

REINTERPRETAÇÃO ANALÍTICO-SIMBÓLICA, DO DESLOCAMENTO

TEMPORAL QUÂNTICO DA FUNÇÃO DE ONDA: UM CASO

PARTICULAR ENSAIADO EM PYTHON

DELMONTE, Samuel¹

TEDESCO, Daniel Guimarães²

RESUMO

A Regressão Simbólica (**RS**) é um procedimento de modelagem matemática, cuja inspiração no Aprendizado de Máquina Profundo, permitiu que suas redes neurais, sugerissem padrões matemáticos completos, em forma de funções, de certos dados correlacionados. É para as Ciências Físicas, um recurso promissor, quanto à expansão do conhecimento, já adquirido. O presente ensaio tem por fim, reinterpretar a natureza da **Equação de Schroedinger**, como um postulado, regedor das leis do movimento, por meio de algoritmos, simuladores e redes neurais, codificados em Python, e submetidos às redes neurais simbólico-regressivas. A finalidade central deste trabalho, é interpretar a sintaxe da mecânica quântica, por meio da linguagem de programação **Python**, desenvolvendo soluções, numéricas e analíticas, da **eqS**, tanto dependente, quanto independente ao tempo, reduzindo, deste modo, a complexidade e dificuldade dos rigores da matemática quântica, proporcionando melhor desempenho nas atividades, cuja Física Quântica, faz-se necessária;; ademais, eles, os scripts em Python, disponibilizam eficiente cenário de aprendizado de ambos os paradigmas (Física Quântica e Ciência de dados); por último, mas não menos importantes, pretende-se, constituir uma coleção de equações, condizentes ao escopo, deste artigo, por meio do método regressivo-simbólico, permitindo, grosso moldo, que trabalhos futuros, verifique-las. O cenário empírico de testes selecionado, é uma representação de um **poço quadrado de potencial infinito**, com uma partícula livre, enclausurada, no seu interior. Para isso, foram empregados os seguintes procedimentos metodológicos: uma abordagem quantitativa, quanto à sua forma, e o recurso bibliográfico, como procedimento técnico; acerca dos instrumentos, utilizar-se-á, sobretudo, um levantamento bibliográfico, composto por livros, artigos, revistas, cartas, repositórios WEB e materiais didáticos. A pesquisa, aqui realizada, culminou em resultados promissores, afinal, pode-se criar trechos algorítmicos, fiéis ao formalismo teórico da mecânica quântica, contudo,

¹ Bacharelando em Física, no Centro Universitário Internacional Uninter.

² Professor orientador convidado pelo Centro Universitário Internacional Uninter.

abstraídos das complexidades e dificuldades matemáticas, resultando assim, em um meio capaz de manipular dados e informações quânticas com bom desempenho, rapidez, confiabilidade e com potencial de gerar nós que ramificam descendentes, muitos deles polimórficos. Os subconjuntos de treinamento e testes, analisados, por meio de um modelo de **Regressão Simbólica**, retornaram, em todos os casos, vasta sugestão de equações, atingindo o objetivo, instaurado a priori, de criar um acervo de padrões matemáticos, a serem analisados em um trabalho futuro. Diante desses resultados, o autor sugere a continuidade intensificada de trabalhos científicos, neste paradigma de pesquisa, aqui esboçado.

Palavras-chave: Schroedinger. Regressão. Simbólica. Python. Reinterpretação.

1. INTRODUÇÃO

A presente proposta de pesquisa, reúne argumentos, cuja temática versa a respeito de uma análise de regressão simbólica, de um caso particular e reduzido de deslocamento temporal quântico, duma função de onda, por meio da perspectiva abstraída de um modelo de partícula numa caixa, mais especificamente, de um poço quadrado de potencial infinito

Para tanto, utilizar-se-á, a linguagem de programação Python, robusta o bastante, para oferecer algoritmos, que retornem simulações, executadas diretamente ou com a inserção das variáveis iniciais, das soluções, numéricas e analíticas, das mensurações plausíveis, do postulado quântico, manifestado, através da **eqS**, restrita aos pressupostos edificadores, do caso especial, aqui já enunciado.

O propósito principal desta empreitada, isto é, a razão pela qual, transcodificar-se- a linguagem algébrica da mecânica quântica, para a sintaxe de programação, lógica e simbólica, do paradigma manifestado pela linguagem de programação Python, é reduzir os índices de complexidade e dificuldade, existentes nos rigores algébricos da mecânica quântica, em geral. Portanto, a hipótese primordial deste ensaio, versa a respeito de reduzir os impactos desta simbiose de adversidades ao manipular dados e informações quânticas; grosso modo, os algoritmos desenvolvidos, funcionam, análogos ao conceito de *caixa-preta*, que por sua vez, abstrai tais dificuldades, das entrelinhas dos seus procedimentos de execução. O escopo, aqui observado, é de vital importância, ao que se refere em fomentar maior robustez do conhecimento atual, localizado nas periferias do perímetro total do conhecimento humano, pois

a teoria da complexidade, ou dos sistemas complexos definem o índice de complexidade, de acordo com o tempo gasto para que certo padrão, solucione um problema a ele direcionado ,de tal modo que o princípio gerador desta ruptura tecnológica, desencadeada pela Inteligência artificial, resume-se na concretude de processar imensas quantidades de dados, em períodos menores que a existência humana, os denominados ***big datas***³, que, por sua vez, são, por si só, vulneráveis, às mensurações complexas.

Justifica-se, a confecção deste ensaio, aqui discorrido, pela contribuição, mesmo que ínfima, que tal trabalho oferecerá, aos conhecimentos e tecnologias limítrofes da humanidade, tais quais são a Mecânica Quântica e a Física Estatística; invariavelmente, o conjunto de contribuições concedidas ao escopo, pelo autor referido, dilatará os limites perimetrais do conhecimento, transmitindo, às gerações futuras, um repertório tecnológico, mais abrangente.

Ademais, o escopo tem os seguintes fins específicos:

Demonstrar: por meio da sintaxe Python, e algumas das suas bibliotecas (sobretudo **Sympy**⁴, rica em simbologia matemática), soluções à propagação duma função de onda, tanto independente, como dependente do tempo, que satisfaça à equação de Schroedinger.

Abstrair: um modelo reduzido de função de onda, deslocando-se no tempo, como uma partícula livre, limitada a um espaço unidimensional, confinada em um poço quadrático e potencial infinito, cenário, cujo potencial enérgico é igual a zero; ademais, certas intensidades foram normalizadas, a valores unitários, contribuindo para simplificar a complexidade matemática, tão comum, na mecânica quântica.

Correlacionar: os resultados obtidos na aplicação, em python, por meio duma solução à equação de Schroedinger, tanto independente, como dependente do tempo e manifestada, necessariamente, como função densidade de probabilidade, com outros resultados satisfatórios como tais. Por fim, coletar, os resultados sugeridos pela aplicação de aprendizado profundo e regressão simbólica, oferecida como função matemática, para futuramente serem submetidos ao método científico.

O autor, como aspirante a bacharel em física, optou por Ciência e Sociedade, como linha de pesquisa, pois este trabalho discorre, sobretudo, sobre Mecânica Quântica e Física Estatística, subitens de tal linha.

³ Conjunto de dados extremamente volumosos e complexos, que excedem a capacidade dos softwares tradicionais de processamento.

⁴ É uma biblioteca Python de código aberto para computação simbólica.

Do critério adotado na composição do repertório bibliográfico, selecionaram-se, primeiramente, cinco autores renomados, com participação direta na criação e divulgação da Equação de Schroedinger, suas citações, a seguir, corroboram para tanto; estão em ordem cronológica:

Henri Poincaré (1902), afirmou na sua obra *Science and hypothesis*, [...] a indução aplicada às ciências é sempre incerta [...], em alusão aos recentes trabalhos em física quântica, que em geral utilizam a dedução na pesquisa.

Albert Einstein (1915), simplesmente publicou sozinho, a maior adversária da física quântica, sua relatividade geral, em 1915; rivalidade essa, que, de certo modo, ainda existe.

Erwin Schroedinger (1946), confessou a Einstein, numa carta enviada em 1946, a respeito da sua própria equação, a seguinte afirmação, [...] eu não gosto dela, lamento profundamente ter tido alguma coisa a ver com ela [...]; ele morreu convencido disso.

Murray Gel-Mann (1996), foi um dos prenunciadores teóricos, da existência dos quarks, publicou também o *Quark e o Jaguar*, em 1996, livro, cujo um entre muitos dos seus convincentes argumentos dizia que, [...] O falso relato de que a medição de um dos fôtons afeta imediatamente o outro leva a todo tipo de conclusões infelizes. [...].

E, Carlo Rovelli (2017), físico italiano, ainda atuante, figura como um dos precursores da teoria da **Gravidade Quântica em Loop**⁵, além do mais, defende a reinterpretação do tempo nas equações vigentes.

Da linguagem Python, destacaram-se as bibliotecas SymPy (2024) e SciPy (2024), sendo elas codificadas e editadas na interface Anaconda Jupyter Notebook (2024).

Das bibliotecas virtuais, consultou-se a obra de David J. Griffiths, *Mecânica Quântica*, cap. 1,2 e 3; segunda edição, ed. Pearson, 2011; e a obra de Vicente Pereira de Barros, *Princípios da Relatividade: o que há de especial no movimento?* cap. 3 e 5, primeira edição, ed. Inter Saberes. Além de algumas outras referências pertinentes ao escopo.

A exposição argumentativa, produziu a definição de vinte e três equações, associadas à mecânica quântica, seis simuladores quânticos, construídos com robustas bibliotecas, pertencentes ao paradigma da linguagem de programação **python**, além de três redes neurais, e vasto repertorio de padrões matemáticos almejados.

⁵ Teoria física que tenta unificar a relatividade geral de Einstein com a mecânica quântica ao postular que o espaço-tempo, em escalas muito pequenas, é composto por "loops" ou unidades discretas, em vez de um tecido contínuo

A hipótese primeira foi verificada, afinal os algoritmos implementados, reduziram satisfatoriamente, a complexidade e a dificuldade dos rigores algébricos da mecânica quântica, resultados estes, que, traduziram-se em desempenho e otimização de etapas de trabalho; ademais suscitaram indícios que, ao autor, parece-lhe, apontar na, na proporção direta, entre melhor desempenho e algoritmos com teor literal e autoral, com tendência à aproximação da linguagem maquinal, evitando excesso métodos encapsulados e oferecidos pelas bibliotecas da linguagem de programação.

Daqui adiante, dá-se continuidade aos esforços de adquirir uma sapiência, cada vez mais profunda, a respeito da realidade, de tal modo que a inspiração para semelhante empreitada, virá das palavras de Richard Dawkins, quando afirmou que:

“A Natureza não é cruel, apenas implacavelmente indiferente. Esta é uma das lições mais duras que os humanos têm de aprender.” (DAWKINS, 1976).

2. METODOLOGIA

O presente ensaio estabelece como arcabouço metodológico, uma abordagem quantitativa, quanto à sua forma, e o recurso bibliográfico, como procedimento técnico.

Acerca dos instrumentos, utilizar-se-á, sobretudo, um levantamento bibliográfico, composto por livros, artigos, revistas, cartas, repositórios WEB e materiais didáticos; todos eles, especificados na próxima seção.

O conteúdo argumentativo, tem como espinha dorsal, trechos programados em Python, transmutados das equações da mecânica quântica, tais quais são, além de testar a solução da aplicação, por meio da análise de regressão simbólica; que segundo Wassim Tenachi (2023), é a ação de calibrar previamente, o algoritmo com equações fundamentais, que serão combinadas com muitos dados, podendo sugerir uma equação nova, que poderá, quem sabe, solucionar fenômenos não esclarecidos.

Trata-se, portanto, de uma pesquisa exploratória, com hipótese estabelecida no semeio de trabalhos futuros, uma vez que propõe a redução da complexidade e dificuldade dos rigores do formalismo matemático da mecânica quântica, reinterpretando-a, por meio de alternativas computacionais.

3. REVISÃO BIBLIOGRÁFICA/ ESTADO DA ARTE

Nos subitens a seguir, explana-se, a respeito dos passos dados, para demonstrar, verificar e solucionar da equação de Schroedinger, além de sua submeter ditas soluções, à uma análise de regressão simbólica.

3.1. CONCEITUAÇÃO DAS ENTIDADES DE ESCOPO PRINCIPAIS:

3.1.1. A equação de Schroedinger (eqS):

De acordo com **Lisbôa** (2020, p. 105), “a eqS descreve a propagação das ondas de matéria, incluindo os elétrons, sendo aplicada a problemas não relativísticos. Ela está em concordância com os resultados experimentais”.

Contrariamente à lógica conceitual clássica, a eqS não é interpretada como constituidora de trajetória, mas sim de estado quântico, dependente do tempo, característica manifestada, grosso modo, como uma função de onda - $\Psi(\vec{r}, t)$.

Eisberg e Resnick (1979, p. 174), concluíram que a obtenção da eqS, é feita por meio de um **postulado**, pois não foi deduzida dos resultados experimentais, mas sim, porque os prevê. Ambos os autores inferiram uma lista de quatro hipóteses, cuja razoabilidade, faria transparecer a eqS verossímil aos estados quânticos

I. O primeiro passo, determina que ela deve ser consistente com a relação de **Broglie-Einstein**, com a seguinte legenda e posterior formulação:

(E) Energia de um fóton; (h) Constante de Planck; (v) Frequência de onda; (\hbar) $\frac{h}{2\pi}$; (ω) Frequência angular; (p) Momento; (λ) Comprimento de onda; (κ) número de onda.

Equações 1 e 2

$$E = hv = \hbar\omega \text{ e } p = \frac{h}{\lambda} = \hbar\kappa$$

II. Deve ser compatível com as seguintes equações:

(m) Massa; (U) Energia potencial

Equações 3

$$E = \frac{p^2}{2m} + U$$

III. Na terceira hipótese, considera-se, a eqS, como **linear**, portanto, se uma onda harmônica unidimensional, na sua forma complexa, apresenta a configuração exposta na **Equação 4**:

(Ψ) Função de onda; (x) Espaço; (t) Tempo; (A) Amplitude de onda; (e) Número de Euller; (i) $\sqrt{-1}$.

Equação 4

$$\Psi = \Psi(x, t) = A e^{i(kx - \omega t)}$$

Então, será solução, para o caso de uma partícula livre, a subsequente **Equação 5**:

Equação 5

$$\Psi(x, t) = A e^{i(kx - \omega t)} = A [\cos(kx - \omega t) + i \sin(kx - \omega t)]$$

IV. Na quarta e última hipótese, utiliza-se um caso especial, existente no estudo das partículas livres, segundo o qual, a **energia potencial**, é uma função, com o seguinte formato:

$$Uo = U(x, t)$$

Ademais, sendo a **força resultante**, atuante na partícula livre, tal qual:

$$F = -\frac{\partial U(x, t)}{\partial x}$$

e, Uo constante, a **força resultante**, será igual a zero ($F=0$).

Cenário que terá, as relações:

$$U = \frac{E}{\hbar} \text{ e } \lambda = \frac{\hbar}{\rho}$$

Constantes; permitindo reescrever a **Equação 03**:

Equações 6

$$\frac{\hbar^2 k^2}{2m} + U(x, t) = \hbar \omega$$

Diante de tal problemática, **Sakurai e Napolitano** (2013, p. 68), demonstraram que, a eqS, por se tratar de uma **equação diferencial linear**, de mais de uma variável independente, pode se obter como uma relação entre sua solução $\Psi(\vec{r}, t)$, e certas derivadas das, já ditas, variáveis independentes. São quatro as possibilidades, aqui consideradas:

$$\frac{\partial \Psi(x,t)}{\partial x} \text{ ou } \frac{\partial \Psi(x,t)}{\partial t} \text{ ou } \frac{\partial^2 \Psi(x,t)}{\partial x^2} \text{ ou } \frac{\partial^2 \Psi(x,t)}{\partial t^2}$$

Além do mais, uma equação diferencial linear, apresenta o seguinte arcabouço:

$$\Psi(x,t) = c_1 \Psi_1(x,t) + c_2 \Psi_2(x,t)$$

Possibilitando, a seguinte proporção:

Com c , sendo uma constante qualquer.

Equações 7

$$\frac{\partial^2 [c \Psi(x,t)]}{\partial x^2} = c \frac{\partial^2 \Psi(x,t)}{\partial x^2}$$

Assim pois, de acordo com as hipóteses 3 (linearidade), e 4 (partícula livre), modifica-se a **Equação 7**, deste modo:

Equações 8

$$\alpha \frac{\partial^2 \Psi(x,t)}{\partial x^2} + U(x,t)\psi(x,t) = \beta \frac{\partial \Psi(x,t)}{\partial t}$$

$$\text{Com } \alpha = \frac{-\hbar^2}{2m} \text{ e } \beta = \pm i\hbar$$

Resultando:

Equações 9

$$\frac{-\hbar^2}{2m} \frac{\partial^2 \Psi(x,t)}{\partial x^2} + U(x,t)\psi(x,t) = i\hbar \frac{\partial \Psi(x,t)}{\partial t}$$

A **Equação 9**, é a ilustre **Equação de Schroedinger, dependente do tempo**.

É possível combiná-la, do seguinte modo:

$$\frac{\partial \Psi(x,t)}{\partial t} = -i\omega\Psi(x,t) ; \frac{\partial^2 \Psi(x,t)}{\partial x^2} = i^2\kappa^2\Psi(x,t) = -\kappa^2\Psi(x,t)$$

Redundando:

Equação 10

$$i\hbar \frac{\partial \Psi(x,t)}{\partial t} = \frac{\hbar^2 \kappa^2}{2m} \Psi(x,t) + U \Psi(x,t)$$

3.1.2. A Regressão Simbólica (RS):

Nos dizeres de, **Morgon e Pereira** (2015, p.1):

Regressão é uma técnica para verificar a relação existente entre duas variáveis observadas, e obter um modelo que possa expressar essa relação. A abordagem tradicional de regressão, usa modelos predefinidos. A regressão simbólica, por outro lado, consiste em encontrar uma função que se ajuste ao conjunto de dados observados, sem que qualquer suposição sobre a forma da função precise ser feita. Diferentemente da tradicional, a regressão simbólica encontra não apenas os coeficientes de uma função, mas a própria função.

E mais, muito mais do que isso, a RS é uma técnica de aprendizado profundo, capaz de sugerir uma expressão simbólica, a respeito duma função desconhecida; grosso modo, refere-se a uma rede neural, calibrada a fim de encontrar funções subjacentes a uma dada correlação de variáveis.

O autor vislumbra, desde uma contemplação metafórica, a simbiose entre o paradigma métrico de acúmulo de aprendizado, que o método regressivo simbólico constrói com a experiência, com os anseios do físico trivial, que perambula pela vida, em um dormir acordado, em busca da descoberta duma engrenagem desconhecida, que seja, da monumental maquinaria que é a realidade. Angustia-se, com frequência, pois teme não atingir tal pretensão, antes de deparar-se com o dia da verdade, dia do irreversível e absoluto fracasso final, imposto no inevitável e redundante início da inexistência.

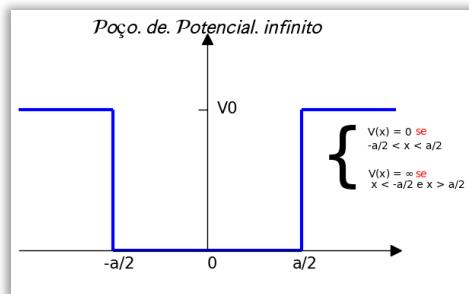
3.1.3. O Poço quadrado de potencial infinito:

É um modelo teórico, cuja descrição se dá, por meio de uma partícula confinada em uma região do espaço, com potencial energético igual a zero; em contrapartida, a região externa ao poço, apresenta potencial infinito.

Embora idealizado, o modelo de potencial infinito é simples, podendo ser utilizado como aproximação a uma série de problemas físicos, sobretudo, os de confinamento.

Tem sua importância caracterizada por permitir que funções de onda, no seu interior, sejam soluções para a equação de Schroedinger, ademais, reduz a complexidade analítica, pois manifesta a energia, de forma quantizada, com valores discretos, além de possuir os estádios quânticos, da partícula, ligados.

Figura 1 - Poço de Potencial Infinito



Fonte: O autor

3.2. A EQS, REDUZIDA E DEPENDENTE DO TEMPO:

Com o propósito de reduzir, ainda mais, a complexidade e dificuldade da matemática, mas não menos adequada, reduz-se a **Equação 10**, por meio da instauração da **Hamiltoniana**, descrito por **Griffiths** (2011, p. 19), como estados de energia total definida. Na mecânica clássica, é assim definida:

Equação 11

$$H(x, p) = \frac{p^2}{2m} + U(x)$$

Por substituição canônica, manipula-se o **Operador Hamiltoniano** (\hat{H}), visto na **Equação 12**, substituindo-o, na **Equação 10**, tal como:

Com $p^2 = \hbar^2 k^2$.

Equação 12

$$\hat{H} = \frac{\hbar^2}{2m} \frac{d^2}{dx^2} + U \hat{\psi}(x)$$

Encontra-se a eqS, reduzida e dependente do tempo, na **Equação 13**, vista abaixo:

Equação 13

$$\hat{H}\Psi = i\hbar \frac{\partial\Psi}{\partial t}$$

3.3. PRIMEIRA SIMULAÇÃO EM PYTHON: Solução da eqS, reduzida e dependente do tempo (ver APÊNDICE B1)

Resolvendo a equação de Schrödinger, reduzida e dependente do tempo

A Equação de Schrödinger como uma equação matricial:

Escrevendo a matriz para \mathbf{H} , obtemos:

$$\mathbf{H} = \frac{-\hbar^2}{2mh^2} \begin{pmatrix} -2 & 1 & 0 & 0 & \dots & \\ 1 & -2 & 1 & 0 & \dots & \\ 0 & 1 & -2 & 1 & \dots & \\ \ddots & \ddots & \ddots & \ddots & \vdots & \\ \dots & \dots & \dots & 1 & -2 & \end{pmatrix} + \begin{pmatrix} V_0 & 0 & 0 & \dots & \vdots & \\ 0 & V_1 & 0 & \dots & \vdots & \\ 0 & 0 & V_2 & \dots & \vdots & \\ \ddots & \ddots & \ddots & \ddots & \vdots & \\ \dots & \dots & \dots & \dots & V_{N-1} & \end{pmatrix}$$

```
In [1]: #Bibliotecas
import numpy as np
import matplotlib.pyplot as plt
from scipy.linalg import eigh

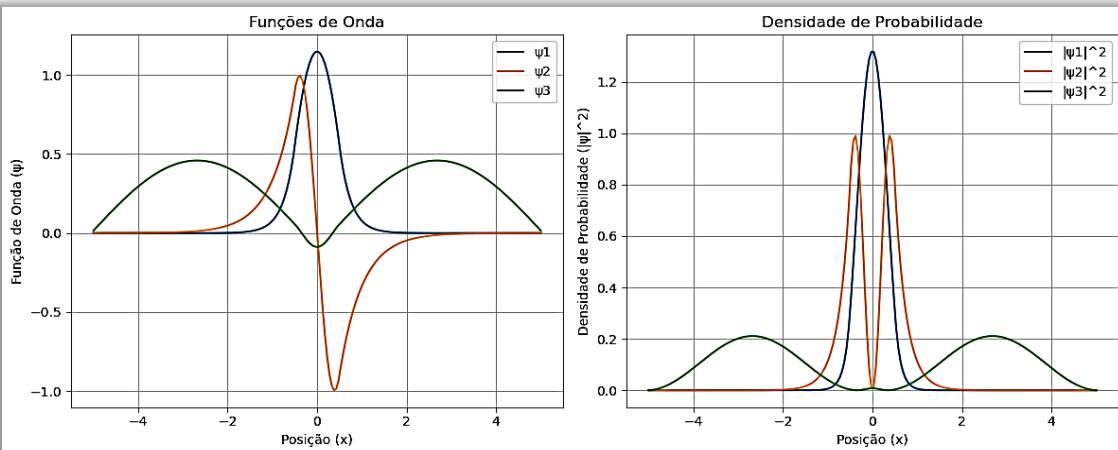
In [2]: #Constantes físicas
hbar = 1.0 #Constante de Planck reduzida
m = 1.0 #Massa da partícula
#Parâmetros do poencial infinito
a=1.0 #Largura do poço
V0 = 10.0 #Profundidade do poço

In [3]: #Função que calcula o potencial
def potencial(x):
    potencial = np.zeros_like(x)
    potencial[(x<-a/2) | (x>a/2)] = V0
    return potencial

#Função para numericamente resolver a eqS
def solve_schrodinger(N,L,V_func):
    """
    N - quantidade de acrescimos em x
    L - largura do potencial
    V_f - função Potencial
    """
    #1. Definir grade e potencial
    x = np.linspace(-L/2, L/2, N)
    dx = x[1] - x[0]
    V = V_func(x)

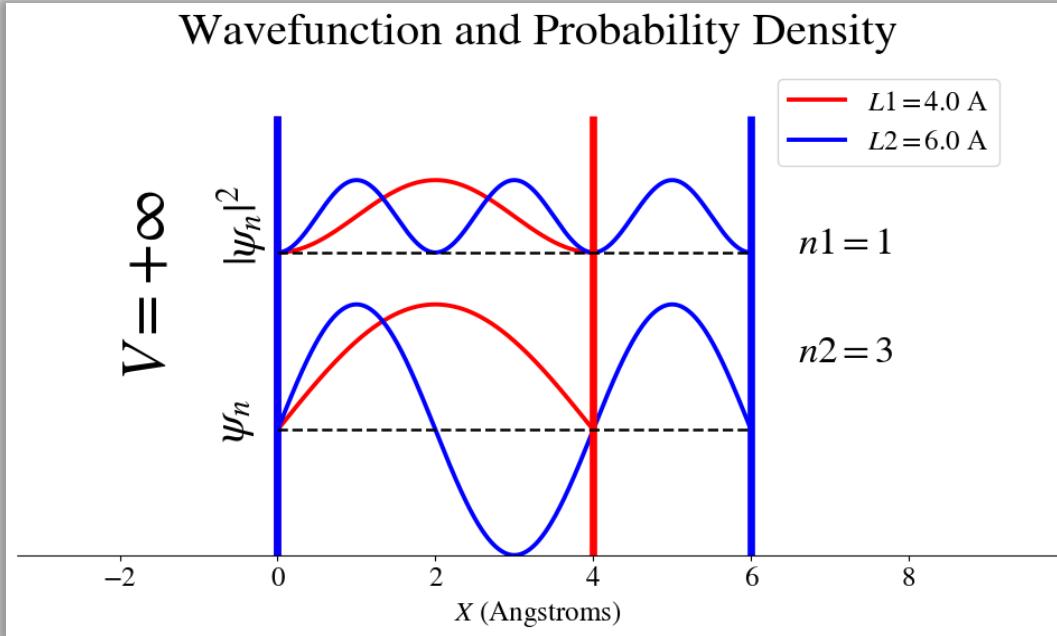
    #2. Construir a matriz hamiltoniana
    #Termo cinético
    main_diag = 1/(m*dx**2) * np.ones(N)
    off_diag = -1/(2*m*dx**2) * np.ones(N-1)
    H = np.diag(main_diag) + np.diag(off_diag, 1) + np.diag(off_diag, -1)
    #Termo potencial
    H += np.diag(V)

    #3. Diagonalizar a matriz hamiltoniana
    energia, psi = eigh(H) # diagonaliza a matriz e calcula autovalores (energia) e autovetores (psi)
    #Normalizar as funções de onda
    for i in range(psi.shape[1]):
        psi[:,i] /= np.sqrt(np.sum(psi[:,i]**2) * dx)
    return x, energia, psi
```



3.4. SEGUNDA SIMULAÇÃO EM PYTHON: Solução da eqS, reduzida e dependente do tempo, com entrada de dado. (ver APÊNDICE B2)

```
In [16]: ## import matplotlib.pyplot as plt
import numpy as np
def psi(x,n,l):
    return np.sqrt(2.0/L)*np.sin(float(n)*np.pi*x/L)
# Reading the input variables from the user
n1 = int(input("Enter the value for the quantum number n for the first box = "))
n2 = int(input("Enter the value for the quantum number n for the second box= "))
# Asking the input boxes sizes from the user, and making sure the values are not larger than 20 Å
L1 = 10.0
L2 = 100.0
while(L>20.0):
    L = float(input(" To compare wavefunctions for boxes of different lengths \n enter the value of L for the first box (in Angstroms and not larger than 20 Å) = "))
L2 = float(input("Enter the value of l for the second box (in Angstroms and not larger than 20) = "))
L = max(L1,L2)
if(L>20.0):
    print ("The sizes of the boxes cannot be larger than 20 Å. Please enter the values again.\n")
# Generating the wavefunction and probability density graphs
plt.rcParams.update({'font.size': 18, 'font.family': 'STIXGeneral', 'mathtext.fontset': 'stix'})
fig, ax = plt.subplots(figsize=(12,6))
ax.spines['right'].set_color('none')
ax.xaxis.tick_bottom()
ax.spines['left'].set_color('none')
ax.axes.get_yaxis().set_visible(False)
ax.spines['top'].set_color('none')
val = 1.1*(L1,L2)
X1 = np.linspace(0.0, L1, 900, endpoint=True)
X2 = np.linspace(0.0, L2, 900, endpoint=True)
ax.axis([-0.5*val,1.5*val,-np.sqrt(2.0/L1),3*np.sqrt(2.0/L1)])
ax.set_xlabel(r'$X$ (Angstroms)')
strA="$psi_{n1}$"
strB="$|\psi_{n1}|^2$"
ax.text(-0.12*val, 0.0, strA, rotation='vertical', fontsize=30, color="black")
ax.text(-0.12*val, np.sqrt(4.0/L1), strB, rotation='vertical', fontsize=30, color="black")
str1="\$L1 = "+str(L1)+"\AA"
str2="\$L2 = "+str(L2)+"\AA"
ax.plot(X1,psi(X1,n1,1)*np.sqrt(l1/l1), color="red", label=str1, linewidth=2.8)
ax.plot(X2,psi(X2,n2,2)*np.sqrt(l2/l2), color="blue", label=str2, linewidth=2.8)
ax.plot(X1,psi(X1,n1,1)*psi(X1,n1,1)*(l1/l1) + np.sqrt(4.0/l1), color="red", linewidth=2.8)
ax.plot(X2,psi(X2,n2,2)*psi(X2,n2,2)*(l2/l2) + np.sqrt(4.0/l2), color="blue", linewidth=2.8)
ax.margins(0.00)
ax.legend(loc=0)
ax.yticks([0,1],["$\psi_n$","$|\psi_n|^2$"])
```



3.5. TERCEIRA SIMULAÇÃO EM PYTHON: Solução da eqS, reduzida e dependente do tempo, com entrada de dados. (ver APÊNDICE B3)

```
In [2]: # import matplotlib.pyplot as plt
import numpy as np

# Defining the wavefunction
def psi(x,n,L): return np.sqrt(2.0/L)*np.sin(float(n)*np.pi*x/L)

# Reading the input variables from the user
n = int(input("Enter the value for the quantum number n = "))
L = float(input("Enter the size of the box in Angstroms L = "))
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(15, 6))

# Generating the wavefunction graph
plt.rcParams.update({'font.size': 18, 'font.family': 'STIXGeneral', 'mathtext.fontset': 'stix'})
x = np.linspace(0, L, 900)

lim1=np.sqrt(2.0/L) # Maximum value of the wavefunction
ax1.axis([0,0,L,-1.1*lim1,1.1*lim1]) # Defining the limits to be plot in the graph
str1="$n = "+str(n)+"$"
str2="$L = "+str(L)+"$"
ax1.spines[['top', 'right','bottom']].set_visible(False)
ax1.spines[['left']].set_position(("data", 0))

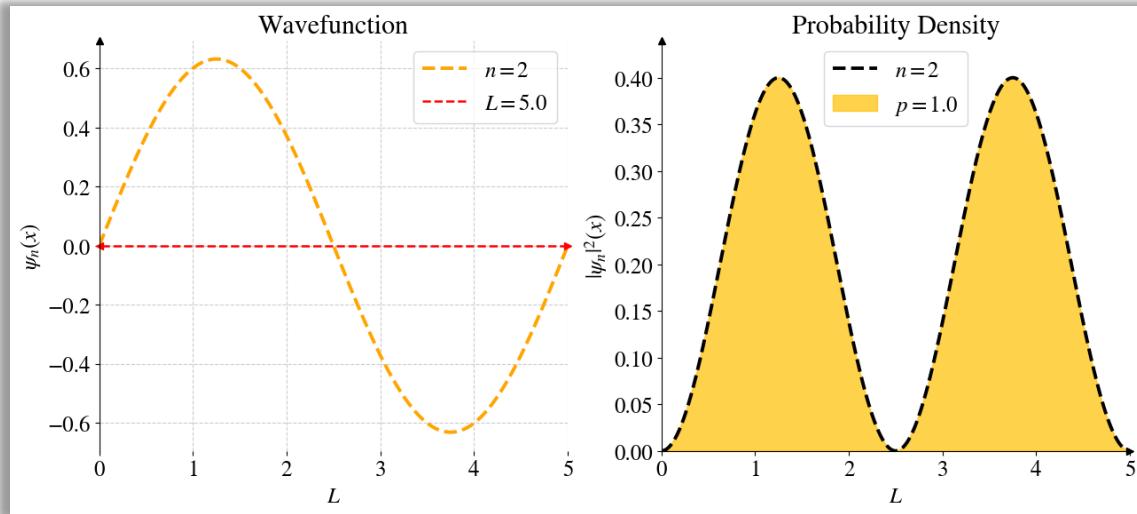
ax1.plot(0, 1, "k", transform=ax1.get_xaxis_transform(), clip_on=False)
ax1.grid(color = '#ccc', linestyle = '--', linewidth = 0.8)

ax1.plot(x, psi(x,n,L), linestyle='--', label=str1, color="orange", linewidth=2.8) # Plotting the wavefunction
ax1.hlines(0, 0.025, L, linewidth=1.8, linestyle='--', label=str2, color="red") # Adding a horizontal line at 0

ax1.plot(1, 0, ">r", transform=ax1.get_yaxis_transform(), clip_on=False)
ax1.plot(0, 0, "<r", transform=ax1.get_yaxis_transform(), clip_on=False)

# Now we define labels, legend, etc
ax1.legend()
ax1.set_xlabel(r'$L$')
ax1.set_ylabel(r'$\psi_n(x)$')
ax1.set_title('Wavefunction')


```



3.6. ENCONTRANDO A EQS, INDEPENDENTE DO TEMPO, ATRAVÉS DA VERSÃO, TEMPORAL DEPENDENTE:

Como descrito por **Lisbôa** (2020, p. 1112), para transformar a eqS, independente do tempo, deve-se utilizar o método da *separação de variáveis*, cuja funcionalidade permite transformar uma equação diferencial parcial, em um conjunto de equações diferenciais ordinárias, reduzindo, portanto, sua dificuldade matemática, além de procurar soluções, cujo resultado, resume-se ao produto de funções de apenas uma variável independente.

De tal modo que a função de onda, permite instaurar a seguinte proporção:

Equação 14

$$\Psi(x, t) = \psi(x)\varphi(t)$$

Substituindo a solução da **equação 14**, na **equação 13**, resulta:

$$\begin{aligned} \frac{-\hbar^2}{2m} \frac{\partial^2 \Psi(x)\varphi(t)}{\partial x^2} + U(x) \Psi(x)\varphi(t) &= i\hbar \frac{\partial \Psi(x)\varphi(t)}{\partial t} = \\ -\frac{\hbar^2}{2m} \varphi(t) \frac{d^2 \psi(x)}{dx^2} + U(x) \psi(x)\varphi(t) &= i\hbar \psi(x) \frac{d\varphi(t)}{dt} = \end{aligned}$$

Equação 15

$$-\frac{\hbar^2}{2m} \frac{1}{\psi(x)} \frac{d^2 \psi(x)}{dx^2} + U(x) = i\hbar \frac{1}{\varphi(t)} \frac{d\varphi(t)}{dt} =$$

Obtém-se, tal qual visto, acima, na **equação 15**, uma simetria do lado esquerdo da proporção, com dependência, somente em x, com o lado direito, dependente de t.

Com efeito, uma vez separadas as variáveis, são, ambos os lados, igualados à constante **c**, denotadas por **Equações 15.1 e 15.2**, respectivamente:

Equações 15.1 e 15.2

$$-\frac{\hbar^2}{2m} \frac{1}{\psi(x)} \frac{d^2 \psi(x)}{dx^2} + U(x) = c \quad i\hbar \frac{1}{\varphi(t)} \frac{d\varphi(t)}{dt} = c$$

Da **equação 15.2**, encontra-se a solução da equação com a variante temporal, tal qual segue, adiante:

$$\frac{d\varphi(t)}{\varphi(t)} = -\frac{ic}{\hbar} dt,$$

Logo,

3.8. A SOLUÇÃO DA EQS, INDEPENDENTE DO TEMPO:

A definição do que é um estado quântico de uma função de onda, é respondida por Griffiths (2011, p. 02), “como uma elucidação provinda da **interpretação estatística** de Max Born, segundo a qual, uma função de onda, cuja sintaxe é: $|\Psi(x, t)|^2$, alude à probabilidade de se encontrar o ponto x , em um instante t ”.

Doutro modo:

Equação 18

$$\int_a^b |\Psi(x, t)|^2 dx = 1$$

Mais do que isso, se substituir na **Equação 18**, a **Equação 14**, obtém-se:

(ψ *) Conjugado

$$\int_a^b |\psi(x)\varphi(t)|^2 dx = 1 =$$

$$\int_a^b |\psi(x)\psi(x)^*|^2 dx = 1 =$$

Podendo ser reformulada, da seguinte maneira:

Equação 19

$$\int_a^b |\psi(x)|^2 dx = 1$$

A **Equação 19**, torna possível descobrir a solução numérica para eqS, independente no tempo, pois, se:

(A) Cte de normalização; (n) número de onda; (L) Comprimento; $Kk_n = \frac{n\pi}{L}$

Equação 20

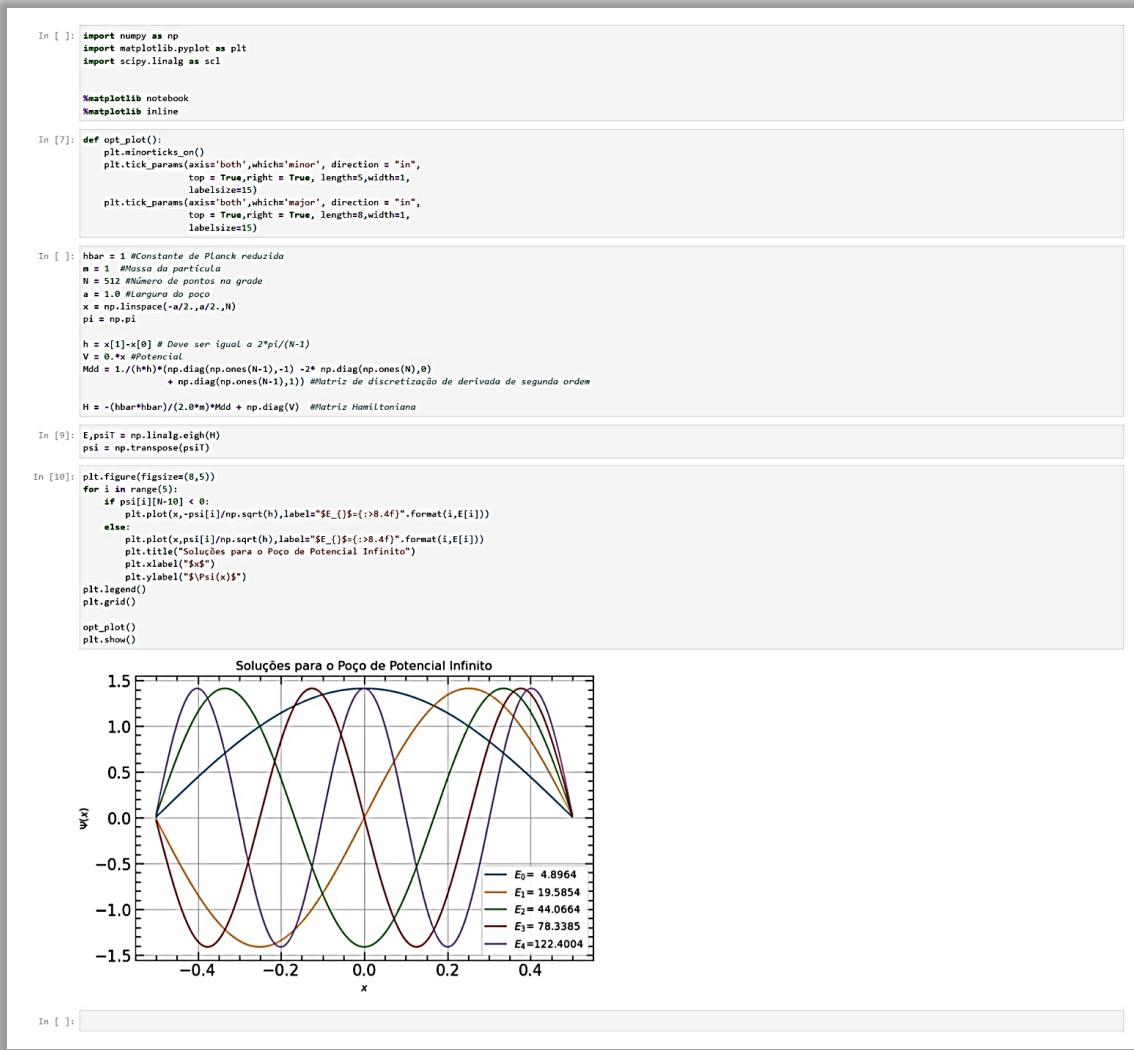
$$\psi(x) = A \sin kL = A \sin \frac{n\pi x}{L}$$

Gerando a **Equação 21**, a seguir, como solução numérica, à eqS, independente do tempo:

Equação 21-

$$\int_a^b A^2 \sin^2 \left(\frac{n\pi x}{L} \right) dx = 1$$

3.9. QUINTA SIMULAÇÃO EM PYTHON: Solução da eqS, independente do tempo, em um cenário de poço quadrado infinito. (ver APÊNDICE B5)



3.10. EXTRAINDO INFORMAÇÕES ADICIONAIS DE CERTA PARTÍCULA:

Uma função de onda, traz consigo, valiosas informações a respeito de sua partícula associada; tal visto que ela, a função de onda, discrimina a densidade de probabilidade da, já dita partícula.

São, especialmente importantes, aqui nesse escopo, os seguintes comportamentos matemáticos, da partícula, em análise de estudo:

3.10.1. Os Valores Esperados:

Recorrendo, mais uma vez, a **Lisbôa** (2020, p. 141), entende-se que os **valores esperados**, nas medições da posição, duma dada partícula, associada a uma função de onda, independente do tempo, como o **valor médio**, de x , considerando-se, a mesma função. Tal valor, pode ser calculado, de acordo com a **Equação 22**:

Equação 22

$$\langle x \rangle = \int_{-\infty}^{+\infty} \psi^*(x) x \psi(x) dx$$

Ademais, computam-se, também, os **valores esperados** do **momento ρ** , da mesma função, por meio do seguinte padrão:

Equação 23

$$\langle \rho \rangle = \int_{-\infty}^{+\infty} \rho(x) |\varphi(\rho, t)|^2 d\rho$$

3.10.2. Operadores:

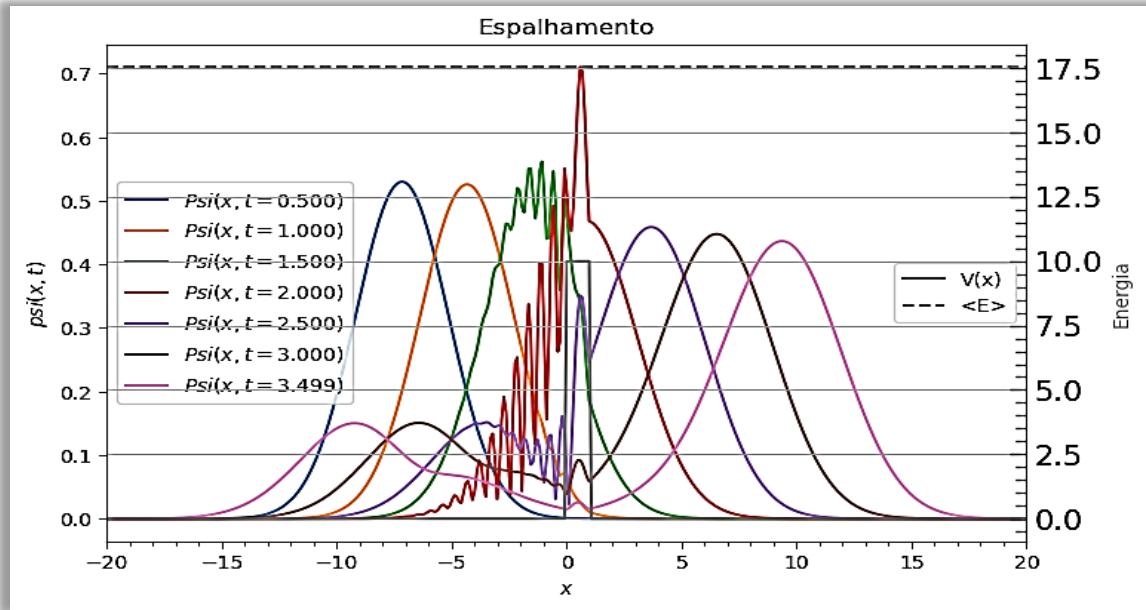
São, grosso modo, objetos matemáticos, capazes de mensurar grandezas físicas, observáveis, de atributos, de dada função de onda, tais quais são: a posição, momento, momento angular ou energia.

A **Tabela 01**, a seguir, relaciona os operadores de posição, e momento, com suas respectivas quantidades físicas:

Tabela 1 - Operadores na representação de posição e momento

Quantidades Físicas	Posição	Momento
Posição \vec{r}	\vec{r}	$i\hbar\nabla_\rho$
Momento $\vec{\rho}$	$(\hbar/i)\nabla$	$\vec{\rho}$
Momento Angular \vec{L}	$\vec{r}(\hbar/i)\nabla$	$\hbar\nabla_\rho\vec{\rho}$
Energia Cinética K	$-(\hbar^2/2m)\nabla^2$	$(\rho^2/2m)$
Energia Potencial V	$V(\vec{r})$	$V(i\hbar\nabla_\rho)$
Energia E	$-(\hbar^2/2m)\nabla^2 + V(\vec{r})$	$(\rho^2/2m) + V(i\hbar\nabla_\rho)$

3.11. SEXTA SIMULAÇÃO EM PYTHON: Análise quantitativa dos valores esperados, de posição e momento, de certa função de onda, independente do tempo. (ver APÊNDICE B6)



3.11. MENSURAÇÕES SIMBÓLICO REGRESSIVAS:

As análises, regressivo simbólicas, recaem sobre a Solução da eqS, de acordo com três paradigmas: Via Clássica, separação de variáveis e integração, por laço de repetição, respectivamente. Ademais, cada rede neural será submetida a duas bibliotecas: **GPyLearn**⁶ e **PYSR**⁷

3.11.1. A Via Clássica. (ver APÊNDICE A1)

```

Complexity Loss Score Equation
1 1.040e-01 0.000e+00 y = -0.011816
3 1.028e-01 5.893e-03 y = x0 * -0.012391
4 8.949e-02 1.382e-01 y = sin(x0) * -0.16923
5 8.797e-02 1.705e-02 y = sin(sin(x0)) * -0.20232
6 8.042e-02 8.972e-02 y = sin(x0 * 1.9962) * -0.22285
7 4.865e-02 5.027e-02 y = sin(x0 + sin(x0)) * -0.43455
9 4.827e-02 3.917e-03 y = (sin(sin(x0) + x0) * -0.47723) + -0.01129
11 4.646e-02 1.909e-02 y = sin(x0 + sin(sin(x0) + sin(x0))) * -0.43453
12 4.437e-02 4.605e-02 y = sin(x0 + sin(sin(x0) + sin(sin(x0)))) * -0.46072
13 1.892e-03 3.155e+00 y = sin(x0 + (sin(x0 + sin(x0)) + sin(x0))) * -0.93668

[38]: PySRRegressor
PySRRegressor.equations_ = [
    pick score equation
    0 0.000000 -0.0118161775
    1 0.005893 x0 * -0.012391238
    2 0.138231 sin(x0) * -0.16922592
    3 0.017049 sin(x0 * 1.9962) * -0.20232285
    4 0.001717 sin(x0 + sin(x0)) * -0.43455496
    5 0.502675 sin(x0 + sin(x0)) * -0.43452933
    6 0.003917 (sin(sin(x0) + x0) * -0.47720566) + -0.011289594
    7 0.019893 sin(x0 + sin(sin(x0) + sin(x0))) * -0.43452933
    8 0.0466049 sin(x0 + sin(sin(x0) + sin(sin(x0)))) * -0.460...
    9 .... 3.155156 -sin(x0 + (sin(x0 + sin(x0)) + sin(x0))) * -0.93668
] outputs/20250913_100247_GOhVey/hall_of_fame.csv

[39]: def round_expr(expr, num_digits=4):
    return expr.xreplace({n: round(n, num_digits) for n in expr.atoms(Number)})

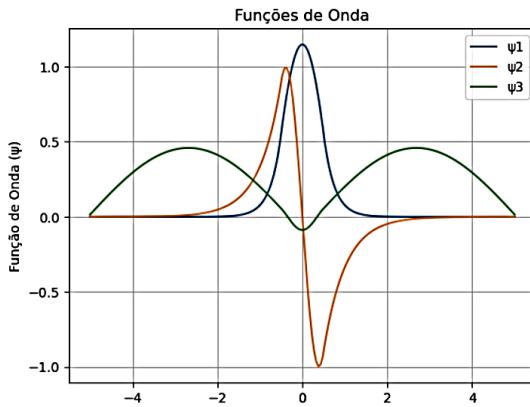
[40]: round_expr(equations.sympy())
[40]: -0.9367 sin(x0 + sin(x0) + sin(x0 + sin(x0)))

```

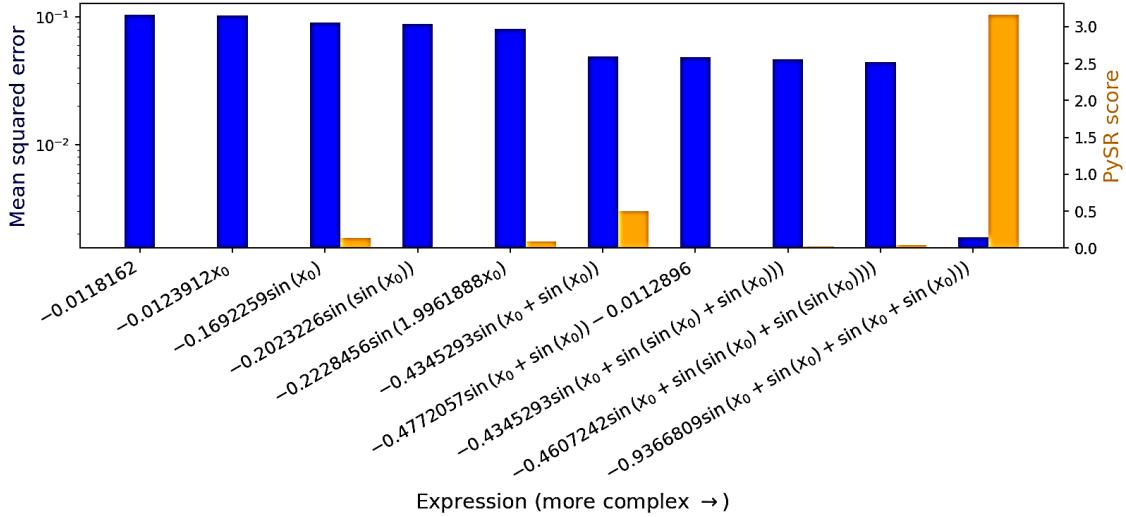
⁶ É uma biblioteca Python criada especificamente para Regressão Simbólica, por meio da Programação Genética.

⁷ É uma biblioteca de código aberto para Regressão Simbólica prática. É um tipo de Aprendizado de Máquina que visa descobrir modelos simbólicos interpretáveis por humanos.

```
[31]: for i in range(min(3, psi.shape[1])): # Plota até o número de funções de onda encontradas
    plt.plot(x, psi[:, i], label=f"\psi{i+1}")
plt.title("Funções de Onda")
plt.xlabel("Posição (x)")
plt.ylabel("Função de Onda (\psi)")
plt.legend()
plt.grid(True)
```



PySR Pareto frontier



3.11.2. Va da independênciā temporal: (ver APÊNDICE A2)

Segunda Análise Regressivo Simbólica da Solução do Poço quadrado de Potencial Infinito

Via da Independênciā Temporal...

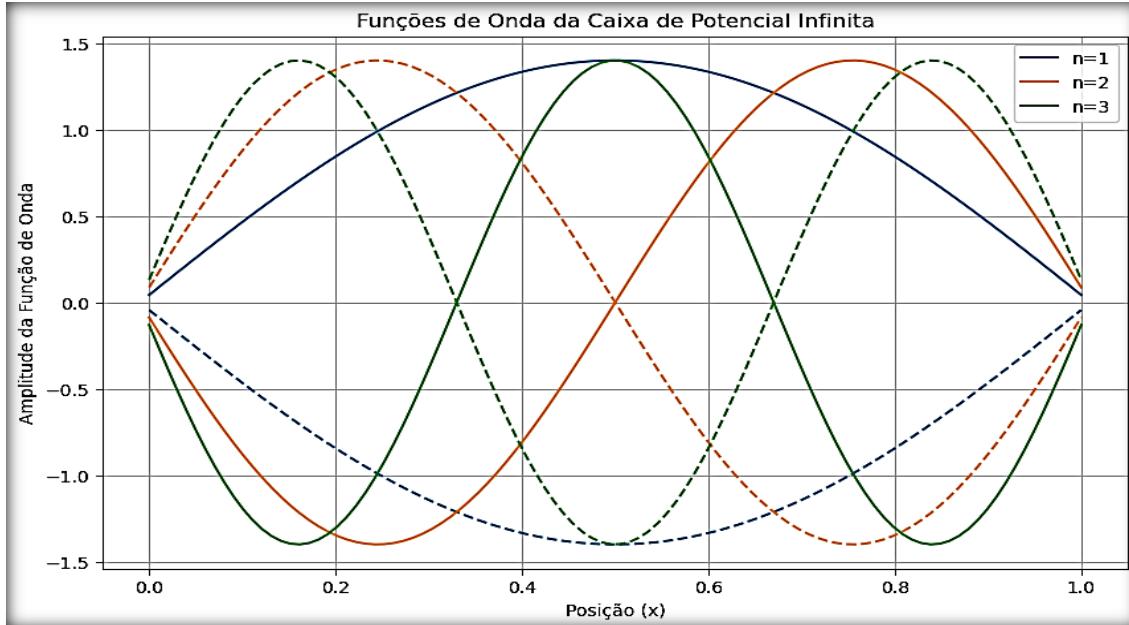
A equação de Schrödinger:

$$\hat{H}\psi(x) = E\psi(x)$$

```
[8]: import numpy as np
import matplotlib.pyplot as plt

L = 1.0 # Comprimento da caixa
N = 100 # Número de pontos de discretização

x = np.linspace(0, L, N)
dx = x[1] - x[0] # Passo espacial
```



```

1      8.337e-01  0.000e+00  y = x₀
2      8.307e-01  3.511e-03  y = sin(x₀)
3      5.887e-01  3.444e-03  y = x₀ + -0.49494
4      8.066e-02  1.988e+00  y = sin(x₀ * -6.1959)
5      7.300e-02  4.992e-02  y = sin((x₀ + -0.49755) * 5.8829)
6      7.292e-02  1.078e-03  y = sin(5.8828 * sin(-0.49753 + x₀))
7      4.760e-02  2.133e-01  y = sin(sin((x₀ + -0.49753) * 5.8828)) * 1.2341
9      3.611e-09  5.465e+00  y = sin(x₀ + ((x₀ + (x₀ * 4.1586)) + -3.0793)) * 1.4002
12     1.068e-09  6.090e-01  y = sin((x₀ + -3.2923) + ((x₀ + (x₀ * 4.1587)) + 0.21301))...
14     * 1.4002

```

- outputs\20250915_133028_79ocBX\hall_of_fame.csv

```

[15]: PySRRegressor
3      1.987582   sin(x₀ * -6.1959267)
4      0.049920   sin((x₀ + -0.4975508) * 5.882884)
5      0.001078   sin(5.882768 * sin(-0.49753174 + x₀))
6      0.213325   sin(sin((x₀ + -0.49753174) * 5.882768)) * 1.2341...
7      5.464764   sin(x₀ + ((x₀ + (x₀ * 4.158655)) + -3.0792732)...
8      >>> 0.608990   sin((x₀ + -3.2923305) + ((x₀ + (x₀ * 4.158689)...)

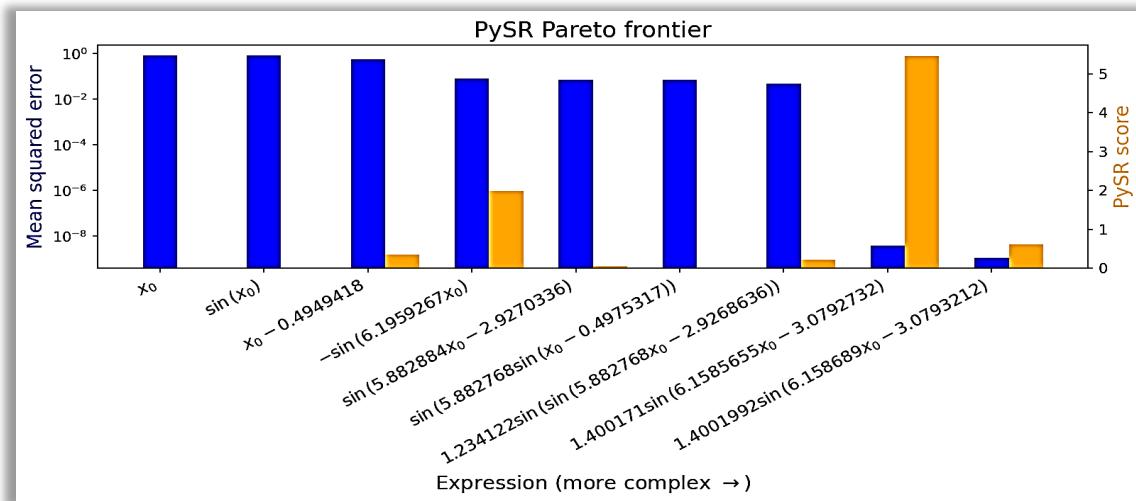
loss  complexity
0  8.336668e-01  1
1  8.307452e-01  2
2  5.886787e-01  3
3  8.066451e-02  4

```

```

[16]: def round_expr(expr, num_digits=4):
    return expr.xreplace({n: round(n, num_digits) for n in expr.atoms(Number)})
[17]: round_expr(equations.sympy())
[17]: 1.4002sin(6.1587x₀ - 3.0793)
[18]: equations.equations_

```



3.11.3. Pela Via Literal: (ver APÊNDICE A3)

Pela Via Literal

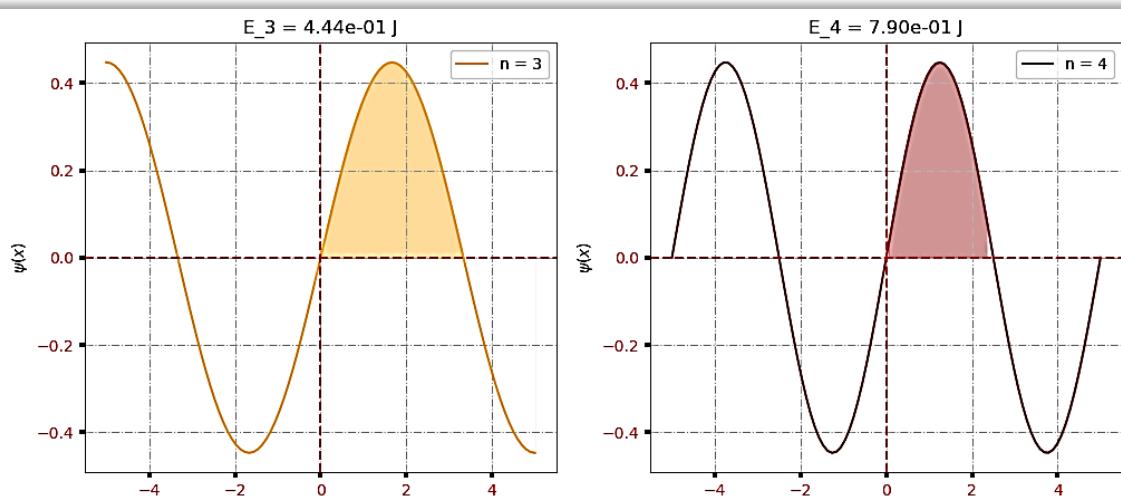
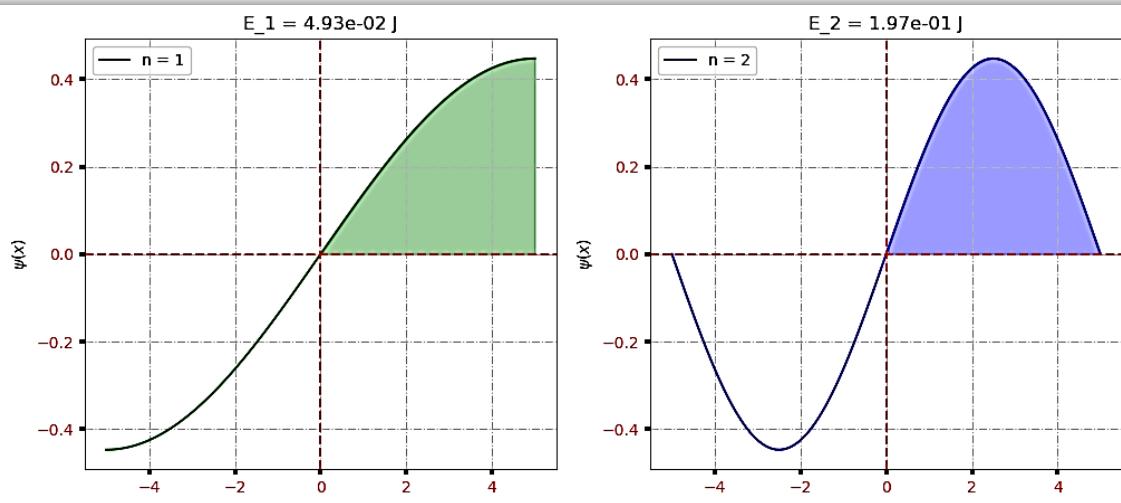
Dada a hipotética função ondulatória:

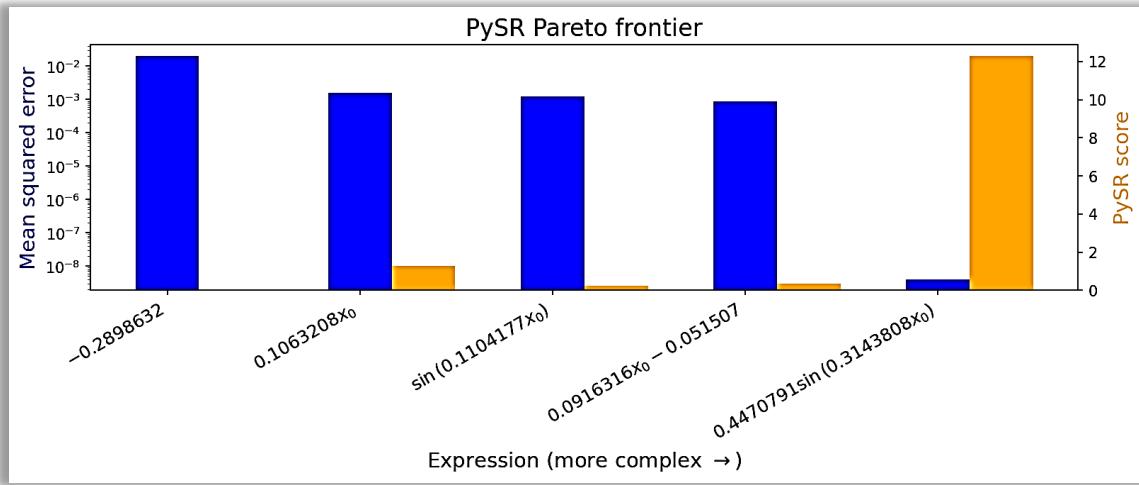
$$\psi_n(x) = \sqrt{\frac{2}{L}} \sin\left(\frac{n\pi x}{L}\right)$$

Solucionar a equação por meio da integração, tendo em vista, aproximar a sintaxe de programação à linguagem de máquina.

[2]:

```
#Bibliotecas
%matplotlib inline
from gplearn.genetic import SymbolicRegressor
from sklearn.ensemble import RandomForestRegressor
from sklearn.tree import DecisionTreeRegressor
from sklearn.utils.random import check_random_state
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
import numpy as np
import graphviz
from scipy.integrate import quad
import sympy as sy
```





4. Considerações finais

O presente trabalho teve como objetivo reinterpretar a Equação de Schrödinger por meio da aplicação da Regressão Simbólica, utilizando a linguagem de programação Python como ferramenta central de modelagem, simulação e análise. O estudo concentrou-se em um caso particular: a função de onda de uma partícula confinada em um poço quadrado de potencial infinito.

A hipótese inicial foi confirmada, uma vez que os algoritmos desenvolvidos em Python, aliados às bibliotecas **Sympy** e **SciPy**, mostraram-se eficazes na redução da complexidade algébrica característica da Mecânica Quântica. Foi possível implementar soluções numéricas e analíticas, além de construir simuladores capazes de representar a evolução temporal da função de onda, mantendo fidelidade ao formalismo teórico, mas de forma mais acessível.

Outro resultado relevante foi a utilização da Regressão Simbólica, que retornou uma ampla variedade de expressões matemáticas coerentes com o problema proposto. Esse acervo de padrões funcionais não apenas confirma a viabilidade da abordagem, mas também abre espaço para a continuidade de investigações futuras, sobretudo na análise, refinamento e validação dessas equações sob o rigor do método científico.

Os achados desta pesquisa sugerem que a integração entre Física Quântica e Ciência de Dados pode oferecer novos caminhos de investigação teórica e computacional, contribuindo para a superação das dificuldades matemáticas que tradicionalmente limitam o estudo da Mecânica Quântica. Além disso, destaca-se o potencial pedagógico dos simuladores desenvolvidos, que

podem servir como instrumentos de ensino e aprendizado tanto em Física quanto em programação científica.

Como perspectivas futuras, recomenda-se:

- Expandir a análise para outros potenciais quânticos além do poço quadrado infinito;
 - Integrar métodos mais avançados de aprendizado de máquina ao processo de regressão simbólica;
 - Desenvolver uma base de dados de funções de onda e padrões matemáticos obtidos, de modo a consolidar um repositório acessível a outros pesquisadores.

Conclui-se, portanto, que a proposta de reinterpretar a dinâmica quântica via algoritmos em Python, apoiada pela Regressão Simbólica, representa uma alternativa promissora tanto para a pesquisa acadêmica quanto para aplicações didáticas e computacionais em Física Moderna.

Referências

- ANACONDA INC. Anaconda Documentation. 2024. Disponível em: . Acesso em: 23 set. 2025.
- BARROS, Vicente Pereira de. Princípios da Relatividade: o que há de especial no movimento? Curitiba: InterSaber, 2011.
- DAWKINS, Richard. O Gene Egoísta. São Paulo: Itatiaia, 1976.
- EINSTEIN, Albert. Relatividade: a teoria especial e a geral. Rio de Janeiro: Contraponto, 1999 [edição original de 1915].
- EISBERG, Robert; RESNICK, Robert. Física Quântica: átomos, moléculas, sólidos, núcleos e partículas. Rio de Janeiro: Campus, 1979.
- GELL-MANN, Murray. O Quark e o Jaguar: aventuras no simples e no complexo. São Paulo: Companhia das Letras, 1996.
- GRIFFITHS, David J. Mecânica Quântica. 2. ed. São Paulo: Pearson, 2011.
- MORGON, Nelson H.; PEREIRA, Maurício A. Introdução à Modelagem Molecular. São Paulo: Editora Livraria da Física, 2015.
- NAPOLITANO, José; SAKURAI, J. J. Mecânica Quântica Moderna. Porto Alegre: Bookman, 2013.
- POINCARÉ, Henri. Science and Hypothesis. Paris: Flammarion, 1902.
- PROJECT JUPYTER. Jupyter Notebook Documentation. 2024. Disponível em: . Acesso em: 23 set. 2025.
- PYTHON SOFTWARE FOUNDATION. Python: Language Reference, version 3.11. 2024. Disponível em: . Acesso em: 23 set. 2025.
- ROVELLI, Carlo. A Ordem do Tempo. São Paulo: Objetiva, 2017.
- SCHRÖDINGER, Erwin. What is Life? The Physical Aspect of the Living Cell. Cambridge: Cambridge University Press, 1946.
- SCIPY COMMUNITY. SciPy Reference Guide. 2024. Disponível em: . Acesso em: 23 set. 2025.
- SYMPY DEVELOPMENT TEAM. SymPy Documentation. 2024. Disponível em: . Acesso em: 23 set. 2025.
- TENACHI, Abdelhakim et al. Symbolic regression for physics-informed discovery of analytical models. *Nature Communications*, v. 14, n. 1, p. 1–12, 2023.

APÊNDICES

APÊNDICE A : CÓDIGOS FONTE DAS REDES NEURAIS BASEADAS EM REGRESSÃO SIMBÓLICA:

A1 - A Via Clássica

Análise Regressivo Simbólica da Solução do Poço quadrado de Potencial Infinito

Pela via Clássica

```
[7]: #Bibliotecas Gerais
import numpy as np
import matplotlib.pyplot as plt
from scipy.linalg import eigh

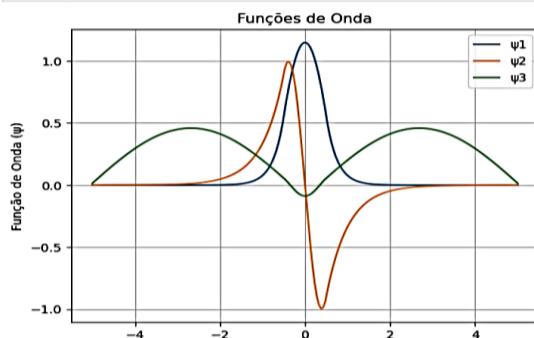
[8]: #Constantes físicas
hbar = 1.0 #Constante de Planck reduzida
m = 1.0 #Massa da partícula
#Parâmetros do poitencial infinito
a=1.0 #Largura do poço
V0 = 10.0 #Profundidade do poço

[14]: #Função que calcula o potencial
def potencial(x):
    potencial = np.zeros_like(x)
    potencial[(x < -a/2) | (x > a/2)] = V0
    return potencial

#Função para, numericamente resolver a eqs
def solve_schrodinger(N,L,V_func):
    """
    N - quantidade de acréscimos em x
    L - largura do potencial
```

```
v_f - função Potencial
"""
#1. Definir grade e potencial
x = np.linspace(-L/2, L/2, N)
dx = x[1] - x[0]
v = v_func(x)
#2. Construir a matriz hamiltoniana
#Termo cinético
main_diag = 1/(m*dx**2) * np.ones(N)
off_diag = -1/(2*m*dx**2) * np.ones(N-1)
H = np.diag(main_diag) + np.diag(off_diag, 1) + np.diag(off_diag, -1)
#Termo potencial
H += np.diag(v)
#3. Diagonalizar a matriz hamiltoniana
energia, psi = eigh(H) # diagonaliza a matriz e calcula autovalores (energia) e autovetores (psi)
#Normalizar as funções de onda
for i in range(psi.shape[1]):
    psi[:,i] /= np.sqrt(np.sum(psi[:,i]**2) * dx)
return x, energia, psi
```

```
[31]: for i in range(min(3, psi.shape[1])): # Plota até o número de funções de onda encontradas
    plt.plot(x, psi[:, i], label=f"\u03c8({i+1})")
plt.title("Funções de Onda")
plt.xlabel("Posição (x)")
plt.ylabel("Função de Onda (\u03c8)")
plt.legend()
plt.grid(True)
```



```

    'pow': lambda x, y : x**y,
    'sin': lambda x : sin(x),
    'cos': lambda x : cos(x),
    'inv': lambda x: 1/x,
    'sqrt': lambda x: x**0.5,
    'pow3': lambda x: x**3
}

function_set = ['add', 'sub', 'mul', 'div','cos','sin','neg','inv']

est_gp = SymbolicRegressor(population_size=5000,
                           generations=20,
                           tournament_size=20,
                           stopping_criteria=0.01, # Stop if R^2 reaches this value
                           function_set=function_set,
                           parsimony_coefficient=0.01, # Penalty for complex equations
                           random_state=0,
                           verbose=1)

```

Método de calibração, via treinamento

```

[35]: est_gp.fit(X_train.reshape(-1,1), y_train)

C:\Users\USER\anaconda3\lib\site-packages\sklearn\base.py:474: FutureWarning: `BaseEstimator._validate_data` is deprecated in 1.6 and will be removed in 1.7. Use `sklearn.utils.validation.validate_data` instead. This function becomes public and is part of the scikit-learn developer API.

warnings.warn(

```

	Population Average	Best Individual				
Gen	Length	Fitness	Length	Fitness	OOB Fitness	Time Left
0	15.94	12.9732	7	0.124853	N/A	1.30m
1	9.10	2.30885	31	0.105479	N/A	1.00m
2	4.27	1.47161	8	0.109029	N/A	51.97s
3	2.75	0.437924	8	0.109029	N/A	45.63s
4	3.63	0.271329	8	0.109029	N/A	41.42s
5	4.01	0.274007	5	0.119818	N/A	38.08s
6	4.01	0.279532	5	0.125077	N/A	32.10s
7	4.01	0.268618	5	0.125077	N/A	32.43s
8	4.01	0.289254	5	0.125077	N/A	26.22s
9	4.03	0.273681	5	0.125077	N/A	25.49s
10	4.01	0.290716	5	0.125077	N/A	23.55s
11	4.03	0.277031	5	0.125077	N/A	21.94s
12	4.01	0.292281	5	0.125077	N/A	20.12s
13	4.01	0.285242	5	0.125077	N/A	17.21s
14	4.01	0.298759	5	0.125077	N/A	13.25s
15	4.05	0.273332	5	0.125077	N/A	11.65s
16	4.03	0.33262	5	0.125077	N/A	8.14s
17	4.04	0.288205	12	0.115558	N/A	5.69s
18	4.03	0.29991	4	0.121506	N/A	2.72s
19	4.02	0.285853	5	0.118375	N/A	0.00s

```

[35]: ▾ SymbolicRegressor
      sin(sin(div(-0.153, X0)))

```

```

[36]: # 4. Evaluate the model
y_pred = est_gp.predict(X_test.reshape(-1,1))
print(f"R^2 score on test data: {r2_score(y_test, y_pred):.4f}")

# 5. Print the discovered equation
print(f"Discovered equation: {est_gp._program}")

next_e_funcao=sympify(str(est_gp._program), locals=converter)
next_e_funcao

```

R² score on test data: 0.7712
Discovered equation: $\sin(\sin(\text{div}(-0.153, X_0)))$

```

[36]: -sin(sin(div(0.153, X0)))

```

Biblioteca PYSR

```

[37]: import pysr

```

```

[38]: equations = pysr.PySRegressor(
    niterations=5,
    binary_operators=["+", "*"], # operators that can combine two terms
    unary_operators=["sin"], # operators that modify a single term
)
equations.fit(X_train.reshape(-1,1), y_train)

C:\Users\USER\anaconda3\lib\site-packages\pysr\sr.py:2811: UserWarning: Note: it looks like you are running in Jupyter. The progress bar will be turned off.
  warnings.warn(
Compiling Julia backend...
[ Info: Started!
[ Info: Final population:
[ Info: Results saved to:

```

```

Complexity Loss Score Equation
1 1.040e-01 0.000e+00 y = -0.011816
3 1.028e-01 5.893e-03 y = x₀ * -0.012391
4 8.949e-01 1.382e-01 y = sin(x₀) * 0.16923
5 8.797e-02 1.705e-02 y = sin(sin(x₀)) * 0.20232
6 8.042e-02 8.972e-02 y = sin(x₀ * 1.9962) * -0.22285
7 4.865e-02 5.027e-01 y = sin(x₀ + sin(x₀)) * -0.43453
9 4.827e-02 3.917e-03 y = (sin(sin(x₀) + x₀) * -0.47721) + -0.01129
11 4.646e-02 1.989e-02 y = sin(x₀ + sin(sin(x₀) + sin(x₀))) * -0.43453
12 4.437e-02 4.695e-02 y = sin(x₀ + sin(sin(x₀) + sin(sin(x₀)))) * -0.46072
13 1.892e-03 3.155e+00 y = sin(x₀ + (sin(x₀ + sin(x₀)) + sin(x₀))) * -0.93668

[38]: PySRRegressor
PySRRegressor.equations_ = [
    pick score equation \
    0 0.000000 -0.0118161775
    1 0.005893 x₀ * -0.012391238
    2 0.138231 sin(x₀) * -0.16922592
    3 0.017049 sin(sin(x₀)) * -0.20232265
    4 0.089717 sin(x₀ * 1.9961888) * -0.2228456
    5 0.502675 sin(x₀ + sin(x₀)) * -0.43452933
    6 0.003917 (sin(sin(x₀) + x₀) * -0.47720566) + -0.011289594
    7 0.019093 sin(x₀ + sin(sin(x₀) + sin(x₀))) * -0.43452933
    8 0.046049 sin(x₀ + sin(sin(x₀) + sin(sin(x₀)))) * -0.46072
    9 >>> 3.155156 sin(x₀ + (sin(x₀ + sin(x₀)) + sin(x₀))) * -0.93668
] - outputs\20250913_100247_GUhVey\hall_of_fame.csv

[39]: def round_expr(expr, num_digits=4):
    return expr.xreplace({n: round(n, num_digits) for n in expr.atoms(Number)})

[40]: round_expr(equations.sympy())
[40]: -0.9367 sin(x₀ + sin(x₀) + sin(x₀ + sin(x₀)))

```

```

[40]: round_expr(equations.sympy())
[40]: -0.9367 sin(x₀ + sin(x₀) + sin(x₀ + sin(x₀)))

[41]: equations.equations_
[41]: complexity loss equation score sympy_format lambda_format
0 1 0.103969 -0.0118161775 0.000000 -0.0118161775000000 PySRFunction(X=>-0.0118161775000000)
1 3 0.102751 x₀ * -0.012391238 0.005893 x₀*(-0.012391238) PySRFunction(X=>x₀*(-0.012391238))
2 4 0.089486 sin(x₀) * -0.16922592 0.138231 sin(x₀)*(-0.16922592) PySRFunction(X=>sin(x₀)*(-0.16922592))
3 5 0.087973 sin(sin(x₀)) * -0.20232265 0.017049 sin(sin(x₀))*(-0.20232265) PySRFunction(X=>sin(sin(x₀))*(-0.20232265))
4 6 0.080424 sin(x₀ * 1.9961888) * -0.2228456 0.089717 sin(x₀*1.9961888)*(-0.2228456) PySRFunction(X=>sin(x₀*1.9961888)*(-0.2228456))
5 7 0.048649 sin(x₀ + sin(x₀)) * -0.43452933 0.502675 sin(x₀ + sin(x₀))*(-0.43452933) PySRFunction(X=>sin(x₀ + sin(x₀))*(-0.43452933))
6 9 0.048270 (sin(sin(x₀) + x₀) * -0.47720566) + -0.011289594 0.003917 sin(x₀ + sin(x₀))*(-0.47720566) - 0.011289594 PySRFunction(X=>sin(x₀ + sin(x₀))*(-0.47720566) - 0.011289594)
7 11 0.046461 sin(x₀ + sin(sin(x₀) + sin(x₀))) * -0.43452933 0.019093 sin(x₀ + sin(sin(x₀) + sin(x₀)))*(-0.43452933) PySRFunction(X=>sin(x₀ + sin(sin(x₀) + sin(x₀)))*(-0.43452933))
8 12 0.044370 sin(x₀ + sin(sin(x₀) + sin(sin(x₀)))) * -0.46072 0.046049 sin(x₀ + sin(sin(x₀) + sin(sin(x₀))))*(-0.46072) PySRFunction(X=>sin(x₀ + sin(sin(x₀) + sin(sin(x₀))))*(-0.46072))
9 13 0.001892 sin(x₀ + (sin(x₀ + sin(x₀)) + sin(x₀))) * -0.93668 3.155156 sin(x₀ + (sin(x₀ + sin(x₀)) + sin(x₀)))*(-0.93668) PySRFunction(X=>sin(x₀ + sin(x₀)) + sin(x₀ + sin(x₀)))*(-0.93668)


```

```

[42]: equations.equations_.loss
[42]: 0 0.103969
1 0.102751
2 0.089486
3 0.087973
4 0.088424
5 0.048649
6 0.048270
7 0.046461
8 0.044370
9 0.001892
Name: loss, dtype: float64

```

```

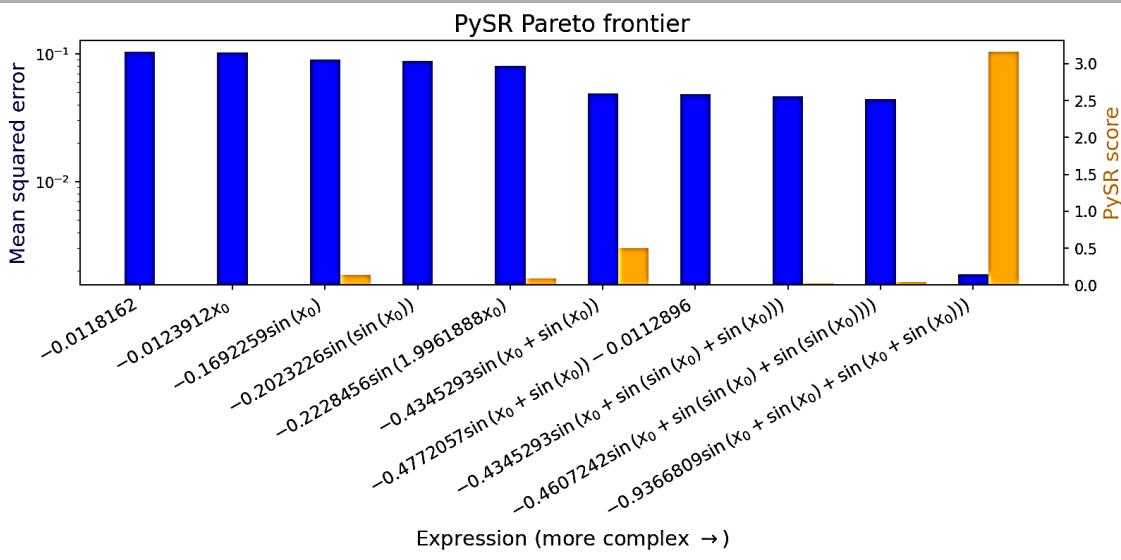
Name: loss5, dtype: float64
[43]: plt.figure(figsize=(12, 3), dpi=150)
plt.bar(
    np.arange(len(equations.equations_)),
    equations.equations_.loss,
    width=0.33,
    color="blue",
)

plt.yscale("log")
plt.ylabel("Mean squared error", fontsize=14, color="blue")
plt.xticks(
    range(len(equations.equations_)),
    [f"${latex(round\_expr(v,7))}$" for v in equations.equations_.sympy_format],
    rotation=30,
    ha="right",
    fontsize=12,
)
plt.title("PySR Pareto frontier", fontsize=16)
plt.xlabel("Expression (more complex $\rightarrow$)", fontsize=14)

ax2 = plt.twinx()
ax2.bar(
    np.arange(len(equations.equations_)) + 0.33,
    equations.equations_.score,
    width=0.33,
    color="orange",
)
ax2.set_ylabel("PySR score", color="orange", fontsize=14)

plt.show()

```



A2. Via da independência temporal:

Segunda Análise Regressivo Simbólica da Solução do Poço quadrado de Potencial Infinito

Via da Independência Temporal...

A equação de Schrödinger:

$$\hat{H}\psi(x) = E\psi(x)$$

```
[8]: import numpy as np
import matplotlib.pyplot as plt

L = 1.0 # Comprimento da caixa
N = 100 # Número de pontos de discretização

x = np.linspace(0, L, N)
dx = x[1] - x[0] # Passo espacial
```

```
# Matriz Hamiltoniana (H = T + V)
# T: Termo de energia cinética (segunda derivada)
# V: Termo de energia potencial

main_diag = 2 * np.ones(N) / dx**2
off_diag = -1 * np.ones(N-1) / dx**2

H = np.diag(main_diag) + np.diag(off_diag, k=1) + np.diag(off_diag, k=-1)

# --- Solução da Equação de Schrödinger Independente do Tempo ---
# H * psi = E * psi
# Encontrar autovalores (E) e autovetores (psi)
autovalores, autovetores = np.linalg.eig(H)

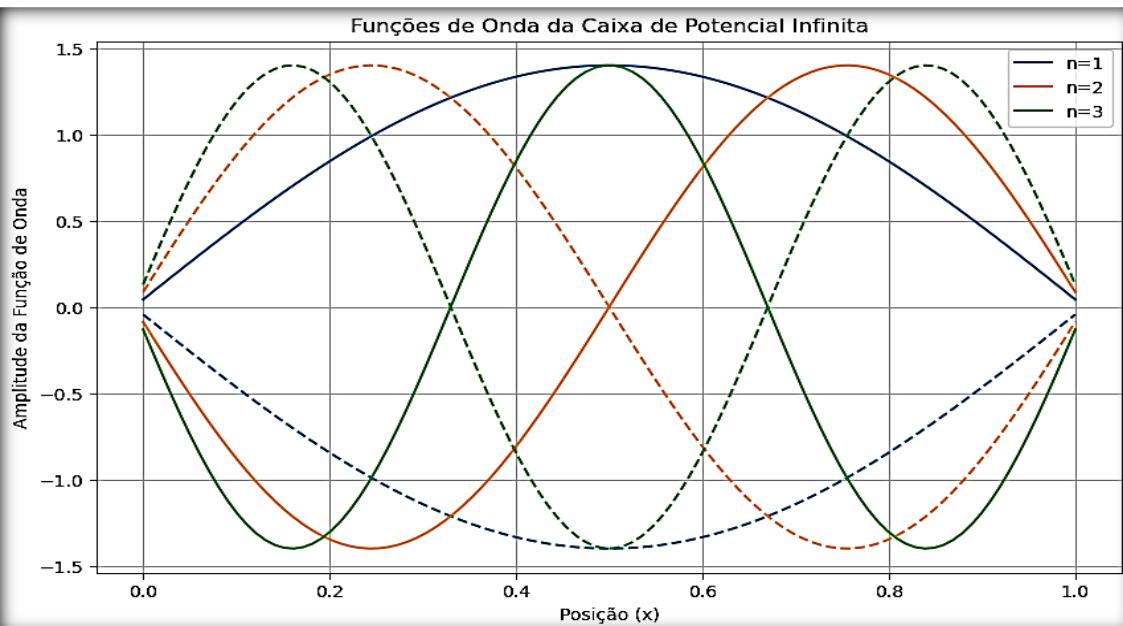
idx_ordenado = np.argsort(autovalores)
autovalores_ordenados = autovalores[idx_ordenado]
autovetores_ordenados = autovetores[:, idx_ordenado]

for i in range(len(autovalores_ordenados)):
    autovetores_ordenados[:, i] = autovetores_ordenados[:, i] / np.sqrt(np.trapezoid(autovetores_ordenados[:, i]**2, x))

print("Primeiras energias (autovalores):")
print(autovalores_ordenados[:5])

plt.figure(figsize=(10, 6))

for i in range(3):
    plt.plot(x, autovetores_ordenados[:, i], label=f'n={i+1}')
    plt.plot(x, -autovetores_ordenados[:, i], '--', color=plt.gca().lines[-1].get_color()) # Adicionar o reflexo para visualizar a forma
y=autovetores_ordenados[:, 1]
```



```

plt.title('Funções de Onda da Caixa de Potencial Infinita')
plt.xlabel('Posição (x)')
plt.ylabel('Amplitude da Função de Onda')
plt.legend()
plt.grid(True)
plt.show()

Primeiras energias (autovalores):
[ 9.48183451 37.91816499 85.28148111 151.52596194 236.5875202 ]

```

Biblioteca GPlearn

```

[9]: import sklearn
from gplearn.genetic import SymbolicRegressor
from sklearn.ensemble import RandomForestRegressor
from sklearn.tree import DecisionTreeRegressor
from sympy import *
from sklearn.model_selection import train_test_split
from sklearn.metrics import r2_score
from sklearn.utils.random import check_random_state
import graphviz

[10]: # 2. Split data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(x, y, test_size=0.2, random_state=42)

```

Método Regressivo Simbólico da Biblioteca GPlearn.

```

[11]: # 3. Initialize and train the SymbolicRegressor
# You can customize functions, population size, generations, etc.

converter = {
    'sub': lambda x, y : x - y,
    'div': lambda x, y : x/y,
    'mul': lambda x, y : x*y,
    'add': lambda x, y : x + y,
    'neg': lambda x : -x,
    'pow': lambda x, y : x**y,
    'sin': lambda x : sin(x),
    'cos': lambda x : cos(x),
    'inv': lambda x: 1/x,
    'sqrt': lambda x: x**0.5,
    'pow3': lambda x: x**3
}

function_set = ['add', 'sub', 'mul', 'div','cos','sin','neg','inv']

est_gp = SymbolicRegressor(population_size=5000,
                            generations=20,
                            tournament_size=20,
                            stopping_criteria=0.01, # Stop if R^2 reaches this value
                            function_set=function_set,
                            parsimony_coefficient=0.01, # Penalty for complex equations
                            random_state=0,
                            verbose=1)

```

Método de calibração, via treinamento

```

[12]: est_gp.fit(X_train.reshape(-1,1), y_train)

C:\Users\USER\anaconda3\lib\site-packages\sklearn\base.py:474: FutureWarning: 'BaseEstimator._validate_data' is deprecated in 1.6 and will be removed in 1.7. Use `sklearn.utils.validation.validate_data` instead. This function becomes public and is part of the scikit-learn developer API.
  warnings.warn(
| Population Average | Best Individual |
-----  

Gen Length Fitness Length Fitness OOB Fitness Time Left
0 15.94 20.5248 14 0.492382 N/A 1.48m
1 7.70 1.60173 10 0.326484 N/A 1.59m
2 7.54 1.38521 14 0.271577 N/A 1.39m
3 8.50 1.73107 18 0.25524 N/A 1.19m
4 9.64 1.1572 13 0.24682 N/A 1.16m
5 11.15 1.06348 13 0.24682 N/A 1.08m
6 11.51 1.06589 22 0.225671 N/A 59.14s
7 11.38 0.973815 24 0.204231 N/A 55.98s
8 11.76 1.6428 16 0.235448 N/A 51.79s
9 11.91 1.08862 16 0.228243 N/A 44.46s
10 11.12 1.19727 11 0.237536 N/A 43.27s
11 11.04 1.11893 11 0.237536 N/A 41.79s
12 11.04 1.12582 11 0.237536 N/A 37.00s
13 11.05 1.12983 14 0.213021 N/A 24.88s
14 10.97 1.17834 11 0.237536 N/A 23.67s
15 11.00 1.13895 11 0.237536 N/A 18.83s
16 10.96 1.14497 15 0.237527 N/A 12.86s
17 11.07 1.15301 11 0.237536 N/A 9.38s
18 10.97 1.17203 11 0.237536 N/A 5.87s
19 11.05 1.16303 11 0.237536 N/A 0.00s

```

[12]:

```

SymbolicRegressor
mul(inv(0.674), cos(mul(inv(-0.318), cos(add(x0, x8)))))


```

```
[13]: # 4. Evaluate the model
y_pred = est_gp.predict(X_test.reshape(-1,1))
print(f"R^2 score on test data: {r2_score(y_test, y_pred):.4f}")

# 5. Print the discovered equation
print(f"Discovered equation: {est_gp._program}")

next_e_funcao=sympify(str(est_gp._program), locals=converter)
next_e_funcao

R2 score on test data: 0.8310
Discovered equation: mul(inv(0.674), cos(mul(inv(-0.318), cos(add(x0, x0)))))

[13]: 1.48367952522255 cos(3.14465408805031 cos(2x0))
```

Biblioteca PYSR

```
[14]: import pysr

[15]: equations = pysr.PySRegressor(
    niterations=5,
    binary_operators=["+", "*"], # operators that can combine two terms
    unary_operators=["sin"], # operators that modify a single term
)
equations.fit(X_train.reshape(-1,1), y_train)

C:\Users\USER\anaconda3\lib\site-packages\pysr\sr.py:2811: UserWarning: Note: it looks like you are running in Jupyter. The progress bar will be turned off.
  warnings.warn(
Compiling Julia backend...
[ Info: Started!
[ Info: Final population:
[ Info: Results saved to:
```

```
1      8.337e-01  0.000e+00  y = x0
2      8.307e-01  3.511e-03  y = sin(x0)
3      5.887e-01  3.444e-03  y = x0 + -0.49494
4      8.066e-02  1.988e+00  y = sin(x0 * -6.1959)
6      7.300e-02  4.992e-02  y = sin((x0 + -0.49755) * 5.8829)
7      7.292e-02  1.078e-03  y = sin(5.8828 * sin(-0.49753 + x0))
9      4.760e-02  2.133e-01  y = sin(sin((x0 + -0.49753) * 5.8828)) * 1.2341
12     3.611e-09  5.465e+00  y = sin(x0 + ((x0 + (x0 * 4.1586)) + -3.0793)) * 1.4002
14     1.068e-09  6.090e-01  y = sin((x0 + -3.2923) + ((x0 + (x0 * 4.1587)) + 0.21301))...
* 1.4002

- outputs\20250915_133028_79ocBX\hall_of_fame.csv
```

PySRegressor

```
[15]:
```

	loss	complexity
0	8.336668e-01	1
1	8.307452e-01	2
2	5.886787e-01	3
3	8.066451e-02	4

```
[16]: def round_expr(expr, num_digits=4):
    return expr.xreplace({n: round(n, num_digits) for n in expr.atoms(Number)})

[17]: round_expr(equations.sympy())

[17]: 1.4002sin(6.1587x0 - 3.0793)

[18]: equations.equations_
```

	complexity	loss	equation	score	sympy_format	lambda_format
0	1	8.336668e-01	x0	0.000000	x0	PySRFunction(X=>x0)
1	2	8.307452e-01	sin(x0)	0.003511	sin(x0)	PySRFunction(X=>sin(x0))
2	3	5.886787e-01	x0 + -0.49494183	0.344443	x0 - 0.49494183	PySRFunction(X=>x0 - 0.49494183)
3	4	8.066451e-02	sin(x0 * -6.1959267)	1.987582	sin(x0*(-6.1959267))	PySRFunction(X=>sin(x0*(-6.1959267)))
4	6	7.300001e-02	sin((x0 + -0.4975508) * 5.882884)	0.049920	sin((x0 - 0.4975508)*5.882884)	PySRFunction(X=>sin(x0 - 0.4975508)*5.882884)
5	7	7.292136e-02	sin(5.882768 * sin(-0.49753174 + x0))	0.001078	sin(5.882768*sin(x0 - 0.49753174))	PySRFunction(X=>sin(5.882768*sin(x0 - 0.497531...))
6	9	4.759515e-02	sin(sin((x0 + -0.49753174) * 5.882768) * 1.23...)	0.213325	sin(sin((x0 - 0.49753174)*5.882768)*1.234122	PySRFunction(X=>sin(sin(x0 - 0.49753174)*5.882768)*1.234122)
7	12	3.610872e-09	sin(x0 + ((x0 + (x0 * 4.158655)) + -3.0792732...)	5.464764	sin(x0 + x0 * 4.158655 - 3.0792732)*1.400171	PySRFunction(X=>sin(x0 + x0 * 4.158655 - 3.0792732)*1.400171)
8	14	1.068194e-09	sin((x0 + -3.2923305) + (x0 + (x0 * 4.158689)...)	0.608990	sin(x0 + x0 + x0*4.158689 - 3.2923305 + 0.2130...)	PySRFunction(X=>sin(x0 + x0 + x0*4.158689 - 3.2923305 + 0.2130...))

```
[19]: equations.equations_.loss
```

```
[19]: 0      8.336668e-01
1      8.307452e-01
2      5.886787e-01
3      8.066451e-02
4      7.300001e-02
5      7.292136e-02
6      4.759515e-02
```

```

5    7.292136e-02
6    4.759515e-02
7    3.610872e-09
8    1.068194e-09
Name: loss, dtype: float64

[20]: plt.figure(figsize=(12, 3), dpi=150)
plt.bar(
    np.arange(len(equations.equations_)),
    equations.equations_.loss,
    width=0.33,
    color="blue",
)

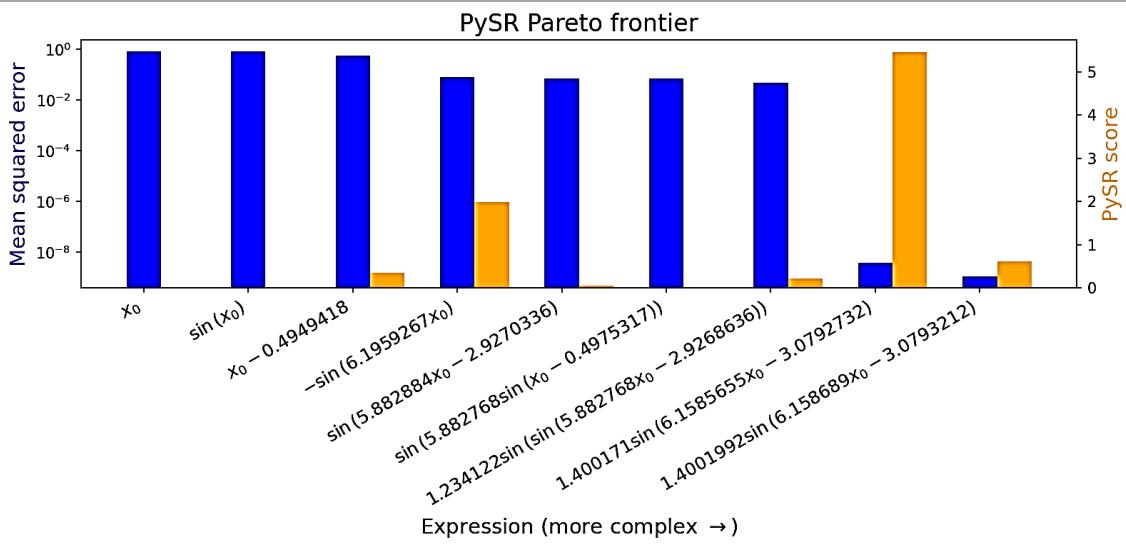
plt.yscale("log")
plt.ylabel("Mean squared error", fontsize=14, color="blue")
plt.xticks(
    range(len(equations.equations_)),
    [f"${ latex( round_expr(v,7) ) }$" for v in equations.equations_.sympy_format],
    rotation=30,
    ha="right",
    fontsize=12,
)
plt.title("PySR Pareto frontier", fontsize=16)
plt.xlabel("Expression (more complex $\rightarrow$)", fontsize=14)

ax2 = plt.twinx()
ax2.bar(
    np.arange(len(equations.equations_)) + 0.33,
    equations.equations_.score,
    width=0.33,
    color="orange",
)
ax2.set_ylabel("PySR score", color="orange", fontsize=14)

plt.show()

```

PySR Pareto frontier



A3. Pela Via Literal:

Pela Via Literal

Dada a hipotética função ondulatória:

$$\psi_n(x) = \sqrt{\frac{2}{L}} \sin\left(\frac{n\pi x}{L}\right)$$

Solucionar a equação por meio da integração, tendo em vista, aproximar a sintaxe de programação à linguagem de máquina.

[2]:

```
#Bibliotecas
%matplotlib inline
from gplearn.genetic import SymbolicRegressor
from sklearn.ensemble import RandomForestRegressor
from sklearn.tree import DecisionTreeRegressor
from sklearn.utils.random import check_random_state
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
import numpy as np
import graphviz
from scipy.integrate import quad
import sympy as sy
```

```
]: #Cálculo, análise e gráficos dos 4 primeiros níveis de energia
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 5))
a=10
m=1
hbar=1
x = np.linspace(-a/2, a/2, 100)
x2 = np.linspace(0, a/2, 100)

def calcular_energia(n, hbar, m, a):
    """Calcula a energia de uma partícula numa caixa unidimensional."""
    return (n**2 * np.pi**2 * hbar**2) / (2 * m * a**2)

y1=np.sqrt(2/a)*(np.sin((1*np.pi*x)/a))
ax1.plot(x,y1, color='green', alpha=1.0, label='n = 1')
ax1.axhline(y=0, color="red", linestyle="--")
ax1.axvline(x=0, color="red", linestyle="--")

ax1.fill_between(x,y1, where = (x > 0), color='green', alpha=0.4)
E1= calcular_energia(1, hbar, m, a)
ax1.set_title(f"E_{1} = {E1:.2e}")
ax1.set_ylabel("\psi(x)")
ax1.legend()

ax1.grid(True, linestyle='-.')
ax1.tick_params(labelcolor='r', labelsize='medium', width=3)

ax2.grid(True, linestyle='-.')
ax2.tick_params(labelcolor='r', labelsize='medium', width=3)

plt.show()
```

```
y2=np.sqrt(2/a)*(np.sin((2*np.pi*x)/a))
ax2.plot(x,y2, color='blue', alpha=1.0, label = 'n = 2')
ax2.axhline(y=0, color="red", linestyle="--")
ax2.axvline(x=0, color="red", linestyle="--")
ax2.fill_between(x,y2, where = (x > 0), color='blue', alpha=0.4)
E2= calcular_energia(2, hbar, m, a)

ax2.set_title(f"E_{2} = {E2:.2e}")
ax2.set_ylabel("\psi(x)")
ax2.legend()

ax1.grid(True, linestyle='-.')
ax1.tick_params(labelcolor='r', labelsize='medium', width=3)

ax2.grid(True, linestyle='-.')
ax2.tick_params(labelcolor='r', labelsize='medium', width=3)

plt.show()

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 5))

y3=np.sqrt(2/a)*(np.sin((3*np.pi*x)/a))
ax1.plot(x,y3, color='orange', alpha=1.0, label = 'n = 3')
ax1.axhline(y=0, color="red", linestyle="--")
ax1.axvline(x=0, color="red", linestyle="--")
ax1.fill_between(x,y3, where = (x > 0), color='orange', alpha=0.4)
```

```

ax1.tri_between(x,y3, where = (x > 3.3), color='white', alpha=1.0)
E3= calcular_energia(3, hbar, m, a)
ax1.set_title(f"E_{3} = {E3:.2e} J")
ax1.set_ylabel("\psi(x)")

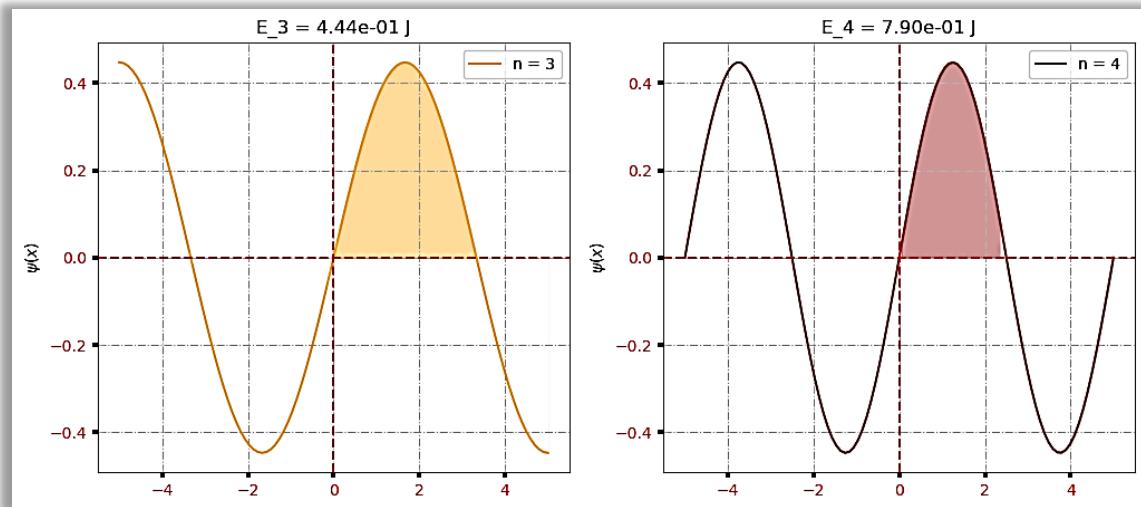
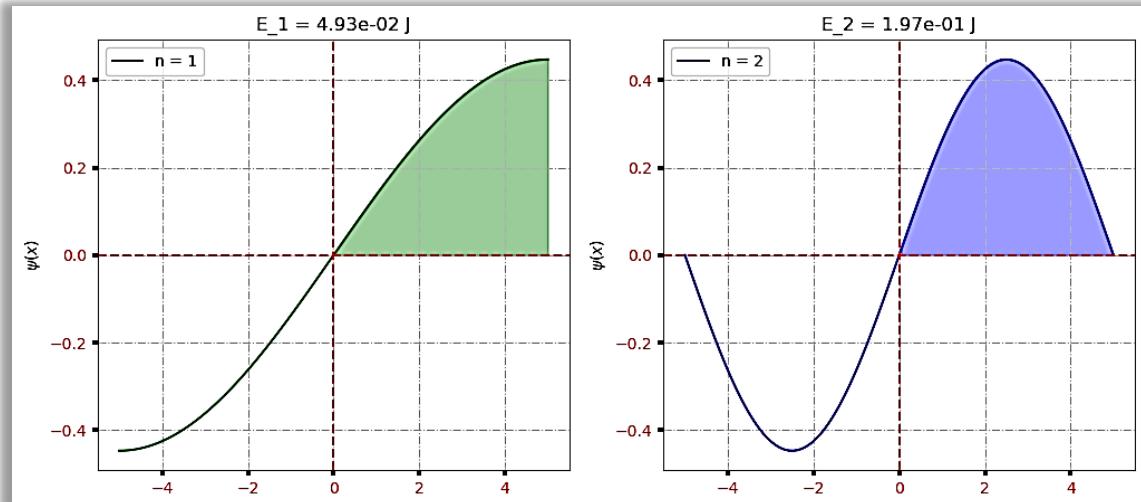
y4=np.sqrt(2/a)*(np.sin((4*np.pi*x)/a))
ax2.plot(x,y4, color='brown', alpha=1.0, label = 'n = 4')
ax2.axhline(y=0, color="red", linestyle="--")
ax2.axhline(x=0, color="red", linestyle="--")
ax2.fill_between(x,y4, where = (x > 0), color='brown', alpha=0.5)
ax2.fill_between(x,y4, where = (x > 2.3), color='white', alpha=1.0)
E4= calcular_energia(4, hbar, m, a)

ax2.set_title(f"E_{4} = {E4:.2e} J")
ax2.set_ylabel("\psi(x)")

ax1.grid(True, linestyle='-.')
ax1.tick_params(labelcolor='r', labelsize='medium', width=3)

ax2.grid(True, linestyle='-.')
ax2.tick_params(labelcolor='r', labelsize='medium', width=3)
ax1.legend()
ax2.legend()

```



Soluções, via Cálculo Integral para:

$$\int_0^L \psi_n^2(x) dx$$

$$\psi_n^2(x) = \frac{2}{L} \sin^2\left(\frac{n\pi x}{L}\right)$$

Dois cenários:

*[4]: #MÉTODO QUAD

```
import numpy as np
from scipy.integrate import quad
import math
L=10
# 1. função a ser integrada
def f(x):
```

```
L=10
# 1. função a ser integrada
def f(x):
    return (2/L) * math.sin(i*math.pi*x/L)**2

# 2. os Limites de integração
a = 0 # Limite inferior
b = 10 # Limite superior

# 3. Calculo da integral usando quad
for i in range(1,5):
    valor_integral, erro = quad(f, a, b)

# 4.resultado
print("A função integrada de {a} a {b} de n = {str(i)} é: {valor_integral} ")
print("O erro estimado foi de: {erro}")


A função integrada de 0 a 10 de n = 1 é: 1.0
O erro estimado foi de: 1.1102230246251565e-14
A função integrada de 0 a 10 de n = 2 é: 1.0
O erro estimado foi de: 7.3398408314000506e-10
A função integrada de 0 a 10 de n = 3 é: 1.0
O erro estimado foi de: 1.1102230246251565e-14
A função integrada de 0 a 10 de n = 4 é: 1.0
O erro estimado foi de: 7.33984109602126e-10
```

[14]: #VIA LITERAL

```
import math
dx=0.1
x=np.linspace(-5,5,100)
sum=0
passo=0
area=0
L=10
for j in range(1,5):
    for i in range(0,100):
        passo=x[i]
        y=(2/L) * 1*math.sin(j*math.pi*passo/L)**2
        area=y*dx
        sum+=area
    print("PDF para n=",j, sum-(j-1))
```

```
PDF para n= 1 1.01
PDF para n= 2 1.0000000000000009
PDF para n= 3 1.0099999999999999
PDF para n= 4 0.9999999999999987
```

Biblioteca GPlearn

```
[5]: import numpy as np
import sklearn
from gplearn.genetic import SymbolicRegressor
from sklearn.ensemble import RandomForestRegressor
from sklearn.tree import DecisionTreeRegressor
from sympy import *
from sklearn.model_selection import train_test_split
from sklearn.metrics import r2_score
from sklearn.utils.random import check_random_state
import graphviz

# 1. Generate synthetic data (e.g.,  $y = 2*X_0 + 3*X_1^2$ )
a=10
x=np.linspace(-a/2,a/2/100)
y=np.sqrt(2/a)*np.sin((1*np.pi*x)/a)

# 2. Split data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(x, y, test_size=0.2, random_state=42)

# 3. Initialize and train the SymbolicRegressor
# You can customize functions, population size, generations, etc.

function_set = ['add', 'sub', 'mul', 'div','cos','sin','neg','inv']
```

```
est_gp = SymbolicRegressor(population_size=5000,
                           generations=20,
                           tournament_size=20,
                           stopping_criteria=0.01, # Stop if R^2 reaches this value
                           function_set=function_set,
                           parsimony_coefficient=0.01, # Penalty for complex equations
                           random_state=0,
                           verbose=1)
```

Método Regressivo Simbólico da Biblioteca GPlearn.

```
[6]: converter = {
    'sub': lambda x, y : x - y,
    'div': lambda x, y : x/y,
    'mul': lambda x, y : x*y,
    'add': lambda x, y : x + y,
    'neg': lambda x : -x,
    'pow': lambda x, y : x**y,
    'sin': lambda x : sin(x),
    'cos': lambda x : cos(x),
    'inv': lambda x: 1/x,
    'sqrt': lambda x: x**0.5,
    'pow3': lambda x: x**3
}
```

Método de calibração, via treinamento

```
[7]: est_gp.fit(X_train.reshape(-1,1), y_train)
| Population Average | Best Individual |
-----+-----+-----+
Gen Length Fitness Length Fitness OOB Fitness Time Left
C:\Users\USER\anaconda3\lib\site-packages\sklearn\base.py:474: FutureWarning: 'BaseEstimator._validate_data' is deprecated in 1.6 and will be removed in 1.7. Use `sklearn.utils.validation.validate_data` instead. This function becomes public and is part of the scikit-learn development API.
warnings.warn(
0 15.94 16.3829 5 0.0319813 N/A 1.48s
1 8.22 2.47508 8 0.0264177 N/A 1.23s
2 5.15 0.945534 8 0.0264177 N/A 1.15s
3 3.77 0.569384 6 0.0259456 N/A 1.33s
4 3.39 0.444887 6 0.0259456 N/A 1.02s
5 3.01 0.418202 3 0.0344377 N/A 54.57s
6 3.02 0.430796 3 0.0344377 N/A 38.30s
7 3.02 0.392775 3 0.0344377 N/A 34.13s
8 3.02 0.448504 3 0.0344377 N/A 28.60s
9 3.03 0.465451 3 0.0344377 N/A 50.24s
10 3.02 0.427395 3 0.0344377 N/A 27.15s
11 3.05 0.429188 3 0.0344377 N/A 21.51s
12 3.02 0.465312 3 0.0344377 N/A 18.17s
13 3.04 0.402725 3 0.0344377 N/A 14.82s
14 3.03 0.419869 3 0.0344377 N/A 13.69s
15 3.03 0.412503 3 0.0344377 N/A 10.90s
16 3.04 0.53365 3 0.0344377 N/A 8.07s
17 3.04 0.398229 3 0.0344377 N/A 4.91s
18 3.02 0.424658 3 0.0344377 N/A 3.89s
19 3.01 0.410373 3 0.0344377 N/A 0.00s
```

[7]:

- ▼ SymbolicRegressor
- mul(X0, 0.106)

```
[8]: # 4. Evaluate the model
y_pred = est_gp.predict(X_test.reshape(-1,1))
print(f'R^2 score on test data: {r2_score(y_test, y_pred):.4f}')

# 5. Print the discovered equation
print(f'Discovered equation: {est_gp._program}')

next_e_funcao=sympify(str(est_gp._program), locals=converter)
next_e_funcao

R2 score on test data: 0.9117
Discovered equation: mul(x0, 0.106)

[8]: 0.106X_0
```

Biblioteca PYSR

```
[3]: import pysr
Detected IPython. Loading juliacall extension. See https://juliapy.github.io/PythonCall.jl/stable/compat/#IPython

[7]: equations = pysr.PySRegressor(
    niterations=5,
    binary_operators=["+", "*"], # operators that can combine two terms
    unary_operators=["sin"], # operators that modify a single term
)
equations.fit(X_train.reshape(-1,1), y_train)

C:\Users\USER\anaconda3\Lib\site-packages\pysr\srv.py:2811: UserWarning: Note: it looks like you are running in Jupyter. The progress bar will be turned off.
  warnings.warn(
Compiling Julia backend...
[ Info: Started!
```

Complexity	Loss	Score	Equation
1	2.062e-02	0.000e+00	$y = -0.28986$
3	1.547e-03	1.295e+00	$y = x_0 * 0.10632$
4	1.228e-03	2.313e-01	$y = \sin(x_0 * 0.11042)$
5	8.628e-04	3.528e-01	$y = (x_0 * 0.09163) + -0.051507$
6	4.029e-09	1.227e+01	$y = \sin(x_0 * 0.31438) * 0.44708$

```
- outputs\20250920_120324_WddiTIV\hall_of_fame.csv
[ Info: Final population:
[ Info: Results saved to:
```

```
[7]: PySRegressor
PySRegressor.equations_ = [
    pick      score          equation      loss
    0   0.000000   -0.28986317  2.062124e-02
    1   1.294869   x0 * 0.10632077  1.547411e-03
    2   0.231305   sin(x0 * 0.11041768)  1.227866e-03
    3   0.352795   (x0 * 0.09163162) + -0.051507015  8.628473e-04
    4 >>> 12.274411   sin(x0 * 0.31438085) * 0.44707912  4.029256e-09

    complexity
    0      1
    1      3
    2      4
```

```
[8]: def round_expr(expr, num_digits=4):
    return expr.xreplace({n: round(n, num_digits) for n in expr.atoms(Number)})

[9]: round_expr(equations.sympy())

[9]: 0.4471sin(0.3144x_0)
```

```
[10]: equations.equations_
[10]:
```

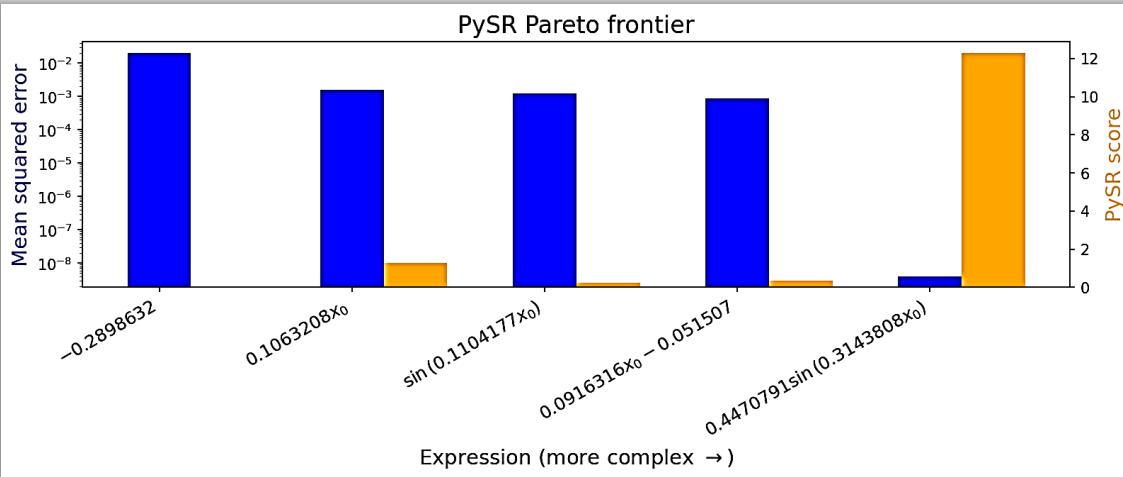
complexity	loss	equation	score	sympy_format	lambda_format
0	2.062124e-02	-0.28986317	0.000000	-0.289863170000000	PySRFunction(X=>-0.289863170000000)
1	1.547411e-03	x0 * 0.10632077	1.294869	x0*0.10632077	PySRFunction(X=>x0*0.10632077)
2	1.227866e-03	sin(x0 * 0.11041768)	0.231305	sin(x0*0.11041768)	PySRFunction(X=>sin(x0*0.11041768))
3	8.628473e-04	(x0 * 0.09163162) + -0.051507015	0.352795	x0*0.09163162 - 0.051507015	PySRFunction(X=>x0*0.09163162 - 0.051507015)
4	4.029256e-09	sin(x0 * 0.31438085) * 0.44707912	12.274411	sin(x0*0.31438085)*0.44707912	PySRFunction(X=>sin(x0*0.31438085)*0.44707912)

```
[11]: equations.equations_.loss
[11]: 0   2.062124e-02
1   1.547411e-03
2   1.227866e-03
3   8.628473e-04
4   4.029256e-09
Name: loss, dtype: float64
```

```
[12]: import matplotlib.pyplot as plt
plt.figure(figsize=(12, 3), dpi=150)
plt.bar(
    np.arange(len(equations.equations_)),
    equations.equations_.loss,
    width=0.33,
    color="blue",
)
plt.yscale("log")
plt.ylabel("Mean squared error", fontsize=14, color="blue")
plt.xticks(
    range(len(equations.equations_)),
    [f"${ latex(round\_expr(v,7)) }$" for v in equations.equations_.sympy_format],
    rotation=30,
    ha="right",
    fontsize=12,
)
plt.title("PySR Pareto frontier", fontsize=16)
plt.xlabel("Expression (more complex $\rightarrow$)", fontsize=14)

ax2 = plt.twinx()
ax2.bar(
    np.arange(len(equations.equations_)) + 0.33,
    equations.equations_.score,
    width=0.33,
    color="orange",
)
ax2.set_ylabel("PySR score", color="orange", fontsize=14)

plt.show()
```



APÊNDICE B : CÓDIGOS FONTE DAS SIMULAÇÕES COMPUTACIONAIS:

B1. SIMULAÇÃO 01 - Solução da eqS, reduzida e dependente do tempo

```
# ## Resolvendo a equação de Schrödinger, reduzida e dependente do tempo
# A Equação de Schrödinger como uma equação matricial:
# Escrevendo a matriz para  $\mathbf{H}$ , obtemos:

#Bibliotecas
import numpy as np
import matplotlib.pyplot as plt
from scipy.linalg import eigh

#Constantes físicas
hbar = 1.,0 #Constante de Planck reduzida
m = 1.0      #Massa da partícula
#Parâmetros do potencial infinito
a=1.0        #Largura do poço
V0 = 10.0     #Profundidade do poço

#Função que calcula o potencial
def potencial(x):
    potencial = np.zeros_like(x)
    potencial[(x < -a/2) | (x > a/2)] = V0
    return potencial

#Função para, numericamente resolver a eqS
def solve_schrodinger(N,L,V_func):
    """
    N - quantidade de acrescimos em x
    L - largura do potencial
    V_f - função Potencial
    """
    #1. Definir grade e potencial
    x = np.linspace(-L/2, L/2, N)
    dx = x[1] - x[0]
    V = V_func(x)
    #2. Construir a matriz hamiltoniana
    #Termo cinético
    main_diag = 1/(m*dx**2) * np.ones(N)
    off_diag = -1/(2*m*dx**2) * np.ones(N-1)
    H = np.diag(main_diag) + np.diag(off_diag, 1) + np.diag(off_diag, -1)
    #Termo potencial
    H += np.diag(V)
    #3. Diagonalizar a matriz hamiltoniana
    energia, psi = eigh(H) # diagonaliza a matriz e calcula autovalores (energia) e
    autovetores (psi)
    #Normalizar as funções de onda
    for i in range(psi.shape[1]):
```

```

    psi[:,i] /= np.sqrt(np.sum(psi[:,i]**2) * dx)
    return x, energia, psi

# Parâmetros da simulação
N = 200 # Número de pontos na grade
L = 10.0 # Comprimento da caixa

# Resolver a equação de Schrödinger
x, energia, psi = solve_schrodinger(N, L, potencial)

# Plotar os resultados
plt.figure(figsize=(12, 5))

# Plotar as primeiras 3 funções de onda
for i in range(min(3, psi.shape[1])): # Plota até o número de funções de onda encontradas
    plt.subplot(1, 2, 1)
    plt.plot(x, psi[:, i], label=f" $\psi_{i+1}$ ")
    plt.title("Funções de Onda")
    plt.xlabel("Posição (x)")
    plt.ylabel("Função de Onda ( $\psi$ )")
    plt.legend()
    plt.grid(True)

    plt.subplot(1, 2, 2)
    plt.plot(x, psi[:, i]**2, label=f" $|\psi_{i+1}|^2$ ")
    plt.title("Densidade de Probabilidade")
    plt.xlabel("Posição (x)")
    plt.ylabel("Densidade de Probabilidade ( $|\psi|^2$ )")
    plt.legend()
    plt.grid(True)

plt.tight_layout()
plt.show()

print("Energias calculadas:")
for i, E in enumerate(energia[:min(3, psi.shape[1])]): # Imprime as primeiras 3 energias
    print(f"E{i+1} = {E:.4f}")

```

B2. SIMULAÇÃO 02 - Solução da eqS, reduzida e dependente do tempo, com entrada de dados

```
### import matplotlib.pyplot as plt
import numpy as np
import matplotlib.pyplot as plt

def psi(x,n,L):
    return np.sqrt(2.0/L)*np.sin(float(n)*np.pi*x/L)

# Reading the input variables from the user
n = int(input("Enter the value for the quantum number n for the first box = "))
n2 = int(input("Enter the value for the quantum number n for the second box= "))

# Reading the input boxes sizes from the user, and making sure the values are not
larger than 20 A
L = 100.0
while(L>20.0):
    L1 = float(input(" To compare wavefunctions for boxes of different lengths \nenter
the value of L for the first box (in Angstroms and not larger then 20 A) = "))
    L2 = float(input("Enter the value of L for the second box (in Angstroms and not
larger then 20) = "))
    L = max(L1,L2)
    if(L>20.0):
        print ("The sizes of the boxes cannot be larger than 20 A. Please enter the
values again.\n")

# Generating the wavefunction and probability density graphs
plt.rcParams.update({'font.size': 18, 'font.family': 'STIXGeneral',
'mathtext.fontset': 'stix'})
fig, ax = plt.subplots(figsize=(12,6))
ax.spines['right'].set_color('none')
ax.xaxis.tick_bottom()
ax.spines['left'].set_color('none')
ax.axes.get_yaxis().set_visible(False)
ax.spines['top'].set_color('none')
val = 1.1*max(L1,L2)
X1 = np.linspace(0.0, L1, 900,endpoint=True)
X2 = np.linspace(0.0, L2, 900,endpoint=True)
ax.axis([-0.5*val,1.5*val,-np.sqrt(2.0/L),3*np.sqrt(2.0/L)])
ax.set_xlabel(r'$x$ (Angstroms)')
strA="$\psi_n$"
strB="$|\psi_n|^2$"
ax.text(-0.12*val, 0.0, strA, rotation='vertical', fontsize=30, color="black")
ax.text(-0.12*val, np.sqrt(4.0/L), strB, rotation='vertical', fontsize=30,
color="black")
str1=r"$L_1 = "+str(L1)+r"$ A"
str2=r"$L_2 = "+str(L2)+r"$ A"
```

```

ax.plot(X1,psi(X1,n,L1)*np.sqrt(L1/L), color="red", label=str1, linewidth=2.8)
ax.plot(X2,psi(X2,n2,L2)*np.sqrt(L2/L), color="blue", label=str2, linewidth=2.8)
ax.plot(X1,psi(X1,n,L1)*psi(X1,n,L1)*(L1/L) + np.sqrt(4.0/L), color="red",
linewidth=2.8)
ax.plot(X2,psi(X2,n2,L2)*psi(X2,n2,L2)*(L2/L) + np.sqrt(4.0/L), color="blue",
linewidth=2.8)
ax.margins(0.00)
ax.legend(loc=9)
str2="$V = +\infty$"
ax.text(-0.3*val, 0.5*np.sqrt(2.0/L), str2, rotation='vertical', fontsize=40,
color="black")
ax.vlines(0.0, -np.sqrt(2.0/L), 2.5*np.sqrt(2.0/L), linewidth=4.8, color="red")
ax.vlines(L1, -np.sqrt(2.0/L), 2.5*np.sqrt(2.0/L), linewidth=4.8, color="red")
ax.vlines(0.0, -np.sqrt(2.0/L), 2.5*np.sqrt(2.0/L), linewidth=4.8, color="blue")
ax.vlines(L2, -np.sqrt(2.0/L), 2.5*np.sqrt(2.0/L), linewidth=4.8, color="blue")
ax.hlines(0.0, 0.0, L, linewidth=1.8, linestyle='--', color="black")
ax.hlines(np.sqrt(4.0/L), 0.0, L, linewidth=1.8, linestyle='--', color="black")
plt.title('Wavefunction and Probability Density', fontsize=30)
str3=r"$n1 = "+str(n)+r"$"
str4=r"$n2 = "+str(n2)+r"$"

ax.text(1.1*L,np.sqrt(4.0/L), r"$n1 = "+str(n)+r"$", fontsize=25, color="black")
ax.text(1.1*L,np.sqrt(4.0/L)-0.5, r"$n2 = "+str(n2)+r"$", fontsize=25, color="black")

plt.legend(bbox_to_anchor=(0.73, 0.95), loc=2, borderaxespad=0.)

# Show the plots on the screen once the code reaches this point
plt.show()

```

B3. SIMULAÇÃO 03 Solução da eqS, reduzida e dependente do tempo, com entrada de dados[] =

```

import matplotlib.pyplot as plt
import numpy as np

# Defining the wavefunction
def psi(x,n,L): return np.sqrt(2.0/L)*np.sin(float(n)*np.pi*x/L)

# Reading the input variables from the user
n = int(input("Enter the value for the quantum number n = "))
L = float(input("Enter the size of the box in Angstroms L = "))
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(15, 6))

# Generating the wavefunction graph
plt.rcParams.update({'font.size': 18, 'font.family': 'STIXGeneral',
'mathtext.fontset': 'stix'})
x = np.linspace(0, L, 900)

lim1=np.sqrt(2.0/L) # Maximum value of the wavefunction

```

```

ax1.axis([0.0,L,-1.1*lim1,1.1*lim1]) # Defining the limits to be plot in the graph
str1=r"$n = "+str(n)+r"$"
str2=r"$L = "+str(L)+r"$"
ax1.spines[['top', 'right','bottom']].set_visible(False)
ax1.spines[["left"]].set_position(("data", 0))

ax1.plot(0, 1, "k", transform=ax1.get_xaxis_transform(), clip_on=False)

ax1.grid(color = '#ccc', linestyle = '--', linewidth = 0.8)

ax1.plot(x, psi(x,n,L), linestyle='--', label=str1, color="orange", linewidth=2.8) # Plotting the wavefunction
ax1.hlines(0, 0.025, L, linewidth=1.8, linestyle='--', label=str2, color="red") # Adding a horizontal line at 0

ax1.plot(1, 0, "r", transform=ax1.get_yaxis_transform(), clip_on=False)
ax1.plot(0, 0, "r", transform=ax1.get_yaxis_transform(), clip_on=False)

# Now we define labels, legend, etc
ax1.legend()
ax1.set_xlabel(r'$L$')
ax1.set_ylabel(r'$|\psi_n(x)|^2$')
ax1.set_title('Wavefunction')

# Generating the probability density graph
ax2.spines[['top', 'right']].set_visible(False)

ax2.axis([0.0,L,0.0,lim1*lim1*1.1])

ax2.plot(1, 0, "k", transform=ax2.get_yaxis_transform(), clip_on=False)
ax2.plot(0, 1, "k", transform=ax2.get_xaxis_transform(), clip_on=False)

str1=r"$n = "+str(n)+r"$"
str2=r"$L = "+str(L)+r"$"
p=1.
str3=r"$p = "+str(p)+r"$"

ax2.plot(x, psi(x,n,L)*psi(x,n,L), label=str1, linewidth=2.8, color='#000', linestyle = '--')
ax2.fill(x, psi(x,n,L)*psi(x,n,L), label=str3, color='fffc000e6', alpha=.7)

# hides borders
ax2.legend();
ax2.set_xlabel(r'$L$')
ax2.set_ylabel(r'$|\psi_n|^2(x)$')

```

```

ax2.legend()
plt.title('Probability Density')

# Show the plots on the screen once the code reaches this point
plt.show()

```

B4. SIMULAÇÃO 04 - Obtendo o valor numérico dos coeficientes c_n .

Computando numericamente os valores dos coeficientes c_n .

Assume-se que a solução para a equação de Schrödinger é conhecida ou já calculada.

As soluções padrão para o poço quadrado infinito, no domínio $0 < x < a$, são dadas por:

$$\langle x | \psi_n \rangle = \psi_n(x) = \sqrt{\frac{2}{a}} \sin\left(\frac{n\pi x}{a}\right)$$

$$E_n = \frac{n^2 \pi^2 \hbar^2}{2ma^2}$$

Para este problema, uma função de onda inicial é dada por $\Psi(x, t = 0)$.

Agora encontrar-se-á os coeficientes c_n , através da seguinte sintaxe:

$$c_n = \langle \psi_n | \Psi(t = 0) \rangle$$

$$= \int_0^a \psi_n(x) \Psi(x, t = 0) dx$$

É preciso fazer essa integral numericamente, para que funcione para qualquer função $\Psi(x, 0)$.

```

%matplotlib inline
import numpy as np
import scipy.integrate as spi
import matplotlib.pyplot as plt

def f(x,a):
    norm = np.sqrt(12./(a*a*a))
    return(np.piecewise(x,[x<a/2.,x>=a/2.],
                        [lambda x: norm*x ,
                         lambda x: norm*(a-x)]))

def psi_n(x,n,a):
    return(np.sqrt(2./a)*np.sin(n*x*np.pi/a))

Pode-se, portanto, calcular o coeficiente  $c_n$ .
Primeiramente, escreve-se uma função simples  $(x, n, a)$  que multiplica  $\Psi(x, t=0)$  por
 $\psi_n(x)$ , de acordo com um  $n$  específico.
Em seguida, encontra-se o coeficiente  $c_n$  integrando tal função, de 0 a  $a$ .
]
;/
```

```

def int_fun(x,n,a):
    return(f(x,a)*psi_n(x,n,a))

def c(n,a):
    if n==0 or n%2==0:
        return(0)
    return( spi.quad(int_fun,0,a,args=(n,a),limit=100)[0] )

Nmax=40
# Nmax, refere-se ao valor limite para a tupla c_n.
#Quanto maior for seu valor, mais lento e preciso será seu processamento
a_1 = 10.
#a_1 contabiliza a largura do poço
a_step = 10./100.
#a_step diz respeito à unidade do paço em x.
#Quanto menor for seu comprimento, maior será a precisão da análise em questão
nl = np.array(range(Nmax))
cx = np.array([c(n,a_1) for n in nl])
print("Posições matriciais do coeficiente cx: ", len(cx))
print('Valores numéricos das posições
iniciais:', 'cx[0]=' ,cx[0], 'cx[1]=' ,cx[1], 'cxz[2]=' ,cx[2], 'cx[3]=' ,cx[3], 'cx[4]=' ,cx[4]
)
print('Valor numérico da última posição:', 'cx[Nmax-1]=' ,cx[Nmax-1])
print('Cálculo da área abaixo da curva gráfica, ou densidade de probabilidade:
','cx|0<x<a|=' ,np.sum(cx*cx))

```

#OUTPUT:

Posições matriciais do coeficiente cx: 40

Valores numéricos das posições iniciais: cx[0]= 0.0 cx[1]= 0.9927408002342286 cxz[2]= 0.0 cx[3]= -0.11030453335935887 cx[4]= 0.0

Valor numérico da última posição: cx[Nmax-1]= -0.0006526895465042513

Cálculo da área abaixo da curva gráfica, ou densidade de probabilidade: cx|0<x<a|= 0.99999743670556410=

B5. SIMULAÇÃO 05 - Solução da eqS, independente do tempo, em um cenário de poço quadrado infinito.

```
import numpy as np
import matplotlib.pyplot as plt
import scipy.linalg as scl

%matplotlib notebook
%matplotlib inline

def opt_plot():
    plt.minorticks_on()
    plt.tick_params(axis='both',which='minor', direction = "in",
                    top = True,right = True, length=5,width=1,
                    labelsize=15)
    plt.tick_params(axis='both',which='major', direction = "in",
                    top = True,right = True, length=8,width=1,
                    labelsize=15)

    hbar = 1 #Constante de Planck reduzida
    m = 1 #Massa da partícula
    N = 512 #Número de pontos na grade
    a = 1.0 #Largura do poço
    x = np.linspace(-a/2.,a/2.,N)
    pi = np.pi

    h = x[1]-x[0] # Deve ser igual a 2*pi/(N-1)
    V = 0.*x #Potencial
    Mdd = 1. / (h*h)*(np.diag(np.ones(N-1),-1) - 2* np.diag(np.ones(N),0)
                  + np.diag(np.ones(N-1),1)) #Matriz de discretização de derivada de
    segunda ordem

    H = -(hbar*hbar)/(2.0*m)*Mdd + np.diag(V) #Matriz Hamiltoniana
    E,psiT = np.linalg.eigh(H)
    psi = np.transpose(psiT)

    plt.figure(figsize=(8,5))
    for i in range(5):
        if psi[i][N-10] < 0:
            plt.plot(x,-psi[i]/np.sqrt(h),label="$E_{%d}=%f$".format(i,E[i]))
        else:
            plt.plot(x,psi[i]/np.sqrt(h),label="$E_{%d}=%f$".format(i,E[i]))
    plt.title("Soluções para o Poço de Potencial Infinito")
    plt.xlabel("$x$")
```

```

        plt.ylabel("\Psi(x)")
plt.legend()
plt.grid()

opt_plot()
plt.show()

```

B6. SIMULAÇÃO 06 - Análise quantitativa dos valores esperados, de posição e momento, de certa função de onda, independente do tempo

- ▼ a equação da função de onda completa:

$$i\hbar \frac{\partial \Psi}{\partial t} = \text{op } H \Psi$$

O opH é um operador de evolução temporal. Pode, ele, também assumir, a seguinte identidade matemática:

$$\text{op } U(\Delta t) = e^{\frac{-i \text{op } H \Delta t}{\hbar}}$$

Resultando, portanto, na solução, descrita abaixo:

$$\Psi(x, t + \Delta t) = e^{\frac{-i \text{op } H \Delta t}{\hbar}} \cdot \Psi(x, t)$$

Podendo tratar o expoente de um operador, por meio de uma série infinita, tem-se:

$$e^{\frac{-i \text{op } H \Delta t}{\hbar}} = \sum_n \frac{\left(\frac{-i \text{op } H \Delta t}{\hbar} \right)^n}{n!}$$

```
#Bibliotecas
```

```

import matplotlib.animation as animation
from IPython.display import HTML
import matplotlib.pyplot as plt
import scipy.fftpack as fft
import scipy.linalg as scl
import numpy as np
import math
%matplotlib inline

```

O Hamiltoniano no espaço:

$$x_n = x_0 + n\Delta x, \text{ com } \Delta x = \frac{(x_N - x_0)}{N}$$

Os valores de entrada:

```

# Os valores de entrada:
hbar = 1
m = 1
N = 2**11
L = 200.0
step_low = 0.
step_high= 1.
V0 = 10.

# Definição do espaço:
n = np.arange(N)
x0 = -L/2.
xN = L/2.
Delta_x = (xN - x0)/N
print("Delta_x = ",Delta_x)

x = x0 + n*Delta_x

# Definição do potencial:
V = np.zeros(N)
for i in range(N):
    if x[i]>= step_low and x[i]<= step_high:
        V[i]= V0

# Configuração do Hamiltoniano para a função `V`, multiplicação com a matriz inversa:
Mdd = 1./(Delta_x**2)*(np.diag(np.ones(N-1),-1)
                     - 2* np.diag(np.ones(N),0)
                     + np.diag(np.ones(N-1),1))
H = -(hbar*hbar)/(2.0*m)*Mdd + np.diag(V)

En,psiT = np.linalg.eigh(H)    # autovalores e os autovetores.
psi = np.transpose(psiT)        # Tomamos a transposta de psiT para os vetores de função
de onda                           # que podem ser acessados como psi[n]

```

```

# Intervalo temporal
dt_max = 2/np.max(En)      # Critério de estabilidade.
dt = 0.001
if dt > dt_max:
    print("ATENÇÃO: dt está na região instável!")

# Função de onda inicial
g_x0=-10.
g_k0=6.
g_sig=2.

```

Definição de um Gaussiano no espaço K, com $p = \hbar k$, um momento k_0 , e o espaço x, $\psi(x, 0) = \left(\frac{2L}{\pi}\right)^{1/4} \cdot e^{-Lx^2}$:

```

def psi0(x,g_x0,g_k0,g_sig):
    _Norm_x=np.sqrt(Delta_x/g_sig)/(np.pi**0.25)

    return(_Norm_x*np.exp(-(x-g_x0)**2/(2.*g_sig*g_sig)+1j*g_k0*x))

psi_t0 = psi0(x,g_x0,g_k0,g_sig)

# H é Hermitiano?
print("Verifique se H é realmente Hermitiano : ",np.array_equal(H.conj().T,H))

Ut_mat = np.diag(np.ones(N,dtype="complex128"),0)

print("Criação de uma matriz U(dt = {})".format(dt))
for n in range(1,3):
    # Realiza a soma. Como se trata de matrizes, o processo irá demorar se N for grande.
    Ut_mat += np.linalg.matrix_power((-1j*dt*H/hbar),n)/math.factorial(n)

p = Ut_mat.dot(psi_t0)

print("O quanto a normalização muda por etapa? Desde {} até {}
{}".format(np.linalg.norm(psi_t0),np.linalg.norm(p)))
print("Nº de etapas em que a norma está errada por um fator 2 :
",1/(np.linalg.norm(p)-1))

# teste do movimento gaussiano:
psi_t0 = psi0(x,g_x0,g_k0,g_sig)
psi_t1 = psi_t0
psi_tu = []

```

```

for t in range(3500):
    psi_t1 = Ut_mat.dot(psi_t1)
    if t>0 and t%500==0:
        psi_tu.append( (t,psi_t1))
psi_tu.append( (t,psi_t1))

```

Teste e verificação de coerência dos resultados:

$\langle E \rangle$ = estado esperado da energia;

$\langle x \rangle$ = estado esperado da posição

```

print("Normalização : ",np.linalg.norm(psi_tu[-1][1]))

vev_E0=float(np.real(np.sum(np.conjugate(psi_t0)*H.dot(psi_t0))))
vev_x0=float(np.real(np.sum(np.conjugate(psi_t0)*x*psi_t0)))

print("<E_(t = 0)> = {:.8.4f} <x_(t = 0)> = {:.8.4f}".format(vev_E0,vev_x0))

for t,p in psi_tu:
    norm = np.linalg.norm(p)
    vev_E1 = float(np.real(np.sum(np.conjugate(p)*H.dot(p))))
    vev_x1 = float(np.real(np.sum(np.conjugate(p)*x*p)))
    print("dt = {:.7.1f} norm = {:.8.5f} <E> = {:.8.4f} <x_(dt)> = {:.8.4g}".format(t,norm,vev_E1,vev_x1))

```

#OUTPUT:

```

-----Normalização : 1.0000447363612588-----
-----<E_(t = 0)> = 17.5429 <x_(t = 0)> = -10.0000-----
dt = 500.0 norm = 1.00001 <E> = 17.5432 <x_(dt)> = -7.164
dt = 1000.0 norm = 1.00001 <E> = 17.5434 <x_(dt)> = -4.335
dt = 1500.0 norm = 1.00002 <E> = 17.5436 <x_(dt)> = -1.55
dt = 2000.0 norm = 1.00003 <E> = 17.5439 <x_(dt)> = 0.8044
dt = 2500.0 norm = 1.00003 <E> = 17.5441 <x_(dt)> = 2.91
dt = 3000.0 norm = 1.00004 <E> = 17.5443 <x_(dt)> = 5.103
dt = 3499.0 norm = 1.00004 <E> = 17.5446 <x_(dt)> = 7.305

```

```

# Dos dados obtidos, resulta:
def opt_plot():
    plt.minorticks_on()
    plt.tick_params(axis='both',which='minor', direction = "in",

```

```

        top = True,right = True, length=5,width=1,
        labelsize=15)
plt.tick_params(axis='both',which='major', direction = "in",
                top = True,right = True, length=8,width=1,
                labelsize=15)

plt.figure(figsize=(8,5))

if vev_E0>max(V):
    plt.title('Espalhamento')
else:
    plt.title('Tunelamento')
plt.ylabel('$\psi(x,t)$')
plt.xlabel('$x$')
# plt.plot(x,np.abs(psi_t0)/np.sqrt(Delta_x),label="$|\Psi(x,t=0)|$")

for t,p in psi_tu:

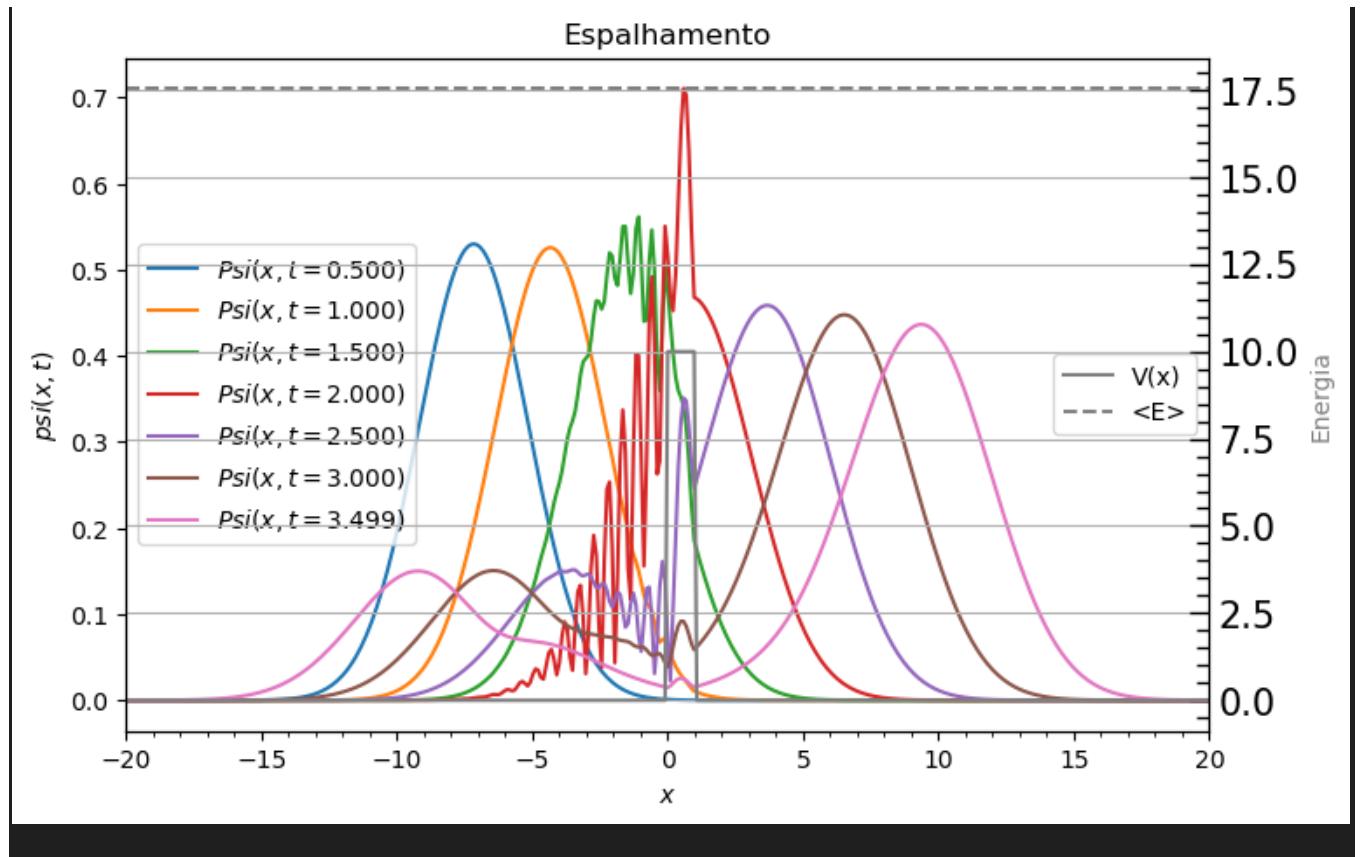
    plt.plot(x,np.abs(p)/np.sqrt(Delta_x),label="$|\Psi(x,t={:6.3f})|${}.format(t*dt)")
    plt.legend(loc = 'center left')

ax1 = plt.twinx()
plt.plot(x,V,color="grey",label="V(x)")
plt.plot([x[0],x[N-1]],[vev_E0,vev_E0],color="grey",linestyle="--",label="<E>")
plt.ylabel("Energia",color="grey")
plt.xlim(g_x0-5*g_sig,-g_x0+5*g_sig)
plt.legend(loc='best')
plt.grid()
opt_plot()

plt.show()

#OUTPUT:

```



APÊNDICE C: LINK DO REPOSITÓRIO DOS CÓDIGOS GITHUB:

https://github.com/SamuelDelmonte/TCC_Bacharelado_Fisica.git