Formalizm QCO w Postaci LaTeXowej

Source:

https://github.com/4i4in/algebraic_trick_abusing_Wick/tree/main

Na podstawie analizy Normy_i_definicje_wersja3.pdf, gdzie QCO (Quantum-Classical Overlap) jest zdefiniowane nielatexowo jako artifact wynikający z nadużycia rotacji Wicka w algebrze hiperzłożonej, mieszający struktury SR z QM-like patologiami w imaginariach (prowadzący do nietrywialnych zer w zderzeniach), proponuję ładny LaTeXowy wzorek formalizujący QCO jako emergentną asymetrię w normie po twist. QCO mierzy gap między real hyperbolic (SR-like) a imag oscillatory (QM-like) reprezentacją, z invariant E^2, ale emergent m_eff z Im part:

$$QCO(v, \phi) = \sqrt{\|\mathrm{Re}(R(i\phi)v) - v'\|^2 + |\mathrm{Im}(R(i\phi)v)|^2}$$

gdzie $R(i\phi)$ to macierz twist, v wektor initial (np. photon), v' target (np. e), z E^2 = ||v||^2 =1 zachowanym.

gdzie R(iϕ)R(i\phi)R(iϕ) to macierz twist, vvv wektor initial (np. photon), v′v'v′ target (np. e), z E^2 = ||v||^2 =1 zachowanym.

Nielatexowa Wersja QCO Jako Funkcja Programu

Definicje zmiennych:

- v: np.array(16, dtype=complex) – wektor initial reprezentacji cząstki (np. [0+0j, 0+0j, ..., 1+0j, ...] dla photon).
- phi: float – kąt twist (w radianach, np. np.pi/4).
- R: np.array(16x16, dtype=complex) – macierz twist dla osi (i,j), np.eye(16, dtype=complex); R[i,i] = np.cos(phi); R[j,j] = np.cos(phi); R[i,j] = 1j * np.sin(phi); R[j,i] = 1j * np.sin(phi).
- v_prime: np.array(16, dtype=complex) – wektor target (np. [0.707+0j, 0+0j, 0.707+0j, ...] dla e normalized).
- qco: float – wynik gap, np.sqrt(np.linalg.norm(np.real(R @ v) - v_prime)**2 + np.abs(np.imag(R @ v))**2), z normalizacją ||R @ v|| =1 jeśli potrzeba (np.divide(R @ v, np.linalg.norm(R @ v))).

Długa funkcja (pseudokod Python): def qco(v, phi, i=0, j=2, v_prime=np.zeros(16, dtype=complex)): R = np.eye(16, dtype=complex) R[i,i] = np.cos(phi) R[j,j] = np.cos(phi) R[i,j] = 1j * np.sin(phi) R[j,i] = 1j * np.sin(phi) v_twist = R @ v v_twist /= np.linalg.norm(v_twist) if np.linalg.norm(v_twist) != 0 else 1 real_gap = np.linalg.norm(np.real(v_twist) - v_prime)**2 imag_abs = np.abs(np.imag(v_twist))**2 return np.sqrt(real_gap + imag_abs)

Formalizm Hiperobrotów w Postaci LaTeXowej

Hiperobrót to operacja R(ϕ) w planie (i,j) sedonionu 16D, zachowująca norm^2 = E^2, ale z twist i*ϕ dla QM-like oscillatory. Formalizm:

$$R_{ij}(\phi) = \begin{pmatrix} \ddots & & & \\ & \cos\phi & i\sin\phi & \\ & i\sin\phi & \cos\phi & \\ & & & \ddots \end{pmatrix}$$

z v' = R v, E^2 = ||v'||^2 = ||v||^2.

z v' = R v, E^2 = ||v'||^2 = ||v||^2.

Nielatexowa Wersja Hiperobrotów Jako Funkcja Programu

Definicje zmiennych:

- i, j: int – indeksy osi (0-15, np. i=0 real masa, j=2 imagin momentum).
- phi: float – kąt (radiany).
- v: np.array(16, dtype=complex) – wektor input.
- R: np.array(16x16, dtype=complex) – macierz, np.eye(16, dtype=complex); R[i,i] = np.cos(phi); R[j,j] = np.cos(phi); R[i,j] = 1j * np.sin(phi); R[j,i] = 1j * np.sin(phi).
- v_prime: np.array(16, dtype=complex) – output R @ v, normalized v_prime /= np.linalg.norm(v_prime).

Długa funkcja: def hyper_rotation(v, phi, i=0, j=2): R = np.eye(16, dtype=complex) R[i,i] = np.cos(phi) R[j,j] = np.cos(phi) R[i,j] = 1j * np.sin(phi) R[j,i] = 1j * np.sin(phi) v_prime = R @ v norm = np.linalg.norm(v_prime) v_prime /= norm if norm != 0 else 1 return v_prime

Formalizm Phi w Postaci LaTeXowej

Phi to kąt hiperobrotu, z twist i*ϕ generujący m_eff ~ sin(ϕ). Formalizm:

$$\phi = \arg\left(\sqrt{\dim_d - \delta}\right) \mod 2\pi$$

gdzie dim_d = 2^k minimal rep, δ rank massless osi.

gdzie dim_d = 2^k minimal rep, δ rank massless osi.

Nielatexowa Wersja Phi Jako Funkcja Programu

Definicje zmiennych:

- dim_d: int – 2**k (k=1 kompleks 2,2 kwaternion 4,3 oktonion 8,4 sedonion 16).
- delta: int – rank massless osi (1-4 dla nu/γ korektora).
- phi: float – np.angle(np.sqrt(dim_d - delta)) % (2 * np.pi).

Długa funkcja: def compute_phi(dim_d=16, delta=2): gap = dim_d - delta sqrt_gap = np.sqrt(gap) phi = np.angle(sqrt_gap) % (2 * np.pi) return phi

Formalizm Klucza, Procesu Szyfrowania i Deszyfrowania Jako Dokument Matematyczny

Niech $\mathcal{S}$ oznacza przestrzeń sedonionów 16D nad $\mathbb{C}$, z normą $\|v\|^2 = \sum_{l=0}^{15} |v_l|^2$. Klucz to sekwencja $\{\phi_k, (i_k, j_k)\}_{k=1}^{K}$, gdzie $\phi_k \in [0, 2\pi)$, $(i_k, j_k) \in \{0, \ldots, 15\}^2$, generowana rekurencyjnie $\phi_k = \phi_{k-1} + s \cdot k + \sin(k) \mod 2\pi$, z seed $s \in \mathbb{R}$.

Proces szyfrowania: Dla wiadomości m zmapowanej na wektor $v \in \mathcal{S}$, zastosuj łańcuch $v' = R_{i_K j_K}(i\phi_K) \circ \cdots \circ R_{i_1 j_1}(i\phi_1)v$, z normalizacją |v'| =1 po każdym. Ciphertext $c = v'$.

Proces deszyfrowania: Zastosuj odwrotny łańcuch $v'' = R_{i_1 j_1}(-i\phi_1) \circ \cdots \circ R_{i_K j_K}(-i\phi_K)c$, z normalizacją. Ze względu na nieasocjatywność, $v'' \approx v$ z asymetrią Im sq $\sim \sum \sin^2(\phi_k) > 0$, strata informacji algebraic.

Nielatexowa Wersja Klucza, Szyfrowania i Deszyfrowania Jako Funkcja Programu

Definicje zmiennych:

- K: int – długość chain (np. 20).
- s: float – seed klucza.
- phi_chain: list[float] – [phi_0] + [(prev + s * k + np.sin(k)) % (2*np.pi) for k in 1 to K].
- axes_seq: list[tuple(int,int)] – [(np.random.randint(0,16), np.random.randint(0,16)) for _ in range(K)].
- v: np.array(16, dtype=complex) – wektor wiadomości (np.frombuffer(message.encode(), dtype=np.complex128).reshape(-1) pad to 16).
- c: np.array(16, dtype=complex) – ciphertext po fwd chain.
- v_dec: np.array(16, dtype=complex) – decrypted approx z asym loss.

Długa funkcja szyfrowania: def encrypt(v, phi_chain, axes_seq, K=20): for k in range(K): i, j = axes_seq[k] phi = phi_chain[k] R = np.eye(16, dtype=complex) R[i,i] = np.cos(phi) R[j,j] = np.cos(phi) R[i,j] = 1j * np.sin(phi) R[j,i] = 1j * np.sin(phi) v = R @ v norm = np.linalg.norm(v) v /= norm if norm != 0 else 1 return v

Długa funkcja deszyfrowania: def decrypt(c, phi_chain, axes_seq, K=20): for k in range(K-1, -1, -1): i, j = axes_seq[k] phi = phi_chain[k] R = np.eye(16, dtype=complex) R[i,i] = np.cos(-phi) R[j,j] = np.cos(-phi) R[i,j] = 1j * np.sin(-phi) R[j,i] = 1j * np.sin(-phi) c = R @ c norm = np.linalg.norm(c) c /= norm if norm != 0 else 1 return c # approx, z imag loss ~ sum sin^2(phi_k) >0