Hagedorn wavepackets for multi-dimensional quantum dynamics

Lipeng Chen
Zhejiang Laboratory, Hangzhou 311100, China

We summarized the general properties of the multi-dimensional Hagedorn wavepackets in Section I, i.e., how to recursively construct the higher-order Hagedorn basis functions in Sec. 1.2, the definition of the basis shapes and basis set expansion in Sec. 1.3, the computation of the gradient of the Hagedorn wavepackets in Sec. 1.4, the calculation of the observables and inner products using the Gaussian Hermite quadrature in Sec. 1.5, the time-stepping algorithm for the propagation of the Hagedorn wavepackets in Sec. 1.6, and the generalization of the Hagedorn wavepacket to the non-adiabatic dynamics in Sec. 1.7. In Sec 2, We list all the necessary C++ classes that implement those properties of the Hagedorn wavepackets. In Sec 3, we present simulation results for three simple systems, 2D harmonic oscillators (Sec. 3.1), 2D torsional potential (Sec. 3.2), and 5D torsional potential (Sec. 3.3). The convergence properties of the Hagedorn wavepackets with the number of the basis functions are carefully checked. Finally, we list properties of the 1D Hagedorn wavepackets and the generalized coherent states in Appendices A and B.

1. Hagedorn wavepackets

1.1. General properties

We consider the time-dependent Schrödinger equation in semiclassical scaling:

$$i\varepsilon \frac{d}{dt}|\psi(t)\rangle = \hat{H}|\psi(t)\rangle \tag{1}$$

where $\psi = \psi(x,t)$ is the wave function depending on the spatial variables $x = (x_1, \cdots, x_D) \in \mathbb{R}^D$ and the time $t \in \mathbb{R}$. ε is a small positive number representing the scaled Planck constant. It is noted that in the limit $\varepsilon \to 0$ we are back in the classical limit and for bigger ε we get more and more quantum effects. The Hamiltonian operator is given by

$$\hat{H} = \hat{T} + \hat{V}(x) \tag{2}$$

with the kinetic energy

$$T = -\sum_{j=1}^{D} \frac{\varepsilon^2}{2m_j}\frac{\partial^2}{\partial x_j^2} \tag{3}$$

where D is the number of degrees of freedom (DOFs).

In Hagedorn's approach [1], a Gaussian wavepacket is parametrized as

$$\phi_0[p,q,Q,P](x) = (\pi\varepsilon)^{-\frac{D}{4}}(\det Q)^{-\frac{1}{2}} \exp\left(\frac{i}{2\varepsilon}(x-q)^T \cdot PQ^{-1}\cdot(x-q) + \frac{i}{\varepsilon}p^T\cdot(x-q)\right) \tag{4}$$

where $q, p \in \mathbb{R}^D$ denote the position and momentum, respectively. The complex matrices $Q, P \in \mathbb{C}^{D\times D}$ satisfy symplecticity conditions,

$$Q^*P - P^*Q = 2iI \tag{5}$$

$$P^TQ - Q^TP = 0 \tag{6}$$

Here Q^T denotes the transpose of Q, and Q^* is the transpose and conjugate of Q. The above two equations guarantee that both Q and P are invertible, and PQ^{-1} is complex symmetric with positive definite imaginary part

$$\mathrm{Im} PQ^{-1} = (QQ^*)^{-1} \tag{7}$$

Furthermore, it should be noted that the above two relations are equivalent to requiring that

$$Y = \begin{pmatrix} \mathrm{Re}Q & \mathrm{Im}Q \\ \mathrm{Re}P & \mathrm{Im}P \end{pmatrix} \tag{8}$$

be symplectic, i.e.,

$$Y^T J Y = J, \quad \text{with} \quad J = \begin{pmatrix} 0 & -I \\ I & 0 \end{pmatrix} \tag{9}$$

Hagedorn constructs a complete L^2-orthonormal set of functions

$$\phi_K(x) = \phi_K^\varepsilon[q,p,Q,P](x) \tag{10}$$

where K is a multi-index

$$K := (k_1, \cdots, k_D) \in \mathbb{N}_0^D \tag{11}$$

Its length is defined as

$$|K| = \sum_{i=1}^{D} k_i \tag{12}$$

and its factorial as

$$K! := (k_1!)\cdots(k_D!) = \prod_{i=1}^{D} k_i! \tag{13}$$

These functions can be recursively constructed as follows [1]: let $\hat{q}$ denote the position operator and $\hat{p} = -i\varepsilon\nabla_x$ the momentum operator, and introducing the raising operator $A^\dagger$ and lowering operator A as

$$A^\dagger = A^\dagger[q,p,Q,P] = \frac{i}{\sqrt{2\varepsilon}}\left(P^*(\hat{q}-q) - Q^*(\hat{p}-p)\right) \tag{14}$$

$$A = A[q,p,Q,P] = -\frac{i}{\sqrt{2\varepsilon}}\left(P^T(\hat{q}-q) - Q^T(\hat{p}-p)\right) \tag{15}$$

It should be noted that for $D = 1$, $\varepsilon = 1$, $q = 0$, $p = 0$, $Q = 1$, $P = i$, these operators reduce to Dirac's ladder operators $\frac{1}{\sqrt{2}}(\hat{q}+i\hat{p})$ and $\frac{1}{\sqrt{2}}(\hat{q}-i\hat{p})$, respectively.

With $\langle j\rangle = e_j = (0\cdots 1 \cdots)$ denoting the jth unit vector, we have

$$\phi_{K+\langle j\rangle} = \frac{1}{\sqrt{k_j+1}} A_j^\dagger \phi_K \tag{16}$$

$$\phi_{K-\langle j\rangle} = \frac{1}{\sqrt{k_j}} A_j \phi_K \tag{17}$$

Furthermore, with the help of $A^\dagger$ we can define all higher order states ϕ_K

$$\begin{aligned} \phi_K &= A^{\dagger K}\phi_0 \\ &= \frac{1}{\sqrt{K!}} A_1^{\dagger k_1} \cdots A_D^{\dagger k_D} \phi_0 \\ &= \frac{1}{\sqrt{\prod_{i=1}^{D} k_i!}} \prod_{i=1}^{D} A_i^{\dagger k_i} \phi_0 \end{aligned} \tag{18}$$

The above relation imply that the functions ϕ_K are polynomials of degree $|K| = k_1 + \cdots + k_D$ multiplied with the ground state Gaussian ϕ_0 (see Appendix A for the explicit formula of 1D case).

1.2. Higher order basis functions

Next we will discuss how to compute the higher order functions ϕ_K recursively in an efficient manner. We note that computing the action of $A^\dagger$ is not straight forward since it contains the differential operator $\hat{p} = -i\varepsilon\nabla_x$. For this purpose, we seek a way to compute $A^\dagger\phi_0$ without ever applying $\hat{p}$ explicitly. We have following formula (see Hagedorn's original paper [1] for detailed derivation, i.e., Eq. (3.28) of Ref.[1])

$$A^\dagger = \sqrt{\frac{2}{\varepsilon}}Q^{-1}(\hat{q}-q) - Q^*Q^{-T}A \tag{19}$$

and

$$A = \sqrt{\frac{2}{\varepsilon}}\overline{Q}^{-1}(\hat{q}-q) - Q^TQ^{*-1}A^\dagger \tag{20}$$

Here $\overline{Q}$ denote the conjugate of the complex matrix Q.

We proceed to the calculation of higher order basis functions. Acting $A^\dagger$ on ϕ_K and applying Eq. 19 gives us

$$A^\dagger\phi_K = \sqrt{\frac{2}{\varepsilon}}Q^{-1}(\hat{q}-q)\phi_K - Q^*Q^{-T}A\phi_K \tag{21}$$

In order to obtain a compact form of above equation, we use following formula for $A^\dagger\phi_K$ and $A\phi_K$

$$\begin{pmatrix} \sqrt{k_1+1}\phi_{K+\langle 1\rangle} \\ \vdots \\ \sqrt{k_D+1}\phi_{K+\langle D\rangle} \end{pmatrix} = \begin{pmatrix} A_1^\dagger\phi_K \\ \vdots \\ A_D^\dagger\phi_K \end{pmatrix} = A^\dagger\phi_K \tag{22}$$

$$\begin{pmatrix} \sqrt{k_1}\phi_{K-\langle 1\rangle} \\ \vdots \\ \sqrt{k_D}\phi_{K-\langle D\rangle} \end{pmatrix} = \begin{pmatrix} A_1\phi_K \\ \vdots \\ A_D\phi_K \end{pmatrix} = A\phi_K \tag{23}$$

We thus have final equation for the recursive calculation of the high order basis functions

$$\begin{pmatrix} \sqrt{k_1+1}\phi_{K+\langle 1\rangle} \\ \vdots \\ \sqrt{k_D+1}\phi_{K+\langle D\rangle} \end{pmatrix} = \sqrt{\frac{2}{\varepsilon}}Q^{-1}(x-q)\phi_K - Q^*Q^{-T}\begin{pmatrix} \sqrt{k_1}\phi_{K-\langle 1\rangle} \\ \vdots \\ \sqrt{k_D}\phi_{K-\langle D\rangle} \end{pmatrix} \tag{24}$$

After resolving the problem of how to recursively compute the functions $\phi_K(x)$, we can take a general set $\mathfrak{K}$ of indices K and use the corresponding ϕ_K to build a basis for $L^2(\mathbb{R}^D)$. The scalar wavepackets Φ can be further constructed by a linear combinations of those basis functions

$$|\Phi\rangle := \Phi[\Pi(t)](x,t) = \exp\left(\frac{iS(t)}{\varepsilon}\right)\sum_{K\in\mathfrak{K}} c_K(t)\phi_K[\Pi(t)](x) \tag{25}$$

where $c_K \in \mathbb{C}$ are expansion coefficients and $\Pi = \{p, q, Q, P, S\}$ are the Hagedorn parameter set.

1.3. Basis shapes and basis set expansion

The basis expansion of Eq. 25 is exact if we take the full lattice $\mathfrak{K} = \mathbb{N}_0^D$ of indices. In all practical calculation, we need to truncate the basis and make the set $\mathfrak{K}$ finite. In this subsection, we will discuss various shapes of a basis set. Specifically, we will mainly focus on the basis set with Hypercubic basis shape, Hyperbolic cut basis shape, Hyperbolic cut basis shape with limits and Simplex basis shape, which have been implemented in our code. The formal definition of those basis shapes are:

Definition 1. (Hypercubic basis shape)

$$\mathfrak{K}(M) := \left\{K \in \mathbb{N}_0^D; k_d < M_d, \forall d \in [1, \cdots, D]\right\} \tag{26}$$

Definition 2. (Hyperbolic cut basis shape)

$$\mathfrak{K}(n) := \left\{K \in \mathbb{N}_0^D : \prod_{d=1}^{D} (1 + k_d) \leq n\right\} \tag{27}$$

Definition 3. (Hyperbolic cut basis shape with limits)

$$\mathfrak{K}(n, M) := \left\{K \in \mathbb{N}_0^D : \prod_{d=1}^{D} (1 + k_d) \leq n \wedge k_d < M_d, \forall d \in [1, \cdots, D]\right\} \tag{28}$$

Definition 4. (Simplex basis shape)

$$\mathfrak{K}(n) := \left\{K \in \mathbb{N}_0^D : \sum_{d=1}^{D} k_d \leq n\right\} \tag{29}$$

It is advantageous to bring the elements K of a basis shape $\mathfrak{K}$ into a fixed total order. This can be done by a certain linearisation mapping

Definition 5. ((Linearisation mapping) A mapping:

$$\begin{aligned} \mu : & \mathfrak{K} \to \mathbb{N}_0 \\ & K = (k_1, \cdots, k_D) \mapsto n \end{aligned} \tag{30}$$

that fixes a total order of the set $\mathfrak{K}$

For computing the gradients of wavepackets we need to extend the basis shape, whose definition is

Definition 6. (Basis shape extension) Given a basis shape $\mathfrak{K}$ we define its extension $\overline{\mathfrak{K}}$ by

$$\overline{\mathfrak{K}} := \mathfrak{K} \cup \left\{K^{'} : K^{'} = K + \langle d \rangle, \forall d \in [1, \cdots, D], \forall K \in \mathfrak{K}\right\} \tag{31}$$

This defines the most tight extension. But any even larger basis shape is a valid extension too. In any case it holds that $\mathfrak{K} \subset \overline{\mathfrak{K}}$

1.4. Gradient computation

In this subsection, we give an explicit formula to calculate the gradient of a scalar wavepacket Φ, i.e., $\hat{p}\Phi = -i\varepsilon\nabla\Phi$, which is needed for the computation of the kinetic energy. The explicit expression of $\hat{p}$ represented in terms of the raising and lowering operator $A^\dagger$ and A is (for a detailed derivation, please see Hagedorn's orginal paper, Eq. (3.29) of Ref. [1])

$$\hat{p} = \sqrt{\frac{\varepsilon}{2}}(PA^\dagger + \overline{P}A) + p \tag{32}$$

With explicit representation of the $\hat{p}$ operator in terms of raising and lowering operators, we can act $\hat{p}$ operator on an arbitrary basis function ϕ_K. Our final goal is to apply $\hat{p}$ to the whole scalar wavepacket Φ in order to obtain its kinetic energy. For this purpose, we first write the gradient in a vector form as

$$\nabla_x := \begin{pmatrix} \frac{\partial}{\partial x_1} \\ \vdots \\ \frac{\partial}{\partial x_D} \end{pmatrix} \tag{33}$$

It is thus expected that the gradient applied to $\phi_K : \mathbb{R}^D \to \mathbb{C}$ yields a vector with D components. We can then easily obtain the explicit expression of $\hat{p}\phi_K$ by using Eq. 32,

$$\hat{p}\phi_K(x) = \sqrt{\frac{\varepsilon}{2}}(PA^\dagger + \overline{P}A)\phi_K(x) + p\phi_K(x) \tag{34}$$

The simple application of the ladder operators yields the following relation for the gradient of a single basis function

$$\hat{p}\phi_K(x) = \sqrt{\frac{\varepsilon}{2}}\left(P\begin{pmatrix} \sqrt{k_1+1}\phi_{K+\langle 1\rangle} \\ \vdots \\ \sqrt{k_D+1}\phi_{K+\langle D\rangle} \end{pmatrix} + \overline{P}\begin{pmatrix} \sqrt{k_1}\phi_{K-\langle 1\rangle} \\ \vdots \\ \sqrt{k_D}\phi_{K-\langle D\rangle} \end{pmatrix}\right) + p\phi_K \tag{35}$$

We then need to compute the gradient of a whole scalar wavepacket $\Phi = \sum_{K\in\mathfrak{K}} c_K\phi_K$ (We skip the phase factor $e^{\frac{iS}{\varepsilon}}$), i.e.,

$$\hat{p}\Phi = \sum_{K\in\mathfrak{K}} \hat{p}c_K\phi_K = \sum_{K\in\mathfrak{K}} c_K\hat{p}\phi_K \tag{36}$$

Since $\hat{p}\phi_{\underline{K}}$ has contributions from all neighbours $\phi_{K+\langle d\rangle}$ and $\phi_{K-\langle d\rangle}$ for all $d \in [1,\cdots,D]$ and from ϕ_K, we need to represent the gradient as linear combinations over the basis functions in the extended basis shape as

$$\hat{p}\Phi = \sum_{K\in\overline{\mathfrak{K}}} c_K^{'}\phi_K \tag{37}$$

where $c_K^{'} \in \mathbb{C}^D$ and $\overline{\mathfrak{K}}$ is the extended basis shape (see the definition of basis shape extension, Eq. 31). Using Eq. 35, we find the following general rule for the coefficient vectors $c_K^{'}$ for all $K \in \overline{\mathfrak{K}}$:

$$c_K^{'} = c_K p + \sqrt{\frac{\varepsilon}{2}}\sum_{d=1}^{D} c_{K+\langle d\rangle}\sqrt{k_d+1}\overline{P}_{:,d} + \sqrt{\frac{\varepsilon}{2}}\sum_{d=1}^{D} c_{K-\langle d\rangle}\sqrt{k_d}P_{:,d} \tag{38}$$

which can be further written in a compact form

$$c_K^{'} = c_K p + \sqrt{\frac{\varepsilon}{2}}\left(\overline{P}\begin{pmatrix} c_{K+\langle 1\rangle}\sqrt{k_1+1} \\ \vdots \\ c_{K+\langle D\rangle}\sqrt{k_D+1} \end{pmatrix} + P\begin{pmatrix} c_{K-\langle 1\rangle}\sqrt{k_1} \\ \vdots \\ c_{K-\langle D\rangle}\sqrt{k_D} \end{pmatrix}\right) \tag{39}$$

1.5. Observables and Inner Products

For any scalar function $f(x)$, our goal is to calculate

$$\langle \Phi_r^{'} | f(x) | \Phi_c \rangle \tag{40}$$

where $|\Phi_r^{'}\rangle$ and $|\Phi_c\rangle$ can be either identical (for the calculation of kinetic, potential, total energies, norm of the wavepacket) or different (for the calculation of autocorrelation function). By using Eq. 25, we explicitly have

$$\begin{aligned}\langle \Phi_r^{'} | f(x) | \Phi_c \rangle &= e^{\frac{i(S_c - S_r)}{\varepsilon}} \left\langle \sum_{K \in \mathfrak{R}_r} c_K^{'} \phi_K^{'} | f(x) | \sum_{L \in \mathfrak{R}_c} c_L \phi_L \right\rangle \\ &= e^{\frac{i(S_c - S_r)}{\varepsilon}} \sum_{K \in \mathfrak{R}_r} \sum_{L \in \mathfrak{R}_c} \bar{c}_K^{'} c_L \left\langle \phi_K^{'} | f(x) | \phi_L \right\rangle \end{aligned} \tag{41}$$

Now our central task is to compute

$$\left\langle \phi_K^{'} | f(x) | \phi_L \right\rangle = \int \cdots \int \overline{\phi_K^{'}(x)} f(x) \phi_L(x) dx \tag{42}$$

numerically by a special, high-order quadrature rule.

Let us start from the one-dimensional Gauss-Hermite quadrature rule

$$\int_{\mathbb{R}} e^{-x^2} f(x) dx \approx \sum_{i=0}^{R-1} \omega_i f(\gamma_i) \tag{43}$$

where ω_i and γ_i are the ith weights ω and nodes γ, respectively, for a quadrature of order R. The quadrature nodes are given as the roots of the Hermite polynomial $H_R(x)$

$$H_R(x) = (-1)^R e^{x^2} \frac{d^R}{dx^R} e^{-x^2} \tag{44}$$

It should be noted that in the practical calculations, we do not compute the nodes by finding the roots of these polynomials since this is inherently unstable. The quadrature weights are then given by

$$\omega_i = \frac{2^{R-1} R! \sqrt{\pi}}{R^2 H_{R-1}^2(\gamma_i)} \tag{45}$$

Since normally our integrals are

$$\int_{\mathbb{R}} g(x) dx \tag{46}$$

without the factor $\exp(-x^2)$, we need to modify our quadrature weights to take that factor into account. For this purpose, we can define new quadrature weights $\omega_i^{'}$ as

$$\omega_i^{'} = \frac{1}{R h_{R-1}^2(\gamma_i)} \tag{47}$$

where h_R are the Hermite functions defined as

$$h_R(x) := \frac{1}{\sqrt{2^R R! \sqrt{\pi}}} e^{-x^2/2} H_R(x). \tag{48}$$

For high-dimensional integrals

$$\int_{\mathbb{R}^D} f(x) dx \approx \sum_m \omega_m f(\gamma_m) \tag{49}$$

where γ_m and ω_m are the multi-dimensional quadrature nodes and weights, respectively for multi-indices $m = (m_1, \cdots, m_D)$, which can be constructed by one-dimensional quadrature nodes and weights as follows

$$\gamma_m = (\gamma_{m_1}, \cdots, \gamma_{m_D}) \tag{50}$$

$$\omega_m = \omega'_{m_1} \cdot \cdots \cdot \omega'_{m_D} \tag{51}$$

Now let us return back to our central task

$$\langle \phi_K[\Pi_k] | f | \phi_L[\Pi_l] \rangle \tag{52}$$

where $\Pi_k = \{q_k, p_k, Q_k, P_k\}$ and $\Pi_l = \{q_l, p_l, Q_l, P_l\}$ are the Hagedorn parameter sets for ϕ_K and ϕ_L, respectively. As discussed in Ref [2], we need to make the transformation of the quadrature nodes in order to compute the matrix elements by Gauss-Hermite quadrature. Explicitly, we need to transform the quadrature nodes by following relation (see Eq. (4.16) of Ref. [3])

$$\gamma'_i = q_0 + \sqrt{\varepsilon} Q_S \gamma_i \tag{53}$$

For the homogenous case ($\Pi_k = \Pi_l = \{q, p, Q, P\}$), we have

$$q_0 = q, \tag{54}$$

$$Q_S = (QQ^*)^{\frac{1}{2}} \tag{55}$$

For the inhomogenous case ($\Pi_k \neq \Pi_l$), we need Algorithm 10 of Ref [3] to obtain the q_0 and Q_S, i.e.,

$$\begin{aligned} \Gamma_k &= P_k Q_k^{-1} \\ \Gamma_l &= P_l Q_l^{-1} \\ \Gamma &= \Im(\Gamma_k - \Gamma_l^*) \\ q &= \Im(\Gamma_k q_k - \Gamma_l^* q_l) \\ q_0 &= \Gamma^{-1} q \\ Q_0 &= \frac{1}{2}\Gamma \\ Q_S &= (\sqrt{Q_0})^{-1} \end{aligned} \tag{56}$$

Finally we obtain the general formula for computing the braket

$$\langle \phi_K | f | \phi_L \rangle \approx \varepsilon^{D/2} \cdot \det(Q_S) \cdot \sum_{r=0}^{R-1} \overline{\phi_K(\gamma'_r)} \cdot f(\gamma'_r) \cdot \phi_L(\gamma'_r) \cdot \omega_r \tag{57}$$

where the two ϕ in general have different Hagedorn parameter sets. It should be noted that for the homogenous case, the above formula can be further simplified. If we omit a prefactor of $\frac{1}{\sqrt{\det(Q)}}$ when calculating $\phi(\gamma'_r)$, then the $\det(Q_S)$ in Eq. 57 cancels nicely with those prefactor, yielding simple formula

$$\langle \phi | f | \phi \rangle \approx \varepsilon^{D/2} \cdot \sum_{r=0}^{R-1} \overline{\phi(\gamma'_r)} \cdot f(\gamma'_r) \cdot \phi(\gamma'_r) \cdot \omega_r \tag{58}$$

Next let us compute the matrix elements like this one

$$F_{u_{\Re}(K), u_{\Re'}(L)} = \langle \phi_K[\Pi] | f | \phi_L[\Pi'] \rangle \tag{59}$$

where $K \in \mathfrak{R}$ and $L \in \mathfrak{R}^{'}$. The matrix F has a size of $|\mathfrak{R}| \times |\mathfrak{R}^{'}|(|\mathfrak{R}|$ denotes the basis size of the basis shape $\mathfrak{R}$), and the order of the entries is given by the linearisation mappings $u_{\mathfrak{R}}$ and $u_{\mathfrak{R}'}$

Finally let us give the explicit expression for the calculation of norm, kinetic energy, potential energy as well as the autocorrelation function. The norm of the wavepacket can be easily obtained as

$$\begin{aligned}\langle \Phi[\Pi] | \Phi[\Pi] \rangle &= \left\langle \exp\left(\frac{iS}{\varepsilon}\right) \sum_{K\in\mathfrak{R}} c_K \phi_K[\Pi](x) | \exp\left(\frac{iS}{\varepsilon}\right) \sum_{L\in\mathfrak{R}} c_L \phi_L[\Pi](x) \right\rangle \\ &= \sum_{K\in\mathfrak{R}} |c_K|^2 \end{aligned} \tag{60}$$

Next, we give the formula of the overlap integrals of wavepackets which is useful for the computation of the autocorrelation function.

$$\begin{aligned}\langle \Phi[\Pi] | \Phi^{'}[\Pi^{'}] \rangle &= \left\langle \exp(\frac{iS}{\varepsilon}) \sum_{K\in\mathfrak{R}} c_K \phi_K[\Pi](x) | \exp\left(\frac{iS'}{\varepsilon}\right) \sum_{L\in\mathfrak{R}'} c_L^{'} \phi_L[\Pi^{'}](x) \right\rangle \\ &= \exp\left(\frac{i}{\varepsilon}(S' - S)\right) \sum_{K\in\mathfrak{R}} \sum_{L\in\mathfrak{R}'} \overline{c_K} c_L^{'} \langle \phi_K[\Pi] | \phi_L[\Pi^{'}] \rangle \end{aligned} \tag{61}$$

Finally, the potential energy is calculated as

$$\langle \Phi | V(x) | \Phi \rangle = \sum_{K\in\mathfrak{R}} \sum_{L\in\mathfrak{R}} \overline{c_K} c_L \langle \phi_K | V(x) | \phi_L \rangle \tag{62}$$

and kinetic energy is

$$\begin{aligned}\langle \Phi(x) | T | \Phi(x) \rangle &= \langle \Phi(x) | -\frac{1}{2}\varepsilon^2 \bigtriangleup | \Phi(x) \rangle \\ &= \frac{1}{2} \langle \Phi(x) | (-i\varepsilon\nabla)(-i\varepsilon\nabla) | \Phi(x) \rangle \\ &= \frac{1}{2} \langle +i\varepsilon\nabla\Phi(x) | -i\varepsilon\nabla\Phi(x) \rangle \\ &= \frac{1}{2} || -i\varepsilon\nabla\Phi(x) ||^2 \end{aligned} \tag{63}$$

The braket simply expresses the squared norm of $-i\varepsilon\nabla\Phi$, which is already discussed in section 1 1.4.

1.6. Wavepacket Propagation

The propagation of q, p, Q, P, and S is the same as in the TGA, where we have kinetic propagation (T propagation) and potential propagation (V propagation). Explicitly,

Kinetic propagation (T propagation, $U_T(\Delta t)$, where U_T is the time evolution operator with Hamiltonian consists only of the kinetic energy)

$$\begin{aligned} q_t &= q_0 + \Delta t m^{-1} \cdot p_0, \\ Q_t &= Q_0 + \Delta t m^{-1} \cdot P_0, \\ S_t &= S_0 + \Delta t T(p_0) \end{aligned}$$

Potential propagation (V propagation, $U_V(\Delta t)$, where U_V is the time evolution operator with Hamiltonian consists only of the potential energy)

$$\begin{aligned} p_t &= p_0 - \Delta t \nabla V(q_0), \\ P_t &= P_0 - \Delta t \nabla^2 V(q_0) Q_0, \\ S_t &= S_0 - \Delta t V(q_0) \end{aligned}$$

In addition, one must propagate the Hagedorn coefficients c_K with the equation of motion

$$i\varepsilon \dot{\mathbf{c}}(t) = \mathbf{F}\mathbf{c}(t) \tag{64}$$

Here, $\mathbf{F} = (f_{KL})_{K,L\in\mathfrak{R}}$ is the Hermitian matrix with entries

$$f_{KL} = \langle \phi_K | V - V_{\mathrm{LHA}} | \phi_L \rangle \tag{65}$$

where $\phi_K = \phi_K[q, p, Q, P]$ are the Hagedorn basis functions, and V_{LHA} is the local harmonic approximation of V, i.e., $V_{\mathrm{LHA}}(x) = V(q) + (x-q)^T \cdot \nabla V(q) + \frac{1}{2}(x-q)^T \cdot \nabla^2 V(q) \cdot (x-q)$.

It should be noted that Lubich's time stepping algorithm to propagate the Hagedorn wavepacet can be simplified as: (1) kinetic propagation $U_T(\Delta t/2)$; (2) potential propagation $U_V(\Delta t)$; (3) Hagedorn coefficient propagation (Eq. 64); (4) kinetic propagation $U_T(\Delta t/2)$. The advantage is that we can easily extend Lubich's time stepping algorithm to the geometric integrators of arbitrary order.

1.7. Generalize the Hagedorn wavepacket to case of the multiple energy surfaces (Continue to implement this in the code)

Let us consider a potential with N energy levels as given by a symmetric real-values $N \times N$ matrix (note that all our discussions are based on the diabatic representation)

$$\mathbf{V}(x) := \begin{pmatrix} V_{1,1}(x) & \cdots & V_{1,N}(x) \\ \vdots & & \vdots \\ V_{N,1}(x) & \cdots & V_{N,N}(x) \end{pmatrix} \tag{66}$$

and the kinetic operator simply has a diagonal form

$$\mathbf{T} := \begin{pmatrix} T & & \\ & \ddots & \\ & & T \end{pmatrix} \tag{67}$$

we then introduce vector-valued wavepackets

$$|\Psi\rangle = \left| \begin{pmatrix} \Phi_1 \\ \vdots \\ \Phi_N \end{pmatrix} \right\rangle \tag{68}$$

and corresponding Schrödinger equation

$$i\varepsilon \frac{\partial}{\partial t} |\Psi\rangle = i\varepsilon \frac{\partial}{\partial t} \left| \begin{pmatrix} \Phi_1 \\ \vdots \\ \Phi_N \end{pmatrix} \right\rangle = \begin{pmatrix} & \\ & \mathbf{H} & \\ & \end{pmatrix} \left| \begin{pmatrix} \Phi_1 \\ \vdots \\ \Phi_N \end{pmatrix} \right\rangle \tag{69}$$

The Hamiltonian operator is matrix-valued now

$$\mathbf{H} := \mathbf{T} + \mathbf{V}(x) \tag{70}$$

We consider two kinds of vector-valued Hagedorn wavepackets. One is the homogeneous vectorial wavepacket where all the Hagedorn wavepackets share the same parameter set Π (something like the singlet set version of the MCTDH), i.e.,

Definition 7. (Homogeneous vectorial wavepacket)

$$|\Psi\rangle := \Psi[\Pi](x,t) = \begin{pmatrix} \Phi_1[\Pi](x,t) \\ \vdots \\ \Phi_N[\Pi](x,t) \end{pmatrix} \tag{71}$$

where $\Pi_i \equiv \Pi_j \equiv \Pi, \forall i,j$

Another one is the inhomogeneous vectorial wavepacket where different Hagedorn wavepackets has different parameter set Π (something like the multi-set version of the MCTDH), i.e.,

Definition 8. (Inhomogeneous vectorial wavepacket)

$$|\Psi\rangle := \Psi[\Pi_1,\cdots,\Pi_N](x,t) = \begin{pmatrix} \Phi_1[\Pi_1](x,t) \\ \vdots \\ \Phi_N[\Pi_N](x,t) \end{pmatrix} \tag{72}$$

where $\Pi_i \neq \Pi_j$ is possible

We now generalize the Lubich's time-stepping propagation algorithm (TVT propagation) to the case of N energy surfaces. We first consider the propagation of the homogenous wavepacket. Since the kinetic operator $\mathbf{T}$ has a block diagonal form, there is no change for the propagation of half time step of the kinetic part (T). For the potential part, we need to split the full potential matrix $\mathbf{V}(x)$ into quadratic part $\mathbf{U}(x)$ and remainder $\mathbf{W}(x)$, i.e.,

$$\mathbf{V}(x) = \begin{pmatrix} U_{\text{ref}}(x) & & \\ & \ddots & \\ & & U_{\text{ref}}(x) \end{pmatrix} + \begin{pmatrix} V_{11}(x) - U_{\text{ref}}(x) & \cdots & V_{1N}(x) \\ \vdots & \ddots & \vdots \\ V_{N,1}(x) & \vdots & V_{NN}(x) - U_{\text{ref}}(x) \end{pmatrix} \tag{73}$$

Here, $U_{\text{ref}}(x)$ is

$$U_{\text{ref}}(x) = \lambda_\chi(q) + (x-q)^T \cdot \nabla\lambda_\chi(q) + \frac{1}{2}(x-q)^T \cdot \nabla^2\lambda_\chi(q) \cdot (x-q) \tag{74}$$

where λ_χ is the χth energy level of the matrix potential $\mathbf{V}(x)$, $\chi \in [1,\cdots,N]$ (in the real calculation, we can choose any single χ). We now need to build the block matrix $\mathbf{F}$ used in the propagation of the coefficient $\{c_K^i\}_{K\in\mathfrak{K}_i}$ of all components $\Phi_i (i = 1,\cdots,N)$. For this purpose, we stack the coefficients $\{c_K^i\}_{K\in\mathfrak{K}_i}$ into a long column vector $\mathbf{c}$,

$$\mathbf{c} = \begin{pmatrix} \cdots & c_K^1 & \cdots & | & \cdots & | & \cdots & c_K^N & \cdots \end{pmatrix}^T \tag{75}$$

of length $\sum_{i=1}^{N} |\mathfrak{R}_i|$ or $N|\mathfrak{R}|$ if all components have a basis shape of same size. Then the block matrix $\mathbf{F}$ can be constructed as follows

$$\mathbf{F} := \begin{pmatrix} \mathbf{F_{1,1}} & \cdots & \mathbf{F_{1,N}} \\ \vdots & \mathbf{F_{i,j}} & \vdots \\ \mathbf{F_{N,1}} & \cdots & \mathbf{F_{N,N}} \end{pmatrix} \tag{76}$$

where each block is of the form

$$\mathbf{F_{i,j}} := \begin{pmatrix} & \vdots & \\ \cdots & \langle \phi_K | \mathbf{W}_{i,j} | \phi_L \rangle & \cdots \\ & \vdots & \end{pmatrix} \tag{77}$$

for $K \in \mathfrak{R}_i$ and $L \in \mathfrak{R}_j$. Finally we can formulate following time-stepping algorithm for the propagation of the homogeneous vectorial wavepackets as follows: (1) kinetic propagation $U_T(\Delta t/2)$; (2) potential propagation $U_{\lambda_\chi}(\Delta t)$; (3) Hagedorn coefficients propagation $\mathbf{c} = \exp(-\Delta t \frac{i}{\varepsilon}\mathbf{F})\mathbf{c}$; (4) kinetic propagation $U_T(\Delta t/2)$

The propagation of the inhomogenous wavepacket is very similar to that of homogenous wavepacket. The only difference is that an inhomogeneous wavepacket Ψ consists of N components Φ_i with its own parameter set Π_i. In order to take this fact into account, we split the potential matrix $\mathbf{V}(x)$ into quadratic part $\mathbf{U}(x)$ and remainder $\mathbf{W}(x)$, i.e.,

$$\mathbf{V}(x) = \begin{pmatrix} U_1(x) & & \\ & \ddots & \\ & & U_N(x) \end{pmatrix} + \begin{pmatrix} V_{11}(x) - U_1(x) & \cdots & V_{1N}(x) \\ \vdots & \ddots & \vdots \\ V_{N,1}(x) & \vdots & V_{NN}(x) - U_N(x) \end{pmatrix} \tag{78}$$

Here, $U_i(x) (i = 1, \cdots, N)$ is the local harmonic approximation of the ith energy level $\lambda_i(x) (i = 1, \cdots, N)$, i.e.,

$$U_i(x) = \lambda_i(q_i) + (x - q_i)^T \cdot \nabla\lambda_i(q_i) + \frac{1}{2}(x - q_i)^T \cdot \nabla^2\lambda_i(q_i) \cdot (x - q_i), \quad i = (1, \cdots, N) \tag{79}$$

We then use the non-quadratic remainder $\mathbf{W}(x)$ to compute the block matrix $\mathbf{F}$.

$$\mathbf{F} := \begin{pmatrix} \mathbf{F_{1,1}} & \cdots & \mathbf{F_{1,N}} \\ \vdots & \mathbf{F_{i,j}} & \vdots \\ \mathbf{F_{N,1}} & \cdots & \mathbf{F_{N,N}} \end{pmatrix} \tag{80}$$

where each $\mathbf{F}_{i,j}$ is of the form

$$\mathbf{F}_{i,j} = \begin{pmatrix} & \cdots & \\ \cdots & \langle \phi_K[\Pi_i] | \mathbf{W}_{i,j} | \phi_L[\Pi_j] \rangle & \cdots \\ & \vdots & \end{pmatrix} \tag{81}$$

for $K \in \mathfrak{R}_i$ and $L \in \mathfrak{R}_j$. It is noted that here we explicitly consider the fact that ϕ_K and ϕ_L have different parameter sets Π_i and Π_j. Finally we formulate following time-stepping algorithm for the propagation of the inhomogenous wavepackets as follows: (1) kinetic propagation $U_{T_i}(\Delta t)$, $i = 1, \cdots, N$; (2) potential propagation $U_{\lambda_i}(\Delta t/2)$, $i = 1, \cdots, N$; (3) Hagedorn coefficients propagation $\mathbf{c} = \exp(-\Delta t \frac{i}{\varepsilon}\mathbf{F})\mathbf{c}$; (4) kinetic propagation $U_{T_i}(\Delta t/2)$, $i = 1, \cdots, N$

Now let us focus on the computation of observables with the vectorized Hagedorn wavepackets. First let us calculate the norm of a vectorized Hagedorn wavepackets $|\Psi\rangle$. We have

$$||\Psi||_{L^2}^2 = \langle\Psi|\Psi\rangle = \sum_{i=1}^{N}\langle\Phi_i|\Phi_i\rangle \tag{82}$$

The squared norm of the vector valued wavepacket is simply the sum of the squared norms of its components, where the latter can be easily calculated by Eq. 60.

The next step is to calculate the energies of a vectorized wavepacket, where we can split the total energy of $|\Psi\rangle$ ($E_{\text{total}} = \langle\Psi|\mathbf{H}|\Psi\rangle$) into the kinetic energy and the potential energy

$$E_{\text{total}} = \langle\Psi|\mathbf{H}|\Psi\rangle = \langle\Psi|\mathbf{T}|\Psi\rangle + \langle\Psi|\mathbf{V}(x)|\Psi\rangle = E_{\text{kinetic}} + E_{\text{potential}}. \tag{83}$$

where the kinetic energy is given as

$$E_{\text{kinetic}} = \langle\Psi|\mathbf{T}|\Psi\rangle = \left\langle \begin{pmatrix}\Phi_1(x)\\ \vdots \\ \Phi_N(x)\end{pmatrix} \middle| \begin{pmatrix} T & & 0 \\ & \ddots & \\ 0 & & T\end{pmatrix} \middle| \begin{pmatrix}\Phi_1(x)\\ \vdots \\ \Phi_N(x)\end{pmatrix} \right\rangle = \sum_{i=1}^{N} \Phi_i(x)|T|\Phi_i(x)\rangle \tag{84}$$

i.e., the kinetic energy of a vectorized Hagedorn wavepacket is the sum of the kinetic energy of each component (see Eq. 63). The potential energy of a vectorized wavepacket is written as

$$\begin{aligned} E_{\text{potential}} &= \langle\Psi|\mathbf{V}(x)|\Psi\rangle \\ &= \left\langle \begin{pmatrix}\Phi_1(x)\\ \vdots \\ \Phi_N(x)\end{pmatrix} \middle| \begin{pmatrix} v_{11}(x) & \cdots & v_{1N}(x) \\ \vdots & & \vdots \\ v_{N1}(x) & \cdots & v_{NN}(x)\end{pmatrix} \middle| \begin{pmatrix}\Phi_1(x)\\ \vdots \\ \Phi_N(x)\end{pmatrix} \right\rangle \\ &= \sum_{i=1}^{N}\sum_{j=1}^{N}\langle\Phi_i(x)|v_{ij}(x)|\Phi_j(x)\rangle \end{aligned} \tag{85}$$

i.e., the potential energy of the vectorized wavepacket $|\Psi\rangle$ can be expressed as a sum of potential energies of its components Φ_i (see Eq. 41).

Finally we give the expression for the calculation of the auto-correlation function of a vectorized Hagedorn wavepacket

$$\begin{aligned} \langle\Psi(0)|\Psi(t)\rangle &= \left\langle \begin{pmatrix}\Phi_1(0)\\ \vdots \\ \Phi_N(0)\end{pmatrix} \middle| \begin{pmatrix}\Phi_1(t)\\ \vdots \\ \Phi_N(t)\end{pmatrix} \right\rangle \\ &= \sum_{i=1}^{N}\langle\Phi_i(0)|\Phi_i(t)\rangle \end{aligned} \tag{86}$$

i.e., the auto-correlation function of the vectorized wavepacket $|\Psi\rangle$ is the sum of auto-correlation functions of its components (see Eq. 61).

In the last, let us describe the basis transformation of vectorized wavepacket $|\Psi\rangle$ (our previous discussions are based on the diabatic representation). Suppose that the basis transformation of our wavepacket $|\Psi\rangle$ from and to the diabatic representation is written as

$$\begin{aligned}|\Psi_{\text{diabatic}}\rangle &= \mathbf{M}(x)|\Psi_{\text{adiabatic}}\rangle \\ |\Psi_{\text{adiabatic}}\rangle = \mathbf{M}^{-1}(x)|\Psi_{\text{diabatic}}\rangle &= \mathbf{M}^{H}(x)|\Psi_{\text{diabatic}}\rangle\end{aligned} \tag{87}$$

Explicitly,

$$\begin{aligned}\mathbf{M}(x)|\Psi_{\text{adiabatic}}(x)\rangle &= \begin{pmatrix} m_{11}(x) & \cdots & m_{1N}(x) \\ \vdots & & \vdots \\ m_{N1}(x) & \cdots & m_{NN}(x)\end{pmatrix}\begin{pmatrix}\Phi_1(x) \\ \vdots \\ \Phi_N(x)\end{pmatrix} \\ &= \begin{pmatrix} m_{11}(x)\Phi_1(x) + \cdots + m_{1N}(x)\Phi_N(x) \\ \vdots \\ m_{N1}(x)\Phi_1(x) + \cdots + m_{NN}(x)\Phi_N(x)\end{pmatrix} \\ &= \begin{pmatrix}\Phi_1^{'}(x) \\ \vdots \\ \Phi_N^{'}(x)\end{pmatrix} = |\Psi_{\text{diabatic}}^{'}\rangle\end{aligned} \tag{88}$$

where $\mathbf{M}(x)$ is the unitary transformation matrix which is composed of the eigenvectors obtained from the diagonalization of the matrix potential $\mathbf{V}(x)$.

2. Technical details for the implementation

1. The multi-index ($\underline{k} := (k_1, \cdots, k_D) \in \mathbb{N}_0^D$)for the basis shape $\mathfrak{K}$ (see types.hpp and multi_index.hpp)

```
//define the Tiny Multi-Index (wrapper of the array with element type int) (defined in types.hpp)
typedef std::size_t dim_t;
template<dim_t D> using TinyMultiIndex=std::array<int, D>;

/* provides less functor (compare) for Standard Template Library containers (notable std::map)
*  Specializes generic std::less<T>
*/
template<dim_t D>
class less< TinyMultiIndex<D> >
{
private:
   typedef TinyMultiIndex<D> MultiIndex;

public:
    typedef MultiIndex first_argument_type;
    typedef MultiIndex second_argument_type;
    typedef bool result_type;

    bool operator()( MultiIndex const& first,  MultiIndex const& second) const
    {
        return lexicographical_compare(first.begin(), first.end(), second.begin(), second.end());
    }
};

/* provides hash functor for Standard Template Library containers (notable std::unordered_map)
*  Specializes generic std::hash<T>.
*/
template<dim_t D>
class hash< TinyMultiIndex<D> >
{
private:
    typedef TinyMultiIndex<D> MultiIndex;

    std::string  to_str( MultiIndex const& index) const
    {
        std::stringstream ss;
        for(auto ii: index)
             ss<<ii;
         return ss.str();
     }
 public:

    std::size_t operator()(MultiIndex const& index) const
    {
    return std::hash<std::string>()(to_str(index));
  }
};

/* Provides equality functor for Standard Template Library containers (notable std::unordered_map)
*  Specializes generic std::equal_to<T>.
*/
template<dim_t D>
class equal_to< TinyMultiIndex<D> >
{
private:
   typedef TinyMultiIndex<D> MultiIndex;

    std::string  to_str( MultiIndex const& index) const
    {
        std::stringstream ss;
        for(auto ii: index)
            ss<<ii;
    return ss.str();
    }

public:
    typedef MultiIndex first_argument_type;
    typedef MultiIndex second_argument_type;
    typedef bool result_type;

    bool operator()(MultiIndex const& first, MultiIndex const& second) const
    {
        std::string first_str=to_str(first);
        std::string second_str=to_str(second);
    return first_str==second_str;
    }
};
```

2 class ContinuousSqrt: This class calculates the square root of the det**Q** ($\sqrt{\det \mathbf{Q}}$), see **continuous_sqrt.hpp**

```
/**
* This class deals with the issue, that the square root of complex numbers is not unique.
* The equation $z^2 = r\exp(i\phi)$ has two solutions, namely
* $z_1 = \sqrt{r}\exp\left(i\frac{\phi}{2}\right)$ and $z_2 = \sqrt{r}\exp\left(i(\frac{\phi}{2}+\pi)\right)$.
* This class chooses the solution, that is nearest to the solution of the previous computation (=
    reference solution). Then this class overrides the stored reference solution with the current
    solution. The distance between the two complex numbers is determined by the angle-distance.
* param T Type of both the real and imaginary components of the complex number.
*/
template<class T> class ContinuousSqrt
{
private:
     std::complex<T> sqrt_;  //stored reference solution
     T state_;   //(angle) of reference solution
     bool empty_; //false if a reference solution is stored

public:
  /**
  * Delays initialization of the stored reference solution.
  * The next call to operator()() yields the principal square root.
  */
  ContinuousSqrt()

  /**
  * Initializes the stored reference solution to a chosen value.
  * param sqrt The initial reference solution.
  */
  ContinuousSqrt(std::complex<T> sqrt)

  /**
  * Chooses the square root angle (argument) that continuates the reference angle the best.
  * Throws an exception if the deviation above an accepted value (by default > π/4).
  * param[in] ref The angle of the reference square root. domain = [−π; π]
  * param[in] arg The angle of the computed square root. domain = [−π; π]
  * return The angle of the continuating square root. domain = [−π; π]
  */
  static T continuate(T ref, T arg)

  /**
  * Solves the quadratic equation $z^2 = c$. Chooses the solution $\hat{z}$ that best continuates the prior
  * result $z_0$ and updates the reference solution ($z_0 \leftarrow \hat{z}$).
  * param input The right—hand—side $c$.
  * return The best solution $\hat{z}$.
  */
  std::complex<T> operator()(std::complex<T> input)

  //Retrieve the stored reference solution.
  std::complex<T> operator()() const

  /**
  * getter for state state
  * return state_[—pi,pi]
  */
  T get_state(void) const
};
```

3 The basis shapes $\mathfrak{R}$, we have implemented Hypercubic basis shape (class HyperCubicShape), Hyperbolic cut basis shape (class HyperbolicCutShape), Hyperbolic cut basis shape with limits (class LimitedHyperbolicCutShape), Simplex basis shape (class SimplexShape), see shape.hpp for details.

```
/**
* Subclasses provide a description of a basis shape.
* A $D$—dimensional basis shape $\mathfrak{K}$ is a set of $D$ dimensional integer tuples (aka node).
* Subclasses provide an description of a basis shape $\mathfrak{K} \subset \mathbb{N}_0^D$.
* It describes, which nodes $\underline{k} \in \mathbb{N}_0^D$ are part of the shape.
* Keep in mind, that basis shapes must fulfill the fundamental property
* $\underline{k} \in \mathfrak{K} \Rightarrow \forall \underline{k} - \underline{e}^d \in \mathfrak{K} \, \forall d \in \{d \mid k_d \geq 1\}$
* where $\underline{e}^d$ is the unit vector in direction $d$. That means, if an arbitrary node is part of the basis
   shape, then all nodes in the backward cone are part of the shape too.
*param D basis shape dimensionality
*/
template<dim_t D>
class AbstractShape
{
public:
  virtual ~AbstractShape()   //virtual destructor
  //get the backward neighbours of multi_index index
  std::array<TinyMultiIndex<D>,D>
  get_backward_neighbours(const TinyMultiIndex<D> &index) const
  //get the forward neighbours of multi_index index
  std::array<TinyMultiIndex<D>,D>
  get_forward_neighbours(const TinyMultiIndex<D> &index) const

  /**
  * Retrieves the length of the minimum bounding box in one direction.
  * The minimum bounding box is given by $L_\alpha = \max_{k_\alpha} \{\underline{k} \in \mathfrak{K}\}$
  * param axis The direction $\alpha$.
  * return Length of the bbox.
  */
  virtual int bbox(dim_t axis) const = 0;
  /**
  * Evaluates the limit of the direction $\alpha$ given a base node, which is defined by
  * $l_\alpha(\underline{n}) = \max_{k_\alpha} \{\underline{k} \in \mathfrak{K} \mid k_d = n_d \; \forall d \neq \alpha\}$
  * Notice that the $\alpha$-th entry of $\underline{n}$ does not influence return value.
  * It can be of any value since it is simply ignored.
  * param base_node The basis node $\underline{n}$. It contains D indices.
  * param axis The direction $\alpha$.
  * return the limit in direction axis
  */
  virtual int limit(int const* base_node, dim_t axis) const = 0;
  /**
  * Prints a pretty description of the shape.
  * param out The output stream.
  */
  virtual void print(std::ostream & out) const = 0;
};

//overridding the operator << used for the output stream
template<dim_t D>
std::ostream & operator<<(std::ostream & out, AbstractShape<D> const& shape)

/**
* This class implements the hyperbolic cut shape.
* This class implements the hyperbolic cut basis shape, which is a special type of a sparse basis
   shape.
* The hyperbolic cut shape in $D$ dimensions with sparsity $S$  is defined as the set
* $\mathfrak{K}(D,S) := \left\{ (k_1,\dots,k_D) \in \mathbb{N}_0^D \mid \prod_{d=1}^{D} (1+k_d) \leq S \right\}$
* param D basis shape dimensionality
*/
template<dim_t D>
class HyperbolicCutShape : public AbstractShape<D>
{
private:
  int S_;   // the sparsity parameter
  std::map<TinyMultiIndex<D>, int> lima_;        //linear map: MultiIndex—> int
  std::map<int,TinyMultiIndex<D>> lima_inv_;    //inverse linear map: int—>MultiIndex
  std::size_t basis_size_;                                //number of basis function
  //rearrange the multi-index into slices where each slice has the same value for the sum of the
    multi-index
  std::vector<std::vector<TinyMultiIndex<D>>> slices_;
public:

  HyperbolicCutShape() = default;  // default constructor
  //constructor, param S;  set the value of  lima_, lima_inv_, slices_, basis_size_
  HyperbolicCutShape(int S) : S_(S)
  HyperbolicCutShape(const HyperbolicCutShape& that)  //copy constructor
  HyperbolicCutShape(HyperbolicCutShape&& that)   //move copy constructor
  HyperbolicCutShape &operator=(const HyperbolicCutShape& that)   //assignment operator
  HyperbolicCutShape &operator=(HyperbolicCutShape&& that)  //move assignment operator
  int& get_item(const TinyMultiIndex<D> &index)  //Given the MultiIndex, get the corresponding
    linear mapping.
  const int& get_item(const TinyMultiIndex<D> &index) const  //same as above, const version
  TinyMultiIndex<D>& get_item(const int &kk)  //Given mapped int value, get the corresponding
    MultiIndex
  const TinyMultiIndex<D>& get_item(const int &kk) const //same as above, const version
  bool contains(const TinyMultiIndex<D> &index) const  //check if a given multi-index is part of
    the basis set
  HyperbolicCutShape extend() const   //return the extended basis shape
  //construct the linear mapping
  std::tuple<std::map<TinyMultiIndex<D>,int>, std::map<int,TinyMultiIndex<D>>, std::vector<std::
    vector<TinyMultiIndex<D>>>> get_index_lex()
  std::map<TinyMultiIndex<D>, int>& get_lima() //return the lima_
  const std::map<TinyMultiIndex<D>, int>& get_lima() const //same as above, const version
  std::map<int,TinyMultiIndex<D>>& get_lima_inv()  //return the lima_inv_
  const std::map<int,TinyMultiIndex<D>>& get_lima_inv() const //same as above, const version
  std::vector<std::vector<TinyMultiIndex<D>>>& get_slices() //return slices_
  const std::vector<std::vector<TinyMultiIndex<D>>>& get_slices() const //same as above, const
    version
  std::size_t size() const  //return basis size: basis_size_
  int sparsity() const      //return sparsity S_
  virtual int bbox(dim_t axis) const override //override function bbox of super class
  virtual int limit(int const* base_node, dim_t axis) const override //override function limit of
    super class
  virtual void print(std::ostream & out) const override //override function print of super class
};

/**
* This class implements the limited hyperbolic cut shape.
* This class implements the limited hyperbolic cut basis shape which is a special type of a sparse
    basis shape.
* The limited hyperbolic cut shape in D dimensions with sparsity S and limits K = (K_1,...,K_D) is
    defined as the set
```

$$* \ \mathfrak{K}(D,S,\boldsymbol{K}) := \left\{ (k_1,\dots,k_D) \in \mathbb{N}_0^D \mid 0 \le k_d < K_d \ \wedge \prod_{d=1}^{D} (1+k_d) \le S \right\}$$

```
* It is an intersection of the hyperbolic cut shape with a hypercubic shape.
* param D basis shape dimensionality
```

```
*/
template<dim_t D>
class LimitedHyperbolicCutShape : public AbstractShape<D>
{
private:
  int S_;                       // the sparsity
  std::array<int,D> limits_;  //  the limits for each dimension
  std::map<TinyMultiIndex<D>, int> lima_;         //linear map: MultiIndex—> int
  std::map<int,TinyMultiIndex<D>> lima_inv_;    //inverse linear map: int—>MultiIndex
  std::size_t basis_size_;     //number of basis function
   //rearrange the multi—index into slices where each slice has the same value for the sum of the
     multi—index
  std::vector<std::vector<TinyMultiIndex<D>>> slices_;
public:
  LimitedHyperbolicCutShape()=default;   // default constructor
  //constructor, param S; param limits; calculate lima_, lima_inv_, basis_size_, slices_
  LimitedHyperbolicCutShape(int S, const std::array<int,D> &limits)
  //constructor, param S; param size; calculate lima_, lima_inv_, basis_size_, slices_
  LimitedHyperbolicCutShape(int S, int size)
  //constructor, param S; param list; calculate lima_, lima_inv_, basis_size_, slices_
  LimitedHyperbolicCutShape(int S, std::initializer_list<int> list)
  LimitedHyperbolicCutShape(const LimitedHyperbolicCutShape& that) // copy constructor
  LimitedHyperbolicCutShape(LimitedHyperbolicCutShape&& that) //move copy constructor
  LimitedHyperbolicCutShape &operator=(const LimitedHyperbolicCutShape& that) //assignment operator
  LimitedHyperbolicCutShape &operator=(LimitedHyperbolicCutShape&& that) //move assignment operator
  int& get_item(const TinyMultiIndex<D> &index) //Given the MultiIndex, get the corresponding
    linear mapping.
  const int& get_item(const TinyMultiIndex<D> &index) const //same as above, const version
  TinyMultiIndex<D>& get_item(const int &kk)  //Given mapped int value, get the corresponding
    MultiIndex
  const TinyMultiIndex<D>& get_item(const int &kk) const  //same as above, const version
  bool contains(const TinyMultiIndex<D> &index) const  //check if a given multi—index is part of
    the basis set
  LimitedHyperbolicCutShape extend() const  //obtain the extended basis shape
  //get the linear mapping
  std::tuple<std::map<TinyMultiIndex<D>,int>, std::map<int,TinyMultiIndex<D>>, std::vector<std::
    vector<TinyMultiIndex<D>>>> get_index_lex()
  std::map<TinyMultiIndex<D>, int>& get_lima() //return lima_
  const std::map<TinyMultiIndex<D>, int>& get_lima() const //same as above, const version
  std::map<int,TinyMultiIndex<D>>& get_lima_inv() //return lima_inv_
  const std::map<int,TinyMultiIndex<D>>& get_lima_inv() const //same as above, const version
  std::vector<std::vector<TinyMultiIndex<D>>>& get_slices() //return slices_
  const std::vector<std::vector<TinyMultiIndex<D>>>& get_slices() const //same as above, const
    version
  std::size_t size() const //return basis size: basis_size_
  int sparsity() const  // return sparsity: S_
  std::array<int,D>& get_limits() //return limit: limits_
  const std::array<int,D>& get_limits() const //same as above, const version
  virtual int bbox(dim_t axis) const override //override function bbox of super class
  virtual int limit(int const* base_node, dim_t axis) const override //override function limit of
    super class
  virtual void print(std::ostream & out) const override  //override function print of super class
};

/**
* This class implements the hypercubic basis shape.
* A $D$-dimensional hypercubic shape with limits $\mathbf{K} = \{K_1, \dots, K_D\}$ is defined as the set
* $\mathfrak{K}(D,\mathbf{K}) := \left\{(k_1, \dots, k_D) \in \mathbb{N}_0^D | k_d < K_d \forall d\right\}$
* param D basis shape dimensionality
*/
template<dim_t D>
class HyperCubicShape : public AbstractShape<D>
{
private:
  std::array<int,D> limits_;                                  // limits
  std::map<TinyMultiIndex<D>, int> lima_;        //linear map: MultiIndex—> int
  std::map<int,TinyMultiIndex<D>> lima_inv_;     //inverse linear map: int—>MultiIndex
  std::size_t basis_size_;     //number of basis function
  //rearrange the multi—index into slices where each slice has the same value of the sum of the
    multi—index
  std::vector<std::vector<TinyMultiIndex<D>>> slices_;
public:
  HyperCubicShape()=default;   //default constructor
  //constructor, param limits; calculate lima_, lima_inv_, basis_size_, slices_
  HyperCubicShape(const std::array<int,D> &limits)
  //constructor, param limit; calculate lima_, lima_inv_, basis_size_, slices_
  HyperCubicShape(int limit)
  //constructor, param list; calculate lima_, lima_inv_, basis_size_, slices_
  HyperCubicShape(std::initializer_list<int> list)
  HyperCubicShape(const HyperCubicShape &that) //copy constructor
  HyperCubicShape(HyperCubicShape&& that) //move copy constructor
  HyperCubicShape &operator=(const HyperCubicShape &that) //assignment operator
  HyperCubicShape &operator=(HyperCubicShape&& that) //copy assignment operator
  int& get_item(const TinyMultiIndex<D> &index)  //Given the MultiIndex, get the corresponding
    linear mapping.
  const int& get_item(const TinyMultiIndex<D> &index) const //same as above, const version
  TinyMultiIndex<D>& get_item(const int &kk) //Given mapped int value, get the corresponding
    MultiIndex
  const TinyMultiIndex<D>& get_item(const int &kk) const //same as above, const version
  bool contains(const TinyMultiIndex<D> &index) const //check if a given multi—index is part of the
     basis set
  HyperCubicShape extend() const //obtain the extended basis shape
  //calculate the linear mapping
  std::tuple<std::map<TinyMultiIndex<D>,int>, std::map<int,TinyMultiIndex<D>>, std::vector<std::
    vector<TinyMultiIndex<D>>>> get_index_lex()
  std::map<TinyMultiIndex<D>, int>& get_lima() //return lima_
  const std::map<TinyMultiIndex<D>, int>& get_lima() const //same as above, const version
  std::map<int,TinyMultiIndex<D>>& get_lima_inv() // return lima_inv_
  const std::map<int,TinyMultiIndex<D>>& get_lima_inv() const  //same as above, const version
  std::vector<std::vector<TinyMultiIndex<D>>>& get_slices() //return slices_
  const std::vector<std::vector<TinyMultiIndex<D>>>& get_slices() const //same as above, const
    version
  std::size_t size() const  //return basis size: basis_size_
  std::array<int,D>& get_limits() //return limits_
  const std::array<int,D>& get_limits() const //same as above, const version
  virtual int limit(int const* base_node, dim_t axis) const override //override function limit of
    the super class
  virtual int bbox(dim_t axis) const override //override function bbox of the super class
  virtual void print(std::ostream & out) const override  //override function print of the super
    class
};

/**
* This class implements the simplex basis shape .
* A $D$-dimensional simplex shape with maximal 1-norm $K$ is defined as the set
* $\mathfrak{K}(D,\mathbf{K}) := \left\{(k_1,\dots,k_D) \in \mathbb{N}_0^D \,|\, \sum_{d=1}^{D} k_d <= K\right\}$
* param D basis shape dimensionality
*/
template<dim_t D>
class SimplexShape : public AbstractShape<D>
{
private:
  int K_;         // maximal 1-norm K
  std::map<TinyMultiIndex<D>, int> lima_;         //linear map: MultiIndex--> int
  std::map<int,TinyMultiIndex<D>> lima_inv_;     //inverse linear map: int-->MultiIndex
  std::size_t basis_size_;      //number of basis function
  //rearrange the multi-index into slices where each slice has the same value for the sum of the
    multi-index
  std::vector<std::vector<TinyMultiIndex<D>>> slices_;
public:
  SimplexShape()=default;    //default constructor
  //constructor, param K; calculate lima_, lima_inv_, basis_size_, slices_
  SimplexShape(int K)
  SimplexShape(const SimplexShape& that) //copy constructor
  SimplexShape(SimplexShape&& that) //move copy constructor
  SimplexShape &operator=(const SimplexShape& that) //assignment operator
  SimplexShape &operator=(SimplexShape&& that) //move assignment operator
  int& get_item(const TinyMultiIndex<D> &index) //Given the MultiIndex, get the corresponding
    linear mapping.
  const int& get_item(const TinyMultiIndex<D> &index) const //same as above, const version
  TinyMultiIndex<D>& get_item(const int &kk) //Given mapped int value, get the corresponding
```

```
    MultiIndex
  const TinyMultiIndex<D>& get_item(const int &kk) const //same as above, const version
  bool contains(const TinyMultiIndex<D> &index) const // check if a given multi-index is part of
    the basis set
  SimplexShape extend() const //obtain the extended basis shape
  //calculate the linear mapping
  std::tuple<std::map<TinyMultiIndex<D>,int>, std::map<int,TinyMultiIndex<D>>, std::vector<std::
    vector<TinyMultiIndex<D>>>> get_index_lex()
  std::map<TinyMultiIndex<D>, int>& get_lima() //return lima_
  const std::map<TinyMultiIndex<D>, int>& get_lima() const //same as above, const version
  std::map<int,TinyMultiIndex<D>>& get_lima_inv() //return lima_inv_
  const std::map<int,TinyMultiIndex<D>>& get_lima_inv() const //same as above, const version
  std::vector<std::vector<TinyMultiIndex<D>>>& get_slices() //return slices_
  const std::vector<std::vector<TinyMultiIndex<D>>>& get_slices() const //same as above, const
    version
  std::size_t size() const  //return the basis size: basis_size_
  int max_norm() const //return K_
  virtual int bbox(dim_t axis) const override //override function bbox for the super class
  virtual int limit(int const* base_node, dim_t axis) const override //override function limit of
    the super class
  virtual void print(std::ostream & out) const override   //override function print of the super
    class
};
```

4 The class HaWpParamSet represents the Hagedorn parameter set $\Pi = \{\underline{q}, \underline{p}, \mathbf{Q}, \mathbf{P}, S\}$, see **hawp_paramset.hpp** for details

```
/**
* This class represents the Hagedorn parameter set Π = {q, p, Q, P, S}.
* The first two parameters q and p are D dimensional real-valued vectors. The second two Q and P
    are complex D × D matrices. The last parameter S is the global complex phase.
*/
template<dim_t D>
struct HaWpParamSet
{
private:
  RMatrix<D,1> q_, p_;    //q and p,
  CMatrix<D,D> Q_, P_;    //Q and P
  complex_t S_;           //global phase S
  math::ContinuousSqrt<real_t> sqrt_detQ_;  //√detQ, see  continuous_sqrt.hpp
public:
  HaWpParamSet() //default constructor
  //constructor, set {q, p, Q, P, S}
  HaWpParamSet(const RMatrix<D,1> &q, const RMatrix<D,1> &p, const CMatrix<D,D> &Q, const CMatrix<D
    ,D> &P, const complex_t &S)
  //constructor, set {q, p, Q, P, S} and √detQ
  HaWpParamSet(const RMatrix<D,1> &q, const RMatrix<D,1> &p, const CMatrix<D,D> &Q, const CMatrix<D
    ,D> &P, const complex_t &S,     math::ContinuousSqrt<real_t> sqrt_detQ)
  HaWpParamSet(const HaWpParamSet &that) //copy constructor
  HaWpParamSet &operator=(const HaWpParamSet &that) // assignment operator
  inline RMatrix<D,1> const& q() const   //return q: q_
  inline RMatrix<D,1> const& p() const   //return p: p_
  inline CMatrix<D,D> const& Q() const  //return Q: Q_
  inline CMatrix<D,D> const& P() const  //return P: P_
  inline complex_t const& S() const      //return S: S_
  inline complex_t /*const*/ sdQ() const  //return √detQ
  inline real_t /*const*/ state() const   //return state of √detQ
  inline void q(const RMatrix<D,1>& qnew)     //set q to qnew
  inline void p(const RMatrix<D,1>& pnew)    // set p to pnew
  inline void Q(const CMatrix<D,D>& Qnew)  // set Q to Qnew
  inline void P(const CMatrix<D,D>& Pnew)  // set P to Pnew
  inline void S(const complex_t& Snew)       // set S to Snew
  inline void updateq(const RMatrix<D,1>& qnew)  //update q: q_ += qnew
  inline void updatep(const RMatrix<D,1>& pnew)  //update p: p_ += pnew
  inline void updateQ(const CMatrix<D,D>& Qnew) //update Q: Q_+= Qnew
  inline void updateP(const CMatrix<D,D>& Pnew)  //update P: P_+=Pnew
  inline void updateS(const complex_t& Snew)       //update S: S_+=Snew
  inline void resync()    //compute the continuous square root of detQ after an update of the Q
    parameter
  bool compatible() const   //check the compatibility relations
  //calculate the mixing of the two parameter set Π_bra and Π_ket
  std::pair< RMatrix<D,1>, RMatrix<D,D> > mix(const HaWpParamSet<D>& ket) const
};

//providing the pretty print of the parameter set
template<dim_t D>
std::ostream &operator<<(std::ostream &out, const HaWpParamSet<D> &parameters)
```

5 The class ScalarHaWp and class VectorHaWp represent the scalar Hagedorn wavepacket of Eq. 25 and the vectorized Hagedorn wavepacket of Eqs. 71 and 72 (**It should be noted that we combine both the homogenous and inhomogenous Hagedorn wavepacket together. The only difference between the homogenous and inhomogenous Hagedorn wavepacket is the way the wavepacket propagate**) (see **hawp_commons.hpp** for details).

```
// Implementation of a scalar Hagedorn wavepacket Φ(x)
template<dim_t D, class Shape>
class ScalarHaWp
{
private:
  // the semiclassical scaling parameter ε
  real_t eps_;
  //Hagedorn paramet set Π = {q, p, Q, P, S} , HaWpParamSet<D> is defined in hawp_paramset.hpp
  HaWpParamSet<D> parameters_;
  //coefficients of the Hagedorn wavepackets (c_k, k ∈ 𝔎), Coefficients is defined in types.hpp
  Coefficients coefficients_;
  Shape shape_;   //basis shape (see shape.hpp for various basis shapes)
  std::vector<real_t> sqrt_;//lookup-table for sqrt

public:
  /**
  * Evaluates all basis functions $\{\phi_{\underline{k}}\}$ on complex grid nodes $\underline{x} \in \gamma$.
  * param grid Complex grid nodes  quadrature points $\gamma$. Complex matrix with shape (dimensionality,
    number of grid nodes).
  * return Complex 2D-array with shape (basis shape size, number of grid nodes)
  * not including $1/\sqrt{\det \mathbf{Q}}$
  */
  template<int N> HaWpBasisVector<N> evaluate_basis(CMatrix<D,N> const& grid) const
  //Same as above, but for the real grid points: rgrid
  template<int N> HaWpBasisVector<N> evaluate_basis(RMatrix<D,N> const& rgrid) const
  /**
  *Evaluates this wavepacket $\Phi(\underline{x})$ at complex grid nodes $\underline{x} \in \gamma$.
  * Notice that this function does not include the prefactor $\frac{1}{\sqrt{\det(Q)}}$ nor the global phase $\exp(\frac{iS}{\varepsilon^2})$.
  * param grid Complex grid nodes: quadrature points $\gamma$. Complex matrix with shape (dimensionality,
    number of grid nodes).
  * return Complex matrix with shape (1, number of grid nodes)
  */
  template<int N> CArray<1,N> evaluate(CMatrix<D,N> const& grid) const
  //Same as above, but for the real grid points: rgrid
  template<int N> CArray<1,N> evaluate(RMatrix<D,N> const& rgrid) const
  //Calculate the gradient of the scalar Hagedorn wavepacket $y\Phi(\underline{x})$, return the new coefficients $\underline{c}'_{\underline{k}}$
  CMatrix<D, Eigen::Dynamic> apply_gradient() const
   real_t & eps()  //return the semiclassical scaling parameter $\varepsilon$
   real_t eps() const //same as above, const version
  HaWpParamSet<D> & parameters()  //return the Hagedorn parameter set $\Pi$
  HaWpParamSet<D> const& parameters() const  //same as above, const version
  Shape & shape()  //return the basis shape $\mathfrak{K}$
  Shape const& shape() const  //same as above, const version
   Coefficients & coefficients()  //return the expansion coefficients $c_{\underline{k}}$
   Coefficients const& coefficients() const //same as above, const version
   complex_t prefactor() const   // return the prefactor: $1/\sqrt{\det \mathbf{Q}}$
   complex_t phasefactor() const  //return the global phase factor: $\exp(\frac{iS}{\varepsilon^2})$
};

/**Represents a vectorized Hagedorn wavepacket $\Psi$ with $N$ components $\Phi_n$.
* Here we combine both the homogenous and inhomogenous Hagedorn wavepacket together
* The only difference between the homogenous and inhomogenous Hagedorn wavepacket is the way the
    wavepacket propagate
* The number of components is determined at runtime.
*param D wavepacket dimensionality
*param packets:  N component wavepackets, tuple of N component wavepackets
*/
template<dim_t D, class... packets>
```

```
class VectorHaWp
{
public:
  VectorHaWp() = default;  // default constructor
  VectorHaWp(real_t eps, const std::tuple<packets...> & components)   //Constructor
  VectorHaWp(const VectorHaWp& that)    //copy constructor
  VectorHaWp &operator=(const VectorHaWp& that)  //assignment operator

  // Grants access to the semi-classical scaling parameter ε of the wavepacket.
  real_t & eps()
  // same as above, const version
  real_t eps() const

  //Grants writeable access to all components {Φ_n} of this wavepacket.
  std::tuple<packets...>  & components()
  //same as above, const version
  std::tuple<packets...> const& components() const

  //Returns the number of components.
  std::size_t n_components() const

private:
  real_t eps_;       //the semiclassical scaling parameter
  std::tuple<packets...> components_;  //represent the Phi_n; n-th component of the vectorized
    Hagedorn wavepacket.
};

/**Represents a vectorized Hagedorn wavepacket Ψ with 2 components Φ_n.
* Here we combine both the homogenous and inhomogenous Hagedorn wavepacket together
* The only difference between the homogenous and inhomogenous Hagedorn wavepacket is the way the
    wavepacket propagate
* The number of components is 2
*param D  wavepacket dimensionality
*param ScalarPacket1:  component 1
*param ScalarPacket2:  component 2
*/

template<dim_t D, class ScalarPacket1, class ScalarPacket2>
class VectorHaWp2
{
public:
  VectorHaWp2() = default;  // default constructor
  VectorHaWp2(real_t eps, const std::tuple<ScalarPacket1, ScalarPacket2> & components) //
    Constructor
  VectorHaWp2(const VectorHaWp2& that) //copy constructor
  VectorHaWp2 &operator=(const VectorHaWp2& that)  //assignment operator

  /**Grants writeable access to the n-th component $\Phi_n$.
  *param  n The index $n$ of the requested component (n=2)
  *return Reference to the requested component.
  */
  decltype(auto)
  component(std::size_t nn)
  //same as above, const version
  decltype(auto)
  component(std::size_t nn)

  //Returns the number of coefficients for all component of the Hagedorn wavepacket (2 components).
  std::size_t size() const

  //Returns the offset vector for the components of the hagedorn wavepackets (2 components)
  std::vector<dim_t>  offset() const

  /** Evaluate the value of all components at once.
  * Evaluates $\Psi(x) = \{\Phi_i(x)\}$, where $x$ is is a complex quadrature point. Notice that this function does
     not include the prefactor
  * $\frac{1}{\sqrt{det(Q)}}$ nor the global phase $\exp(\frac{iS}{\varepsilon^2})$ for each component of the vector Hagedorn wavepacket
  * param grid  Complex quadrature points. Complex matrix of shape (dimensionality, number of
    quadrature points)
  * return Complex matrix of shape (number of components, number of quadrature points)
  * param N  Number of quadrature points.
  * Don't use Eigen::Dynamic. It works, but performance is bad.
  */
  template<int N>
  CArray<Eigen::Dynamic,N> evaluate(CMatrix<D,N> const& grid) const

  //same as above, version for the real quadrature points
  template<int N>
  CArray<Eigen::Dynamic,N> evaluate(RMatrix<D,N> const& rgrid) const

  //Grants access to the semi-classical scaling parameter $\varepsilon$ of the wavepacket.
  real_t & eps()
  //same as above, const version
  real_t eps() const

  //Grants writeable access to all components $\{\Phi_n\}$ of this wavepacket.
  std::tuple<ScalarPacket1, ScalarPacket2>  & components()
  //same as above, const version
  std::tuple<ScalarPacket1, ScalarPacket2> const& components() const

  //Returns the number of components.
  std::size_t n_components() const

private:
  real_t eps_;       //the semiclassical scaling parameter
  std::tuple<ScalarPacket1, ScalarPacket2> components_;  //represent the Φn; n-th component of the
    vectorized Hagedorn wavepacket.
};
```

6 All the classes implementing the quadrature rules are in the folder innerproducts. Specifically, **gauss_hermite_qr.hpp** provides **struct GaussHermiteQR** for representing the one-dimensional Gauss Hermite quadrature (weights and nodes), **genz_keister_qr.hpp** provides **struct GenzKeisterQR** for representing the Genz-Keister quadrature (weights and nodes), **tensor_product_qr.hpp** provides **struct TensorProductQR** for representing Tensor Product quadrature (weights and nodes), **homogeneous_inner_product.hpp** provides **class HomogeneousInnerProduct** for performing the integral with the same parameter set Π, **inhomogeneous_inner_product.hpp** provides **class InhomogeneousInnerProduct** for performing the integral with different parameter sets Π_r and Π_c, **vector_inner_product.hpp** provides **class VectorInnerProduct** for performing integral regarding the vectorized Hagedorn wavepacket

Some helper files are: **quadrature_rule.hpp** provides struct QuadratureRule for representing the nodes and weight values of a 1D quadrature rule (should only be used internally), **tables_gausshermite.hpp** provides array gauss_hermite_rules for Gauss-Hermite quadrature nodes and weights with different orders. **tables_genzkeister.hpp** provides Genz-Keister weighting factor and generator tables.

Basically this folder is what we need to extend for the future work, i.e., implementing more advanced quadrature rules for efficient calculation of the high-dimensional integral.

```
/** (in gauss_hermite_gr.hpp)
* Structure providing weighted nodes for Gauss Hermite quadrature.
* param ORDER requested order of the quadrature rule
*/
template <dim_t ORDER>
struct GaussHermiteQR
{
  static const dim_t D = 1;
  static const dim_t order = ORDER;
  using NodeMatrix = Eigen::Matrix<real_t,1,Eigen::Dynamic>;    // Node, sizes (1*|𝔑|), where |𝔑| is
    the number of nodes
  using WeightVector = Eigen::Matrix<real_t,1,Eigen::Dynamic>; // Weight, sizes (1*|𝔑|), where |𝔑| is
     the number of nodes

  static dim_t number_nodes() //Return the number of nodes for the given order.
  static NodeMatrix nodes() //Return the quadrature nodes.
  static WeightVector weights() //Return the quadrature weights.
  static std::tuple<NodeMatrix,WeightVector> nodes_and_weights() //Return the quadrature nodes and
    weights.
};

/** (in genz_keister_qr.hpp)
* Structure providing weighted nodes for Genz-Keister quadrature.
* param DIM dimensionality of the Genz-Keister rule
```

```cpp
* param LEVEL the level of the Genz–Keister rule, must be between 1 and 30 inclusive
*/
template <dim_t DIM, dim_t LEVEL>
struct GenzKeisterQR
{
  static const dim_t D = DIM; //dimensionality of the Genz–Keister rule
  static const dim_t level = LEVEL; //the level of the Genz–Keister rule, must be between 1 and 30 inclusive
  using NodeMatrix = Eigen::Matrix<real_t,DIM,Eigen::Dynamic>;  //Node, sizes (DIM*|ℜ|), where |ℜ| is the number of nodes
  using WeightVector = Eigen::Matrix<real_t,1,Eigen::Dynamic>; //Weight, sizes (1*|ℜ|), where |ℜ| is the number of nodes

  static dim_t number_nodes() //Return the number of nodes for the given order.
  static NodeMatrix nodes()      // Return the quadrature nodes.
  static WeightVector weights() //Return the quadrature weights.
  static std::tuple<NodeMatrix,WeightVector> nodes_and_weights() //Return the quadrature nodes and weights.
};

/**
* Structure providing weighted nodes for Tensor Product quadrature.
* param RULES list of other quadrature rules to use as components of the tensor product.
*/
template <class... RULES>
struct TensorProductQR
{
  static const dim_t D = sizeof...(RULES);
  using NodeMatrix = Eigen::Matrix<real_t,D,Eigen::Dynamic>;
  using WeightVector = Eigen::Matrix<real_t,1,Eigen::Dynamic>;

  /**
  * Return the number of nodes for the given order.
  * In the case of TensorProductQR, this is the product of the numbers of nodes of the constituent quadrature rules (RULES).
  */
  static dim_t number_nodes()
  static NodeMatrix nodes()  //Return the quadrature nodes.
  static WeightVector weights() //Return the quadrature weights.
  static std::tuple<NodeMatrix,WeightVector> nodes_and_weights() //Return the quadrature nodes and weights.
};

/**
* This class provides homogeneous inner product calculation of scalar wavepackets.
* param D dimensionality of processed wavepackets
* param QR quadrature rule to use, with |ℜ| nodes
```

```
*/
template<dim_t D, class QR>
class HomogeneousInnerProduct
{
public:
  using CMatrixXX = CMatrix<Eigen::Dynamic, Eigen::Dynamic>;
  using CMatrix1X = CMatrix<1, Eigen::Dynamic>;
  using CMatrixX1 = CMatrix<Eigen::Dynamic, 1>;
  using CMatrixD1 = CMatrix<D, 1>;
  using CMatrixDD = CMatrix<D, D>;
  using CMatrixDX = CMatrix<D, Eigen::Dynamic>;
  using RMatrixD1 = RMatrix<D, 1>;
  using CDiagonalXX = Eigen::DiagonalMatrix<complex_t, Eigen::Dynamic>;
  using NodeMatrix = typename QR::NodeMatrix;
  using WeightVector = typename QR::WeightVector;
  using op_t = std::function<CMatrix1X(CMatrixDX,RMatrixD1)>;

  /**
  * Calculate the matrix of the inner product.
  * Returns the matrix elements ⟨Φ|f|Φ⟩ with an operator f. The coefficients of the wavepacket are
    ignored.
  * param ScalarPacket: the type of the Scalar Hagedorn wavepacket
  * param[in] packet wavepacket Φ
  * param[in] op operator f(x,q) : ℂ^{D×R} × ℝ^D → ℂ^R which is evaluated at the nodal points x
    and position q; default functor default_op returns a vector of ones
  */
  template<class ScalarPacket>
  static CMatrixXX build_matrix(const ScalarPacket& packet, const op_t& op=default_op)

  /** Perform quadrature.
  * param ScalarPacket: the type of the Scalar Hagedorn wavepacket
  * Evaluates the scalar ⟨Φ|f|Φ⟩. See build_matrix() for the parameters.
  */
  template<class ScalarPacket>
  static complex_t quadrature(const ScalarPacket& packet, const op_t& op=default_op)

private:
  // The default functor returns a vector of ones
  static CMatrix1X default_op(const CMatrixDX& nodes, const RMatrixD1& pos)
};

/**
* This class provides inhomogeneous inner product calculation of scalar wavepackets.
* param D dimensionality of processed wavepackets
* param QR quadrature rule to use, with |ℜ| nodes
*/
template<dim_t D, class QR>
```

```
class InhomogeneousInnerProduct
{
public:
  using CMatrixXX = CMatrix<Eigen::Dynamic, Eigen::Dynamic>;
  using CMatrix1X = CMatrix<1, Eigen::Dynamic>;
  using CMatrixX1 = CMatrix<Eigen::Dynamic, 1>;
  using CMatrixD1 = CMatrix<D, 1>;
  using CMatrixDD = CMatrix<D, D>;
  using CMatrixDX = CMatrix<D, Eigen::Dynamic>;
  using RMatrixDD = RMatrix<D, D>;
  using RMatrixD1 = RMatrix<D, 1>;
  using CDiagonalXX = Eigen::DiagonalMatrix<complex_t, Eigen::Dynamic>;
  using NodeMatrix = typename QR::NodeMatrix;
  using WeightVector = typename QR::WeightVector;
  using op_t = std::function<CMatrix1X(CMatrixDX,RMatrixD1)>;

  /**
  * Calculate the matrix of the inner product.
  * Returns the matrix elements ⟨Φ|f|Φ'⟩ with an operator f.
  * The coefficients of the wavepackets are ignored.
  * param ScalarPacbra: the type of the bra of Scalar Hagedorn wavepacket
  * param ScalarPacket: the type of the ket of Scalar Hagedorn wavepacket
  * param[in] pacbra wavepacket Φ
  * param[in] packet wavepacket Φ'
  * param[in] op operator f(x,q) : C^{D×R} × R^D → C^R which is evaluated at the nodal points x
    and position q; The default default_op returns a vector of ones
  */
  template<class ScalarPacbra, class ScalarPacket>
  static CMatrixXX build_matrix(const ScalarPacbra& pacbra, const ScalarPacket& packet, const op_t&
     op=default_op)

  /**
  * Perform quadrature.
  * param ScalarPacbra: the type of the bra of Scalar Hagedorn wavepacket
  * param ScalarPacket: the type of the ket of Scalar Hagedorn wavepacket
  * Evaluates the scalar ⟨Φ|f|Φ'⟩. See build_matrix() for the parameters.
  */
  template<class ScalarPacbra, class ScalarPacket>
  static complex_t quadrature(const ScalarPacbra& pacbra, const ScalarPacket& packet, const op_t&
    op=default_op)

private:
  // The default operator return a vector of ones
  static CMatrix1X default_op(const CMatrixDX& nodes, const RMatrixD1& pos)
};

/**
```

```
* This class provides inner product calculation of the vectorized wavepackets
* param D dimensionality of processed wavepackets
* param QR quadrature rule to use, with |ℜ| nodes
*/
template<dim_t D, class QR>
class VectorInnerProduct
{
public:
  using CMatrixNN = CMatrix<Eigen::Dynamic, Eigen::Dynamic>;
  using CMatrix1N = CMatrix<1, Eigen::Dynamic>;
  using CMatrixN1 = CMatrix<Eigen::Dynamic, 1>;
  using CMatrixD1 = CMatrix<D, 1>;
  using CMatrixDD = CMatrix<D, D>;
  using CMatrixDN = CMatrix<D, Eigen::Dynamic>;
  using RMatrixD1 = RMatrix<D, 1>;
  using CDiagonalNN = Eigen::DiagonalMatrix<complex_t, Eigen::Dynamic>;
  using NodeMatrix = typename QR::NodeMatrix;
  using WeightVector = typename QR::WeightVector;
  using op_t = std::function<CMatrix1N(CMatrixDN,RMatrixD1, dim_t, dim_t)>;

  /**
  * Calculate the matrix of the inner product.
  * Returns the matrix elements ⟨Ψ|f|Ψ⟩ with an operator f.
  * The matrix consists of N × N blocks (N: number of components), each of size |ℜ| × |ℜ|.
  * The coefficients of the wavepacket are ignored.
  * param  Packet: the type of the vectorized Hagedorn wavepacket
  * param[in] packet multi-component wavepacket Ψ
  * param[in] op operator f(x,q,i,j) : C^{D×R} × R^D × N × N → C^R which is evaluated at the nodal points x
    and position q, between components Φ_i and Φ_j;
    default returns a vector of ones if i==j, zeros otherwise
  */
  template<class Packet>
  static CMatrixNN build_matrix(const Packet& packet, const op_t& op=default_op)

  /**
  * Calculate the matrix of the inner product.
  * Returns the matrix elements ⟨Ψ|f|Ψ'⟩ with an operator f.
  * The matrix consists of N × N' blocks (N,N': number of components of Ψ,Ψ'), each of size |ℜ_i| × |ℜ'_j|.
     The coefficients of the wavepacket are ignored.
  * param[in] pacbra multi-component wavepacket Ψ
  * param[in] packet multi-component wavepacket Ψ'
  * param[in] op operator f(x,q,i,j) : C^{D×R} × R^D × N × N → C^R which is evaluated at the
    nodal points x and position q, between components Φ_i and Φ_j;
    default returns a vector of ones if i==j, zeros otherwise
  * param Pacbra packet type of Ψ
  * param Packet packet type of Ψ'
  */
  template<class Pacbra, class Packet>
  static CMatrixNN build_matrix_inhomog(const Pacbra& pacbra, const Packet& packet, const op_t& op=
    default_op)

  /**Perform quadrature.
  * Returns an $N^2$–sized vector of scalars $\langle\Phi_i|f|\Phi_j\rangle$.
  * See build_matrix() for the parameters.
  */
  template<class Packet>
  static CMatrixN1 quadrature(const Packet& packet, const op_t& op=default_op)

  /**Perform quadrature.
  * Returns an $N \cdot N'$–sized vector of scalars $\langle\Phi_i|f|\Phi'_j\rangle$.
  * See build_matrix_inhomog() for the parameters.
  */
  template<class Pacbra, class Packet>
  static CMatrixN1 quadrature_inhomog(const Pacbra& pacbra, const Packet& packet, const op_t& op=
    default_op)

private:
  static CMatrix1N default_op(const CMatrixDN& nodes, const RMatrixD1& pos, dim_t i, dim_t j)
};
```

7 All the potential files are included in the folder potentials.

We have 1D potentials

1D harmonic oscillator (**harmonic_1D.hpp**):

$$V(x) = \frac{\sigma x^2}{2} \tag{89}$$

1D quartic oscillator (**quartic_1D.hpp**)

$$V(x) = \frac{\sigma x^4}{4} \tag{90}$$

1D cosin oscillator (**cos_osc_1D.hpp**)

$$V(x) = a(-\cos(bx) + 1) \tag{91}$$

1D cosh oscillator (**cosh_osc_1D.hpp**)

$$V(x) = a * \cosh(bx) \tag{92}$$

1D double well (**double_well_1D.hpp**)

$$V(x) = \sigma * (x^2 - 1)^2 \tag{93}$$

1D double well (**double_well2_1D.hpp**)

$$V(x) = a * x^4 - b * x^2 \tag{94}$$

1D eckart potential (**eckart_1D.hpp**)

$$V(x) = \frac{\sigma}{\cosh^2(x/a)} \tag{95}$$

1D morse potential (**morse_1D.hpp**)

$$V(x) = D_e(1 - e^{-a(x-x_0)})^2 \tag{96}$$

1D morse potential (**morse_zero_1D.hpp**)

$$V(x) = D_e(-2e^{-a(x-x_0)} + e^{-2a(x-x_0)}) \tag{97}$$

2D potentials

2D cosin oscillator (**cos_osc_2D.hpp**)

$$V(x,y) = a_x(-\cos(b_x x) + 1) + a_y(-\cos(b_y y) + 1) \tag{98}$$

2D cosin oscillator (**cos_osc_mul_2D.hpp**)

$$V(x) = -\cos(ax) * \cos(by) \tag{99}$$

2D harmonic oscillator (**harmonic_2D.hpp**)

$$V(x,y) = 0.5(\sigma_x x^2 + \sigma_y y^2) \tag{100}$$

2D quartic oscillator (**quartic_2D.hpp**)

$$V(x) = \sigma_x x^4 + \sigma_y y^4 \tag{101}$$

D-dimensional potentials

D-dimensional torsional potential (**torsion_XD.hpp**)

$$V(\underline{x}) = \sum_{j=1}^{D}(1 - \cos(x_j)) \tag{102}$$

Henon-Heiles potential (**henon-heiles.hpp**)

$$V(\underline{x}) = \frac{1}{2}\sum_{i=1}^{D} x_i^2 + \lambda \sum_{i=1}^{D-1}\left(x_i^2 x_{i+1} - x_{i+1}^3/3\right) \tag{103}$$

2D delta gap potential (delta_gap_2D_2N**.hpp**)

$$\mathbf{V}(x,y) = \begin{pmatrix} \frac{1}{2}\tanh(\sqrt{x^2+y^2}) & \lambda Q_{10a} \\ \lambda Q_{10a} & -\frac{1}{2}\tanh(\sqrt{x^2+y^2}) \end{pmatrix} \tag{104}$$

4D pyrazine model (4D_pyrazine**.hpp**)

$$\mathbf{V}(x,y) = \begin{pmatrix} -\Delta + \sum_{l=10a,6a,1,9a} \frac{\omega_l}{2} Q_l^2 + \sum_{m=6a,1,9a} \kappa_m^{(1)} Q_m & \lambda Q_{10a} \\ \lambda Q_{10a} & \Delta + \sum_{l=10a,6a,1,9a} \frac{\omega_l}{2} Q_l^2 + \sum_{m=6a,1,9a} \kappa_m^{(2)} Q_m \end{pmatrix} \tag{105}$$

It should be noted that we can add any potentials into our potential library in the folder potentials, they just need to implement following general interfaces for the scalar potential

```
/**
* calculate the value of potential V(x) at position pos
* param[in] pos  the position at which we calculate the value of V(x)
*/
real_t evaluate_pes(const RMatrixD1& pos) const

/**
* calculate the gradient of potential V(x) [∇V(x)] at position pos
* param[in] pos the position at which we calculate the gradient of potential V(x)
*/
RMatrixD1 evaluate_grad(const RMatrixD1& pos) const

/**
* calculate the hessian of potential V(x) [∇^2V(x)] at position pos
* param[in] pos the position at which we calculate the potential V(x)
*/
RMatrixDD evaluate_hess(const RMatrixD1& pos) const

/**
* evaluate the values of the PES at the quadrature points nodes
* param[in] nodes the quadrature points at which we calculate the values of the PES
*/
CMatrix1X evaluate_pes_node(const CMatrixDX& nodes) const

//calculate the local quadratic of V(x) at node: nodes and position: pos,
```

$$U(\underline{x}) = V(\underline{q}) + \nabla V(\underline{q})(\underline{x}-\underline{q}) + \frac{1}{2}\left(\underline{x}-\underline{q}\right)^T \nabla^2 V(\underline{q})\left(\underline{x}-\underline{q}\right)$$

```
CMatrix1X local_quadratic(const CMatrixDX& nodes, const RMatrixD1& pos) const

//calculate the local remainder of the V(x) at node: nodes and position: pos, i.e., W(x) = V(x) − U(x)
CMatrix1X local_remainder(const CMatrixDX& nodes, const RMatrixD1& pos) const
```

and for the matrix potential

```
//evaluate the value of V_{ij} at the qudrature node: nodes
CMatrix1X evaluate_pes_node(const CMatrixDX& nodes, dim_t ii, dim_t jj) const

//evaluate the value of V_{ii}
real_t evaluate_pes(const RMatrixD1& pos, dim_t ii) const

//evaluate the value of V_{leading_order,leading_order}
real_t evaluate_pes(const RMatrixD1& pos) const

//evaluate the gradient of V_{ii}
RMatrixD1 evaluate_grad(const RMatrixD1& pos, dim_t ii) const

//evaluate the gradient of V_{leading_order, leading_order}
RMatrixD1 evaluate_grad(const RMatrixD1& pos) const
```

```

//evaluate the hessian of V_{ii}
RMatrixDD evaluate_hess(const RMatrixD1& pos, dim_t ii) const

//evaluate the hessian of V_{leading_order, leading_order}
RMatrixDD evaluate_hess(const RMatrixD1& pos) const

//evaluate the local remainder of the vector PES
CMatrix1X local_remainder(const CMatrixDX& nodes, const RMatrixD1& pos, dim_t ii, dim_t jj) const

//evaluate the local remainder of the vector PES (homogenous case)
CMatrix1X local_remainder_homogenous(const CMatrixDX& nodes, const RMatrixD1& pos, dim_t ii, dim_t
    jj) const
```

For simplicity, we also provides a wrapper class MatrixPotential1S (**potentials.hpp**) representing any scalar potential $V(\underline{x})$ and a wrapper class MatrixPotentialMS (**potentials.hpp**) for any matrix potential $\mathbf{V}(\underline{x})$

```
/**
* This class represents a scalar potential V(x)
* param D dimensionality of PES (number of variables)
* param PES specific potential experession providing the potential value, gradient and hessian at
    certain point
*/
template<dim_t D, class PES>
class MatrixPotential1S
{
private:
  PES pes_;        // scalar potential V(x) (see potentialLib.hpp for various scalar potential V(x))
public:
  MatrixPotential1S() = default;  // default constructor
  MatrixPotential1S(const PES& pes) //constructor
  MatrixPotential1S(const MatrixPotential1S& that) //copy constructor
  MatrixPotential1S &operator=(const MatrixPotential1S& that) //assignment operator
  PES & pes() //return the object: potential V(x)
  PES const& pes() const // same as above, const version
};

/**
* This class represents a matrix potential V(x)
* param D dimensionality of PES (number of variables)
* param PES specific potential experession providing the potential value, gradient and hessian at
    certain point
*/
template<dim_t D, class PES>
class MatrixPotentialMS
{
private:
  PES pes_;        // matrix  potential V(x) (see potentialLib.hpp for various matrix potential V(x))
public:
  MatrixPotentialMS() = default;  // default constructor
  MatrixPotentialMS(const PES& pes) //constructor
  MatrixPotentialMS(const MatrixPotentialMS& that) //copy constructor
  MatrixPotentialMS &operator=(const MatrixPotentialMS& that) //assignment operator
  PES & pes() //return the object: potential V(x)
  PES const& pes() const // same as above, const version
};
```

8 class ScalarHaWp_Propagator (**ScalarHaWp_Propagator.hpp**) implements Lubich's time-stepping algorithm (see section 1 1.6) for a scalar Hagedorn wavepacket. class VectorHaWp_Propagator (**VectorHaWp_Propagator.hpp**) implments extended time-stepping algorithm (see section 1 1.7) for a vector Hagedorn wavepacket.

```
/**
* This class implements Lubich's time-steping algorithm for a scalar wavepacket
* param D  The dimension of the system
* param ScalarPacket  the type of the scalar wavepacket
* param QR  The quadrature rule
* param PES the scalar potential V(x)
*/
template<dim_t D, class ScalarPacket, class QR, class PES>
class ScalarHaWp_Propagator{
public:

  using IP = innerproducts::HomogeneousInnerProduct<D, QR>;   //the homogeneousInnerProduct type
  using ScalarPES=potentials::MatrixPotential1S<D, PES>;              //the Scalar potential type

  using CMatrix1X = CMatrix<1, Eigen::Dynamic>;
  using RMatrixD1 = RMatrix<D, 1>;
  using CMatrixDX = CMatrix<D, Eigen::Dynamic>;

  ScalarHaWp_Propagator() = default;  // default constructor
  ScalarHaWp_Propagator(const ScalarPES& matrix1s, const ScalarPacket& packet)  //constructor
  ScalarHaWp_Propagator(const ScalarHaWp_Propagator& that)  //copy constructor
  ScalarHaWp_Propagator &operator=(const ScalarHaWp_Propagator& that) //assignment operator
  //Propagate scalar hagedorn wavepacket based on Lubich's time stepping algorithm
  void propagate(real_t dt)
  ScalarPES & matrix1s()  //Grants access to the scalar potential V(x)
  ScalarPES const& matrix1s() const //same as above, const version
  ScalarPacket & packet() //Grants access to the scalar hagedorn wavepacket Φ(x)
  ScalarPacket const& packet() const //same as above, const version
private:
  ScalarPES matrix1s_;      //the potential wrapper for any scalar potential V(x)
  ScalarPacket     packet_;       //the scalar hagedorn wavepacket Φ(x)
  RMatrix<D, D>   mass_inv_;  //the inverse mass matrix M^-1
};

```

```
/**
* This class implements extended Lubich's time—steping algorithm for a vector wavepacket
* param D  The dimension of the system
* param VectorPacket  the type of the vector wavepacket
* param QR  The quadrature rule
* param PES the scalar potential V(x)
*/
template<dim_t D, class VectorPacket, class QR, class PES>
class VectorHaWp_Propagator{
public:

  using IP=innerproducts::VectorInnerProduct<D, QR>;          //the VectorInnerProduct type
  using MatrixPES=potentials::MatrixPotentialMS<D, PES>;      //the matrix potential type
  using CMatrix1X = CMatrix<1, Eigen::Dynamic>;
  using RMatrixD1 = RMatrix<D, 1>;
  using CMatrixDX = CMatrix<D, Eigen::Dynamic>;

  VectorHaWp_Propagator()=default;     //default constructor
  VectorHaWp_Propagator(const MatrixPES& matrixms, const VectorPacket& packet)  //Constructor
  VectorHaWp_Propagator(const VectorHaWp_Propagator& that)        //copy constructor
  VectorHaWp_Propagator &operator=(const VectorHaWp_Propagator& that)  //assignment operator

  //propagate homogenous hagedorn wavepacket based on the extended Lubich time—stepping algorithm
  void propagate_homo(real_t dt)

  //propagate inhomogenous hagedorn wavepacket based on the extended Lubich time—stepping algorithm
  void propagate_inhomo(real_t dt)

  //Grants access to the matrix potential
  MatrixPES & matrixms()
  //same as above, const version
  MatrixPES const& matrixms() const

  //Grants access to the vectorized Hagedorn wavepacket
  VectorPacket & packet()
  //same as above, const version
  VectorPacket const& packet() const

private:
  MatrixPES matrixms_;            //the potential wrapper for any matrix potential
  VectorPacket    packet_;        //the scalar hagedorn wavepacket
  RMatrix<D, D>   mass_inv_;      //the inverse mass matrix
};
```

9 file Observables.hpp provides several routines to calculate the potential energy, kinetic energy, norm and autocorrelation function for both scalar Hagedorn wavepacket and vector Hagedorn wavepacket (**in the diabatic representation**)

```
using wavepackets::ScalarHaWp;                    // Scalar Hagedorn wavepacket
using innerproducts::HomogeneousInnerProduct;     // used for the energy, norm calculation
using innerproducts::InhomogeneousInnerProduct;   // used for the autocorrelation function
    calculation
/**
* Compute the expectation value of the scalar potential ⟨Φ|V|Φ⟩
* param [in] packet   the scalar Hagedorn wavepacket Φ(x)
* param [in] V        the scalar potential V(x)
* param D        the dimension of the system
* param ScalarPacket   the type of the scalar wavepacket
* param QR       the quadrature rule used to calculate the innerproduct
* param PES      the type for the scalar potential V(x)
*/
template<dim_t D, class ScalarPacket, class QR, class PES>
real_t potential_energy(const ScalarPacket& packet, const PES& V){

  HomogeneousInnerProduct<D, QR> ip;
  return ip.quadrature(packet,
                                [&V] (const CMatrix<D,Eigen::Dynamic>& nodes,
                                      const RMatrix<D,1>& pos)
                                -> CMatrix<1,Eigen::Dynamic> {
                                   return V.pes().evaluate_pes_node(nodes);
                                }).real();
}
/**
* Computes kinetic energy of a Hagedorn Wavepacket ⟨Φ|T|Φ⟩
* param[in] packet       the scalar Hagedorn wavepacket Φ(x)
* param[in] mass_inv     the inverse of mass matrix M^-1
* param D            Dimension of the system
* param ScalarPacket       the type of the scalar wavepacket
*/
template<int D, class ScalarPacket>
real_t kinetic_energy(const ScalarPacket& packet, const RMatrix<D, D>& mass_inv){

  CMatrix<D, Eigen::Dynamic> cprime=packet.apply_gradient();  //get the new expansion coefficient

  complex_t result(0,0);
  for(dim_t ii=0; ii<D; ii++){
    result+=mass_inv(ii,ii)*cprime.row(ii).dot(cprime.row(ii));
  }

  return 0.5*result.real();
}
/**
* Compute the norm of the Hagedorn wavepacket ⟨Φ|Φ⟩
* param[in] packet   the scalar Hagedorn wavepacket Φ(x)
* param D         the dimension of the system
* param ScalarPacket    the type of the scalar wavepacket
*/
template<dim_t D, class ScalarPacket>
real_t norm(const ScalarPacket& packet){

  return packet.coefficients().norm();
}

/**
* Compute the auto correlation function of the hagedorn wavepacket ⟨Φ(0)|Φ(t)⟩
* In fact this routine can calculate the overlap of any two scalar Hagedorn wavepacket ⟨Φ_r|Φ_c⟩
* param [in] pacbra   the bra Hagedorn wavepacket Φ(0)
* param [in] packet   the ket Hagedorn wavepacket Φ(t)
* param D         the dimension of the system
* param ScalarPacbra    the type of the bra of scalar wavepacket
* param ScalarPacket    the type of the ket of scalar wavepacket
* param QR        the quadrature rule used to calculate the innerproduct
*/
template<dim_t D, class ScalarPacbra, class ScalarPacket, class QR>
complex_t auto_corr(const ScalarPacbra& pacbra, const ScalarPacket& packet){

  InhomogeneousInnerProduct<D, QR> ip;
  return ip.quadrature(pacbra, packet);
}

/**
* Compute the potential energy of the vectorized Hagedorn wavepackets
* param [in] packet   the vectorized Hagedorn wavepacket |Ψ⟩
* param [in] V        the matrix potential V(x)
* param  D         the dimension of the system
* param VectorPacket    the type for the vectorized hagedorn wavepackeet
* param QR        the quadrature rule used to calculate the innerproduct
* param PES       the type for the matrix potential V
*/
template<dim_t D, class VectorPacket, class QR, class PES>
real_t potential_energy_vec(const VectorPacket& packet, const PES& V){

  VectorInnerProduct<D, QR> ip;

  return ip.quadrature(packet,
                                   [&V]  (const CMatrix<D,Eigen::Dynamic>& nodes,
                                          const RMatrix<D,1>& pos, dim_t ii, dim_t jj)
                                   -> CMatrix<1,Eigen::Dynamic> {
                                      return V.pes().evaluate_pes_node(nodes, ii, jj);
                                   }).sum().real();
}
```

```

/**
* Computes kinetic energy of a vectorized Hagedorn Wavepacket.
* param [in] packet   the vectorized Hagedorn wavepacket |Ψ⟩
* param [in] mass_inv    the inverse of the mass matrix
* param  D         the dimension of the system
* param VectorPacket    the type for the vectorized hagedorn wavepackeet
*/
template<dim_t D, class VectorPacket>
real_t kinetic_energy_vec(const VectorPacket& packet, const RMatrix<D, D>& mass_inv){

  const dim_t n_comps = packet.n_components();  //number of component of the hagedorn wavepacket

  RMatrix<Eigen::Dynamic, 1> kin(n_comps, 1);
  for(dim_t ii=0; ii<n_comps; ii++){
    kin(ii)=kinetic_energy(packet.component(ii), mass_inv);
  }

return kin.sum();
}

/**
*  Compute the norm of a vectorized Hagedorn wavepacket
* param [in]  packet   the vectorized Hagedorn wavepacket
* param D         the dimension of the system
* param VectorPacket    the type of the vectorized hagedorn wavepacket
*/
template<dim_t D, class VectorPacket>
real_t norm_vec(const VectorPacket& packet){

  /*
  VectorInnerProduct<D, QR> ip;
  return std::sqrt(ip.quadrature(packet).sum().real());
  */

  const dim_t n_comps= packet.n_components();  //number of component of the hagedorn wavepacket

  RMatrix<Eigen::Dynamic, 1> pop_components(n_comps, 1);
  for(dim_t ii=0; ii<n_comps; ii++){
    pop_components(ii)=packet.component(ii).coefficients().squaredNorm();
  }
  return std::sqrt(pop_components.sum());
}

/**
* Compute the population of a vectorized Hagedorn wavepacket
* param [in] packet   the vectorized Hagedorn wavepacket
* param D        the dimension of the system
* param VectorPacket    the type of the vectorized hagedorn wavepacket
*/
template<dim_t D, class VectorPacket>
RMatrix<Eigen::Dynamic, 1> pop_vec(const VectorPacket& packet){

  const dim_t n_comps = packet.n_components();  //number of component of the hagedorn wavepacket

  RMatrix<Eigen::Dynamic, 1> pop_components(n_comps, 1);
  for(dim_t ii=0; ii<n_comps; ii++){
    pop_components(ii)=packet.component(ii).coefficients().squaredNorm();
  }

  return pop_components;
}

/**
* Compute the auto correlation of the two vectorized hagedorn wavepacket
* param [in] pacbra   the bra vectorized Hagedorn wavepacket
* param [in] packet   the ket vectorized Hagedorn wavepacket
* param D        the dimension of the system
* param VectorPacbra    the type of the bra of the vectorized hagedorn wavepacket
* param VectorPacket    the type of the ket of the vectorized hagedorn wavepacket
* param QR       the quadrature rule used to calculate the innerproduct
*/
template<dim_t D, class VectorPacbra, class VectorPacket, class QR>
complex_t auto_corr_vec(const VectorPacbra& pacbra, const VectorPacket& packet){

  /*
  VectorInnerProduct<D, QR> ip;
  return ip.quadrature_inhomog(pacbra, packet).sum();
  */

  InhomogeneousInnerProduct<D, QR> ip;
  const dim_t n_comps = packet.n_components();  //number of component of the hagedorn wavepacket
  CMatrix<Eigen::Dynamic, 1> auto_components(n_comps, 1);
  for(dim_t ii=0; ii<n_comps; ii++){
    auto_components(ii)=ip.quadrature(pacbra.component(ii), packet.component(ii));
  }
  return auto_components.sum();
}
```

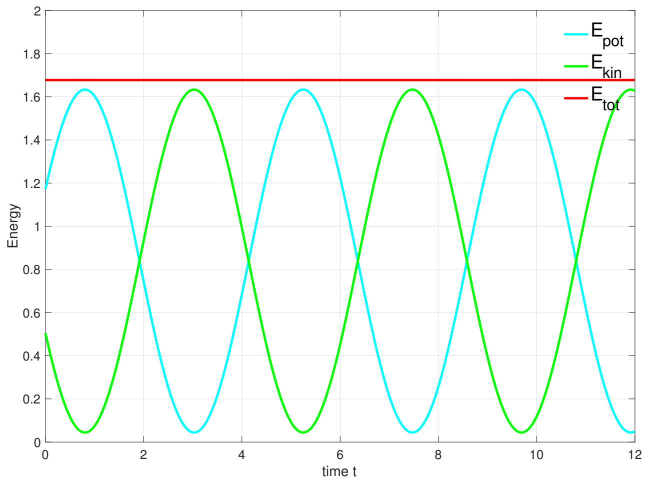

Figure 1. The kinetic, potential and total energy for a two-dimensional harmonic oscillator $V(x,y) = \frac{1}{2}\left(\frac{1}{2}x^2 + \frac{1}{2}y^2\right)$

3. Simulation Results

3.1. Harmonic oscillators

First let us test a simple 2D harmonic potential in order to check whether our code works properly. The potential is given by

$$V(x,y) := \frac{1}{2}\left(\frac{1}{2}x^2 + \frac{1}{2}y^2\right) \tag{106}$$

We consider a single Gaussian wavepacket with the following parameters

$$\underline{q} = \begin{pmatrix} 1.8 \\ 1.2 \end{pmatrix} \quad \underline{p} = \begin{pmatrix} 0.6 \\ 0.8 \end{pmatrix} \quad \mathbf{Q} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \quad \mathbf{P} = \begin{pmatrix} i & 0 \\ 0 & i \end{pmatrix} \quad S = 0 \tag{107}$$

and a scaling parameter of $\varepsilon = 0.1$.

The kinetic, potential, and total energies are plotted in Fig. 1. Fig. 2 displays the time-evolution of the parameter set $\Pi = \{\underline{q}, \underline{p}, \mathbf{Q}, \mathbf{P}, S\}$ of a wavepacket in a two-dimensional harmonic oscillator. Next let us plot the trajectory of the parameter set Π. Fig. 3 depicts the trajectories of the parameters $\underline{q}$ and $\underline{p}$, and Fig. 4 shows the trajectories of $\det\mathbf{Q}$ and $\det\mathbf{P}$ in the complex plane.

3.2. Torsional potential in two dimensions

In this section, we will test the torsional potential in two dimensions as given by the expression

$$V(x,y) := (1 - \cos(x)) + (1 - \cos(y)) \tag{108}$$

The initial parameter set $\Pi = \{\underline{q}, \underline{p}, \mathbf{Q}, \mathbf{P}, S\}$ for the wavepacket $\Phi = \phi_{0,0}$ is given as

$$\underline{q} = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \quad \underline{p} = \begin{pmatrix} 0 \\ 0 \end{pmatrix} \quad \mathbf{Q} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \quad \mathbf{P} = \begin{pmatrix} i & 0 \\ 0 & i \end{pmatrix} \quad S = 0 \tag{109}$$

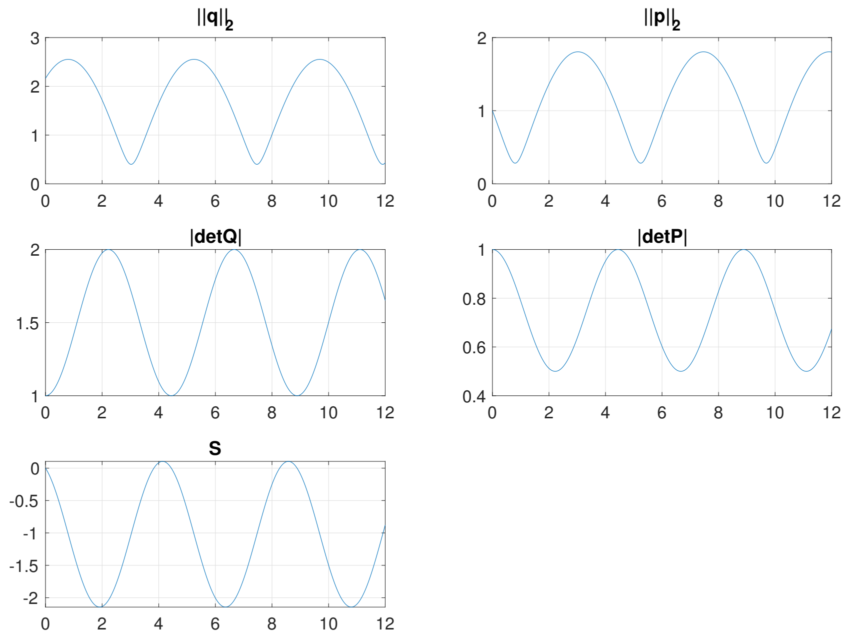

Figure 2. The time-evolution of the parameter set $\Pi = \{\underline{q}, \underline{p}, \mathbf{Q}, \mathbf{P}, S\}$ of a wavepacket in a two-dimensional harmonic oscillator.

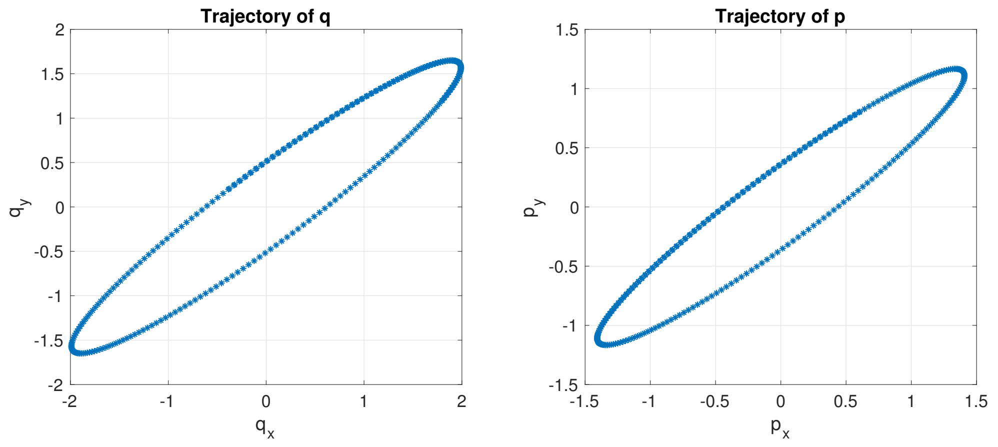

Figure 3. Trajectories of the parameters $\underline{\text{q}}$ and $\underline{\text{p}}$

We use a hyperbolic cut basis shape $\mathfrak{K}$ with a cutoff value of $K = 8, 12, 16, 20, 24$. This yields 20, 35, 50, 66 and 84 basis functions in total. For each simulation we use a time step $\Delta t = 0.005$ and and total propagation time $T = 20$. We perform three simulations for different values of the semi-classical scaling parameter $\varepsilon = \sqrt{0.1}, \sqrt{0.01}, \sqrt{0.001}$, i.e., gradually change from the more quantum to more semi-classical cases. Figs. 5 and 6 plot the energies and auto-correlation function for more quantum case of $\varepsilon = \sqrt{0.1}$, Figs.7 and 8 shows the energies and auto-correlation function for $\varepsilon = \sqrt{0.01}$. Figs. 9 and 10 displays the energies and auto-correlation function for $\varepsilon = \sqrt{0.001}$. As expected, It is very hard to get converged results at longer time for the case of large ε.

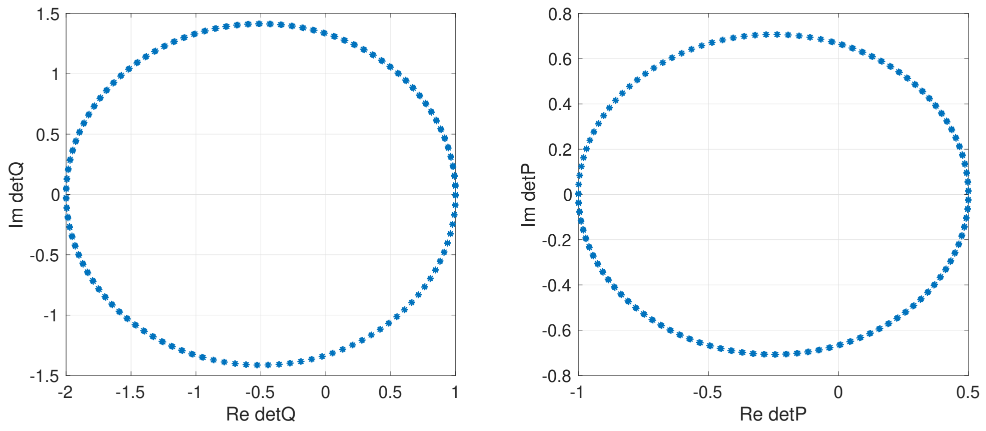

Figure 4. Trajectories of det$\mathbf{Q}$ and det$\mathbf{P}$ in the complex plane.

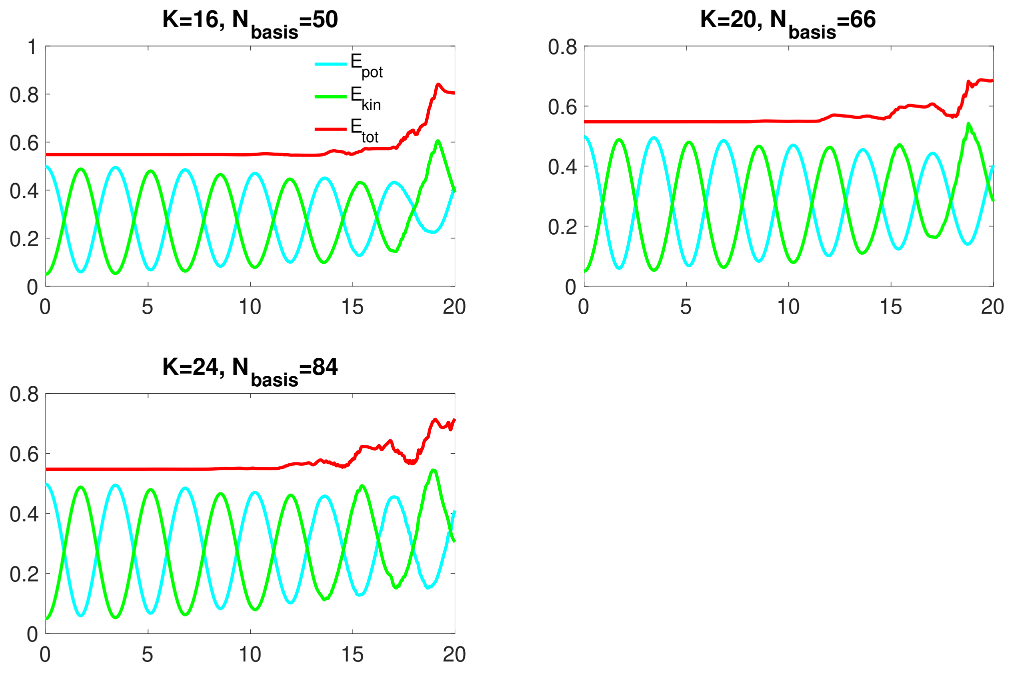

Figure 5. Energies for the case of $\varepsilon = \sqrt{0.1}$

3.3. Torsional potential in five dimensions

In this section, we will test the torsional potential in five dimensions as given by the expression

$$V(x) = \sum_{j=1}^{N}(1 - \cos(x_j)) \tag{110}$$

with $N = 5$.

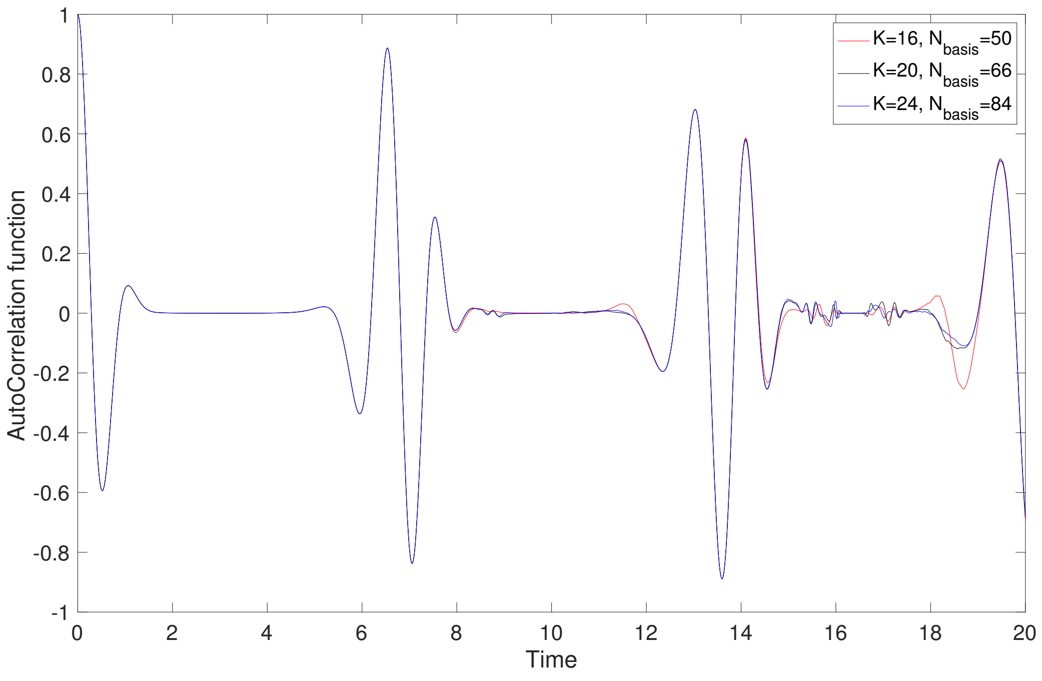

Figure 6. Auto-correlation function for the case of $\varepsilon = \sqrt{0.1}$

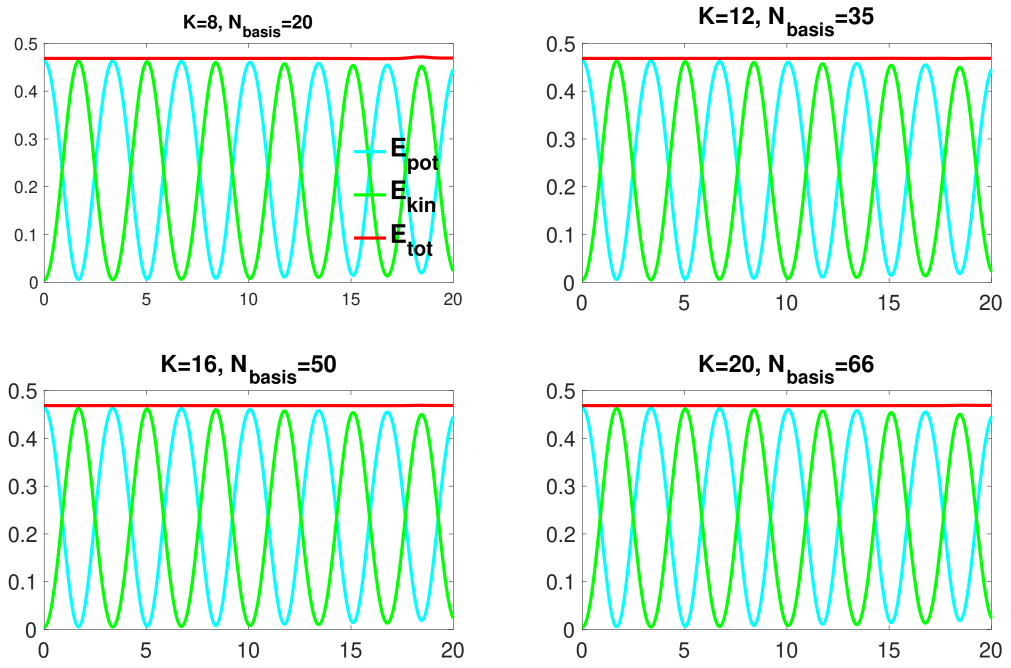

Figure 7. Energies for the case of $\varepsilon = \sqrt{0.01}$

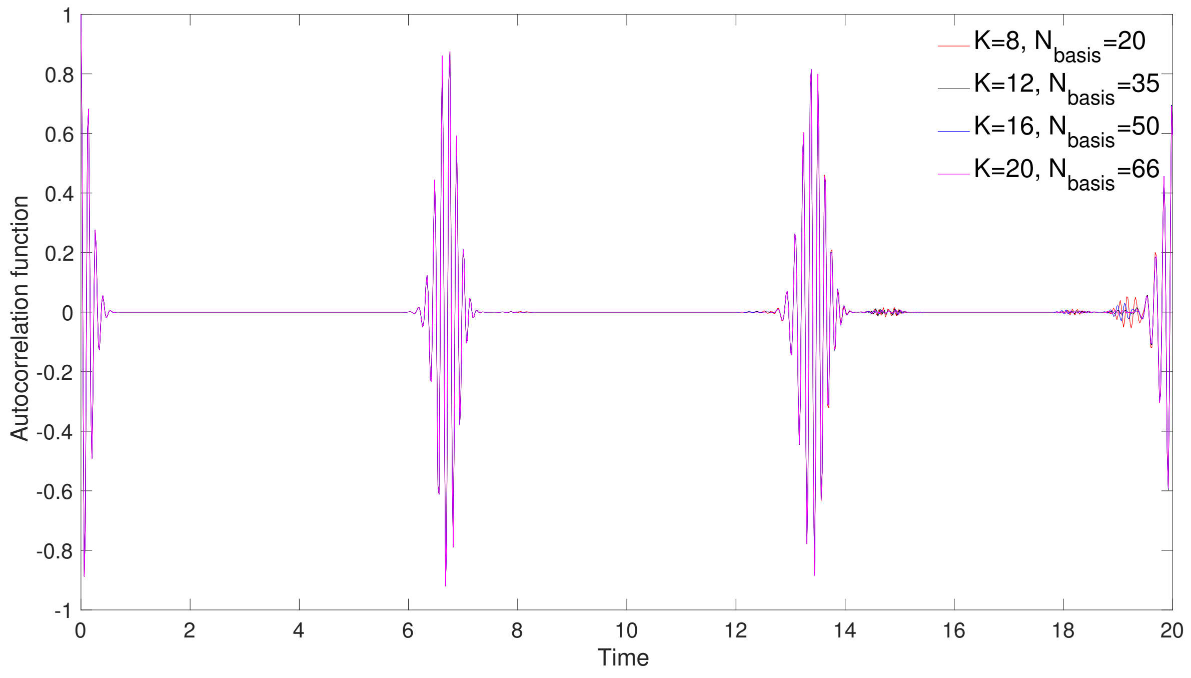

Figure 8. Auto-correlation function for the case of $\varepsilon = \sqrt{0.01}$

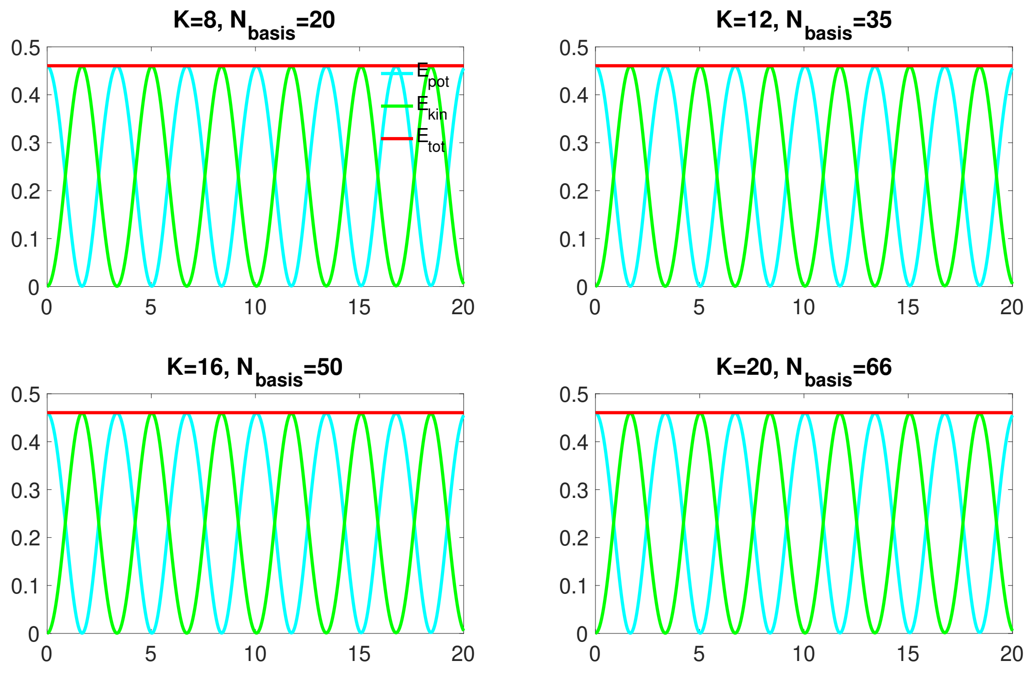

Figure 9. Energies for the case of $\varepsilon = \sqrt{0.001}$

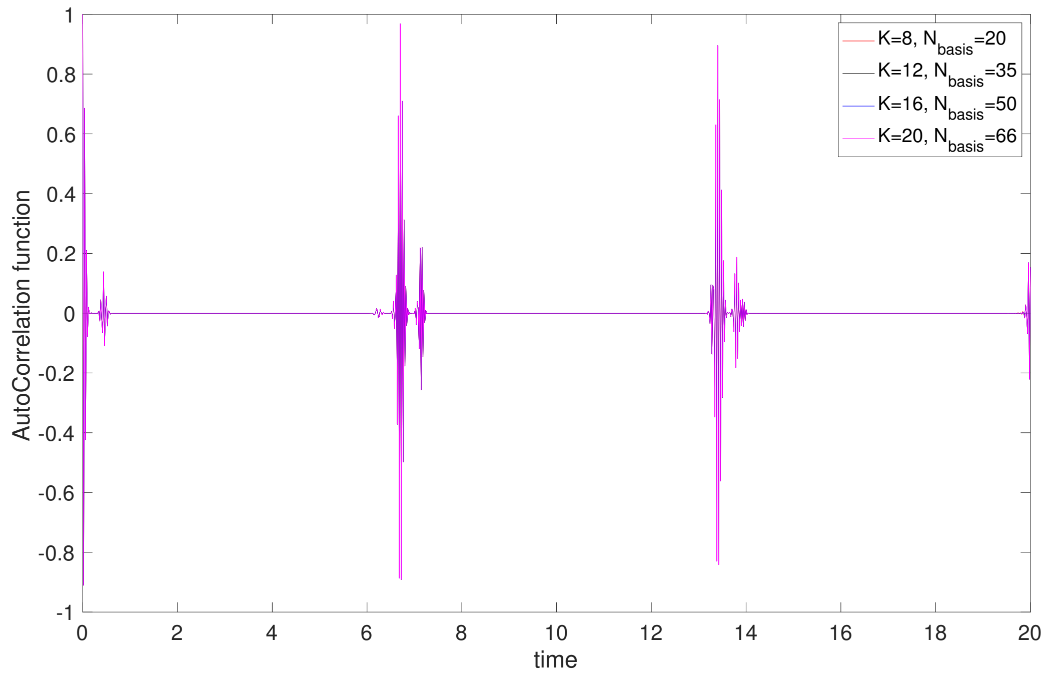

Figure 10. Auto-correlation function for the case of $\varepsilon = \sqrt{0.001}$

The initial parameter set $\Pi = \{\underline{q}, \underline{p}, \mathbf{Q}, \mathbf{P}, S\}$ for the wavepacket $\Phi = \phi_{0,0,0,0,0}$ is given as

$$\underline{q} = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{pmatrix} \quad \underline{p} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} \quad \mathbf{Q} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} \quad \mathbf{P} = \begin{pmatrix} i & 0 & 0 & 0 & 0 \\ 0 & i & 0 & 0 & 0 \\ 0 & 0 & i & 0 & 0 \\ 0 & 0 & 0 & i & 0 \\ 0 & 0 & 0 & 0 & i \end{pmatrix} \quad S = 0 \tag{111}$$

We use a hyperbolic cut basis shape $\mathfrak{K}$ with a cutoff value of $K = 4, 6, 8, 10$. This yields 26, 56, 96 and 136 basis functions in total. For each simulation we use a time step $\Delta t = 0.005$ and and total propagation time $T = 10$. The semiclassical scaling parameter $\varepsilon = \sqrt{0.01}$. Figs. 11 and 12 display the energies and auto-correlation function for different cutoff values of $K = 4, 6, 8, 10$.

3.4. 4-D pyrazine model

In this section, we consider a linear electronic-vibrational coupling for 4-D pyrazine model

$$H = \sum_{l=10a,6a,1,9a} \frac{\omega_l}{2} \left(-\frac{\partial^2}{\partial Q_l^2} + Q_l^2 \right) \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} + \begin{pmatrix} -\Delta & 0 \\ 0 & \Delta \end{pmatrix} + \begin{pmatrix} 0 & \lambda \\ \lambda & 0 \end{pmatrix} Q_{10a} + \sum_{m=6a,1,9a} \begin{pmatrix} \kappa_m^{(1)} & 0 \\ 0 & \kappa_m^{(2)} \end{pmatrix} Q_m \tag{112}$$

The numerical values of the parameters of the system Hamiltonian are collected in Table I.

To obtain a Schrödinger equation of the semiclassical form, we set the smallest of the four oscillator frequency as

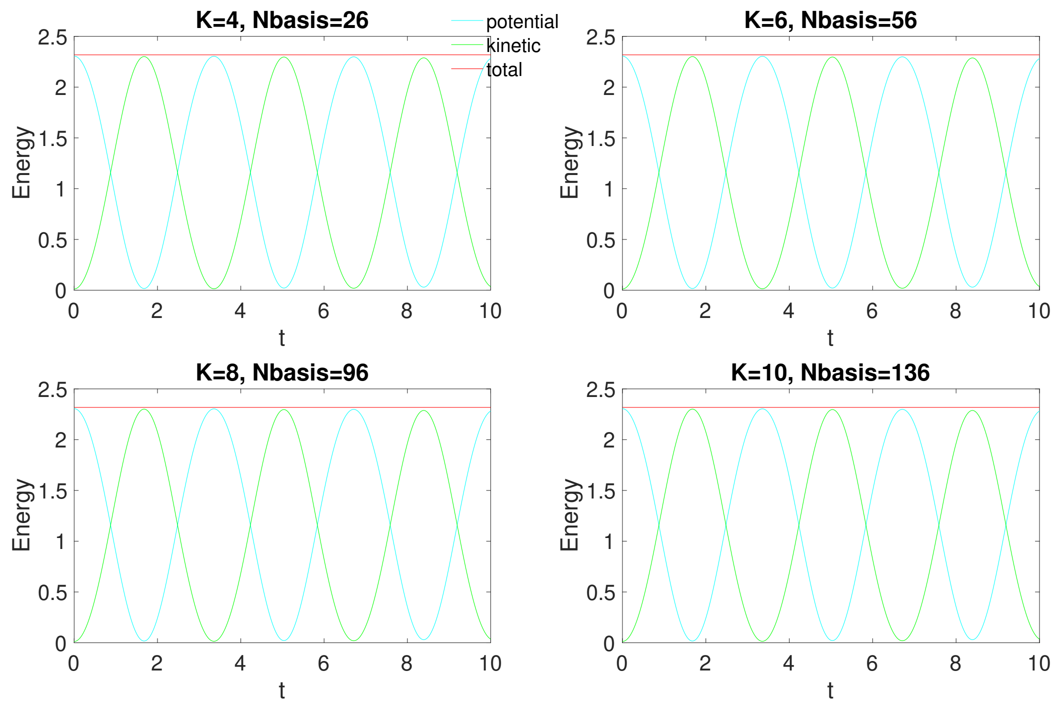

Figure 11. Energies for different cutoff values of $K = 4, 6, 8, 10$

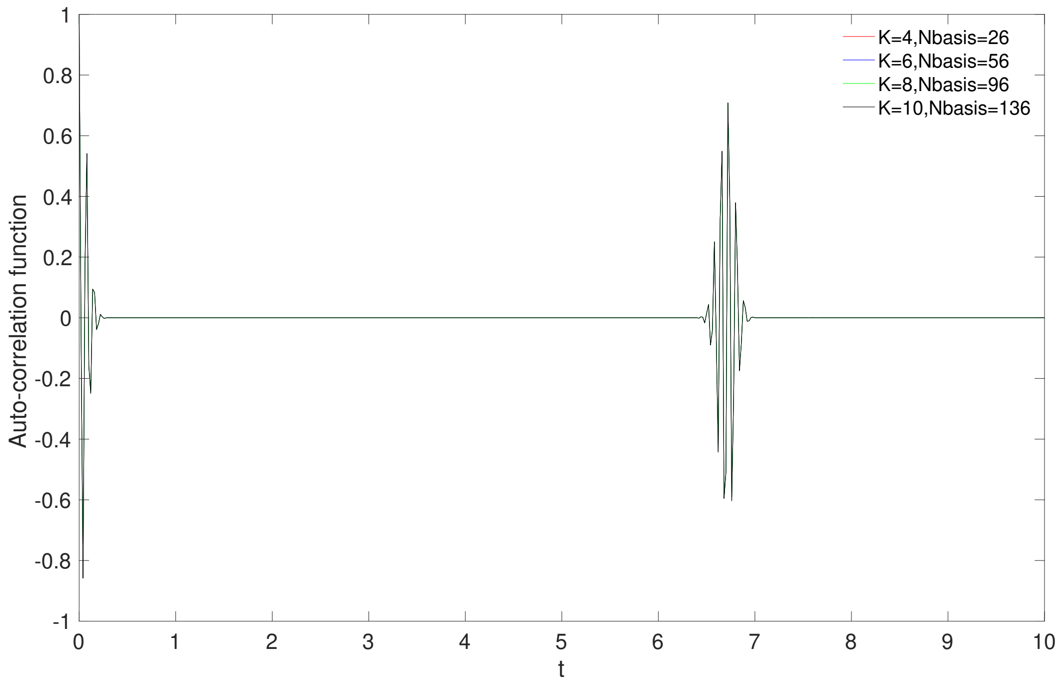

Figure 12. Auto-correlation function for different cutoff values of $K = 4, 6, 8, 10$

Table I. Vibrational frequencies ω_l (eV), vibrational periods τ_l (fs), and electron-vibration coupling constants $\kappa_l^{(1)}$ and $\kappa_l^{(2)}$ (in eV) of the four modes of the system Hamiltonian ($l = 1, 6a, 9a, 10a$) along with the inter-state electronic coupling constant λ (eV) and the vertical energy gap Δ (eV).

l	ω_l	$\tau_l = 2\pi/\omega_l$	$\kappa_l^{(1)}$	$\kappa_l^{(2)}$
$10a$	0.0936	44.2	0	0
$6a$	0.0740	55.9	-0.0964	0.1194
1	0.1273	32.5	0.0470	0.2012
$9a$	0.1568	26.3	0.1594	0.0484
	$\lambda = 0.1825$		$\Delta = 0.4617$	

Table II. Vibrational frequencies ω_l (no unit), and electron-vibration coupling constants $\kappa_l^{(1)}$ and $\kappa_l^{(2)}$ (in hatree$^{\frac{1}{2}}$) of the four modes of the system Hamiltonian ($l = 1, 6a, 9a, 10a$) along with the inter-state electronic coupling constant λ (hatree$^{\frac{1}{2}}$), the vertical energy gap Δ (hatree) and semiclassical scaling parameter ε^2 (hatree)

l	ω_l	$\kappa_l^{(1)}$	$\kappa_l^{(2)}$
$10a$	1.264864864864865	0	0
$6a$	1	-0.067933789407902	0.084142058664975
1	1.720270270270270	0.043441519397318	0.185966674526393
$9a$	2.118918918918919	0.163513755411313	0.049649095118617
	$\lambda = 0.144641774101584$	$\varepsilon^2 = 0.002719449840998$	$\Delta = 0.016967162048500$

$\varepsilon^2 = \omega_{6a}$ and rescale the coordinates according to

$$Q_l \to \frac{\varepsilon^2}{\sqrt{\omega_l}} Q_l \tag{113}$$

The resulting Schrödinger equation ($t \to t\varepsilon^2$) is

$$i\varepsilon^2 \frac{\partial}{\partial t}|\Psi\rangle = \mathbf{H}|\Psi\rangle \tag{114}$$

where

$$\mathbf{H} = \sum_{l=10a,6a,1,9a} \left(-\frac{\varepsilon^4}{2}\frac{\partial^2}{\partial Q_l^2} + \frac{\omega_l^2}{2\varepsilon^4}Q_l^2\right)\begin{pmatrix}1 & 0\\ 0 & 1\end{pmatrix} + \begin{pmatrix}-\Delta & 0\\ 0 & \Delta\end{pmatrix} + \begin{pmatrix}0 & \lambda\\ \lambda & 0\end{pmatrix}\frac{\sqrt{\omega_{10a}}}{\varepsilon^2}Q_{10a} + \sum_{m=6a,1,9a}\begin{pmatrix}\kappa_m^{(1)} & 0\\ 0 & \kappa_m^{(2)}\end{pmatrix}\frac{\sqrt{\omega_m}}{\varepsilon^2}Q_m \tag{115}$$

Now let us introduce scaled frequency $\omega_l \to \frac{\omega_l}{\varepsilon^2}$, scaled $\lambda \to \frac{\lambda\sqrt{\omega_{10a}}}{\varepsilon^2}$, and scaled $\kappa_m^{(1)} \to \frac{\kappa_m^{(1)}\sqrt{\omega_m}}{\varepsilon^2}$, $\kappa_m^{(2)} \to \frac{\kappa_m^{(2)}\sqrt{\omega_m}}{\varepsilon^2}$, the above Hamiltonian can be written as

$$\mathbf{H} = \sum_{l=10a,6a,1,9a} \left(-\frac{\varepsilon^4}{2}\frac{\partial^2}{\partial Q_l^2} + \frac{\omega_l^2}{2}Q_l^2\right)\begin{pmatrix}1 & 0\\ 0 & 1\end{pmatrix} + \begin{pmatrix}-\Delta & 0\\ 0 & \Delta\end{pmatrix} + \begin{pmatrix}0 & \lambda\\ \lambda & 0\end{pmatrix}Q_{10a} + \sum_{m=6a,1,9a}\begin{pmatrix}\kappa_m^{(1)} & 0\\ 0 & \kappa_m^{(2)}\end{pmatrix}Q_m \tag{116}$$

The scaled value of the system Hamiltonian are collected in Table II.

The initial vibrational function is now the ground state of the quantum mechanical harmonic oscillator in semiclassical scaling $\frac{1}{2}\sum_l \left(-\varepsilon^4 \frac{\partial^2}{\partial Q_l^2} + \omega_l^2 Q_l^2\right)$, i.e.,

$$\prod_{l=10a,6a,1,9a} \left(\frac{\omega_l}{\pi\varepsilon^2}\right)^{\frac{1}{4}} \exp(-\frac{\omega_l}{2\varepsilon^2} Q_l^2) \tag{117}$$

The energy level for the matrix potential (here, $\omega_l \to \omega_l^2$)is thus

$$\begin{aligned} \chi_1 &= \frac{1}{2}\left[\sum_l \omega_l Q_l^2 + \sum_m (\kappa_m^{(1)} + \kappa_m^{(2)}) Q_m\right] - \frac{1}{2}\sqrt{\left[2\Delta + \sum_m (\kappa_m^{(2)} - \kappa_m^{(1)}) Q_m\right]^2 + 4\lambda^2 Q_{10a}^2} \\ \chi_2 &= \frac{1}{2}\left[\sum_l \omega_l Q_l^2 + \sum_m (\kappa_m^{(1)} + \kappa_m^{(2)}) Q_m\right] + \frac{1}{2}\sqrt{\left[2\Delta + \sum_m (\kappa_m^{(2)} - \kappa_m^{(1)}) Q_m\right]^2 + 4\lambda^2 Q_{10a}^2} \end{aligned} \tag{118}$$

A. Hagedorn wavepackets for 1D system

We focus on here the construction and propagation algorithms of the Hagedorn wavepackets in one-dimensional systems, where we consider the one-dimensional time-dependent Schrödinger equation

$$i\hbar \frac{\partial \psi}{\partial t} = H\psi \tag{A1}$$

where $\psi = \psi(x,t)$ is the wave function depending on the spatial variable x and time t. The Hamiltonian operator H is written as

$$H = T + V \tag{A2}$$

with the kinetic and potential energy operator

$$T = -\frac{\hbar^2}{2m}\frac{\partial^2}{\partial x^2} \quad \text{and} \quad V = V(x) \tag{A3}$$

In Hagedorn's approach, a Gaussian wavepacket is parametrized as

$$\varphi_0[q,p,Q,P](x) = (\pi\hbar)^{-1/4}(Q)^{-1/2} \exp\left(\frac{i}{2\hbar} P Q^{-1}(x-q)^2 + \frac{i}{\hbar} p(x-q)\right) \tag{A4}$$

where $q, p \in \mathbb{R}$ represent the position and momentum, respectively, and $Q, P \in \mathbb{C}$ satisfy the compatibility conditions.

$$QP - PQ = 0, \tag{A5}$$

$$Q^* P - P^* Q = 2i \tag{A6}$$

The last two relations are equivalent to requiring that

$$Y = \begin{pmatrix} \mathrm{Re}Q & \mathrm{Im}Q \\ \mathrm{Re}P & \mathrm{Im}P \end{pmatrix} \tag{A7}$$

be symplectic, i.e.,

$$Y^T J Y = J, \quad \text{with} \quad J = \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix} \tag{A8}$$

Hagedorn constructs a complete L^2-orthonormal set of functions

$$\varphi_k(x) = \varphi_k[q,p,Q,P](x) \tag{A9}$$

for non-negative integers k, which can be recursively constructed as follows [1]: let x denote the position operator (acting on functions of x by multiplication with x) and $y = -i\hbar\partial/\partial_x$ the momentum operator, and introducing the raising operator $\mathcal{R}$ and lowering operator $\mathcal{L}$ as

$$\mathcal{R} = -\frac{i}{\sqrt{2\hbar}}\left(-P^*(x-q) + Q^*(y-p)\right) \tag{A10}$$

$$\mathcal{L} = \frac{i}{\sqrt{2\hbar}}\left(-P(x-q) + Q(y-p)\right) \tag{A11}$$

Define

$$\varphi_{k+1} = \frac{1}{\sqrt{k+1}}\mathcal{R}\varphi_k \tag{A12}$$

It then turns out that these functions are orthonormal, as the eigenfunctions of the Hermitian operator $\mathcal{LR} = \mathcal{RL} + I$. Moreover, we have

$$\varphi_{k-1} = \frac{1}{\sqrt{k}}\mathcal{L}\varphi_k \tag{A13}$$

(the right-hand side is zero if $k = 0$), and the functions φ_k are polynomials of degree k multiplied by the Gaussian φ_0. Since the above relations imply that [see Ref.[1] Eqs. (2.22) and (2.23)]

$$x - q = \sqrt{\frac{\hbar}{2}}(Q\mathcal{R} + Q^*\mathcal{L}) \tag{A14}$$

and

$$y - p = \sqrt{\frac{\hbar}{2}}\left(P\mathcal{R} + P^*\mathcal{L}\right) \tag{A15}$$

we obtain the recurrence relation

$$Q\sqrt{k+1}\varphi_{k+1}(x) = \sqrt{\frac{2}{\hbar}}(x-q)\varphi_k(x) - Q^*\sqrt{k}\varphi_{k-1}(x). \tag{A16}$$

and

$$P\sqrt{k+1}\varphi_{k+1}(x) = \sqrt{\frac{2}{\hbar}}(y-p)\varphi_k(x) - P^*\sqrt{k}\varphi_{k-1}(x) \tag{A17}$$

Formulas Eq. A14 shows that in the basis $\{\varphi_k(x)\}$, the operator $x - q$ is represented by the infinite matrix

$$\sqrt{\hbar/2}\begin{pmatrix} 0 & Q^*\sqrt{1} & 0 & 0 & 0 & 0 & \cdots \\ Q\sqrt{1} & 0 & Q^*\sqrt{2} & 0 & 0 & 0 & \cdots \\ 0 & Q\sqrt{2} & 0 & Q^*\sqrt{3} & 0 & 0 & \cdots \\ 0 & 0 & Q\sqrt{3} & 0 & Q^*\sqrt{4} & 0 & \cdots \\ 0 & 0 & 0 & Q\sqrt{4} & 0 & Q^*\sqrt{5} & \cdots \\ 0 & 0 & 0 & 0 & Q\sqrt{5} & 0 & \cdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots \end{pmatrix} \tag{A18}$$

By a routine induction, the k, j matrix element of $(x-q)^m$ is zero unless $m-|k-j|$ is a non-negative, even integer, and

$$\langle \varphi_k|(x-q)^m|\varphi_j\rangle = \hbar^{m/2} Q^{(m+k-j)/2}(Q^*)^{(m+j-k)/2} M(k,m,j) \tag{A19}$$

where $M(k,m,j)$ is the matrix element of x^m in the basis of eigenstates of the standard harmonic oscillator Hamiltonian, whose explicit formula can be found in section 4 of Ref. [1]. Alternatively, one can also use Wick's theorem to derive the explicit formula of $M(k,m,j)$. It should be noted that $\varphi_k[q,p,Q,P](x)$ can be also defined in terms of the Hermite polynomial

$$\begin{aligned}\varphi_k[q,p,Q,P](x) =& 2^{-k/2}(k!)^{-1/2}\pi^{-1/4}\hbar^{-1/4}Q^{-(k+1)/2}(Q^*)^{k/2} \times H_k(\hbar^{-1/2}|Q|^{-1}(x-q)) \\ & \times \exp\left\{iPQ^{-1}(x-q)^2/(2\hbar) + ip(x-q)/\hbar\right\} \\ =& 2^{-k/2}(k!)^{-1/2}Q^{-k/2}(Q^*)^{k/2} \times H_k(\hbar^{-1/2}|Q|^{-1}(x-q))\varphi_0[q,p,Q,P](x). \end{aligned} \tag{A20}$$

Next let us calculate overlap of two hagedorn wavepacket $\langle\varphi_m[q_\alpha,p_\alpha,Q_\alpha,P_\alpha]|\varphi_n[q_\beta,p_\beta,Q_\beta,P_\beta]\rangle$. For this purpose, we denote $D_{m,n} = \langle\varphi_m[q_\alpha,p_\alpha,Q_\alpha,P_\alpha]|\varphi_n[q_\beta,p_\beta,Q_\beta,P_\beta]\rangle = \langle\varphi_{m\alpha}|\varphi_{n\beta}\rangle$, $\mathcal{L}_\alpha = \mathcal{L}(q_\alpha,p_\alpha,Q_\alpha,P_\alpha)$, $\mathcal{L}_\beta = \mathcal{L}(q_\beta,p_\beta,Q_\beta,P_\beta)$ and derive following recursive relations

$$\begin{aligned}\sqrt{m+1}D_{m+1,n} =& \sqrt{m+1}\langle\varphi_{m+1,\alpha}|\varphi_{n,\beta}\rangle \\ =& \langle\varphi_{m,\alpha}|\mathcal{L}_\alpha|\varphi_{n,\beta}\rangle \\ =& \langle\varphi_{m,\alpha}|\frac{i}{\sqrt{2\hbar}}\left(-P_\alpha(x-q_\alpha)+Q_\alpha(y-p_\alpha)\right)|\varphi_{n,\beta}\rangle \\ =& \langle\varphi_{m,\alpha}|\frac{i}{\sqrt{2\hbar}}\left(-P_\alpha(x-q_\beta+q_\beta-q_\alpha)+Q_\alpha(y-p_\beta+p_\beta-p_\alpha)\right)|\varphi_{n,\beta}\rangle \\ =& \frac{i}{\sqrt{2\hbar}}\left[-P_\alpha(q_\beta-q_\alpha)+Q_\alpha(p_\beta-p_\alpha)\right]D_{m,n} - \frac{i}{\sqrt{2\hbar}}P_\alpha\langle\varphi_{m,\alpha}|(x-q_\beta)|\varphi_{n,\beta}\rangle \\ & + \frac{i}{\sqrt{2\hbar}}Q_\alpha\langle\varphi_{m,\alpha}|(y-p_\beta)|\varphi_{n,\beta}\rangle \\ =& \frac{i}{\sqrt{2\hbar}}\left[-P_\alpha(q_\beta-q_\alpha)+Q_\alpha(p_\beta-p_\alpha)\right]D_{m,n} - \frac{i}{\sqrt{2\hbar}}P_\alpha\langle\varphi_{m,\alpha}|\sqrt{\frac{\hbar}{2}}(Q_\beta\mathcal{R}_\beta+Q_\beta^*\mathcal{L}_\beta)|\varphi_{n,\beta}\rangle \\ & + \frac{i}{\sqrt{2\hbar}}Q_\alpha\langle\varphi_{m,\alpha}|\sqrt{\frac{\hbar}{2}}(P_\beta\mathcal{R}_\beta+P_\beta^*\mathcal{L}_\beta)|\varphi_{n,\beta}\rangle \\ =& \frac{i}{\sqrt{2\hbar}}\left[-P_\alpha(q_\beta-q_\alpha)+Q_\alpha(p_\beta-p_\alpha)\right]D_{m,n} - \frac{i}{2}P_\alpha\left[Q_\beta\sqrt{n+1}D_{m,n+1}+Q_\beta^*\sqrt{n}D_{m,n-1}\right] \\ & + \frac{i}{2}Q_\alpha\left[P_\beta\sqrt{n+1}D_{m,n+1}+P_\beta^*\sqrt{n}D_{m,n-1}\right]\end{aligned} \tag{A21}$$

and

$$\begin{aligned}
\sqrt{n+1}D_{m,n+1} =&\sqrt{n+1}\langle\varphi_{m,\alpha}|\varphi_{n+1,\beta}\rangle \\
=&\langle\varphi_{m,\alpha}|\mathcal{R}_\beta|\varphi_{n,\beta}\rangle \\
=&\langle\varphi_{m,\alpha}| - \frac{i}{\sqrt{2\hbar}}\left(-P_\beta^*(x-q_\beta)+Q_\beta^*(y-p_\beta)\right)|\varphi_{n,\beta}\rangle \\
=&\langle\varphi_{m,\alpha}| - \frac{i}{\sqrt{2\hbar}}\left(-P_\beta^*(x-q_\alpha+q_\alpha-q_\beta)+Q_\beta^*(y-p_\alpha+p_\alpha-p_\beta)\right)|\varphi_{n,\beta}\rangle \\
=&-\frac{i}{\sqrt{2\hbar}}\left[-P_\beta^*(q_\alpha-q_\beta)+Q_\beta^*(p_\alpha-p_\beta)\right]D_{m,n}+\frac{i}{\sqrt{2\hbar}}P_\beta^*\langle\varphi_{m,\alpha}|(x-q_\alpha)|\varphi_{n,\beta}\rangle \\
&-\frac{i}{\sqrt{2\hbar}}Q_\beta^*\langle\varphi_{m,\alpha}|(y-p_\alpha)|\varphi_{n,\beta}\rangle \\
=&-\frac{i}{\sqrt{2\hbar}}\left[-P_\beta^*(q_\alpha-q_\beta)+Q_\beta^*(p_\alpha-p_\beta)\right]D_{m,n}+\frac{i}{\sqrt{2\hbar}}P_\beta^*\langle\varphi_{m,\alpha}|\sqrt{\frac{\hbar}{2}}(Q_\alpha\mathcal{R}_\alpha+Q_\alpha^*\mathcal{L}_\alpha)|\varphi_{n,\beta}\rangle \\
&-\frac{i}{\sqrt{2\hbar}}Q_\beta^*\langle\varphi_{m,\alpha}|\sqrt{\frac{\hbar}{2}}(P_\alpha\mathcal{R}_\alpha+P_\alpha^*\mathcal{L}_\alpha)|\varphi_{n,\beta}\rangle \\
=&-\frac{i}{\sqrt{2\hbar}}\left[-P_\beta^*(q_\alpha-q_\beta)+Q_\beta^*(p_\alpha-p_\beta)\right]D_{m,n}+\frac{i}{2}P_\beta^*\left(Q_\alpha\sqrt{m}D_{m-1,n}+Q_\alpha^*\sqrt{m+1}D_{m+1,n}\right) \\
&-\frac{i}{2}Q_\beta^*\left(P_\alpha\sqrt{m}D_{m-1,n}+P_\alpha^*\sqrt{m+1}D_{m+1,n}\right)
\end{aligned} \tag{A22}$$

$D_{0,0} = \langle\varphi_0[q_\alpha,p_\alpha,Q_\alpha,P_\alpha]|\varphi_0[q_\beta,p_\beta,Q_\beta,P_\beta]\rangle$ can be easily calculated as

$$\begin{aligned}
D_{0,0} =&\int_{-\infty}^{\infty}dx(\pi\hbar)^{-1/2}(Q_\alpha^*Q_\beta)^{-1/2}\exp\left(-\frac{i}{2\hbar}P_\alpha^*(Q_\alpha^*)^{-1}(x-q_\alpha)^2-\frac{i}{\hbar}p_\alpha(x-q_\alpha)\right) \\
&\times\exp\left(\frac{i}{2\hbar}P_\beta(Q_\beta)^{-1}(x-q_\beta)^2+\frac{i}{\hbar}p_\beta(x-q_\beta)\right) \\
=&(\pi\hbar)^{-1/2}(Q_\alpha^*Q_\beta)^{-1/2}\sqrt{\frac{\pi}{a}}e^{\frac{b^2}{4a}+c}
\end{aligned} \tag{A23}$$

where

$$a = \frac{i}{2\hbar}\left(P_\alpha^*(Q_\alpha^*)^{-1}-P_\beta Q_\beta^{-1}\right),$$

$$b = \frac{i}{\hbar}\left(P_\alpha^*(Q_\alpha^*)^{-1}q_\alpha-P_\beta Q_\beta^{-1}q_\beta+p_\beta-p_\alpha\right), \tag{A24}$$

$$c = -\frac{i}{2\hbar}\left(P_\alpha^*(Q_\alpha^*)^{-1}q_\alpha^2-P_\beta Q_\beta^{-1}q_\beta^2-2p_\alpha q_\alpha+2p_\beta q_\beta\right) \tag{A25}$$

Those equations are useful for the calculation of correlation function which involves Hagedorn wavepacket at different time.

We approximate solutions to Schrödinger equation in the form of a finite linear combination of wave packets, with a common highly oscillatory phase factor

$$\psi(x,t)\approx u(x,t) = e^{iS(t)/\hbar}\sum_{k=0}^{K}c_k(t)\varphi_k[q(t),p(t),Q(t),P(t)](x) \tag{A26}$$

This ansatz is motivated by the remarkable fact that in the case of a quadratic (possibly time-dependent) potential V, the functions $e^{iS(t)/\hbar}\varphi_k[q(t),p(t),Q(t),P(t)]$ are exact solutions to the Schrödinger equation if the position and momentum parameters follow the classical equation of motion,

$$\dot{q} = p/m, \quad \dot{p} = -\nabla V(q), \tag{A27}$$

the linearized equations of motion

$$\dot{Q} = P/m, \quad \dot{P} = -\nabla^2 V(q)Q, \tag{A28}$$

and $S(t) = \int_0^t \left(\frac{1}{2m}p(s)^2 - V(q(s))\right) ds$ is the classical action. On the other hand, for a nonquadratic potential, we can employ the time-stepping algorithm described in Ref. [2]. Assume that the stepsize Δt is given, and let the real scalars q^n, p^n, S^n, the complex Q^n, P^n, and the complex coefficient vector $c^n = (c_k^n)(k = 0, \cdots, K)$, be such that

$$u^n = e^{iS^n/\hbar} \sum_{k=0}^{K} c_k^n \varphi_k[q^n, p^n, Q^n, P^n] \tag{A29}$$

is an approximation to the solution of the Schrödinger equation at time $t^n = n\Delta t$. To compute the approximation u^{n+1} at time t^{n+1}, we proceed as follows.

(1) Compute $q^{n+1/2}$, $Q^{n+1/2}$, and $S^{n+1/2,-}$ via

$$\begin{aligned} q^{n+1/2} &= q^n + \frac{\Delta t}{2} p^n/m, \\ Q^{n+1/2} &= Q^n + \frac{\Delta t}{2} P^n/m, \\ S^{n+1/2,-} &= S^n + \frac{\Delta t}{4}(p^n)^2/m \end{aligned} \tag{A30}$$

(2) Compute p^{n+1}, P^{n+1}, and $S^{n+1/2,+}$ via

$$\begin{aligned} p^{n+1} &= p^n - \Delta t \nabla V(q^{n+1/2}), \\ P^{n+1} &= P^n - \Delta t \nabla^2 V(q^{n+1/2}) Q^{n+1/2}, \\ S^{n+1/2,+} &= S^{n+1/2,-} - \Delta t V(q^{n+1/2}) \end{aligned} \tag{A31}$$

(3) Update the coefficient vector $c^{n+1} = (c_k^{n+1})(k = 0, \cdots, K)$ as

$$c^{n+1} = \exp(-\Delta t \frac{i}{\hbar} F^{n+1/2}) c^n \tag{A32}$$

Here, $F^{n+1/2} = (f_{kl})(k, l = 0, \cdots, K)$ is the Hermitian matrix with entries

$$f_{kl} = \langle \varphi_k^{n+1/2} | W^{n+1/2} | \varphi_l^{n+1/2} \rangle, \tag{A33}$$

where $\varphi_k^{n+1/2} = \varphi_k[q^{n+1/2}, p^{n+1}, Q^{n+1/2}, P^{n+1}]$ are the Hagedorn basis functions and

$$W^{n+1/2}(x) = V(x) - U^{n+1/2}(x) \tag{A34}$$

is the remainder in the local quadratic approximation to V, given at $q = q^{n+1/2}$ by $U^{n+1/2}(x) = V(q) + \nabla V(q)(x - q) + \frac{1}{2}\nabla^2 V(q)(x-q)^2$.

4. Compute q^{n+1}, Q^{n+1}, and S^{n+1} via

$$\begin{aligned} q^{n+1} &= q^{n+1/2} + \frac{\Delta t}{2} p^{n+1}/m, \\ Q^{n+1} &= Q^{n+1/2} + \frac{\Delta t}{2} P^{n+1}/m, \\ S^{n+1} &= S^{n+1/2,+} + \frac{\Delta t}{4}(p^{n+1})^2/m \end{aligned} \tag{A35}$$

B. The generalized coherent states

The one-dimensional harmonic oscillator is defined as

$$H = \frac{p^2}{2m} + \frac{m\omega^2}{2} q^2 \tag{B1}$$

we then introduce the creation and annihilation operators

$$\hat{a}^\dagger = \sqrt{\frac{m\omega}{2\hbar}}(q - \frac{i}{m\omega}p), \tag{B2}$$

$$\hat{a} = \sqrt{\frac{m\omega}{2\hbar}}(q + \frac{i}{m\omega}p), \tag{B3}$$

thus

$$q = \sqrt{\frac{\hbar}{2m\omega}}(\hat{a}^\dagger + \hat{a}), \tag{B4}$$

$$p = i\sqrt{\frac{m\omega\hbar}{2}}(\hat{a}^\dagger - \hat{a}) \tag{B5}$$

The one-dimensional harmonic oscillator (Eq. B1) can thus be written in the second-quantization representation as

$$H = \frac{1}{2}\hbar\omega(\hat{a}^\dagger\hat{a} + \hat{a}\hat{a}^\dagger) = \hbar\omega(\hat{a}^\dagger\hat{a} + \frac{1}{2}) \tag{B6}$$

we further have several commutation relations

$$[\hat{a}, \hat{a}^\dagger] = 1, \tag{B7}$$

$$[H, \hat{a}] = -\hbar\omega\hat{a}, \tag{B8}$$

$$\left[H, \hat{a}^\dagger\right] = \hbar\omega\hat{a}^\dagger \tag{B9}$$

The eigenstate and eigenvalue of the one-dimensional harmonic oscillator are $|k\rangle$ and $\mathrm{E}_k = \hbar\omega(k + \frac{1}{2})$, respectively. we further have

$$\hat{a}|k\rangle = \sqrt{k}|k-1\rangle, \tag{B10}$$

$$\hat{a}^\dagger|k\rangle = \sqrt{k+1}|k+1\rangle, \tag{B11}$$

$$|k\rangle = \frac{1}{\sqrt{k!}}(\hat{a}^\dagger)^k|0\rangle \tag{B12}$$

In the coordinate representation, $|k\rangle$ can be written as

$$\phi_k(x) = \langle x|k\rangle = \pi^{-1/4}2^{-k/2}(k!)^{-1/2}H_k(x)\exp(-\frac{1}{2}x^2) \tag{B13}$$

where

$$H_k(x) = (-1)^k e^{x^2}\frac{d^k}{dx^k}e^{-x^2} \tag{B14}$$

is the kth order Hermite polynomial. The generating function of the Hermite polynomial is

$$\exp(-s^2 + 2xs) = \sum_{k=0}^{\infty}\frac{H_k(x)}{k!}s^k \tag{B15}$$

It can be further shown that using the above generating function, one obtains

$$\int_{-\infty}^{\infty} H_k(x)H_j(x)\exp(-x^2)dx = \pi^{1/2}2^k k!\delta_{kj} \tag{B16}$$

One can also obtain the following recursive relations

$$H_{k+1}(x) - 2xH_k(x) + 2kH_{k-1}(x) = 0, \tag{B17}$$

$$H_k^{'}(x) = 2kH_{k-1}(x) \tag{B18}$$

which can be used to obtain the recursive relations for the eigenstate of the harmonic oscillator

$$x\varphi_k(x) = \left[\sqrt{\frac{k}{2}}\varphi_{k-1}(x) + \sqrt{\frac{k+1}{2}}\varphi_{k+1}(x)\right], \tag{B19}$$

$$x^2\varphi_k(x) = \frac{1}{2}\left[\sqrt{k(k-1)}\varphi_{k-2} + (2k+1)\varphi_k + \sqrt{(k+1)(k+2)}\varphi_{k+2}(x)\right], \tag{B20}$$

$$\frac{d}{dx}\varphi_k(x) = (\sqrt{\frac{k}{2}}\varphi_{k-1} - \sqrt{\frac{k+1}{2}}\varphi_{k+1}), \tag{B21}$$

$$\frac{d^2}{dx^2}\varphi_k(x) = \frac{1}{2}\left[\sqrt{k(k-1)}\varphi_{k-2} - (2k+1)\varphi_k + \sqrt{(k+1)(k+2)}\varphi_{k+2}\right] \tag{B22}$$

The first two of above relations are useful for the calculation of matrix element in Eq. A19 ($M(k,m,j)$), while the last two of above relations are useful for the calculation of the matrix element of the kinetic operator.

Next let us introduce the coherent state, which is defined as the eigenstate of the annihilation operator $\hat{a}$, i.e.,

$$\hat{a}|\alpha\rangle = \alpha|\alpha\rangle \tag{B23}$$

where α is a complex (we will show later that it relate to the momentum and position of the harmonic oscillator) The coherent state $|\alpha\rangle$ can be obtained by acting the displacement operator $\hat{D}(\alpha) = \exp\left[\alpha\hat{a}^\dagger - \alpha^*\hat{a}\right]$ on the ground state $|k=0\rangle$ of the harmonic oscillator

$$|\alpha\rangle = \hat{D}(\alpha)|0\rangle = \exp\left[\alpha\hat{a}^\dagger - \alpha^*\hat{a}\right]|0\rangle = \exp\left[-\frac{|\alpha|^2}{2}\right]\exp\left[\alpha\hat{a}^\dagger\right]\exp\left[-\alpha^*\hat{a}\right]|0\rangle = \exp\left[-\frac{|\alpha|^2}{2}\right]\exp\left[\alpha\hat{a}^\dagger\right]|0\rangle \tag{B24}$$

where we have used the simplified Zassenhaus formula

$$\exp\left[\hat{A}+\hat{B}\right] = \exp\left[\hat{A}\right]\exp\left[\hat{B}\right]\exp\left\{-\frac{1}{2}[\hat{A},\hat{B}]\right\} \tag{B25}$$

valid in the case that the operators $\hat{A}$, $\hat{B}$ commute with their commutator, and the complex number α is related to the expectation values of the position and momentum operator

$$\alpha = \sqrt{\frac{1}{2}}(\langle\hat{q}\rangle + i\langle\hat{p}\rangle) = \sqrt{\frac{1}{2}}(q+ip) \tag{B26}$$

It can be easily verified that the displacement operator is unitary

$$\hat{D}^{-1}[\alpha] = \hat{D}^\dagger[\alpha] = \hat{D}[-\alpha] \tag{B27}$$

Furthermore, the displacement operator has the properties

$$\hat{D}^\dagger[\alpha]\,\hat{a}\hat{D}[\alpha] = \hat{a} + \alpha, \tag{B28}$$

$$\hat{D}^\dagger[\alpha]\,\hat{a}^\dagger\hat{D}[\alpha] = \hat{a}^\dagger + \alpha^*. \tag{B29}$$

they can be proven by using the Hadamard lemma

$$\exp\left[\hat{B}\right]\hat{A}\exp\left[-\hat{B}\right] = \hat{A} + \left[\hat{B},\hat{A}\right] + \frac{1}{2!}\left[\hat{B},\left[\hat{B},\hat{A}\right]\right] + \cdots \tag{B30}$$

We can also represent the coherent state in the coordinate representation

$$\langle x|\alpha\rangle = \pi^{-1/4}\exp\left(-\frac{1}{2\hbar}(x-q)^2 + \frac{i}{\hbar}p(x-q) + \frac{ipq}{2\hbar}\right) \tag{B31}$$

which is differently from the $|q,p\rangle$ notation commonly employed by chemical physicists by a phase factor $\exp\left(\frac{ipq}{2\hbar}\right)$

$$\langle x|p,q\rangle = \pi^{-1/4}\exp\left(-\frac{1}{2\hbar}(x-q)^2 + \frac{i}{\hbar}p(x-q)\right) \tag{B32}$$

Now let us introduce the generalized coherent state (also called displaced number states in quantum optics [4]), which can be obtained by applying the displacement operator $\hat{D}(\alpha) = \exp\left[\alpha\hat{a}^\dagger - \alpha^*\hat{a}\right]$ to any harmonic oscillator eigenstate $|k\rangle$ [5]

$$|\alpha,k\rangle = \hat{D}(\alpha)|k\rangle \tag{B33}$$

where $k=0$ corresponds to the standard Glauber coherent states. Unlike the standard coherent state, the generalized coherent state are not eigenstates of the annihilation operator. It can be easily verified that

$$\hat{a}|\alpha,k\rangle = \sqrt{k}|\alpha,k-1\rangle + \alpha|\alpha,k\rangle, \tag{B34}$$

$$\hat{a}^\dagger|\alpha,k\rangle = \sqrt{k+1}|\alpha,k+1\rangle + \alpha^*|\alpha,k\rangle \tag{B35}$$

The generalized coherent state in the coordinate representation can be written as

$$\varphi_k(x,q,p) = 2^{-k/2}(k!)^{-1/2}\pi^{-1/4}H_k\left[(x-q)\right]e^{-(x-q)^2/2+ip(x-q/2)} \tag{B36}$$

where the two real variables, q,p are related to the complex variable α by

$$\alpha = \frac{q+ip}{\sqrt{2}} \tag{B37}$$

Correspondingly, the Hagedorn wave packet (see Eq. A20) is

$$\begin{aligned}\varphi_k[q,p,Q,P](x) =& 2^{-k/2}(k!)^{-1/2}\pi^{-1/4}\hbar^{-1/4}Q^{-(k+1)/2}(Q^*)^{k/2}\times H_k(\hbar^{-1/2}|Q|^{-1}(x-q)) \\ &\times\exp\left\{iPQ^{-1}(x-q)^2/(2\hbar) + ip(x-q)/\hbar\right\} \\ =& 2^{-k/2}(k!)^{-1/2}Q^{-k/2}(Q^*)^{k/2}\times H_k(\hbar^{-1/2}|Q|^{-1}(x-q))\varphi_0[q,p,Q,P](x).\end{aligned} \tag{B38}$$

By setting $Q=1$ and $P=i$ (they satisfy the relations $QP-PQ=0$ and $Q^*P-P^*Q=2i$), it can be easily verify that (setting $\hbar$=1)

$$\varphi_k[q,p,1,i](x) = 2^{-k/2}(k!)^{-1/2}\pi^{-1/4}H_k\left[(x-q)\right]\exp\left(-(x-q)^2/2 + ip(x-q)\right) \tag{B39}$$

Now it becomes clear that in fact the generalized coherent state is equivalent to the Hagedorn wave packet except of a phase factor $\exp\left(\frac{ipq}{2\hbar}\right)$ (compare Eqs. B36 and B39), which can be attributed to the definition of the standard coherent state (see Eq. B31 and Eq. B32). It is also possible to give the number state expansion of a generalized coherent state in the form

$$|\alpha,k\rangle = \frac{e^{-|\alpha|^2/2}}{(k!)^{1/2}}\sum_{n=0}^{\infty}(-1)^{k+n}(n!)^{1/2}L_n^{k-n}(|\alpha|^2)(\alpha^*)^{k-n}|n\rangle \tag{B40}$$

where $L_n^{k-n}(\cdot)$ being the associated Laguerre polynomials. For $k = 0$, $|\alpha, 0\rangle = |\alpha\rangle$, and we recover the formula for Glauber coherent states $|\alpha\rangle = e^{-|\alpha|^2/2}\sum_{n=0}^{\infty}\frac{\alpha^n}{\sqrt{n!}}|n\rangle$.

Next let us calculate the one-dimensional Franck-Condon integrals $\langle k, \alpha|\beta, j\rangle$ (overlap of the two generalized coherent states $|\alpha, k\rangle$ and $|\beta, j\rangle$) by recurrence formulae [6–8]. Indeed, we first observe that

$$\langle k, \alpha|\beta, j\rangle = \langle k|\hat{D}^\dagger(\alpha)\hat{D}(\beta)|j\rangle = \exp(\frac{\alpha^*\beta - \alpha\beta^*}{2})\langle k|D(\gamma)|j\rangle \tag{B41}$$

where $\gamma = \beta - \alpha$, then, using the generating function [9]

$$G(\rho^*, \tau) = \sum_{mn} D_{mn}(\gamma)\frac{\rho^{*m}\tau^n}{\sqrt{m!n!}} = \exp(-|\gamma|^2/2)\exp(\rho^*\tau + \rho^*\gamma - \tau\gamma^*) \tag{B42}$$

and applying well known mathematical procedures, it is possible to derive the recurrence formulae

$$\sqrt{m+1}D_{m+1,n} = \gamma D_{mn} + \sqrt{n}D_{m,n-1}, \tag{B43}$$

$$\sqrt{n+1}D_{m,n+1} = -\gamma^* D_{mn} + \sqrt{m}D_{m-1,n} \tag{B44}$$

with

$$D_{00}(\gamma) = \exp(-|\gamma|^2/2) \tag{B45}$$

which allow to compute FC integrals for any pair of quantum numbers. Alternatively, one can explicitly calculate $\langle k|\hat{D}(\gamma)|j\rangle$ as

$$\begin{aligned}\langle k|\hat{D}(\gamma)|j\rangle =& e^{-|\gamma|^2/2}\langle k|\exp(\gamma\hat{a}^\dagger)\exp(-\gamma^*\hat{a})|j\rangle \\ =& e^{-|\gamma|^2/2}\sum_{n=0}^{\min(k,j)}\frac{\gamma^{k-n}(-\gamma^*)^{j-n}}{(k-n)!(j-n)!n!}\sqrt{k!j!}\end{aligned} \tag{B46}$$

we thus have $D_{00}(\gamma) = \langle 0|\hat{D}(\gamma)|0\rangle = \exp(-|\gamma|^2/2)$. One can also easily derive above recurrence formulae.

$$\begin{aligned}\sqrt{m+1}D_{m+1,n} =& \sqrt{m+1}\langle m+1|\hat{D}(\gamma)|n\rangle = e^{-|\gamma|^2/2}\sqrt{m+1}\langle m+1|\exp(\gamma\hat{a}^\dagger)\exp(-\gamma^*\hat{a})|n\rangle \\ =& e^{-|\gamma|^2/2}\langle m|\hat{a}\exp(\gamma\hat{a}^\dagger)\exp(-\gamma^*\hat{a})|n\rangle \\ =& e^{-|\gamma|^2/2}\langle m|\left(\gamma\exp(\gamma\hat{a}^\dagger) + \exp(\gamma\hat{a}^\dagger)\hat{a}\right)\exp(-\gamma^*\hat{a})|n\rangle \\ =& e^{-|\gamma|^2/2}\gamma\langle m|\exp(\gamma\hat{a}^\dagger)\exp(-\gamma^*\hat{a})|n\rangle + e^{-|\gamma|^2/2}\sqrt{n}\langle m|\exp(\gamma\hat{a}^\dagger)\exp(-\gamma^*\hat{a})|n-1\rangle \\ =& \gamma D_{m,n} + \sqrt{n}D_{m,n-1}\end{aligned} \tag{B47}$$

and

$$\begin{aligned}\sqrt{n+1}D_{m,n+1} =& \sqrt{n+1}\langle m|\hat{D}(\gamma)|n+1\rangle = \sqrt{n+1}e^{-|\gamma|^2/2}\langle m|\exp(\gamma\hat{a}^\dagger)\exp(-\gamma^*\hat{a})|n+1\rangle \\ =& e^{-|\gamma|^2/2}\langle m|\exp(\gamma\hat{a}^\dagger)\exp(-\gamma^*\hat{a})\hat{a}^\dagger|n\rangle \\ =& e^{-|\gamma|^2/2}\langle m|\exp(\gamma\hat{a}^\dagger)\left(\hat{a}^\dagger\exp(-\gamma^*\hat{a}) - \gamma^*\exp(-\gamma^*\hat{a})\right)|n\rangle \\ =& e^{-|\gamma|^2/2}\sqrt{m}\langle m-1|\exp(\gamma\hat{a}^\dagger)\exp(-\gamma^*\hat{a})|n\rangle - \gamma^* e^{-|\gamma|^2/2}\langle m|\exp(\gamma\hat{a}^\dagger)\exp(-\gamma^*\hat{a})|n\rangle \\ =& -\gamma^* D_{m,n} + \sqrt{m}D_{m-1,n}\end{aligned} \tag{B48}$$

In the derivation, we have used following commutation relations

$$[\hat{a}, f(\hat{a}, \hat{a}^\dagger)] = \frac{\partial f}{\partial \hat{a}^\dagger}, \tag{B49}$$

$$\left[\hat{a}^\dagger, f(\hat{a}, \hat{a}^\dagger)\right] = -\frac{\partial f}{\partial \hat{a}} \tag{B50}$$

By introducing the associated Laguerre polynomials ($n, k \in \mathbb{N}, x \in \mathbb{C}$)

$$L_n^k(x) = \sum_{i=0}^{n} \frac{1}{i!} \begin{pmatrix} k+n \\ n-i \end{pmatrix} (-x)^i, \tag{B51}$$

Eq. B46 can be rewritten as

$$\langle k|\hat{D}(\gamma)|j\rangle = e^{-|\gamma|^2/2} \times \begin{cases} (-\gamma^*)^{j-k}\sqrt{\frac{k!}{j!}} L_k^{j-k}(|\gamma|^2) & \text{for } k \leq j \\ \gamma^{k-j}\sqrt{\frac{j!}{k!}} L_j^{k-j}\left(|\gamma|^2\right) & \text{for } j \leq k \end{cases} \tag{B52}$$

The associated Laguerre polynomials satisfy following relations

$$L_n^k(x) = L_n^{k+1}(x) - L_{n-1}^{k+1}(x), \tag{B53}$$

$$\frac{d^\alpha}{dx^\alpha} L_n^k(x) = (-1)^\alpha L_{n-\alpha}^{k+\alpha}(x) \tag{B54}$$

[1] G. A. Hagedorn. Raising and Lowering Operators for Semiclassical Wave Packets. Ann. Phys. **269**, 77-104 (1998).

[2] E. Faou, V. Gradinaru, and C. Lubich. Computing Semiclassical quantum dynamics with Hagedorn wavepackets. SIAM J. Sci. Comput. **31**, 3027-3041 (2009)

[3] R. Bourquin, Wavepacket propagation in D-Dimensional Non-adiabatic crossings. Master Thesis, ETH, Zurich.

[4] M. M. Nieto, Displaced and squeezed number states. Phys. Lett. A. **229**, 135-143 (1997).

[5] C. Monique, R. Didier, Coherent states and Applications in Mathematical Physics, Theoretical and Mathematical Physics, Springer, Netherlands, 2012.

[6] T. E. Sharp, H. M. Rosenstock, Frank-Condon factors for Polyatomic molecules. J. Chem. Phys. **41**, 3453(1964).

[7] R. Borrelli, A. Capobianco, A. Peluso, Franck-Condon factors-Computational approaches and recent developments. Can. J. Chem. **91**, 495 (2013).

[8] R. Borrelli, M. F. Gelin, The Generalized Coherent State ansatz: Application to quantum electron-vibrational dynamics. Chem. Phys. **481**, 91(2016).

[9] A. Perelomov, Generalized Coherent States and Their Applications, Text and Monographs in Physics, Springer-Verlag, 1986.