempos anteriores.

Assim, a decomposição de dados será feita a nível de estados, ou seja, cada unidade de trabalho irá propagar um estado (x, y, z).

Como ferramenta de paralelização foi utilizado Python OpenCL, em função das facilidades oferecidas pelos pacotes numpy e pandas no que se refere à formatação e manipulação de dados. O dispositivo OpenCl adotado foi uma placa de vídeo NVIDIA GeForce GTX 1050 Ti, que conta com 4Gb de memória global e 768 CUDA Cores.

Inicialmente, a paralelização havia sido pensada de modo que a cada passo de tempo fosse enviada uma fila de instruções ao device openCL, que calcularia os estados do tempo seguinte e retornaria a informação ao host. No host, os novos estados seriam adicionados a um dataframe pandas e enviados novamente ao device OpenCL, para que fossem calculados os novos estados no instante de tempo seguinte. Esse processo se repetiria até atingir o tempo final.

Contudo, posteriormente os códigos do kernel e do host foram modificados de maneira que se possa escolher quantos passos de tempo cada work-item irá calcular a cada fila de instruções. Sendo assim, quanto mais passos no tempo forem calculados a cada fila, menor será o número de interações entre o host e o device OpenCL, de modo que espera-se que melhor seja o desempenho. Uma análise comparativa será feita na Seção 5.

Por praticidade, os três tensores que descrevem as variáveis de estado iniciais foram convertidos em três arrays unidimensionais com 9261 elementos cada, evitando-se assim complicações com controle de índices. Foi utilizado um dataframe com quatro colunas (uma para o tempo e uma para cada uma das variáveis de estado) para armazenar todas as trajetórias calculadas (isto é, para cada instante de tempo há 9261 linhas no dataframe, referentes aos 9261 estados considerados). Ao fim, o dataframe é exportado para um arquivo csv para que possa ser animado externamente.

3. Versão sequencial

O trecho sequencial da propagação, implementada em Python, segue abaixo, sendo que

- `x0,y0,z0` são três arrays numpy unidimensionais contendo os estados iniciais. São consistentes entre si, no sentido de que para uma mesma posição os três arrays se referem a um mesmo estado.
- `df` é o dataframe pandas para armazenar as trajetórias
- `numIter` é o número de passos de tempo utilizados
- `n_estados` é o número de estados considerados
- `rk4_nextStep` é uma rotina que recebe como parâmetros o estado atual (`pos`, array numpy unidimensional de três elementos) e o passo de tempo adotado (`timeStep`) e retorna um array numpy referente ao estado seguinte.

```
x=np.copy(x0)
y=np.copy(y0)
z=np.copy(z0)
tdf=np.full(n_estados,0.0)
df = pd.DataFrame({"time": tdf ,"x" : x, "y" : y, "z" : z})
start_time = time()
for j in range (numIter):
    for i in range(n_estados):
        pos=np.array([x[i],y[i],z[i]])
```

```
        newPos=rk4_nextStep(pos,timeStep)
        x[i]=newPos[0]
        y[i]=newPos[1]
        z[i]=newPos[2]
    t=(j+1)*timeStep
    tdf=np.full(n_estados,t)
    df2 = pd.DataFrame({"time": tdf ,"x" : x, "y" : y, "z" : z})
    df=pd.concat([df,df2])
run_time = time() - start_time
```

O pacote `time` foi usado para medir o tempo de execução. A definição do método `rk4_nextStep`, bem como das equações de movimento podem ser vistas abaixo

```
def drdt(r):
    return np.array([
        sigma*(r[1]-r[0]),
        r[0]*(rho-r[2])-r[1],
        (r[0]*r[1])-(beta*r[2])
    ])

def rk4_nextStep(r,step):
    k1 = step*drdt(r)
    k2 = step*drdt(r + 0.5*k1)
    k3 = step*drdt(r + 0.5*k2)
    k4 = step*drdt(r + k3)
    r=r+(1/ 6.0)*(k1 + 2*k2 + 2*k3 + k4)
    return r
```

4. Versão Paralela

A parte de interesse no código do host foi desenvolvida tendo como referência os códigos apresentados durante a disciplina e é apresentada logo abaixo, sendo que

- `n_estados` é o número de estados considerados
- `numStepsWI` é o número de passos no tempo calculados por cada work-item;
- `numQ` numéro de filas de execução enviadas ao device;
- `h_x0,h_y0,h_z0` são arrays unidimensionais de `n_estados` elementos, cada, contendo as variáveis de estado iniciais;
- `h_x,h_y,h_z` são arrays unidimensioais com `n_estados x numStepsWI` elementos cada. Estes vetores são inicializados de modo que os `n_estados` elementos finais correspondem aos estados iniciais e os demais elementos, se houver, são nulos. A partir destes arrays são criados os buffers do device `d_x,d_y,d_z`. Optou-se por implementar dessa maneira, pois o código do kernel utiliza os últimos estados do buffer (referentes ao instante de tempo mais recente) para calcular os `numStepsWI` instantes seguintes.

```
tdf=np.full(n_estados,0.0)
df = pd.DataFrame({"time": tdf ,"x" : h_x0, "y" : h_y0, "z" : h_z0})

#inicia medicao de tempo
start_time = time()

#definicao dos buffers do host
```

```
h_x=np.concatenate([np.zeros((numStepsWI-1)*n_estados).astype(np.
    float32),h_x0])
h_y=np.concatenate([np.zeros((numStepsWI-1)*n_estados).astype(np.
    float32),h_y0])
h_z=np.concatenate([np.zeros((numStepsWI-1)*n_estados).astype(np.
    float32),h_z0])

#definicao dos buffers do dispositivo
d_x = cl.Buffer(context, cl.mem_flags.READ_WRITE | cl.mem_flags.
    COPY_HOST_PTR, hostbuf=h_x)
d_y = cl.Buffer(context, cl.mem_flags.READ_WRITE | cl.mem_flags.
    COPY_HOST_PTR, hostbuf=h_y)
d_z = cl.Buffer(context, cl.mem_flags.READ_WRITE | cl.mem_flags.
    COPY_HOST_PTR, hostbuf=h_z)

#A cada iteracao sao calculados numStepsWI passos de tempo por cada
    work-item
for j in range(numQ):
    prop(queue, (n_estados,), None,numStepsWI, n_estados, sigma,rho,
    beta,timeStep, d_x,d_y, d_z)
    queue.finish()
    run_time = time() - start_time
    cl.enqueue_copy(queue, h_x, d_x)
    cl.enqueue_copy(queue, h_y, d_y)
    cl.enqueue_copy(queue, h_z, d_z)

    tdf = np.array([np.ones(n_estados)*((i+1)+(j*numStepsWI))*timeStep
    for i in range(numStepsWI)]).flatten()
    df2 = pd.DataFrame({"time": tdf ,"x" : h_x, "y" : h_y, "z" : h_z})
    df=pd.concat([df,df2])
run_time = time() - start_time
```

O códido do kernel implementa o método RK4 de modo a calcular `numStepsWI` passos no tempo e armazenar os resultados na memória global. Dessa maneira, ao final da fila de execução, o device retorna ao host `numStepsWI` passos no tempo para cada um dos `n_estados`.

5. Análise de Escalabilidade

Para cada caso de estudos foram realizadas dez execuções e calculadas as médias dos tempos de execução, o desvio padrão e os intervalos de confiança de 95%. Os resultados obtidos para a execução sequencial são apresentados na Tabela (1).

Table 1. Tempo médio de execução, desvio padrão e intervalo de confinça para propagação sequencial.

Média [s]	22,8991
Desvio Padrão [s]	0,8043
Intervalo de Confiança (95%)	0,4985

Para a implementação paralela foram escolhidos diferentes valores para o número de passos de tempo executados por cada kernel. Como, para os parâmetros de tempo adotados, são calculados 100 passos totais, foram avaliados, então, os casos em que cada

work-item calcula 1, 2, 4, 5, 10, 20, 25, 50 e 100 passos a cada fila de execução, de modo que a divisão entre o número de passos totais no tempo e número passos a cada fila deixe resto zero. As médias dos tempos de execução, desvios padrão e intervalo de confiança de 95% são apresentados na Tabela (2). Na Tabela (2) são apresentados ainda o speedup obtido para cada caso, calculado dividindo o tempo de execução paralelo pelo tempo de execução sequencial.

Table 2. Tempo médio de execução, desvio padrão e intervalo de confinça para propagação paralela.

Passos por work-item	1	2	4	5	10	20	25	50	100
Média [s]	0,82818	0,44881	0,22630	0,18620	0,10761	0,07569	0,06712	0,06293	0,07111
Desvio Padrão [s]	0,02508	0,01356	0,00992	0,00745	0,00424	0,00157	0,00179	0,00192	0,00200
Intervalo de Confiança (95%)	0,01554	0,00841	0,00615	0,00462	0,00263	0,00098	0,00111	0,00119	0,00124
Speedup	27,6498	51,0215	101,1903	122,9787	212,7891	302,5442	341,1622	363,8534	322,0216

Enquanto a implementação sequencial leva, em média, $22,8991s$ para executar, o caso paralelizado de menor desempenho (isto é, o caso em que cada work-item calcula apenas um passo no tempo) leva em média $0,8282s$, correspondendo a um speedup de, aproximadamente, $27,6498$. Pode-se concluir, portanto, que a paralelização do problema foi efetiva. Os dados da Tabela (2) são sintetizados no gráfico da Figura (1).

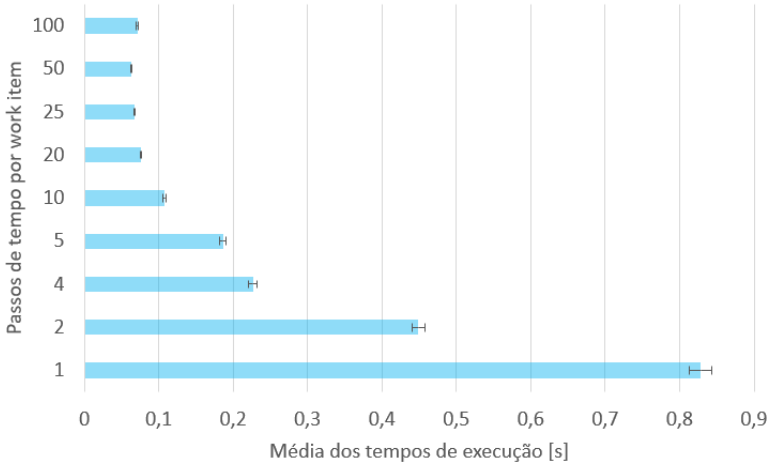

Figure 1. Tempos de execução médios para paralelização da propagação de acordo com o número de passos por work-item.

Nota-se, observando a Tabela (2) e a Figura (1) que, na medida em que maior é o número de passos calculados por cada work-item, melhor é o desempenho. Isso se deve, possivelmente, ao fato de que, quanto menos passos são computados por work-item, mais trocas de informação são realizadas entre o host e o device (ou seja, mais filas de execução são necessárias) e, dada a latência na comunicação entre os dispositivos, maior é o tempo de processamento. Uma excessão notável é o caso em que todos os 100 passos de tempo são calculados em uma única fila de instruções, sendo o tempo de processamento ligeiramente maior que o caso com 50 passos por work-item. Um fator que poderia explicar esse fenômeno é que o grande volume de escritas e leituras na memória global do device OpenCL dificultaria o gerenciamento das informações internamente e contribuiria para um maior tempo de processamento.

6. Discussão Sobre a Eficiência da Solução

Conforme apresentado na Seção anterior os casos paralelizados apresentaram um grande ganho de desempenho em relação à implementação sequencial. Tal eficiência está relacionada à natureza massivamente paralela do problema estudado, uma vez que não há necessidade de comunicação entre work-items, já que a propagação de cada estado é completamente independente dos demais. Isso faz com não haja a necessidade de realizar qualquer tipo de sincronização de dados no device.

Além disso, é importante ressaltar que a tarefa de propagar um estado é bastante simples, de maneira que o maior custo está na quantidade de vezes que essa tarefa deve ser executada. Sendo assim, aumentar o número de unidades de processamento, executando os cálculos em uma GPU, se demonstrou muito eficiente.

É necessário, contudo, atentar-se à quantidade de informação enviada ao dispositivo OpenCL, haja vista que, ao lidar com grandes volumes de dados simultaneamente o desempenho pode cair, como visto acima.

7. Animação

Para a animação, foi utilizado um script python para carregar os dados em um dataframe e atualizar a plotagem de acordo com cada instante de tempo através da função `FuncAnimation`. Tal script foi adaptado de um código disponibilizado neste link.

Os estados do sistema para alguns instantes de tempo são apresentados abaixo, na Figura (2).

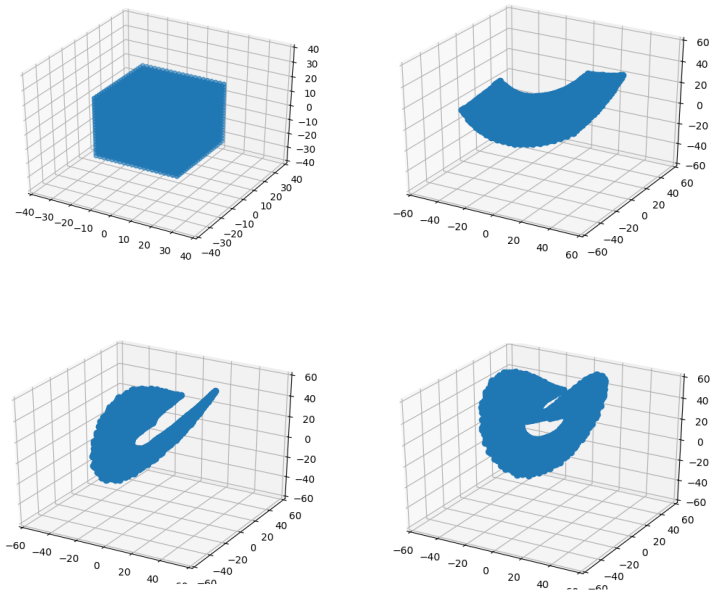

Figure 2. Estados dos sistemas em diferentes instantes de tempo.

Na imagem é possível notar que o cubo se contrai na medida em os estados evoluem. Como muitos estados são propagados simultaneamente eles povoam a região atratora de modo que é possíver observar sua forma no espaço de fases. Como se supunha, a forma da região atratora assemelha-se às asas de um borboleta.

8. Conclusões

A paralelização da simulação do cubo de estados iniciais segundo as equações de Lorenz com pyopencl se mostrou altamente eficiente, se comparada com a execução sequencial. Observou-se que, geralmente, quanto maior o número de passos de tempo calculados por work-item, melhor é o desempenho. Contudo, caso a quantidade de informação processada por work-item seja muito grande, o desempenho pode cair. Isso se deve, possivelmente, à adição de overhead devido a dificuldades de gerenciamento de informações no device OpenCL. Ao animar o dataFrame gerado, pôde-se, como pretendido, observar algumas características importantes do sistema estudado, como a contração volumétrica de estados e a forma do atrator de Lorenz.

9. Códigos completos

Os códigos completos podem ser acessado no repositório disponível no seguinte link: https://github.com/felipe-paulino/lorenz-propagation.

References

[Birkhoff 1927] Birkhoff, G. D. (1927). *Dynamical systems*, volume 9. American Mathematical Soc.

[Cartwright and Littlewood 1945] Cartwright, M. and Littlewood, J. (1945). On non-linear differential equations of the second order. *J. London Math. Soc.*, 20:180–189.

[Kolmogorov 1941] Kolmogorov, A. (1941). The local structure of turbulence in incompressible viscous fluid for very large reynolds numbers, philos. Technical report, TR Soc. S.-A, 434, 9–13.

[Lorenz 1963] Lorenz, E. N. (1963). Deterministic nonperiodic flow. *Journal of the atmospheric sciences*, 20(2):130–141.

[Poincaré and Popp 2017] Poincaré, H. and Popp, B. (2017). *The Three-Body Problem and the Equations of Dynamics: Poincaré's Foundational Work on Dynamical Systems Theory*. Astrophysics and Space Science Library. Springer International Publishing.

[Procaccia 1988] Procaccia, I. (1988). Universal properties of dynamically complex systems: the organization of chaos. *Nature*, 333(6174):618–623.

[R. Bulirsch 1996] R. Bulirsch, J. S. (1996). *Introduction to numerical analysis*. Texts in Applied Mathematics, No 12. Springer, 2nd edition.

[Strogatz 2007] Strogatz, S. H. (2007). *Nonlinear Dynamics And Chaos*. Studies in nonlinearity. Sarat Book House.