

Simulating the Klein-Gordon Field on a Lattice using a Custom C++ Statevector Simulator

Darsh Maheshwari

September 2025

Abstract

This project presents a first-principles simulation of particle dynamics governed by the 1D Klein-Gordon equation, a foundational model in quantum field theory. The theoretical framework involves discretizing the continuous field onto a spatial lattice and mapping the resulting Hamiltonian to an equivalent XY model in a transverse field. To execute the simulation, a high-performance statevector simulator was developed from scratch in C++ using the Eigen library for efficient linear algebra. The simulator's output is piped in real-time to a separate Python script, which leverages Matplotlib and SciPy to generate a smooth, animated visualization of the particle's probability wavepacket. The simulation successfully models the time evolution of a single-particle excitation, clearly demonstrating the characteristic propagation and delocalization of its wavepacket across the lattice, in agreement with theoretical predictions. This work serves as a comprehensive demonstration of the pipeline from abstract field theory to concrete computational physics, providing a robust foundation for future explorations into more complex interacting theories.

Contents

1	Introduction	3
1.1	Klein-Gordon Equation	3
1.2	Challenge and Objective	3
2	Theoretical Foundations	3
2.1	The Continuum Klein-Gordon Hamiltonian	3
2.2	Discretization on a 1D Lattice	4
2.3	Quantization and Qubit Mapping	4
2.3.1	Fermion-Qubit Mapping:	4
2.3.2	Hard-Core Boson Approximation:	5
2.4	The Evolution	5
3	Implementation	6
3.1	Site Operator:	6
3.2	Evolution:	7
3.3	The Hamiltonian:	7
4	Results	8

1 Introduction

Quantum Field Theory is a technique for dealing with a quantum system of many particles. It is necessary when the number of particles is not conserved, as is the case in a system of photons and in high-energy collisions of two electrons, which can produce electron-positron pairs besides two electrons. It is also a convenient tool when the number of particles is large but conserved.

1.1 Klein-Gordon Equation

We shall work in units $\hbar = 1$ and $c = 1$. In any formula, we can recover powers of \hbar and c using dimensional analysis.

$$-\frac{\partial^2 \phi}{\partial t^2} = -\vec{\nabla}^2 \phi + m^2 \phi$$

This is the **Klein-Gordon** equation.

1.2 Challenge and Objective

As there are infinite degrees of freedom, simulating QFT is computationally hard, and thus classical computers are insufficient. Hence, there is a need for quantum simulation or advanced classical techniques.

This report details the development of a C++ based quantum simulator from first principles to model the real-time dynamics of a particle in the 1D Klein-Gordon field. We will demonstrate the propagation of a localised field excitation on a discrete lattice. To achieve this, we will first lay out the theoretical foundation, transforming the continuous field theory into a discrete, simulatable model.

2 Theoretical Foundations

2.1 The Continuum Klein-Gordon Hamiltonian

Firstly, we want to define the Hamiltonian for our field. We do so by first constructing the Lagrangian of the field and defining its conjugate momentum.

The Lagrangian from which we can derive the Klein-Gordon equations is,

$$L = \frac{1}{2} \int d^3r [(\dot{\phi})^2 - (\vec{\nabla}\phi)^2 - m^2\phi^2]$$

$$\implies \Pi(t, \vec{r}') = \frac{\delta L}{\delta \dot{\phi}(t, \vec{r}')} = \int d^3r \dot{\phi}(t, \vec{r}) \delta^{(3)}(\vec{r} - \vec{r}') = \dot{\phi}(t, \vec{r}')$$

where $\Pi(t, \vec{r}')$ is the conjugate momentum.

Now, we find the Hamiltonian by Legendre transformation as,

$$H = \int d^3r \Pi(t, \vec{r}) \dot{\phi}(t, \vec{r}) - L$$

$$\Rightarrow H = \frac{1}{2} \int d^3r [\Pi(t, \vec{r})^2 + (\vec{\nabla} \phi(t, \vec{r}))^2 + m^2 \phi(t, \vec{r})^2]$$

For a continuous 1-D field, we can define our Hamiltonian as:

$$H = \frac{1}{2} \int d^3r [\Pi(x)^2 + (\frac{\partial \phi(x)}{\partial x})^2 + m^2 \phi^2(x)]$$

Here, $\Pi^2(x)$ represents the kinetic energy, $(\frac{\partial \phi(x)}{\partial x})^2$ represents the spatial energy, and $m^2 \phi^2(x)$ represents the mass energy.

2.2 Discretization on a 1D Lattice

A computer cannot handle an infinite, continuous space. So, we replace the continuous 1D line with our chain of N points separated by distance a.

$$H = \sum_{j=0}^{N-1} a \cdot \frac{1}{2} [\Pi_j^2 + \frac{1}{a^2} (\phi_{j+1} - \phi_j)^2 + m^2 \phi_j^2]$$

This describes a system of N coupled harmonic oscillators!

The term $(\phi_{j+1} - \phi_j)$ is the coupling term. It connects each oscillator to its neighbours. It's like having a spring between each site.

2.3 Quantization and Qubit Mapping

2.3.1 Fermion-Qubit Mapping:

An important thing about fermions is that **a given index cannot be repeated more than once**, since the result will vanish by anti-symmetry. This means that occupancy number m_n of the n -th state can be either 0 or 1, but not > 1 . This is the famous **Pauli exclusion principle**.

A system with only two states per site is mathematically identical (isomorphic) to a system of spin- $\frac{1}{2}$ particles, which is what qubits represent.

$$|1100\rangle \rightarrow |\uparrow\uparrow\downarrow\downarrow\rangle$$

Similarly, the creation and annihilation operators can be mapped to the spin operators:

$$\hat{a}_j \rightarrow \sigma_j^{(-)}$$

$$\hat{a}_j^\dagger \rightarrow \sigma_j^{(+)}$$

This truncation is what makes the model so useful for quantum computation.

2.3.2 Hard-Core Boson Approximation:

We are going to assume a theoretical framework where bosons cannot occupy the same site due to infinite repulsive interaction. This interaction **mimics** a sort of exclusion principle, where a site can be either empty or occupied by one boson. Now, rewriting our operators:

$$\hat{\phi}_j = k_1(\hat{a}_j + \hat{a}_j^\dagger)$$

$$\hat{\Pi}_j = k_2(\hat{a}_j - \hat{a}_j^\dagger)$$

where k_1, k_2 are constants which we'll ignore for now.

Rewriting our Hamiltonian,

$$(\phi_{j+1} - \phi_j)^2 \rightarrow (\hat{a}_{j+1} + \hat{a}_{j+1}^\dagger - \hat{a}_j - \hat{a}_j^\dagger)^2$$

here, terms like \hat{a}_j^2 will vanish due to our fermionic approximation. Since our simulation is set up for a single, long-lived particle, we are explicitly interested in the low-energy sector of the theory where the number of particles is constant. Hence, terms like $\hat{a}_{j+1}\hat{a}_j$ will also vanish. So, we are left only with the cross terms that annihilate the particle at one site and simultaneously create a particle at another site.

$$\rightarrow \hat{a}_{j+1}^\dagger \hat{a}_j + \hat{a}_j^\dagger \hat{a}_{j+1}$$

Similarly, $\hat{\Pi}_j$ and $\hat{\phi}_j^2$ will produce terms that only operates at site j .

Now, using the Fermion-Qubit Mapping:

$$\hat{a}_j = \sigma_j^{(+)} = \frac{\hbar}{2}(X_j + iY_j)$$

where X, Y are the Pauli spin matrices.

Using properties of the Pauli matrices, our Hamiltonian simplifies to:

$$H = -J \sum_{j=0}^{N-2} (X_j X_{j+1} + Y_j Y_{j+1}) + h \sum_{j=0}^{N-1} Z_j$$

where J, h absorbs all the constants that we've been ignoring.

2.4 The Evolution

By Schrodinger's equation,

$$|\psi(t)\rangle = e^{-iHt} |\psi(0)\rangle$$

3 Implementation

3.1 Site Operator:

```

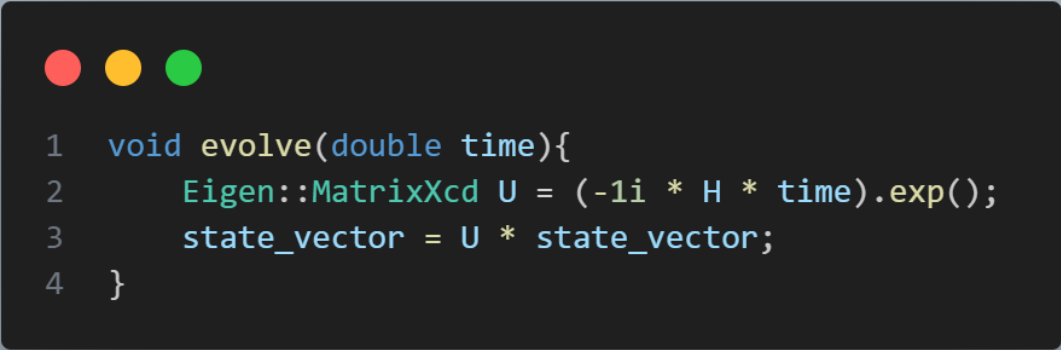
1  Eigen::MatrixXcd get_operator_for_site(const std::string& gate, int site) {
2      std::vector<Eigen::Matrix2cd> op_list(n, GATES_["I"]);
3      op_list[site] = GATES_[gate];
4
5      Eigen::MatrixXcd full_matrix = op_list[0];
6      for (int i = 1; i < n; ++i) {
7          full_matrix = Eigen::kroneckerProduct(full_matrix, op_list[i]).eval();
8      }
9      return full_matrix;
10 }
11
12 Eigen::MatrixXcd get_operator_for_two_sites(const std::string& g1, int s1, const std::string& g2, int s2) {
13     std::vector<Eigen::Matrix2cd> op_list(n, GATES_["I"]);
14     op_list[s1] = GATES_[g1];
15     op_list[s2] = GATES_[g2];
16
17     Eigen::MatrixXcd full_matrix = op_list[0];
18     for (int i = 1; i < n; ++i) {
19         full_matrix = Eigen::kroneckerProduct(full_matrix, op_list[i]).eval();
20     }
21     return full_matrix;
22 }

```

This function returns the full system matrix with operator "gate" on some site "j".
e.g. for $N = 5$, `get_operator_for_site("Z", 2)` returns

$$I \otimes I \otimes Z \otimes I \otimes I$$

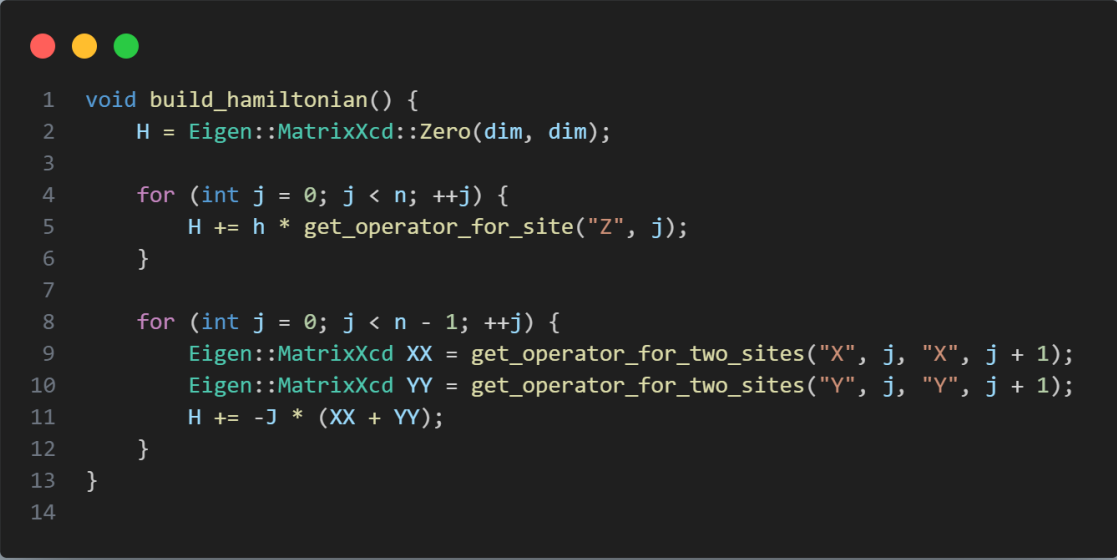
3.2 Evolution:



```
1 void evolve(double time){
2     Eigen::MatrixXcd U = (-1i * H * time).exp();
3     state_vector = U * state_vector;
4 }
```

This function uses the Schrodinger equation to update our statevector.

3.3 The Hamiltonian:

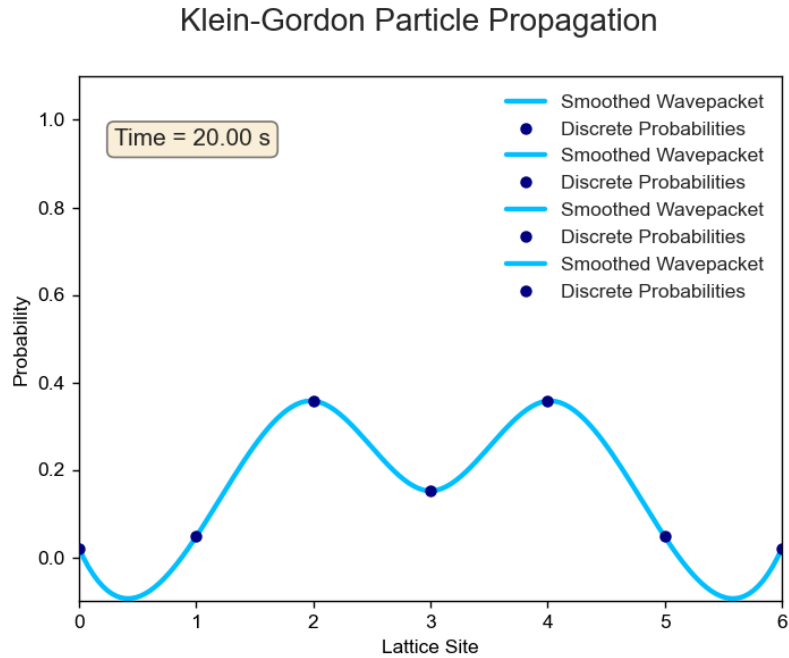
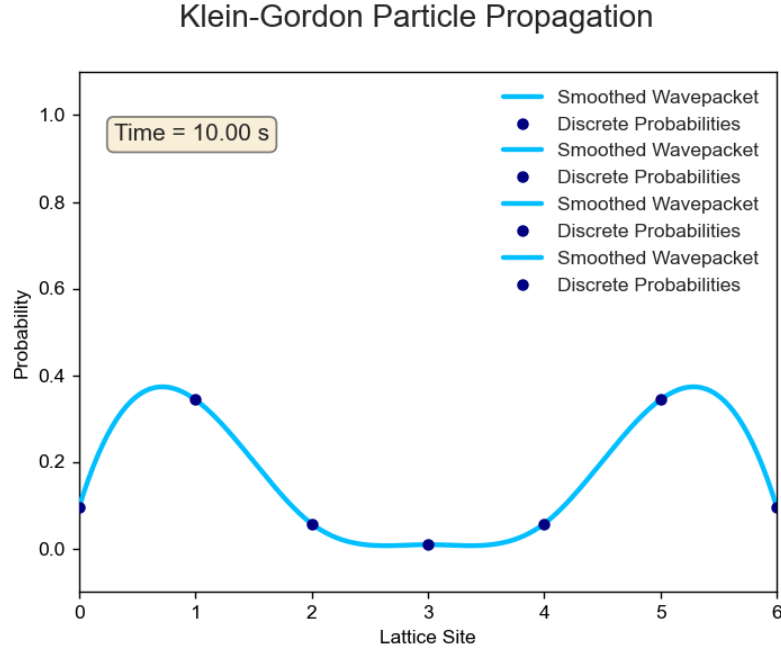


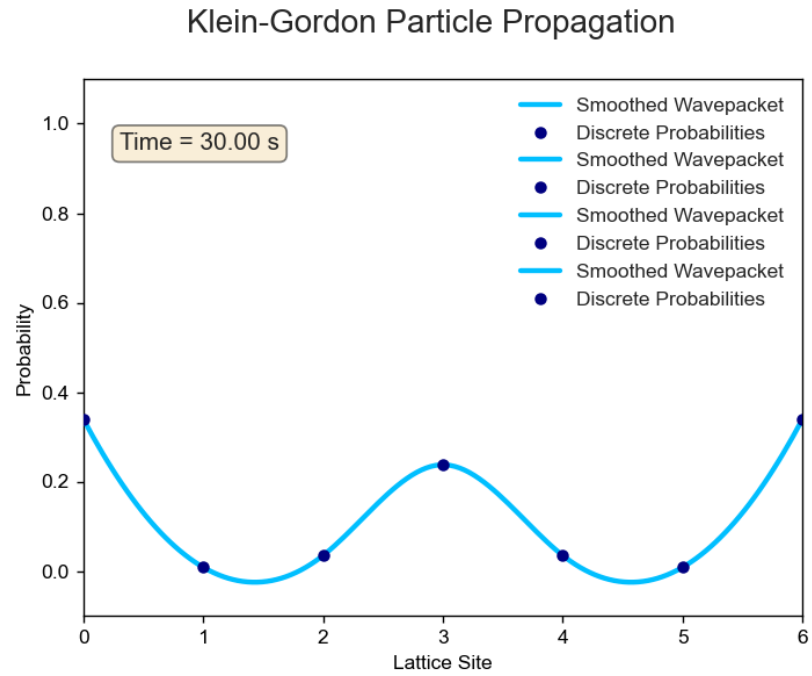
```
1 void build_hamiltonian() {
2     H = Eigen::MatrixXcd::Zero(dim, dim);
3
4     for (int j = 0; j < n; ++j) {
5         H += h * get_operator_for_site("Z", j);
6     }
7
8     for (int j = 0; j < n - 1; ++j) {
9         Eigen::MatrixXcd XX = get_operator_for_two_sites("X", j, "X", j + 1);
10        Eigen::MatrixXcd YY = get_operator_for_two_sites("Y", j, "Y", j + 1);
11        H += -J * (XX + YY);
12    }
13 }
14
```

This is the most important function. It builds the Hamiltonian as per the equation that we derived.

4 Results

Time evolution of the particle probability wavepacket on a 7-site lattice with $J=1.0$ and $\hbar=0.5$:





Appendix A: Full Source Code

main.cpp

```

1  #include<iostream>
2  #include<vector>
3  #include<map>
4  #include<complex>
5  #include<string>
6  #include<iomanip>
7  #include<Eigen/Dense>
8  #include <unsupported/Eigen/KroneckerProduct>
9  #include <unsupported/Eigen/MatrixFunctions>
10 using namespace std::complex_literals;
11 using namespace std;
12
13 using complex_d = complex<double>;
14
15 class KleinGordonSimulator{
16     public:
17     KleinGordonSimulator(int n, double J, double h) : n(n), J(J), h(h), dim(1LL << n) {
18         GATES_["I"] = Eigen::Matrix2cd::Identity();
19         GATES_["X"] = (Eigen::Matrix2cd() << 0, 1, 1, 0).finished();
20         GATES_["Y"] = (Eigen::Matrix2cd() << 0, -1i, 1i, 0).finished();
21         GATES_["Z"] = (Eigen::Matrix2cd() << 1, 0, 0, -1).finished();
22         ;
23
24         state_vector = Eigen::VectorXcd::Zero(dim);
25         state_vector(0) = 1.0;
26
27         build_hamiltonian();
28     }
29
30     void initial_particle(int site){
31         if(site >=n || site < 0){
32             throw std::runtime_error("Error: site index out of range.");
33         }
34         state_vector = Eigen::VectorXcd::Zero(dim);
35         long long index = 1LL << site;
36         state_vector(index) = 1.0;
37     }
38
39     void evolve(double time){
40         Eigen::MatrixXcd U = (-1i * H * time).exp();
41         state_vector = U * state_vector;

```

```

41     }
42
43     vector<double> measure(){
44         vector<double> probabilities(n, 0.0);
45         for(long long i = 0; i < n; ++i){
46             Eigen::MatrixXcd n_op = get_operator_for_site("Z", i);
47             complex_d expectation_z = (state_vector.adjoint() * n_op
48                                     * state_vector)(0);
49
50             probabilities[i] = 0.5*(1.0 - expectation_z.real());
51         }
52         return probabilities;
53     }
54 private:
55     int n;
56     double J;
57     double h;
58     long long dim;
59
60     Eigen::VectorXcd state_vector;
61     Eigen::MatrixXcd H;
62     std::map<string, Eigen::Matrix2cd> GATES_;
63
64     void build_hamiltonian() {
65         H = Eigen::MatrixXcd::Zero(dim, dim);
66
67         for (int j = 0; j < n; ++j) {
68             H += h * get_operator_for_site("Z", j);
69         }
70
71         for (int j = 0; j < n - 1; ++j) {
72             Eigen::MatrixXcd XX = get_operator_for_two_sites("X", j,
73                     "X", j + 1);
74             Eigen::MatrixXcd YY = get_operator_for_two_sites("Y", j,
75                     "Y", j + 1);
76             H += -J * (XX + YY);
77         }
78     }
79
80     Eigen::MatrixXcd get_operator_for_site(const std::string& gate,
81     int site) {
82         std::vector<Eigen::Matrix2cd> op_list(n, GATES_["I"]);
83         op_list[site] = GATES_[gate];
84
85         Eigen::MatrixXcd full_matrix = op_list[0];
86         for (int i = 1; i < n; ++i) {
87             full_matrix = Eigen::kroneckerProduct(full_matrix,

```

```

84         op_list[i]).eval();
85     }
86     return full_matrix;
87 }
88 Eigen::MatrixXcd get_operator_for_two_sites(const std::string&
89     g1, int s1, const std::string& g2, int s2) {
90     std::vector<Eigen::Matrix2cd> op_list(n, GATES_["I"]);
91     op_list[s1] = GATES_[g1];
92     op_list[s2] = GATES_[g2];
93
94     Eigen::MatrixXcd full_matrix = op_list[0];
95     for (int i = 1; i < n; ++i) {
96         full_matrix = Eigen::kroneckerProduct(full_matrix,
97             op_list[i]).eval();
98     }
99     return full_matrix;
100 }
101 };
102
103 void stream_data_for_plotting(double time, const std::vector<double
104     >& probs) {
105     std::cout << time;
106     for(const auto& p : probs) {
107         std::cout << ", " << p;
108     }
109     std::cout << std::endl;
110 }
111
112 int main() {
113     const int N_SITES = 7;
114     const double J_COUPLING = 1.0;
115     const double H_FIELD = 0.5;
116     const double TOTAL_TIME = 100;
117     const double TIME_STEP = 1;
118
119     KleinGordonSimulator sim(N_SITES, J_COUPLING, H_FIELD);
120
121     int middle_site = N_SITES / 2;
122     sim.initial_particle(middle_site);
123
124     double current_time = 0.0;
125     while (current_time <= TOTAL_TIME) {
126         auto probs = sim.measure();
127         stream_data_for_plotting(current_time, probs);

```

```

127         sim.evolve(TIME_STEP);
128         current_time += TIME_STEP;
129     }
130
131     return 0;
132 }

```

plot.py

```

1  import subprocess
2  import matplotlib.pyplot as plt
3  import matplotlib.animation as animation
4  import numpy as np
5  import os
6  from scipy.interpolate import CubicSpline
7
8  EXECUTABLE_NAME = "./kg_sim"
9  if os.name == 'nt':
10     EXECUTABLE_NAME = "kg_sim.exe"
11
12  fig, ax = plt.subplots()
13  plt.style.use('seaborn-v0_8-darkgrid')
14
15  line = None
16  points = None
17  time_text = None
18  sites = []
19  sites_smooth = []
20
21  ax.set_ylim(-0.1, 1.1)
22  ax.set_xlabel("Lattice_Site")
23  ax.set_ylabel("Probability")
24  fig.suptitle("Klein-Gordon_Particle_Propagation", fontsize=16)
25
26  try:
27     process = subprocess.Popen(EXECUTABLE_NAME, stdout=subprocess.
28                                PIPE, text=True)
29  except FileNotFoundError:
30     print(f"Error: The executable '{EXECUTABLE_NAME}' was not found.")
31     print("Please make sure you have compiled the C++ code first.")
32     exit()
33
34  def init():
35     global line, points, sites, sites_smooth, time_text
36
37     first_line = process.stdout.readline()
38     if not first_line:

```

```

38         return []
39
40     data = [float(val) for val in first_line.strip().split(',')]
41     probabilities = data[1:]
42     num_sites = len(probabilities)
43     sites = np.arange(num_sites)
44     sites_smooth = np.linspace(sites.min(), sites.max(), 150)
45
46     spline = CubicSpline(sites, probabilities)
47     probs_smooth = spline(sites_smooth)
48
49     ax.set_xlim(sites.min(), sites.max())
50
51     line, = ax.plot(sites_smooth, probs_smooth, '-', lw=2.5, color='
    deepskyblue', label='Smoothed_Wavepacket')
52     points, = ax.plot(sites, probabilities, 'o', color='navy',
    markersize=5, label='Discrete_Probabilities')
53
54     time_text = ax.text(0.05, 0.9, '', transform=ax.transAxes,
    fontsize=12,
55                        verticalalignment='top', bbox=dict(boxstyle='
    round', facecolor='wheat', alpha=0.5))
56
57     ax.legend(loc='upper_right')
58
59     return line, points, time_text
60
61 def update(frame):
62     line_data = process.stdout.readline()
63
64     if not line_data:
65         ani.event_source.stop()
66         return ()
67
68     data = [float(val) for val in line_data.strip().split(',')]
69     current_time = data[0]
70     probabilities = data[1:]
71
72     spline = CubicSpline(sites, probabilities)
73     probs_smooth = spline(sites_smooth)
74
75     line.set_ydata(probs_smooth)
76     points.set_ydata(probabilities)
77
78     time_text.set_text(f'Time = {current_time:.2f} s')
79
80     return line, points, time_text

```

```
81  
82 ani = animation.FuncAnimation(fig, update, init_func=init, blit=True  
    , interval=75, repeat=False, cache_frame_data=False)  
83  
84 plt.show()  
85  
86 process.terminate()
```