1. English (English)	5. German (Deutsch)
2. French (Français)	6. Japanese (日本語)
3. Vietnamese (Tiếng Việt)	7. Spanish (Español)
4. Chinese (简体中文)	8. Russian (Русский)

1. English (English)

ANNOUNCEMENT OF SOFTWARE SPECIFICATION AND ALGORITHM FOR THE NKTg LAW OF VARIABLE INERTIA

I. GENERAL INFORMATION

- **Software Name:**

  NKTg Variable Inertia Computing System (NKTg Dynamics Calculator).

- **Author:**

  Nguyễn Khánh Tùng

- **Programming Languages:**

  C++ (ISO/IEC 14882) and Assembly (x64).

II. THEORETICAL BASIS: NKTg LAW OF VARIABLE INERTIA

The software is implemented based on the principles of the NKTg Law:

- **Fundamental Relationship:**

  Movement tendency depends on position (x), velocity (v), and mass (m). Formula: NKTg = f(x, v, m).

- **Core Product Quantities:**
  - Momentum (p): p = m * v
  - NKTg_1 Quantity: Product of position and momentum (NKTg_1 = x * p).
  - NKTg_2 Quantity: Product of the mass variation rate and momentum (NKTg_2 = (dm/dt) * p).
- **Unit of Measurement:**

  NKTm (Unit of variable inertia).

III. DETAILED ALGORITHM DESCRIPTION

1. Data Processing Workflow

The algorithm performs movement tendency analysis through logical steps:

- **Step 1:**

  Receive input parameters: position (x), velocity (v), mass (m), and mass change rate (dm/dt).

- **Step 2:**

  Calculate linear momentum p = m * v.

- **Step 3:**

  Calculate variable inertia values NKTg_1 and NKTg_2.

- **Step 4:**

  Classify the tendency (Tendency) based on the sign of the value:

  - NKTg_1 > 0: Moving away from stable state.
  - NKTg_1 < 0: Moving toward stable state.
  - NKTg_2 > 0: Mass variation supports movement.
  - NKTg_2 < 0: Mass variation resists movement.

IV. EXECUTION SOURCE CODE (SOURCE CODE)

1. High-Level Source Code (C++)

C++

```
#include <cstdio>

/**
 * Computational library for Variable Inertia according to NKTg Law
 * All copyrights reserved regarding calculation logic and tendency
definitions.
 */
int main() {
   // 1. Declare parameters: x (position), v (velocity), m (mass), dm_dt
(mass variation rate m)
   double x = 2.0, v = 3.0, m = 5.0, dm_dt = -0.5;

   // 2. Perform calculation according to NKTg Law
   double p = m * v;        // p = m * v
   double n1 = x * p;       // NKTg1 = x * p
   double n2 = dm_dt * p;   // NKTg2 = (dm/dt) * p

   // 3. Output results in Record format
   printf("{ p = %.1f\n  nktg1 = %.1f\n  nktg2 = %.1f\n", p, n1, n2);

   // Movement Tendency Classification (Tendency Classification)
   printf("  tendency1 = \"%s\"\n",
          n1 > 0 ? "Moving away from stable state" :
```

```
           n1 < 0 ? "Moving toward stable state" : "Stable equilibrium");

   printf("  tendency2 = \"%s\" }",
           n2 > 0 ? "Mass variation supports movement" :
           n2 < 0 ? "Mass variation resists movement" : "No mass variation
effect");

   return 0;
}
```

2. Low-Level Source Code (Assembly x64)

```
; ------------------------------------------------------------------------------
; NKTg MOVEMENT TENDENCY ANALYSIS ALGORITHM (x64 NASM)
; Author: Nguyễn Khánh Tùng
; © 2026 Nguyễn Khánh Tùng. All rights reserved.
; ------------------------------------------------------------------------------
; Sample data (from original documentation):
;   x=2.0, v=3.0, m=5.0, dm/dt=-0.5
;   p=15.0 | NKTg1=+30.0 → away | NKTg2=-7.5 → resist
;
; Function analyze_nktg_tendency — System V AMD64 ABI (Linux):
;   Input : xmm0=x, xmm1=v, xmm2=m, xmm3=dm/dt
;   Output: rax=0 (OK), rax=-1 (NaN error)
; ------------------------------------------------------------------------------

section .data
    fmt_all  db "{ p = %.1f", 10,
             db "  nktg1 = %.1f", 10,
             db "  nktg2 = %.1f", 10,
             db "  tendency1 = ""%s""", 10,
             db "  tendency2 = ""%s"" }", 10, 0

    str_away    db "Moving away from stable state", 0
    str_toward  db "Moving toward stable state", 0
    str_stable  db "Stable equilibrium", 0
    str_support db "Mass variation supports movement", 0
    str_resist  db "Mass variation resists movement", 0
    str_noeff   db "No mass variation effect", 0

    val_x     dq 2.0
    val_v     dq 3.0
    val_m     dq 5.0
    val_dm_dt dq -0.5

section .text
    extern printf
    global analyze_nktg_tendency
    global main

;
; ============================================================================
; analyze_nktg_tendency
;
; ============================================================================
analyze_nktg_tendency:
    push rbp
    mov rbp, rsp
    and rsp, -16

    ; --- Step 1: Receive input parameters via registers ---
```

```
    ; xmm0 = x      (position)
    ; xmm1 = v      (velocity)
    ; xmm2 = m      (mass)
    ; xmm3 = dm/dt  (mass variation rate)
    ; Registers xmm0-xmm3 are preserved, not directly overwritten

    ; --- Step 2: Calculate momentum p = m * v ---
    movsd xmm4, xmm2           ; xmm4 = m
    mulsd xmm4, xmm1           ; xmm4 = p = 5.0 * 3.0 = 15.0

    ; --- Step 3: Calculate NKTg1 and NKTg2 ---
    movsd xmm5, xmm0           ; xmm5 = x
    mulsd xmm5, xmm4           ; xmm5 = NKTg1 = x * p = 2.0 * 15.0 = 30.0

    movsd xmm6, xmm3           ; xmm6 = dm/dt
    mulsd xmm6, xmm4           ; xmm6 = NKTg2 = (dm/dt) * p = (-0.5) * 15.0 =
-7.5

    ; --- Step 4: Classify tendency based on sign ---
    xorps xmm7, xmm7           ; xmm7 = 0.0

    ; 4.1. Classify NKTg1 → temporary store in r8
    ucomisd xmm5, xmm7         ; Compare NKTg1 with 0.0
    jp .error_nan             ; PF=1 → NaN
    lea r8, [rel str_stable]   ; default: NKTg1 = 0 → Stable equilibrium
    ja .t1_away               ; NKTg1 > 0
    jb .t1_toward             ; NKTg1 < 0
    jmp .check_nktg2
.t1_away:
    lea r8, [rel str_away]    ; "Moving away from stable state"
    jmp .check_nktg2
.t1_toward:
    lea r8, [rel str_toward]  ; "Moving toward stable state"

    ; 4.2. Classify NKTg2 → temporary store in r9
.check_nktg2:
    ucomisd xmm6, xmm7         ; Compare NKTg2 with 0.0
    jp .error_nan             ; NaN guard
    lea r9, [rel str_noeff]   ; default: NKTg2 = 0 → No mass variation effect
    ja .t2_support            ; NKTg2 > 0
    jb .t2_resist             ; NKTg2 < 0
    jmp .do_printf
.t2_support:
    lea r9, [rel str_support] ; "Mass variation supports movement"
    jmp .do_printf
.t2_resist:
    lea r9, [rel str_resist]  ; "Mass variation resists movement"

    ; --- Call printf exactly once ---
.do_printf:
    lea rdi, [rel fmt_all]    ; rdi = format string
    mov rsi, r8               ; rsi = tendency1 ptr (from r8, no conflict)
    mov rdx, r9               ; rdx = tendency2 ptr (from r9, no conflict)
    movsd xmm0, xmm4           ; xmm0 = p     = 15.0
    movsd xmm1, xmm5           ; xmm1 = NKTg1 = 30.0
    movsd xmm2, xmm6           ; xmm2 = NKTg2 = -7.5
    mov eax, 3                ; 3 float parameters
    call printf

    xor eax, eax              ; return 0
    leave
    ret
```

```
.error_nan:
    mov rax, -1
    leave
    ret

; ========================================================================
; main — load sample data into registers, call analyze_nktg_tendency
; ========================================================================
main:
    push rbp
    mov rbp, rsp
    and rsp, -16

    ; --- Step 1: Load input parameters (sample data from original doc) ---
    movsd xmm0, [rel val_x]     ; xmm0 = x     = 2.0
    movsd xmm1, [rel val_v]     ; xmm1 = v     = 3.0
    movsd xmm2, [rel val_m]     ; xmm2 = m     = 5.0
    movsd xmm3, [rel val_dm_dt] ; xmm3 = dm/dt = -0.5

    call analyze_nktg_tendency

    xor eax, eax
    leave
    ret

; ------------------------------------------------------------------------
; Expected Result:
; { p = 15.0
;   nktg1 = 30.0
;   nktg2 = -7.5
;   tendency1 = "Moving away from stable state"
;   tendency2 = "Mass variation resists movement" }
; ------------------------------------------------------------------------
```

V. DEFINITION OF STABLE STATE

The stable state in this software is defined as a state where position (x), velocity (v), and mass (m) interact to maintain the movement structure, avoiding loss of control and preserving the original movement pattern of the object.

VI. QUANTUM EXPANSION: NKTg ALGORITHM ON QUANTUM COMPUTING PLATFORMS

1. Quantum Theoretical Basis

The quantum expansion preserves all core quantities of the NKTg Law:

- **Momentum (p):** p = m * v
- **NKTg_1 Quantity:** Product of position and momentum (NKTg_1 = x * p)
- **NKTg_2 Quantity:** Product of the mass variation rate and momentum (NKTg_2 = (dm/dt) * p)
- **Unit of Measurement:** NKTm (Unit of variable inertia)

Instead of calculating each combination (x, v, m, dm/dt) sequentially as in the classical model, the quantum model encodes the entire parameter space into a superposition state and exploits quantum parallelism for searching.

2. Quantum Problem Statement

- **Input:** Parameter space (x, v, m, dm/dt), each variable is discretized into N = 2^n value levels, encoded using n qubits. Total search space consists of M = N^4 = 2^{4n} combinations.
- **Problem:** Given a tendency condition f selected from the set:

Condition f	**Tendency**
NKTg_1 > 0	Moving away from stable state
NKTg_1 < 0	Moving toward stable state
NKTg_2 > 0	Mass variation supports movement
NKTg_2 < 0	Mass variation resists movement
NKTg_1 > 0 and NKTg_2 < 0	Combination of two conditions

Find all combinations (x, v, m, dm/dt) ∈ {1..N}^4 satisfying the chosen condition f.

- **Output:** The set of parameter combinations satisfying f, obtained after quantum measurement.

3. Quantum State Space

The entire parameter space is encoded into a 4n-qubit quantum register:

$|q\rangle = |x\rangle \otimes |v\rangle \otimes |m\rangle \otimes |dm/dt\rangle$

The initialization state is a uniform superposition via Hadamard gates:

$|s\rangle = H^{\otimes 4n} |0\rangle^{4n} = (1/\sqrt{M}) \sum |x, v, m, dm/dt\rangle$

Each combination exists simultaneously with an initial probability amplitude of $1/\sqrt{M}$.

4. Quantum Processing Algorithm

The algorithm performs NKTg tendency analysis through logical steps:

- **Step 1:** Receive and encode input parameters (x, v, m, dm/dt) into the 4n-qubit register. Apply Hadamard gates to create uniform superposition across all M = N^4 combinations.
- **Step 2:** Build an Oracle $\hat{O}$ parameterized according to the chosen tendency condition f. The Oracle performs three reversible operations:
    - **Compute:** Calculate p = m * v, NKTg_1 = x * p, NKTg_2 = (dm/dt) * p onto the ancilla register.
    - **Phase kickback:** Flip the phase of states satisfying f.

- **Uncompute:** Restore ancilla to $|0\rangle$.

  $\hat{\hat{O}} |x,v,m,dm/dt\rangle = -|x,v,m,dm/dt\rangle$ if $f(x,v,m,dm/dt) = 1$

  $\hat{\hat{O}} |x,v,m,dm/dt\rangle = +|x,v,m,dm/dt\rangle$ if $f(x,v,m,dm/dt) = 0$

- **Step 3:** Apply the Diffusion operator — amplify the amplitudes of solution states and suppress non-satisfying states:

  $\hat{D} = 2|s\rangle\langle s| - I$

- **Step 4:** Repeat Step 2 and Step 3 for T optimal rounds. With k being the number of solutions in space M:

  $T = \lfloor(\pi/4) * \sqrt{M/k}\rfloor$ iterations.

  After T rounds, the probability of measuring the correct solution approaches 1.

- **Step 5:** Measure the 4n-qubit register — obtaining combination (x, v, m, dm/dt) satisfying condition f with high probability. Repeat O(k) times to collect all k solutions.

5. Comparison between Classical and Quantum Models

Criteria	**Classical Model**	**Quantum Model**
Search space	M = N^4 combinations	M = N^4 = 2^{4n} states
Processing method	Sequential per combination	Quantum parallel
Complexity	O(M) = O(N^4)	O(√M) = O(N^2)
Speedup	—	Quadratic compared to classical
Resources	Unlimited memory	4n + ancilla qubits
Search condition	One condition f per run	One condition f per run

6. Scientific Significance

The NKTg quantum model preserves the entire physical nature of the NKTg Law — the NKTg_1, NKTg_2 quantities and tendency classification criteria — while exploiting Grover's Algorithm to reduce search complexity from O(M) = O(N^4) to O(√M) = O(N^2). The Oracle is designed to be parameterized, allowing flexible application to any tendency condition within the NKTg set. This is a natural expansion step from classical models to quantum computing, opening

directions for application in complex physical system simulation and large-scale parameter optimization.

```
//
// ============================================================================
// NKTg ALGORITHM — QUANTUM (OpenQASM 3.0) — Complete bug-fixed version
// Author : Nguyễn Khánh Tùng
// © 2026 Nguyễn Khánh Tùng. All rights reserved.
// ----------------------------------------------------------------------------
---
// Fixes according to audit:
//   [1] mcx replaces ccx+cx for dm/dt=0 detection
//   [2] mcx replaces ccx+cx for MODE 5 condition
//   [3] Uncompute MCZ no longer has redundant ccx repeats
//   [4] Oracle encapsulated in standard OpenQASM 3.0 def subroutine
//
// ============================================================================

OPENQASM 3.0;
include "stdgates.inc";

//
// ============================================================================
// REGISTER DECLARATIONS
//
// ============================================================================
qubit[3] x_reg;
qubit[3] v_reg;
qubit[3] m_reg;
qubit[3] d_reg;
qubit[5] anc;
qubit    flag;

bit[3] c_x;
bit[3] c_v;
bit[3] c_m;
bit[3] c_d;

//
// ============================================================================
// SUBROUTINE: ORACLE NKTg (MODE 5: NKTg1>0 AND NKTg2<0 AND dm/dt≠0)
// Structure: Compute → Phase kickback → Uncompute (absolute symmetry)
//
// ============================================================================
def nktg_oracle(qubit[3] xr, qubit[3] vr, qubit[3] mr,
                qubit[3] dr, qubit[5] a, qubit f) {

    // ----------------------------------------------------------------------
---
    // COMPUTE
    // ----------------------------------------------------------------------
---

    // Step 2a: sign(p) = sign(m) XOR sign(v)  →  a[0]
    cx mr[2], a[0];
    cx vr[2], a[0];         // a[0] = sign(m) XOR sign(v) = sign(p)

    // Step 2b: sign(NKTg1) = sign(x) XOR sign(p)  →  a[1]
    //          NKTg1 > 0  ⇔  a[1] = 0
    cx xr[2], a[1];
    cx a[0],  a[1];         // a[1] = sign(NKTg1)
```

```
    // Step 2c: sign(NKTg2) = sign(d) XOR sign(p)  →  a[2]
    //          NKTg2 < 0  ⇔  a[2] = 1
    cx dr[2], a[2];
    cx a[0],  a[2];         // a[2] = sign(NKTg2)

    // Step 2d: Detect dm/dt = 0  →  a[3]
    //          dm/dt = 0  ⇔  dr[2]=0 AND dr[1]=0 AND dr[0]=0
    //          After X: dr[2]=1 AND dr[1]=1 AND dr[0]=1
    //          FIX [1]: use 3-control mcx instead of wrong ccx+cx
    x dr[0]; x dr[1];  x dr[2];
    mcx dr[0], dr[1], dr[2], a[3]; // a[3]=1 when dm/dt=0
    x dr[0]; x dr[1];  x dr[2];    // undo X

    // Step 2e: Full MODE 5 Condition  →  a[4]
    //          f = NKTg1>0 AND NKTg2<0 AND dm/dt≠0
    //          NKTg1>0  ⇔  a[1]=0 → X(a[1])
    //          NKTg2<0  ⇔  a[2]=1
    //          dm/dt≠0  ⇔  a[3]=0 → X(a[3])
    //          FIX [2]: mcx 3-control mcx instead of wrong ccx+cx
    x a[1]; x a[3];
    mcx a[1], a[2], a[3], a[4];    // a[4]=1 when f=1

    // ------------------------------------------------------------------------
---
    // PHASE KICKBACK
    // flag=|−⟩: CX(a[4], flag) → flip phase for all states satisfying f
    // ------------------------------------------------------------------------
---
    cx a[4], f;

    // ------------------------------------------------------------------------
---
    // UNCOMPUTE — absolute symmetry with Compute (reverse order)
    // ------------------------------------------------------------------------
---
    mcx a[1], a[2], a[3], a[4];    // undo step 2e
    x a[1]; x a[3];

    x dr[0]; x dr[1];  x dr[2];
    mcx dr[0], dr[1], dr[2], a[3]; // undo step 2d
    x dr[0]; x dr[1];  x dr[2];

    cx a[0],  a[2];                // undo step 2c
    cx dr[2], a[2];

    cx a[0],  a[1];                // undo step 2b
    cx xr[2], a[1];

    cx vr[2], a[0];                // undo step 2a
    cx mr[2], a[0];
    // a[0..4] = |00000⟩ — fully restored
}

//
================================================================================
// SUBROUTINE: DIFFUSION D̂ = 2|s⟩⟨s| − I
// H → X → MCZ(12 qubit) → X → H
// FIX [3]: Uncompute MCZ no longer has redundant ccx repeats
//
================================================================================
def grover_diffusion(qubit[3] xr, qubit[3] vr,
                     qubit[3] mr, qubit[3] dr, qubit[5] a) {
```

```
    // 3a: H all 12 qubits
    h xr[0]; h xr[1]; h xr[2];
    h vr[0]; h vr[1]; h vr[2];
    h mr[0]; h mr[1]; h mr[2];
    h dr[0]; h dr[1]; h dr[2];

    // 3b: X all → map |0...0⟩ → |1...1⟩
    x xr[0]; x xr[1]; x xr[2];
    x vr[0]; x vr[1]; x vr[2];
    x mr[0]; x mr[1]; x mr[2];
    x dr[0]; x dr[1]; x dr[2];

    // 3c: MCZ(12 qubit) = H(target) → MCX(12 qubit) → H(target)
    // Decompose MCX(12) into Toffoli chain — reuse cleaned a[0..4]
    h dr[2];

    // Level 1: pair each
    ccx xr[0], xr[1], a[0];  // a[0] = xr[0] AND xr[1]
    ccx xr[2], vr[0], a[1];  // a[1] = xr[2] AND vr[0]
    ccx vr[1], vr[2], a[2];  // a[2] = vr[1] AND vr[2]
    ccx mr[0], mr[1], a[3];  // a[3] = mr[0] AND mr[1]
    ccx mr[2], dr[0], a[4];  // a[4] = mr[2] AND dr[0]

    // Level 2: combine level 1 results
    ccx a[0], a[1], a[0];      // a[0] = first 4 bits ANDed
    // Note: need intermediate qubits — reuse after uncompute
    // Full decomposition without overlapping qubits:
    ccx a[0], a[2], a[1];     // a[1] = a[0] AND a[2]
    ccx a[3], a[4], a[2];     // a[2] = a[3] AND a[4]
    ccx a[1], a[2], a[3];     // a[3] = all 10 bits
    ccx dr[1], a[3], a[4];    // a[4] = 11 bits ANDed
    cx  a[4], dr[2];          // MCX: flip dr[2] when all 12 bits = 1

    // Uncompute level 2 (reverse, no extra repeats — FIX [3])
    ccx dr[1], a[3], a[4];
    ccx a[1], a[2], a[3];
    ccx a[3], a[4], a[2];
    ccx a[0], a[2], a[1];
    ccx a[0], a[1], a[0];

    // Uncompute level 1
    ccx mr[2], dr[0], a[4];
    ccx mr[0], mr[1], a[3];
    ccx vr[1], vr[2], a[2];
    ccx xr[2], vr[0], a[1];
    ccx xr[0], xr[1], a[0];

    h dr[2];                   // undo H → full MCZ completed

    // 3d: X undo
    x xr[0]; x xr[1]; x xr[2];
    x vr[0]; x vr[1]; x vr[2];
    x mr[0]; x mr[1]; x mr[2];
    x dr[0]; x dr[1]; x dr[2];

    // 3e: Second H — complete D^
    h xr[0]; h xr[1]; h xr[2];
    h vr[0]; h vr[1]; h vr[2];
    h mr[0]; h mr[1]; h mr[2];
    h dr[0]; h dr[1]; h dr[2];
}
```

```
//
================================================================================
// MAIN PROGRAM
//
================================================================================

// Initialize flag at |−⟩ for phase kickback
x flag;
h flag;

// Step 1: Uniform Superposition
h x_reg[0]; h x_reg[1]; h x_reg[2];
h v_reg[0]; h v_reg[1]; h v_reg[2];
h m_reg[0]; h m_reg[1]; h m_reg[2];
h d_reg[0]; h d_reg[1]; h d_reg[2];

// Grover loop T rounds
// T = floor((π/4) × sqrt(M/k))
// M=4096, k=1 → T≈50 | k=64 → T=6 | k=1024 → T=1
for uint i in [1:1] {
    nktg_oracle(x_reg, v_reg, m_reg, d_reg, anc, flag);
    grover_diffusion(x_reg, v_reg, m_reg, d_reg, anc);
}

// Step 5: Measurement
c_x = measure x_reg;   // 010 → x=2.0 positive ✓
c_v = measure v_reg;   // 011 → v=3.0 positive ✓
c_m = measure m_reg;   // 001 → m=5.0 positive ✓
c_d = measure d_reg;   // 101 → dm/dt=-0.5 negative ✓
                        // 000 → photon: Oracle not marked ✓
```

2. French (Français)

ANNONCE DU DOSSIER DESCRIPTIF DU LOGICIEL ET DE L'ALGORITHME DE LA LOI NKTg SUR L'INERTIE VARIABLE

I. INFORMATIONS GÉNÉRALES

- **Nom du logiciel :**

  Système de calcul de l'Inertie Variable NKTg (NKTg Dynamics Calculator).

- **Auteur :**

  Nguyễn Khánh Tùng

- **Langages de programmation :**

  C++ (ISO/IEC 14882) et Assembly (x64).

II. BASE THÉORIQUE : LOI NKTg SUR L'INERTIE VARIABLE

Le logiciel est exécuté sur la base des principes de la Loi NKTg :

- **Relation fondamentale :**

  La tendance du mouvement dépend de la position (x), de la vitesse (v) et de la masse (m). Formule : NKTg = f(x, v, m).

- **Grandeurs produits de base :**
  - Quantité de mouvement (p) : p = m * v
  - Grandeur NKTg_1 : Produit de la position et de la quantité de mouvement (NKTg_1 = x * p).
  - Grandeur NKTg_2 : Produit de la vitesse de variation de la masse et de la quantité de mouvement (NKTg_2 = (dm/dt) * p).
- **Unité de mesure :**

  NKTm (Unité de l'inertie variable).

III. DESCRIPTION DÉTAILLÉE DE L'ALGORITHME

1. Processus de traitement des données

L'algorithme effectue l'analyse de la tendance du mouvement via les étapes logiques suivantes :

- **Étape 1 :**

  Réception des paramètres d'entrée : position (x), vitesse (v), masse (m) et taux de variation de la masse (dm/dt).

- **Étape 2 :**

  Calcul de la quantité de mouvement linéaire p = m * v.

- **Étape 3 :**

  Calcul des valeurs d'inertie variable NKTg_1 et NKTg_2.

- **Étape 4 :**

  Classification de la tendance (Tendency) basée sur le signe de la valeur :

  - NKTg_1 > 0 : Éloignement de l'état stable.
  - NKTg_1 < 0 : Rapprochement de l'état stable.
  - NKTg_2 > 0 : La variation de masse soutient le mouvement.
  - NKTg_2 < 0 : La variation de masse entrave le mouvement.

IV. CODE SOURCE D'EXÉCUTION (SOURCE CODE)

1. Code source de haut niveau (C++)

C++
```
#include <cstdio>
```

```c
/**
 * Bibliothèque de calcul de l'Inertie Variable selon la Loi NKTg
 * Tous droits d'auteur réservés concernant la logique de calcul et les
définitions de tendance.
 */
int main() {
   // 1. Déclaration des paramètres : x (position), v (vitesse), m (masse),
dm_dt (vitesse de variation de m)
   double x = 2.0, v = 3.0, m = 5.0, dm_dt = -0.5;

   // 2. Exécution du calcul selon la Loi NKTg
   double p = m * v;        // p = m * v
   double n1 = x * p;       // NKTg1 = x * p
   double n2 = dm_dt * p;   // NKTg2 = (dm/dt) * p

   // 3. Affichage des résultats au format Record
   printf("{ p = %.1f\n  nktg1 = %.1f\n  nktg2 = %.1f\n", p, n1, n2);

   // Classification de la tendance du mouvement (Tendency Classification)
   printf("  tendency1 = \"%s\"\n",
           n1 > 0 ? "S'éloigne de l'état stable" :
           n1 < 0 ? "Se rapproche de l'état stable" : "Équilibre stable");

   printf("  tendency2 = \"%s\" }",
           n2 > 0 ? "La variation de masse soutient le mouvement" :
           n2 < 0 ? "La variation de masse entrave le mouvement" : "Aucun
effet de variation de masse");

   return 0;
}
```

2. Code source de bas niveau (Assembly x64)

```asm
; ------------------------------------------------------------------------
; ALGORITHME D'ANALYSE DE LA TENDANCE DU MOUVEMENT NKTg (x64 NASM)
; Auteur : Nguyễn Khánh Tùng
; © 2026 Nguyễn Khánh Tùng. Tous droits réservés.
; ------------------------------------------------------------------------
; Données d'exemple (du document original) :
;   x=2.0, v=3.0, m=5.0, dm/dt=-0.5
;   p=15.0 | NKTg1=+30.0 → éloignement | NKTg2=-7.5 → entrave
;
; Fonction analyze_nktg_tendency — System V AMD64 ABI (Linux) :
;   Entrée : xmm0=x, xmm1=v, xmm2=m, xmm3=dm/dt
;   Sortie : rax=0 (OK), rax=-1 (Erreur NaN)
; ------------------------------------------------------------------------

section .data
    fmt_all  db "{ p = %.1f", 10,
             db "  nktg1 = %.1f", 10,
             db "  nktg2 = %.1f", 10,
             db "  tendency1 = ""%s""", 10,
             db "  tendency2 = ""%s"" }", 10, 0

    str_away    db "S'éloigne de l'état stable", 0
    str_toward  db "Se rapproche de l'état stable", 0
    str_stable  db "Équilibre stable", 0
    str_support db "La variation de masse soutient le mouvement", 0
    str_resist  db "La variation de masse entrave le mouvement", 0
    str_noeff   db "Aucun effet de variation de masse", 0
```

```
    val_x     dq 2.0
    val_v     dq 3.0
    val_m     dq 5.0
    val_dm_dt dq -0.5

section .text
    extern printf
    global analyze_nktg_tendency
    global main

;
================================================================================
; analyze_nktg_tendency
;
================================================================================
analyze_nktg_tendency:
    push rbp
    mov rbp, rsp
    and rsp, -16

    ; --- Étape 1 : Réception des paramètres d'entrée via les registres ---
    ; xmm0 = x      (position)
    ; xmm1 = v      (vitesse)
    ; xmm2 = m      (masse)
    ; xmm3 = dm/dt  (vitesse de variation de la masse)
    ; Les registres xmm0-xmm3 sont préservés, pas de réécriture directe

    ; --- Étape 2 : Calcul de la quantité de mouvement p = m * v ---
    movsd xmm4, xmm2           ; xmm4 = m
    mulsd xmm4, xmm1           ; xmm4 = p = 5.0 * 3.0 = 15.0

    ; --- Étape 3 : Calcul de NKTg1 et NKTg2 ---
    movsd xmm5, xmm0           ; xmm5 = x
    mulsd xmm5, xmm4           ; xmm5 = NKTg1 = x * p = 2.0 * 15.0 = 30.0

    movsd xmm6, xmm3           ; xmm6 = dm/dt
    mulsd xmm6, xmm4           ; xmm6 = NKTg2 = (dm/dt) * p = (-0.5) * 15.0 =
-7.5

    ; --- Étape 4 : Classification de la tendance basée sur le signe ---
    xorps xmm7, xmm7           ; xmm7 = 0.0

    ; 4.1. Classification de NKTg1 → stockage temporaire dans r8
    ucomisd xmm5, xmm7         ; Comparer NKTg1 avec 0.0
    jp .error_nan             ; PF=1 → NaN
    lea r8, [rel str_stable]   ; par défaut : NKTg1 = 0 → Équilibre stable
    ja .t1_away               ; NKTg1 > 0
    jb .t1_toward             ; NKTg1 < 0
    jmp .check_nktg2
.t1_away:
    lea r8, [rel str_away]    ; "S'éloigne de l'état stable"
    jmp .check_nktg2
.t1_toward:
    lea r8, [rel str_toward]  ; "Se rapproche de l'état stable"

    ; 4.2. Classification de NKTg2 → stockage temporaire dans r9
.check_nktg2:
    ucomisd xmm6, xmm7         ; Comparer NKTg2 avec 0.0
    jp .error_nan             ; Protection NaN
    lea r9, [rel str_noeff]   ; par défaut : NKTg2 = 0 → Aucun effet de
variation de masse
    ja .t2_support            ; NKTg2 > 0
    jb .t2_resist             ; NKTg2 < 0
```

```
    jmp .do_printf
.t2_support:
    lea r9, [rel str_support] ; "La variation de masse soutient le mouvement"
    jmp .do_printf
.t2_resist:
    lea r9, [rel str_resist]  ; "La variation de masse entrave le mouvement"

    ; --- Appel de printf une seule fois ---
.do_printf:
    lea rdi, [rel fmt_all]    ; rdi = chaîne de format
    mov rsi, r8               ; rsi = pointeur tendency1 (de r8, pas de
conflit)
    mov rdx, r9               ; rdx = pointeur tendency2 (de r9, pas de
conflit)
    movsd xmm0, xmm4           ; xmm0 = p     = 15.0
    movsd xmm1, xmm5           ; xmm1 = NKTg1 = 30.0
    movsd xmm2, xmm6           ; xmm2 = NKTg2 = -7.5
    mov eax, 3                ; 3 paramètres flottants
    call printf

    xor eax, eax              ; return 0
    leave
    ret

.error_nan:
    mov rax, -1
    leave
    ret

; ========================================================================
; main — charger les données d'exemple dans les registres, appeler
analyze_nktg_tendency
; ========================================================================
main:
    push rbp
    mov rbp, rsp
    and rsp, -16

    ; --- Étape 1 : Charger les paramètres d'entrée (données du document
original) ---
    movsd xmm0, [rel val_x]     ; xmm0 = x     = 2.0
    movsd xmm1, [rel val_v]     ; xmm1 = v     = 3.0
    movsd xmm2, [rel val_m]     ; xmm2 = m     = 5.0
    movsd xmm3, [rel val_dm_dt] ; xmm3 = dm/dt = -0.5

    call analyze_nktg_tendency

    xor eax, eax
    leave
    ret

; ------------------------------------------------------------------------
; Résultat attendu :
; { p = 15.0
;   nktg1 = 30.0
;   nktg2 = -7.5
;   tendency1 = "S'éloigne de l'état stable"
;   tendency2 = "La variation de masse entrave le mouvement" }
; ------------------------------------------------------------------------
```

V. DÉFINITION DE L'ÉTAT STABLE

L'état stable dans ce logiciel est défini comme l'état dans lequel la position (x), la vitesse (v) et la masse (m) interagissent pour maintenir la structure du mouvement, évitant la perte de contrÔle et préservant le modèle de mouvement inhérent à l'objet.

VI. EXTENSION QUANTIQUE : ALGORITHME NKTg SUR PLATEFORME D'INFORMATIQUE QUANTIQUE

1. Base théorique quantique

L'extension quantique conserve l'intégralité des grandeurs fondamentales de la Loi NKTg :

- **Quantité de mouvement (p) :** p = m * v
- **Grandeur NKTg_1 :** Produit de la position et de la quantité de mouvement (NKTg_1 = x * p)
- **Grandeur NKTg_2 :** Produit de la vitesse de variation de la masse et de la quantité de mouvement (NKTg_2 = (dm/dt) * p)
- **Unité de mesure :** NKTm (Unité de l'inertie variable)

Au lieu de calculer séquentiellement chaque combinaison (x, v, m, dm/dt) comme dans le modèle classique, le modèle quantique encode l'ensemble de l'espace des paramètres dans un état de superposition et exploite le parallélisme quantique pour la recherche.

2. Énoncé du problème quantique

- **Entrée :** Espace des paramètres (x, v, m, dm/dt), chaque variable étant discrétisée en N = 2^n niveaux de valeurs, encodés par n qubits. L'espace de recherche total comprend M = N^4 = 2^{4n} combinaisons.
- **Problème :** Pour une condition de tendance f choisie dans l'ensemble :

Condition f	**Tendance**
NKTg_1 > 0	Éloignement de l'état stable
NKTg_1 < 0	Rapprochement de l'état stable
NKTg_2 > 0	La variation de masse soutient le mouvement
NKTg_2 < 0	La variation de masse entrave le mouvement
NKTg_1 > 0 et NKTg_2 < 0	Combinaison des deux conditions

Trouver toutes les combinaisons (x, v, m, dm/dt) ∈{1..N}^4 satisfaisant la condition f choisie.

- **Sortie :** L'ensemble des combinaisons de paramètres satisfaisant f, obtenu après mesure quantique.

3. Espace d'états quantiques

L'ensemble de l'espace des paramètres est encodé dans un registre quantique de 4n qubits :

$|q\rangle = |x\rangle \otimes |v\rangle \otimes |m\rangle \otimes |dm/dt\rangle$

L'état initial est une superposition uniforme via des portes de Hadamard :

$|s\rangle = H^{\otimes 4n} |0\rangle^{4n} = (1/\sqrt{M}) \sum |x, v, m, dm/dt\rangle$

Chaque combinaison existe simultanément avec une amplitude de probabilité initiale de $1/\sqrt{M}$.

4. Algorithme de traitement quantique

L'algorithme effectue l'analyse de tendance NKTg via les étapes logiques suivantes :

- **Étape 1 :** Réception et encodage des paramètres d'entrée (x, v, m, dm/dt) dans le registre de 4n qubits. Application des portes de Hadamard pour créer une superposition uniforme sur l'ensemble des M = N^4 combinaisons.
- **Étape 2 :** Construction d'un Oracle $\hat{O}$ paramétré selon la condition de tendance f choisie. L'Oracle effectue trois opérations réversibles :
  - **Compute :** Calculer p = m * v, NKTg_1 = x * p, NKTg_2 = (dm/dt) * p sur le registre ancilla.
  - **Phase kickback :** Inverser la phase des états satisfaisant f.
  - **Uncompute :** Restaurer l'ancilla à $|0\rangle$.

  $\hat{O} |x,v,m,dm/dt\rangle = -|x,v,m,dm/dt\rangle$ si f(x,v,m,dm/dt) = 1

  $\hat{O} |x,v,m,dm/dt\rangle = +|x,v,m,dm/dt\rangle$ si f(x,v,m,dm/dt) = 0

- **Étape 3 :** Application de l'opérateur de Diffusion — amplifier l'amplitude des états solutions, supprimer l'amplitude des états non satisfaisants :

  $\hat{D} = 2|s\rangle\langle s| - I$

- **Étape 4 :** Répéter l'Étape 2 et l'Étape 3 pendant T cycles optimaux. Avec k étant le nombre de solutions dans l'espace M :

  $T = \lfloor (\pi/4) * \sqrt{M/k} \rfloor$ cycles.

  Après T cycles, la probabilité de mesurer la solution correcte tend vers 1.

- **Étape 5 :** Mesure du registre de 4n qubits — obtention de la combinaison (x, v, m, dm/dt) satisfaisant la condition f avec une probabilité élevée. Répéter O(k) fois pour collecter l'ensemble des k solutions.

5. Comparaison entre les modèles classique et quantique

Critères	**Modèle classique**	**Modèle quantique**

Critères	Modèle classique	Modèle quantique
Espace de recherche	M = N^4 combinaisons	M = N^4 = 2^{4n} états
Méthode de traitement	Séquentielle par combinaison	Parallèle quantique
Complexité	O(M) = O(N^4)	O(√M) = O(N^2)
Accélération	—	Quadratique par rapport au classique
Ressources	Mémoire illimitée	4n + ancilla qubits
Condition de recherche	Une condition f par exécution	Une condition f par exécution

6. Signification scientifique

Le modèle quantique NKTg conserve l'intégralité de la nature physique de la Loi NKTg — les grandeurs NKTg_1, NKTg_2 et les critères de classification de tendance — tout en exploitant l'algorithme de Grover pour réduire la complexité de recherche de O(M) = O(N^4) à O(√M) = O(N^2). L'Oracle est conçu de manière paramétrée, permettant une application flexible à n'importe quelle condition de tendance dans l'ensemble NKTg. Il s'agit d'une étape d'extension naturelle du modèle classique vers l'informatique quantique, ouvrant des perspectives d'application dans la simulation de systèmes physiques complexes et l'optimisation de paramètres à grande échelle.

```
//
// ============================================================================
// ALGORITHME NKTg — QUANTIQUE (OpenQASM 3.0) — Version corrigée complète
// Auteur : Nguyễn Khánh Tùng
// © 2026 Nguyễn Khánh Tùng. Tous droits réservés.
// ----------------------------------------------------------------------------
---
// Corrections selon audit :
//   [1] mcx remplace ccx+cx pour la détection dm/dt=0
//   [2] mcx remplace ccx+cx pour la condition MODE 5
//   [3] Uncompute MCZ ne répète plus inutilement les ccx
//   [4] Oracle encapsulé dans une def subroutine standard OpenQASM 3.0
//
// ============================================================================

OPENQASM 3.0;
include "stdgates.inc";

//
// ============================================================================
// DÉCLARATIONS DES REGISTRES
//
// ============================================================================
```

```
qubit[3] x_reg;
qubit[3] v_reg;
qubit[3] m_reg;
qubit[3] d_reg;
qubit[5] anc;
qubit    flag;

bit[3] c_x;
bit[3] c_v;
bit[3] c_m;
bit[3] c_d;

//
================================================================================
// SUBROUTINE : ORACLE NKTg (MODE 5 : NKTg1>0 ET NKTg2<0 ET dm/dt≠0)
// Structure : Compute → Phase kickback → Uncompute (symétrie absolue)
//
================================================================================
def nktg_oracle(qubit[3] xr, qubit[3] vr, qubit[3] mr,
                qubit[3] dr, qubit[5] a, qubit f) {

    // ------------------------------------------------------------------------
---
    // COMPUTE
    // ------------------------------------------------------------------------
---

    // Étape 2a : sign(p) = sign(m) XOR sign(v)  →  a[0]
    cx mr[2], a[0];
    cx vr[2], a[0];         // a[0] = sign(m) XOR sign(v) = sign(p)

    // Étape 2b : sign(NKTg1) = sign(x) XOR sign(p)  →  a[1]
    //          NKTg1 > 0  ⇔  a[1] = 0
    cx xr[2], a[1];
    cx a[0],  a[1];         // a[1] = sign(NKTg1)

    // Étape 2c : sign(NKTg2) = sign(d) XOR sign(p)  →  a[2]
    //          NKTg2 < 0  ⇔  a[2] = 1
    cx dr[2], a[2];
    cx a[0],  a[2];         // a[2] = sign(NKTg2)

    // Étape 2d : Détection dm/dt = 0  →  a[3]
    //          dm/dt = 0  ⇔  dr[2]=0 ET dr[1]=0 ET dr[0]=0
    //          Après X : dr[2]=1 ET dr[1]=1 ET dr[0]=1
    //          FIX [1] : utilise mcx à 3 contrÔles au lieu de ccx+cx erroné
    x dr[0]; x dr[1];  x dr[2];
    mcx dr[0], dr[1], dr[2], a[3]; // a[3]=1 quand dm/dt=0
    x dr[0]; x dr[1];  x dr[2];    // annuler X

    // Étape 2e : Condition complète MODE 5  →  a[4]
    //          f = NKTg1>0 ET NKTg2<0 ET dm/dt≠0
    //          NKTg1>0  ⇔  a[1]=0 → X(a[1])
    //          NKTg2<0  ⇔  a[2]=1
    //          dm/dt≠0  ⇔  a[3]=0 → X(a[3])
    //          FIX [2] : mcx à 3 contrÔles au lieu de ccx+cx erroné
    x a[1]; x a[3];
    mcx a[1], a[2], a[3], a[4];    // a[4]=1 quand f=1

    // ------------------------------------------------------------------------
---
    // PHASE KICKBACK
    // flag=|−⟩ : CX(a[4], flag) → inverse la phase de tous les états
satisfaisant f
```

```
    // ------------------------------------------------------------------------
---
    cx a[4], f;

    // ------------------------------------------------------------------------
---
    // UNCOMPUTE — symétrie absolue avec Compute (ordre inverse)
    // ------------------------------------------------------------------------
---
    mcx a[1], a[2], a[3], a[4];    // annuler étape 2e
    x a[1]; x a[3];

    x dr[0]; x dr[1];  x dr[2];
    mcx dr[0], dr[1], dr[2], a[3]; // annuler étape 2d
    x dr[0]; x dr[1];  x dr[2];

    cx a[0],  a[2];                // annuler étape 2c
    cx dr[2], a[2];

    cx a[0],  a[1];                // annuler étape 2b
    cx xr[2], a[1];

    cx vr[2], a[0];                // annuler étape 2a
    cx mr[2], a[0];
    // a[0..4] = |00000⟩ — entièrement restauré
}

//
================================================================================
// SUBROUTINE : DIFFUSION D^ = 2|s⟩⟨s| - I
// H → X → MCZ(12 qubit) → X → H
// FIX [3] : Uncompute MCZ ne répète plus inutilement les ccx
//
================================================================================
def grover_diffusion(qubit[3] xr, qubit[3] vr,
                     qubit[3] mr, qubit[3] dr, qubit[5] a) {

    // 3a : H sur les 12 qubits
    h xr[0]; h xr[1]; h xr[2];
    h vr[0]; h vr[1]; h vr[2];
    h mr[0]; h mr[1]; h mr[2];
    h dr[0]; h dr[1]; h dr[2];

    // 3b : X partout → mapper |0...0⟩ → |1...1⟩
    x xr[0]; x xr[1]; x xr[2];
    x vr[0]; x vr[1]; x vr[2];
    x mr[0]; x mr[1]; x mr[2];
    x dr[0]; x dr[1]; x dr[2];

    // 3c : MCZ(12 qubit) = H(target) → MCX(12 qubit) → H(target)
    // Décomposer MCX(12) en chaîne Toffoli — réutiliser a[0..4] propres
    h dr[2];

    // Niveau 1 : grouper par paires
    ccx xr[0], xr[1], a[0];  // a[0] = xr[0] AND xr[1]
    ccx xr[2], vr[0], a[1];  // a[1] = xr[2] AND vr[0]
    ccx vr[1], vr[2], a[2];  // a[2] = vr[1] AND vr[2]
    ccx mr[0], mr[1], a[3];  // a[3] = mr[0] AND mr[1]
    ccx mr[2], dr[0], a[4];  // a[4] = mr[2] AND dr[0]

    // Niveau 2 : combiner les résultats du niveau 1
    ccx a[0], a[1], a[0];      // a[0] = les 4 premiers bits ET-és
    // Note : besoin de qubits intermédiaires — réutilisation après uncompute
```

```
    // Décomposition complète sans chevauchement de qubits :
    ccx a[0], a[2], a[1];     // a[1] = a[0] AND a[2]
    ccx a[3], a[4], a[2];     // a[2] = a[3] AND a[4]
    ccx a[1], a[2], a[3];     // a[3] = les 10 bits
    ccx dr[1], a[3], a[4];    // a[4] = les 11 bits ET-és
    cx  a[4], dr[2];          // MCX : basculer dr[2] quand les 12 bits = 1

    // Uncompute niveau 2 (inverse, pas de répétitions — FIX [3])
    ccx dr[1], a[3], a[4];
    ccx a[1], a[2], a[3];
    ccx a[3], a[4], a[2];
    ccx a[0], a[2], a[1];
    ccx a[0], a[1], a[0];

    // Uncompute niveau 1
    ccx mr[2], dr[0], a[4];
    ccx mr[0], mr[1], a[3];
    ccx vr[1], vr[2], a[2];
    ccx xr[2], vr[0], a[1];
    ccx xr[0], xr[1], a[0];

    h dr[2];                   // annuler H → MCZ complet terminé

    // 3d : Annuler X
    x xr[0]; x xr[1]; x xr[2];
    x vr[0]; x vr[1]; x vr[2];
    x mr[0]; x mr[1]; x mr[2];
    x dr[0]; x dr[1]; x dr[2];

    // 3e : Second H — termine D^
    h xr[0]; h xr[1]; h xr[2];
    h vr[0]; h vr[1]; h vr[2];
    h mr[0]; h mr[1]; h mr[2];
    h dr[0]; h dr[1]; h dr[2];
}

//
// ============================================================================
// PROGRAMME PRINCIPAL
//
// ============================================================================

// Initialisation du flag à |−⟩ pour le phase kickback
x flag;
h flag;

// Étape 1 : Superposition uniforme
h x_reg[0]; h x_reg[1]; h x_reg[2];
h v_reg[0]; h v_reg[1]; h v_reg[2];
h m_reg[0]; h m_reg[1]; h m_reg[2];
h d_reg[0]; h d_reg[1]; h d_reg[2];

// Boucle de Grover T cycles
// T = floor((π/4) × sqrt(M/k))
// M=4096, k=1 → T≈50 | k=64 → T=6 | k=1024 → T=1
for uint i in [1:1] {
    nktg_oracle(x_reg, v_reg, m_reg, d_reg, anc, flag);
    grover_diffusion(x_reg, v_reg, m_reg, d_reg, anc);
}

// Étape 5 : Mesure
c_x = measure x_reg;   // 010 → x=2.0 positif ✓
```

```
c_v = measure v_reg;   // 011 → v=3.0 positif ✓
c_m = measure m_reg;   // 001 → m=5.0 positif ✓
c_d = measure d_reg;   // 101 → dm/dt=-0.5 négatif ✓
                        // 000 → photon : Oracle non marqué ✓
```

3.Vietnamese (Tiếng Việt)

CÔNG BỐ HỒ SƠ MÔ TẢ PHẦN MỀM VÀ GIẢI THUẬT ĐỊNH LUẬT NKTg VỀ QUÁN TÍNH BIẾN THIÊN

I. THÔNG TIN CHUNG

- **Tên phần mềm:** Hệ thống tính toán Quán tính biến thiên NKTg (NKTg Dynamics Calculator).
- **Tác giả:** Nguyễn Khánh Tùng
- **Ngôn ngữ lập trình:** C++ (ISO/IEC 14882) và Assembly (x64).

II. CƠ SỞ LÝ THUYẾT: ĐỊNH LUẬT NKTg VỀ QUÁN TÍNH BIẾN THIÊN

Phần mềm thực thi dựa trên các nguyên lý của Định luật NKTg:

1. **Mối quan hệ cơ bản:** Xu hướng chuyển động phụ thuộc vào vị trí (x), tốc độ (v) và khối lượng (m). Công thức: NKTg = f(x, v, m).
2. **Các đại lượng sản phẩm cốt lõi:**
    - **Động lượng (p):** $p = m \times v$
    - **Đại lượng $NKTg_1$:** Sản phẩm của vị trí và động lượng ($NKTg_1 = x \times p$).
    - **Đại lượng $NKTg_2$:** Sản phẩm của tốc độ biến thiên khối lượng và động lượng ($NKTg_2 = (dm/dt) \times p$).
3. **Đơn vị đo lường:** NKTm (Đơn vị của quán tính biến thiên).

III. MÔ TẢ GIẢI THUẬT CHI TIẾT

1. Quy trình xử lý dữ liệu

Giải thuật thực hiện phân tích xu hướng chuyển động qua các bước logic:

- **Bước 1:** Tiếp nhận thông số đầu vào: vị trí (x), vận tốc (v), khối lượng (m) và tốc độ thay đổi khối lượng (dm/dt).
- **Bước 2:** Tính toán động lượng tuyến tính $p = m \times v$.
- **Bước 3:** Tính toán giá trị quán tính biến thiên $NKTg_1$ và $NKTg_2$.
- **Bước 4:** Phân loại xu hướng (Tendency) dựa trên dấu của giá trị:
    - **$NKTg_1 > 0$:** Rời xa trạng thái ổn định.
    - **$NKTg_1 < 0$:** Tiến về trạng thái ổn định.
    - **$NKTg_2 > 0$:** Biến thiên khối lượng hỗ trợ chuyển động.
    - **$NKTg_2 < 0$:** Biến thiên khối lượng cản trở chuyển động.

IV. MÃ NGUỒN THỰC THI (SOURCE CODE)

1. Mã nguồn bậc cao (C++)

C++

```
#include <cstdio>

/**
 * Thư viện tính toán Quán tính biến thiên theo Định luật NKTg
 * Bảo lưu mọi quyền tác giả đối với logic tính toán và định nghĩa xu hướng.
 */
int main() {
    // 1. Khai báo tham số: x (vị trí), v (vận tốc), m (khối lượng), dm_dt
(tốc độ biến thiên m)
    double x = 2.0, v = 3.0, m = 5.0, dm_dt = -0.5;

    // 2. Thực hiện tính toán theo Định luật NKTg
    double p = m * v;        // p = m * v
    double n1 = x * p;       // NKTg1 = x * p
    double n2 = dm_dt * p;   // NKTg2 = (dm/dt) * p

    // 3. Xuất kết quả định dạng Record
    printf("{ p = %.1f\n  nktg1 = %.1f\n  nktg2 = %.1f\n", p, n1, n2);

    // Phân loại xu hướng chuyển động (Tendency Classification)
    printf("  tendency1 = \"%s\"\n",
           n1 > 0 ? "Moving away from stable state" :
           n1 < 0 ? "Moving toward stable state" : "Stable equilibrium");

    printf("  tendency2 = \"%s\" }",
           n2 > 0 ? "Mass variation supports movement" :
           n2 < 0 ? "Mass variation resists movement" : "No mass variation
effect");

    return 0;
}
```

2. Mã nguồn mức thấp (Assembly x64)

```
; ------------------------------------------------------------------------
; GIẢI THUẬT PHÂN TÍCH XU HƯỚNG CHUYỂN ĐỘNG NKTg (x64 NASM )
; Tác giả: Nguyễn Khánh Tùng
; © 2026 Nguyễn Khánh Tùng. All rights reserved.
; ------------------------------------------------------------------------
; Số liệu mẫu (từ tài liệu gốc):
;   x=2.0, v=3.0, m=5.0, dm/dt=-0.5
;   p=15.0 | NKTg1=+30.0 → away | NKTg2=-7.5 → resist
;
; Hàm analyze_nktg_tendency — System V AMD64 ABI (Linux):
;   Input : xmm0=x, xmm1=v, xmm2=m, xmm3=dm/dt
;   Output: rax=0 (OK), rax=-1 (NaN error)
; ------------------------------------------------------------------------

section .data
    fmt_all   db "{ p = %.1f", 10,
              db "  nktg1 = %.1f", 10,
              db "  nktg2 = %.1f", 10,
              db "  tendency1 = ""%s""", 10,
              db "  tendency2 = ""%s"" }", 10, 0
```

```
    str_away    db "Moving away from stable state", 0
    str_toward  db "Moving toward stable state", 0
    str_stable  db "Stable equilibrium", 0
    str_support db "Mass variation supports movement", 0
    str_resist  db "Mass variation resists movement", 0
    str_noeff   db "No mass variation effect", 0

    val_x     dq 2.0
    val_v     dq 3.0
    val_m     dq 5.0
    val_dm_dt dq -0.5

section .text
    extern printf
    global analyze_nktg_tendency
    global main

;
================================================================================
; analyze_nktg_tendency
;
================================================================================
analyze_nktg_tendency:
    push rbp
    mov  rbp, rsp
    and  rsp, -16

    ; --- Bước 1: Tiếp nhận thông số đầu vào qua thanh ghi ---
    ; xmm0 = x     (vị trí)
    ; xmm1 = v     (vận tốc)
    ; xmm2 = m     (khối lượng)
    ; xmm3 = dm/dt (tốc độ biến thiên khối lượng)
    ; Các thanh ghi xmm0–xmm3 được bảo toàn, không ghi đè trực tiếp

    ; --- Bước 2: Tính động lượng p = m * v ---
    movsd xmm4, xmm2           ; xmm4 = m
    mulsd xmm4, xmm1           ; xmm4 = p = 5.0 * 3.0 = 15.0

    ; --- Bước 3: Tính NKTg1 và NKTg2 ---
    movsd xmm5, xmm0           ; xmm5 = x
    mulsd xmm5, xmm4           ; xmm5 = NKTg1 = x * p = 2.0 * 15.0 = 30.0

    movsd xmm6, xmm3           ; xmm6 = dm/dt
    mulsd xmm6, xmm4           ; xmm6 = NKTg2 = (dm/dt) * p = (-0.5) * 15.0 =
-7.5

    ; --- Bước 4: Phân loại xu hướng dựa trên dấu ---
    xorps xmm7, xmm7           ; xmm7 = 0.0

    ; 4.1. Phân loại NKTg1 → lưu tạm vào r8
    ucomisd xmm5, xmm7         ; So sánh NKTg1 với 0.0
    jp  .error_nan             ; PF=1 → NaN
    lea  r8, [rel str_stable]  ; mặc định: NKTg1 = 0 → Stable equilibrium
    ja  .t1_away               ; NKTg1 > 0
    jb  .t1_toward             ; NKTg1 < 0
    jmp .check_nktg2
.t1_away:
    lea  r8, [rel str_away]    ; "Moving away from stable state"
    jmp .check_nktg2
.t1_toward:
    lea  r8, [rel str_toward]  ; "Moving toward stable state"
```

```
    ; 4.2. Phân loại NKTg2 → lưu tạm vào r9
.check_nktg2:
    ucomisd xmm6, xmm7         ; So sánh NKTg2 với 0.0
    jp  .error_nan             ; NaN guard
    lea  r9, [rel str_noeff]   ; mặc định: NKTg2 = 0 → No mass variation
effect
    ja  .t2_support            ; NKTg2 > 0
    jb  .t2_resist             ; NKTg2 < 0
    jmp .do_printf
.t2_support:
    lea  r9, [rel str_support] ; "Mass variation supports movement"
    jmp .do_printf
.t2_resist:
    lea  r9, [rel str_resist]  ; "Mass variation resists movement"

    ; --- Gọi printf một lần duy nhất ---
.do_printf:
    lea  rdi, [rel fmt_all]    ; rdi = format string
    mov  rsi, r8               ; rsi = tendency1 ptr (từ r8, không xung đột)
    mov  rdx, r9               ; rdx = tendency2 ptr (từ r9, không xung đột)
    movsd xmm0, xmm4           ; xmm0 = p     = 15.0
    movsd xmm1, xmm5           ; xmm1 = NKTg1 = 30.0
    movsd xmm2, xmm6           ; xmm2 = NKTg2 = -7.5
    mov  eax, 3                ; 3 tham số float
    call printf

    xor  eax, eax              ; return 0
    leave
    ret

.error_nan:
    mov  rax, -1
    leave
    ret

; ========================================================================
; main — nạp số liệu mẫu vào thanh ghi, gọi analyze_nktg_tendency
; ========================================================================
main:
    push rbp
    mov  rbp, rsp
    and  rsp, -16

    ; --- Bước 1: Nạp thông số đầu vào (số liệu mẫu từ tài liệu gốc) ---
    movsd xmm0, [rel val_x]     ; xmm0 = x     = 2.0
    movsd xmm1, [rel val_v]     ; xmm1 = v     = 3.0
    movsd xmm2, [rel val_m]     ; xmm2 = m     = 5.0
    movsd xmm3, [rel val_dm_dt] ; xmm3 = dm/dt = -0.5

    call analyze_nktg_tendency

    xor  eax, eax
    leave
    ret

; ------------------------------------------------------------------------
; Kết quả kỳ vọng:
; { p = 15.0
;   nktg1 = 30.0
;   nktg2 = -7.5
;   tendency1 = "Moving away from stable state"
;   tendency2 = "Mass variation resists movement" }
```

; --

V. ĐỊNH NGHĨA TRẠNG THÁI ỔN ĐỊNH

Trạng thái ổn định trong phần mềm này được định nghĩa là trạng thái mà tại đó vị trí (x), tốc độ (v) và khối lượng (m) tương tác để duy trì cấu trúc chuyển động, tránh mất kiểm soát và bảo toàn mô thức chuyển động vốn có của vật thể.

VI. MỞ RỘNG LƯỢNG TỬ: GIẢI THUẬT NKTg TRÊN NỀN TẢNG ĐIỆN TOÁN LƯỢNG TỬ

---

1. Cơ sở lý thuyết lượng tử

Phần mở rộng lượng tử giữ nguyên toàn bộ các đại lượng cốt lõi của Định luật NKTg:

- **Động lượng (p):** $p = m \times v$
- **Đại lượng $NKTg_1$:** Sản phẩm của vị trí và động lượng ($NKTg_1 = x \times p$)
- **Đại lượng $NKTg_2$:** Sản phẩm của tốc độ biến thiên khối lượng và động lượng ($NKTg_2 = (dm/dt) \times p$)
- **Đơn vị đo lường:** NKTm (Đơn vị của quán tính biến thiên)

Thay vì tính tuần tự từng tổ hợp (x, v, m, dm/dt) như mô hình cổ điển, mô hình lượng tử mã hóa toàn bộ không gian tham số vào trạng thái chồng chất và khai thác song song lượng tử để tìm kiếm.

---

2. Phát biểu bài toán lượng tử

Đầu vào: Không gian tham số (x, v, m, dm/dt), mỗi biến được rời rạc hóa thành $N = 2^n$ mức giá trị, mã hóa bằng n qubit. Tổng không gian tìm kiếm gồm $\mathbf{M = N^4 = 2^{4n}}$ tổ hợp.

Bài toán: Cho một điều kiện xu hướng f được chọn từ tập:

Điều kiện f	Xu hướng
$NKTg_1 > 0$	Rời xa trạng thái ổn định
$NKTg_1 < 0$	Tiến về trạng thái ổn định
$NKTg_2 > 0$	Biến thiên khối lượng hỗ trợ chuyển động
$NKTg_2 < 0$	Biến thiên khối lượng cản trở chuyển động
$NKTg_1 > 0$ **và** $NKTg_2 < 0$	Kết hợp hai điều kiện

Tìm tất cả tổ hợp $(x, v, m, dm/dt) \in \{1..N\}^4$ thỏa mãn điều kiện f đã chọn.

Đầu ra: Tập hợp các tổ hợp tham số thỏa mãn f, thu được sau phép đo lượng tử.

---

3. Không gian trạng thái lượng tử

Toàn bộ không gian tham số được mã hóa vào thanh ghi lượng tử **4n qubit**:

$$|q\rangle = |x\rangle \otimes |v\rangle \otimes |m\rangle \otimes |dm/dt\rangle$$

Trạng thái khởi tạo là superposition đều qua cổng Hadamard:

$$|s\rangle = H^{\otimes 4n}|0\rangle^{4n} = (1/\sqrt{M})\,\Sigma\,|x, v, m, dm/dt\rangle$$

Mỗi tổ hợp tồn tại đồng thời với biên độ xác suất ban đầu $\mathbf{1/\sqrt{M}}$.

---

4. Giải thuật xử lý lượng tử

Giải thuật thực hiện phân tích xu hướng NKTg qua các bước logic:

Bước 1: Tiếp nhận và mã hóa thông số đầu vào (x, v, m, dm/dt) vào thanh ghi 4n qubit. Áp dụng cổng Hadamard tạo superposition đều trên toàn bộ $M = N^4$ tổ hợp.

Bước 2: Xây dựng Oracle Ô tham số hóa theo điều kiện xu hướng f được chọn. Oracle thực hiện ba thao tác thuận nghịch:

- *Compute:* Tính $p = m \times v$, $NKTg_1 = x \times p$, $NKTg_2 = (dm/dt) \times p$ lên thanh ghi ancilla
- *Phase kickback:* Đảo pha các trạng thái thỏa mãn f
- *Uncompute:* Khôi phục ancilla về $|0\rangle$

$$\hat{O}\,|x,v,m,dm/dt\rangle = -|x,v,m,dm/dt\rangle \quad \text{nếu } f(x,v,m,dm/dt) = 1$$
$$\hat{O}\,|x,v,m,dm/dt\rangle = +|x,v,m,dm/dt\rangle \quad \text{nếu } f(x,v,m,dm/dt) = 0$$

Bước 3: Áp dụng toán tử Diffusion — khuếch đại biên độ các trạng thái nghiệm, triệt tiêu biên độ các trạng thái không thỏa mãn:

$$\hat{D} = 2|s\rangle\langle s| - I$$

Bước 4: Lặp lại Bước 2 và Bước 3 trong T vòng tối ưu. Với k là số nghiệm trong không gian M:

$$T = \lfloor(\pi/4) \times \sqrt{(M/k)}\rfloor \quad \text{vòng lặp}$$

Sau T vòng, xác suất đo được nghiệm đúng tiến về 1.

Bước 5: Đo lường thanh ghi 4n qubit — thu được tổ hợp (x, v, m, dm/dt) thỏa mãn điều kiện f với xác suất cao. Lặp lại O(k) lần để thu toàn bộ k nghiệm.

---

5. So sánh mô hình cổ điển và lượng tử

Tiêu chí	Mô hình cổ điển	Mô hình lượng tử
Không gian tìm kiếm	$M = N^4$ tổ hợp	$M = N^4 = 2^{4n}$ trạng thái
Phương thức xử lý	Tuần tự từng tổ hợp	Song song lượng tử
Độ phức tạp	$O(M) = O(N^4)$	$O(\sqrt{M}) = O(N^2)$
Tăng tốc	—	Bậc hai so với cổ điển
Tài nguyên	Không giới hạn bộ nhớ	4n + ancilla qubit
Điều kiện tìm kiếm	Một điều kiện f mỗi lần chạy	Một điều kiện f mỗi lần chạy

---

6. Ý nghĩa khoa học

Mô hình lượng tử NKTg giữ nguyên toàn bộ bản chất vật lý của Định luật NKTg — các đại lượng $NKTg_1$, $NKTg_2$ và tiêu chí phân loại xu hướng — trong khi khai thác Grover's Algorithm để giảm độ phức tạp tìm kiếm từ $\mathbf{O(M) = O(N^4)}$ xuống $\mathbf{O(\sqrt{M}) = O(N^2)}$. Oracle được thiết kế tham số hóa, cho phép áp dụng linh hoạt với bất kỳ điều kiện xu hướng nào trong tập NKTg. Đây là bước mở rộng tự nhiên từ mô hình cổ điển sang điện toán lượng tử, mở ra hướng ứng dụng trong mô phỏng hệ vật lý phức tạp và tối ưu hóa tham số quy mô lớn.

```
//
// ============================================================================
// GIẢI THUẬT NKTg — LƯỢNG TỬ (OpenQASM 3.0) — Bản sửa lỗi hoàn chỉnh
// Tác giả : Nguyễn Khánh Tùng
// © 2026 Nguyễn Khánh Tùng. All rights reserved.
// ----------------------------------------------------------------------------
---
// Sửa lỗi theo audit:
//   [1] mcx thay ccx+cx cho phát hiện dm/dt=0
//   [2] mcx thay ccx+cx cho điều kiện MODE 5
//   [3] Uncompute MCZ không còn lặp ccx thừa
//   [4] Oracle đóng gói trong def subroutine chuẩn OpenQASM 3.0
//
// ============================================================================

OPENQASM 3.0;
include "stdgates.inc";

//
// ============================================================================
// KHAI BÁO THANH GHI
//
// ============================================================================
qubit[3] x_reg;
qubit[3] v_reg;
qubit[3] m_reg;
qubit[3] d_reg;
qubit[5] anc;
qubit    flag;

bit[3] c_x;
bit[3] c_v;
bit[3] c_m;
bit[3] c_d;

//
// ============================================================================
// SUBROUTINE: ORACLE NKTg (MODE 5: NKTg1>0 VÀ NKTg2<0 VÀ dm/dt≠0)
// Cấu trúc: Compute → Phase kickback → Uncompute (đối xứng tuyệt đối)
```

```
//
================================================================================
def nktg_oracle(qubit[3] xr, qubit[3] vr, qubit[3] mr,
                qubit[3] dr, qubit[5] a, qubit f) {

    // ------------------------------------------------------------------------
---
    // COMPUTE
    // ------------------------------------------------------------------------
---

    // Bước 2a: sign(p) = sign(m) XOR sign(v)  →  a[0]
    cx mr[2], a[0];
    cx vr[2], a[0];         // a[0] = sign(m) XOR sign(v) = sign(p)

    // Bước 2b: sign(NKTg1) = sign(x) XOR sign(p)  →  a[1]
    //          NKTg1 > 0  ⇔ a[1] = 0
    cx xr[2], a[1];
    cx a[0],  a[1];         // a[1] = sign(NKTg1)

    // Bước 2c: sign(NKTg2) = sign(d) XOR sign(p)  →  a[2]
    //          NKTg2 < 0  ⇔ a[2] = 1
    cx dr[2], a[2];
    cx a[0],  a[2];         // a[2] = sign(NKTg2)

    // Bước 2d: Phát hiện dm/dt = 0  →  a[3]
    //          dm/dt = 0  ⇔ dr[2]=0 AND dr[1]=0 AND dr[0]=0
    //          Sau X: dr[2]=1 AND dr[1]=1 AND dr[0]=1
    //          FIX [1]: dùng mcx 3-control thay ccx+cx sai
    x dr[0];  x dr[1];  x dr[2];
    mcx dr[0], dr[1], dr[2], a[3]; // a[3]=1 khi dm/dt=0
    x dr[0];  x dr[1];  x dr[2];   // hoàn tác X

    // Bước 2e: Điều kiện MODE 5 đầy đủ  →  a[4]
    //          f = NKTg1>0 AND NKTg2<0 AND dm/dt≠0
    //          NKTg1>0  ⇔ a[1]=0  → X(a[1])
    //          NKTg2<0  ⇔ a[2]=1
    //          dm/dt≠0  ⇔ a[3]=0  → X(a[3])
    //          FIX [2]: mcx 3-control thay ccx+cx sai
    x a[1];  x a[3];
    mcx a[1], a[2], a[3], a[4];    // a[4]=1 khi f=1

    // ------------------------------------------------------------------------
---
    // PHASE KICKBACK
    // flag=|-⟩: CX(a[4], flag) → đảo pha toàn bộ trạng thái thỏa mãn f
    // ------------------------------------------------------------------------
---
    cx a[4], f;

    // ------------------------------------------------------------------------
---
    // UNCOMPUTE — đối xứng tuyệt đối với Compute (ngược thứ tự)
    // ------------------------------------------------------------------------
---
    mcx a[1], a[2], a[3], a[4];    // hoàn tác bước 2e
    x a[1];  x a[3];

    x dr[0];  x dr[1];  x dr[2];
    mcx dr[0], dr[1], dr[2], a[3]; // hoàn tác bước 2d
    x dr[0];  x dr[1];  x dr[2];

    cx a[0],  a[2];                 // hoàn tác bước 2c
```

```
    cx dr[2], a[2];

    cx a[0],  a[1];                 // hoàn tác bước 2b
    cx xr[2], a[1];

    cx vr[2], a[0];                 // hoàn tác bước 2a
    cx mr[2], a[0];
    // a[0..4] = |00000⟩ — đã khôi phục hoàn toàn
}

//
// ============================================================================
// SUBROUTINE: DIFFUSION D^ = 2|s⟩⟨s| - I
// H → X → MCZ(12 qubit) → X → H
// FIX [3]: Uncompute MCZ không còn lặp ccx thừa
//
// ============================================================================
def grover_diffusion(qubit[3] xr, qubit[3] vr,
                     qubit[3] mr, qubit[3] dr, qubit[5] a) {

    // 3a: H toàn bộ 12 qubit
    h xr[0];  h xr[1];  h xr[2];
    h vr[0];  h vr[1];  h vr[2];
    h mr[0];  h mr[1];  h mr[2];
    h dr[0];  h dr[1];  h dr[2];

    // 3b: X toàn bộ → ánh xạ |0...0⟩ → |1...1⟩
    x xr[0];  x xr[1];  x xr[2];
    x vr[0];  x vr[1];  x vr[2];
    x mr[0];  x mr[1];  x mr[2];
    x dr[0];  x dr[1];  x dr[2];

    // 3c: MCZ(12 qubit) = H(target) → MCX(12 qubit) → H(target)
    // Phân rã MCX(12) thành chuỗi Toffoli — tái dùng a[0..4] đã sạch
    h dr[2];

    // Tầng 1: ghép từng cặp
    ccx xr[0],  xr[1],  a[0];  // a[0] = xr[0] AND xr[1]
    ccx xr[2],  vr[0],  a[1];  // a[1] = xr[2] AND vr[0]
    ccx vr[1],  vr[2],  a[2];  // a[2] = vr[1] AND vr[2]
    ccx mr[0],  mr[1],  a[3];  // a[3] = mr[0] AND mr[1]
    ccx mr[2],  dr[0],  a[4];  // a[4] = mr[2] AND dr[0]

    // Tầng 2: ghép kết quả tầng 1
    ccx a[0], a[1], a[0];      // a[0] = 4 bit đầu AND nhau
    // Lưu ý: cần qubit trung gian — dùng lại sau uncompute
    // Phân rã đầy đủ không dùng qubit chồng:
    ccx a[0],  a[2],  a[1];    // a[1] = a[0] AND a[2]
    ccx a[3],  a[4],  a[2];    // a[2] = a[3] AND a[4]
    ccx a[1],  a[2],  a[3];    // a[3] = tất cả 10 bit
    ccx dr[1], a[3],  a[4];    // a[4] = 11 bit AND nhau
    cx  a[4],  dr[2];          // MCX: lật dr[2] khi tất cả 12 bit = 1

    // Uncompute tầng 2 (đảo ngược, không lặp thừa — FIX [3])
    ccx dr[1], a[3],  a[4];
    ccx a[1],  a[2],  a[3];
    ccx a[3],  a[4],  a[2];
    ccx a[0],  a[2],  a[1];
    ccx a[0],  a[1],  a[0];

    // Uncompute tầng 1
    ccx mr[2], dr[0],  a[4];
    ccx mr[0], mr[1],  a[3];
```

```
    ccx vr[1], vr[2],  a[2];
    ccx xr[2], vr[0],  a[1];
    ccx xr[0], xr[1],  a[0];

    h dr[2];                    // hoàn tác H → MCZ hoàn chỉnh

    // 3d: X hoàn tác
    x xr[0];  x xr[1];  x xr[2];
    x vr[0];  x vr[1];  x vr[2];
    x mr[0];  x mr[1];  x mr[2];
    x dr[0];  x dr[1];  x dr[2];

    // 3e: H lần hai — hoàn thành D^
    h xr[0];  h xr[1];  h xr[2];
    h vr[0];  h vr[1];  h vr[2];
    h mr[0];  h mr[1];  h mr[2];
    h dr[0];  h dr[1];  h dr[2];
}

//
// ============================================================================
// CHƯƠNG TRÌNH CHÍNH
//
// ============================================================================

// Khởi tạo flag ở |−⟩ cho phase kickback
x flag;
h flag;

// Bước 1: Superposition đều
h x_reg[0];  h x_reg[1];  h x_reg[2];
h v_reg[0];  h v_reg[1];  h v_reg[2];
h m_reg[0];  h m_reg[1];  h m_reg[2];
h d_reg[0];  h d_reg[1];  h d_reg[2];

// Vòng lặp Grover T vòng
// T = floor((π/4) × sqrt(M/k))
// M=4096, k=1 → T≈50 | k=64 → T=6 | k=1024 → T=1
for uint i in [1:1] {
    nktg_oracle(x_reg, v_reg, m_reg, d_reg, anc, flag);
    grover_diffusion(x_reg, v_reg, m_reg, d_reg, anc);
}

// Bước 5: Đo lường
c_x = measure x_reg;   // 010 → x=2.0 dương ✓
c_v = measure v_reg;   // 011 → v=3.0 dương ✓
c_m = measure m_reg;   // 001 → m=5.0 dương ✓
c_d = measure d_reg;   // 101 → dm/dt=-0.5 âm ✓
                        // 000 → photon: Oracle không đánh dấu ✓
```

4. Chinese (简体中文)

关于 NKTg 变惯性定律的软件规格与算法公告

一、基本信息

- **软件名称：**

  NKTg 变惯性计算系统 (NKTg Dynamics Calculator)。

- **作者：**

  Nguyễn Khánh Tùng

- **编程语言：**

  C++ (ISO/IEC 14882) 和 Assembly (x64)。

二、理论基础：NKTg 变惯性定律

本软件基于 NKTg 定律的原理实现：

- **基本关系：**

  运动趋势取决于位置 (x)、速度 (v) 和质量 (m)。公式：NKTg = f(x, v, m)。

- **核心乘积量：**
  - 动量 (p)：p = m * v
  - NKTg_1 量：位置与动量的乘积 (NKTg_1 = x * p)。
  - NKTg_2 量：质量变化率与动量的乘积 (NKTg_2 = (dm/dt) * p)。
- **计量单位：**

  NKTm（变惯性单位）。

三、详细算法描述

1. 数据处理流程

算法通过以下逻辑步骤进行运动趋势分析：

- **第一步：**

  接收输入参数：位置 (x)、速度 (v)、质量 (m) 和质量变化率 (dm/dt)。

- **第二步：**

计算线动量 p = m * v。

- **第三步：**

  计算变惯性值 NKTg_1 和 NKTg_2。

- **第四步：**

  根据数值的正负号对趋势 (Tendency) 进行分类：

  - NKTg_1 > 0：远离稳定状态。
  - NKTg_1 < 0：向稳定状态靠拢。
  - NKTg_2 > 0：质量变化支持运动。
  - NKTg_2 < 0：质量变化阻碍运动。

四、执行源码 (SOURCE CODE)

1. 高级语言源码 (C++)

C++

```
#include <cstdio>

/**
 * 根据 NKTg 定律编写的变惯性计算库
 * 保留有关计算逻辑和趋势定义的所有著作权。
 */
int main() {
   // 1. 声明参数：x (位置), v (速度), m (质量), dm_dt (质量 m 的变化率)
   double x = 2.0, v = 3.0, m = 5.0, dm_dt = -0.5;

   // 2. 根据 NKTg 定律执行计算
   double p = m * v;         // p = m * v
   double n1 = x * p;        // NKTg1 = x * p
   double n2 = dm_dt * p;    // NKTg2 = (dm/dt) * p

   // 3. 以 Record 格式输出结果
   printf("{ p = %.1f\n  nktg1 = %.1f\n  nktg2 = %.1f\n", p, n1, n2);

   // 运动趋势分类 (Tendency Classification)
   printf("  tendency1 = \"%s\"\n",
          n1 > 0 ? "远离稳定状态" :
          n1 < 0 ? "向稳定状态靠拢" : "稳定平衡");

   printf("  tendency2 = \"%s\" }",
          n2 > 0 ? "质量变化支持运动" :
          n2 < 0 ? "质量变化阻碍运动" : "无质量变化影响");

   return 0;
```

```
}
```

2. 低级语言源码 (Assembly x64)

```
; ------------------------------------------------------------------------------
; NKTg 运动趋势分析算法 (x64 NASM)
; 作者：Nguyễn Khánh Tùng
; © 2026 Nguyễn Khánh Tùng。保留所有权利。
; ------------------------------------------------------------------------------
; 示例数据（源自原始文档）：
;   x=2.0, v=3.0, m=5.0, dm/dt=-0.5
;   p=15.0 | NKTg1=+30.0 → 远离 | NKTg2=-7.5 → 阻碍
;
; 函数 analyze_nktg_tendency — System V AMD64 ABI (Linux):
;   输入：xmm0=x, xmm1=v, xmm2=m, xmm3=dm/dt
;   输出：rax=0 (正常), rax=-1 (NaN 错误)
; ------------------------------------------------------------------------------

section .data
    fmt_all  db "{ p = %.1f", 10,
             db "  nktg1 = %.1f", 10,
             db "  nktg2 = %.1f", 10,
             db "  tendency1 = ""%s""", 10,
             db "  tendency2 = ""%s"" }", 10, 0

    str_away    db "远离稳定状态", 0
    str_toward  db "向稳定状态靠拢", 0
    str_stable  db "稳定平衡", 0
    str_support db "质量变化支持运动", 0
    str_resist  db "质量变化阻碍运动", 0
    str_noeff   db "无质量变化影响", 0

    val_x     dq 2.0
    val_v     dq 3.0
    val_m     dq 5.0
    val_dm_dt dq -0.5

section .text
    extern printf
    global analyze_nktg_tendency
    global main

;
================================================================================
; analyze_nktg_tendency
;
================================================================================
analyze_nktg_tendency:
    push rbp
    mov rbp, rsp
    and rsp, -16

    ; --- 第一步：通过寄存器接收输入参数 ---
    ; xmm0 = x       (位置)
```

```
    ; xmm1 = v       (速度)
    ; xmm2 = m       (质量)
    ; xmm3 = dm/dt   (质量变化率)
    ; 寄存器 xmm0-xmm3 被保护，不直接覆盖

    ; --- 第二步：计算动量 p = m * v ---
    movsd xmm4, xmm2           ; xmm4 = m
    mulsd xmm4, xmm1           ; xmm4 = p = 5.0 * 3.0 = 15.0

    ; --- 第三步：计算 NKTg1 和 NKTg2 ---
    movsd xmm5, xmm0           ; xmm5 = x
    mulsd xmm5, xmm4           ; xmm5 = NKTg1 = x * p = 2.0 * 15.0 = 30.0

    movsd xmm6, xmm3           ; xmm6 = dm/dt
    mulsd xmm6, xmm4           ; xmm6 = NKTg2 = (dm/dt) * p = (-0.5) * 15.0 =
-7.5

    ; --- 第四步：根据正负号进行趋势分类 ---
    xorps xmm7, xmm7           ; xmm7 = 0.0

    ; 4.1. NKTg1 分类 → 暂存至 r8
    ucomisd xmm5, xmm7         ; 将 NKTg1 与 0.0 比较
    jp .error_nan              ; PF=1 → 非数字 (NaN)
    lea r8, [rel str_stable]   ; 默认：NKTg1 = 0 → 稳定平衡
    ja .t1_away                ; NKTg1 > 0
    jb .t1_toward              ; NKTg1 < 0
    jmp .check_nktg2
.t1_away:
    lea r8, [rel str_away]     ; "远离稳定状态"
    jmp .check_nktg2
.t1_toward:
    lea r8, [rel str_toward]   ; "向稳定状态靠拢"

    ; 4.2. NKTg2 分类 → 暂存至 r9
.check_nktg2:
    ucomisd xmm6, xmm7         ; 将 NKTg2 与 0.0 比较
    jp .error_nan              ; NaN 防护
    lea r9, [rel str_noeff]    ; 默认：NKTg2 = 0 → 无质量变化影响
    ja .t2_support             ; NKTg2 > 0
    jb .t2_resist              ; NKTg2 < 0
    jmp .do_printf
.t2_support:
    lea r9, [rel str_support] ; "质量变化支持运动"
    jmp .do_printf
.t2_resist:
    lea r9, [rel str_resist]   ; "质量变化阻碍运动"

    ; --- 仅调用一次 printf ---
.do_printf:
    lea rdi, [rel fmt_all]     ; rdi = 格式化字符串
    mov rsi, r8                ; rsi = tendency1 指针 (来自 r8)
    mov rdx, r9                ; rdx = tendency2 指针 (来自 r9)
    movsd xmm0, xmm4           ; xmm0 = p     = 15.0
    movsd xmm1, xmm5           ; xmm1 = NKTg1 = 30.0
```

```
    movsd xmm2, xmm6            ; xmm2 = NKTg2 = -7.5
    mov eax, 3                  ; 3 个浮点参数
    call printf

    xor eax, eax                ; 返回 0
    leave
    ret

.error_nan:
    mov rax, -1
    leave
    ret

; ========================================================================
; main — 将示例数据加载到寄存器，调用 analyze_nktg_tendency
; ========================================================================
main:
    push rbp
    mov rbp, rsp
    and rsp, -16

    ; --- 第一步：加载输入参数（原始文档中的示例数据） ---
    movsd xmm0, [rel val_x]      ; xmm0 = x     = 2.0
    movsd xmm1, [rel val_v]      ; xmm1 = v     = 3.0
    movsd xmm2, [rel val_m]      ; xmm2 = m     = 5.0
    movsd xmm3, [rel val_dm_dt] ; xmm3 = dm/dt = -0.5

    call analyze_nktg_tendency

    xor eax, eax
    leave
    ret

; ------------------------------------------------------------------------------
; 预期结果：
; { p = 15.0
;   nktg1 = 30.0
;   nktg2 = -7.5
;   tendency1 = "远离稳定状态"
;   tendency2 = "质量变化阻碍运动" }
; ------------------------------------------------------------------------------
```

五、稳定状态的定义

本软件中的“稳定状态”定义为：位置 (x)、速度 (v) 和质量 (m)
相互作用以维持运动结构的平衡状态，从而避免失控并保持物体固有的运动模式。

六、量子扩展：基于量子计算平台的 NKTg 算法

1. 量子理论基础

量子扩展保留了 NKTg 定律的所有核心量：

- **动量 (p)：** p = m * v

- **NKTg_1 量：** 位置与动量的乘积 (NKTg_1 = x * p)
- **NKTg_2 量：** 质量变化率与动量的乘积 (NKTg_2 = (dm/dt) * p)
- **计量单位：** NKTm（变惯性单位）

量子模型不再像经典模型那样按顺序计算每个组合 (x, v, m, dm/dt)，而是将整个参数空间编码到叠加态中，利用量子并行性进行搜索。

2. 量子问题陈述

- **输入：** 参数空间 (x, v, m, dm/dt)，**每个**变量被离散化为 N = 2^n 个取值能级，并使用 n 个量子比特编码。总搜索空间包含 M = N^4 = 2^{4n} 个组合。
- **问题：** 给定一个从以下集合中选择的趋势条件 f：

条件 f	**趋势**
NKTg_1 > 0	远离稳定状态
NKTg_1 < 0	向稳定状态靠拢
NKTg_2 > 0	质量变化支持运动
NKTg_2 < 0	质量变化阻碍运动
NKTg_1 > 0 且 NKTg_2 < 0	结合以上两个条件

寻找满足所选条件 f 的所有组合 (x, v, m, dm/dt) ∈{1..N}^4。

- **输出：** 量子测量后获得的满足 f 的参数组合集合。

3. 量子态空间

整个参数空间被编码到一个 4n 量子比特的寄存器中：

$|q\rangle = |x\rangle \otimes |v\rangle \otimes |m\rangle \otimes |dm/dt\rangle$

初始化状态是通过 Hadamard 门生成的均匀叠加态：

$|s\rangle = H^{\otimes 4n} |0\rangle^{4n} = (1/\sqrt{M}) \sum |x, v, m, dm/dt\rangle$

每个组合同时存在，初始概率振幅为 $1/\sqrt{M}$。

4. 量子处理算法

算法通过以下逻辑步骤进行 NKTg 趋势分析：

- **第一步**： 接收输入参数 (x, v, m, dm/dt) 并将其编码到 4n 量子比特寄存器中。应用 Hadamard 门在所有 $M = N^4$ 个组合上创建均匀叠加。
- **第二步**： 根据所选的趋势条件 f 构建参数化的 Oracle $\hat{O}$。该 Oracle 执行三个可逆操作：
    - **Compute：** 计算 $p = m * v$, $NKTg\_1 = x * p$, $NKTg\_2 = (dm/dt) * p$ 并存入辅助寄存器 (ancilla)。
    - **Phase kickback：** 翻转满足 f 的状态相位。
    - **Uncompute：** 将辅助寄存器恢复为 $|0\rangle$。

      若 $f(x,v,m,dm/dt) = 1$， 则 $\hat{O} |x,v,m,dm/dt\rangle = -|x,v,m,dm/dt\rangle$

      若 $f(x,v,m,dm/dt) = 0$， 则 $\hat{O} |x,v,m,dm/dt\rangle = +|x,v,m,dm/dt\rangle$

- **第三步**： 应用扩散算子 (Diffusion operator) —— 放大解状态的振幅，抑制不满足条件状态的振幅：

  $\hat{D} = 2|s\rangle\langle s| - I$

- **第四步**： 重**复第二步和第三步共** T 个最优轮次。设 k 为空间 M 中的解数量：

  $T = \lfloor (\pi/4) * \sqrt{M/k} \rfloor$ 次迭代。

  T 轮后，测量到正确解的概率趋于 1。

- **第五步**： 测量 4n 量子比特寄存器 —— 以高概率获得满足条件 f 的组合 (x, v, m, dm/dt)。重**复** O(k) 次以收集全部 k 个解。

5. 经典模型与量子模型的比较

标准	经典模型	量子模型
搜索空间	M = N^4 个组合	M = N^4 = 2^{4n} 个状态
处理方式	按组合顺序执行	量子并行处理
复杂度	O(M) = O(N^4)	O(√M) = O(N^2)
加速效果	—	相比经典模型呈平方级加速
资源需求	无限制内存	4n + ancilla 量子比特
搜索条件	每次运行一个条件 f	每次运行一个条件 f

6. 科学意义

NKTg 量子模型保留了 NKTg 定律的所有物理本质（即 NKTg_1, NKTg_2 量和趋势分类标准），同时利用 Grover 算法将搜索复杂度从 O(M) = O(N^4) 降低到 O(√M) = O(N^2)。Oracle 设计为参数化结构，允许灵活应用于 NKTg 集合中的任何趋势条件。这是从经典模型向量子计算迈出的自然扩展步骤，为复杂物理系统模拟和大规模参数优化开辟了应用方向。

```
//
================================================================================
// NKTg 算法 — 量子版 (OpenQASM 3.0) — 完整修复版
// 作者：Nguyễn Khánh Tùng
// © 2026 Nguyễn Khánh Tùng。保留所有权利。
// --------------------------------------------------------------------------------
// 根据审核进行的修复：
//   [1] 使用 mcx 替换 ccx+cx 以检测 dm/dt=0
//   [2] 使用 mcx 替换 ccx+cx 以处理 MODE 5 条件
//   [3] Uncompute MCZ 阶段不再出现重复的 ccx 循环
//   [4] Oracle 被封装在标准的 OpenQASM 3.0 子程序 (def subroutine) 中
//
================================================================================
```

```
OPENQASM 3.0;
include "stdgates.inc";

//
// ============================================================================
// 寄存器声明
//
// ============================================================================
qubit[3] x_reg;
qubit[3] v_reg;
qubit[3] m_reg;
qubit[3] d_reg;
qubit[5] anc;
qubit    flag;

bit[3] c_x;
bit[3] c_v;
bit[3] c_m;
bit[3] c_d;

//
// ============================================================================
// 子程序：ORACLE NKTg (模式 5: NKTg1>0 且 NKTg2<0 且 dm/dt≠0)
// 结构：计算 (Compute) → 相位翻转 (Phase kickback) → 逆计算 (Uncompute) (绝对对称)
//
// ============================================================================
def nktg_oracle(qubit[3] xr, qubit[3] vr, qubit[3] mr,
                qubit[3] dr, qubit[5] a, qubit f) {

    // ------------------------------------------------------------------------
    // 计算 (COMPUTE)
    // ------------------------------------------------------------------------

    // 步骤 2a: sign(p) = sign(m) XOR sign(v)  →  a[0]
    cx mr[2], a[0];
    cx vr[2], a[0];         // a[0] = sign(m) 异或 sign(v) = sign(p)

    // 步骤 2b: sign(NKTg1) = sign(x) XOR sign(p)  →  a[1]
    //          NKTg1 > 0  ⇔  a[1] = 0
    cx xr[2], a[1];
    cx a[0],  a[1];         // a[1] = sign(NKTg1)

    // 步骤 2c: sign(NKTg2) = sign(d) XOR sign(p)  →  a[2]
    //          NKTg2 < 0  ⇔  a[2] = 1
    cx dr[2], a[2];
    cx a[0],  a[2];         // a[2] = sign(NKTg2)

    // 步骤 2d: 检测 dm/dt = 0  →  a[3]
    //          dm/dt = 0  ⇔  dr[2]=0 且 dr[1]=0 且 dr[0]=0
    //          经过 X 门后: dr[2]=1 且 dr[1]=1 且 dr[0]=1
    //          修复 [1]: 使用 3 控制位的 mcx 替换错误的 ccx+cx
    x dr[0]; x dr[1];  x dr[2];
    mcx dr[0], dr[1], dr[2], a[3]; // 当 dm/dt=0 时 a[3]=1
    x dr[0]; x dr[1];  x dr[2];    // 撤销 X 门
```

```
    // 步骤 2e: 完整的模式 5 条件  →  a[4]
    //           f = NKTg1>0 且 NKTg2<0 且 dm/dt≠0
    //           NKTg1>0  ⇔  a[1]=0 → X(a[1])
    //           NKTg2<0  ⇔  a[2]=1
    //           dm/dt≠0  ⇔  a[3]=0 → X(a[3])
    //           修复 [2]: 使用 3 控制位的 mcx 替换错误的 ccx+cx
    x a[1]; x a[3];
    mcx a[1], a[2], a[3], a[4];    // 当 f=1 时 a[4]=1

    // ------------------------------------------------------------------------
---
    // 相位翻转 (PHASE KICKBACK)
    // flag=|-⟩: CX(a[4], flag) → 翻转所有满足条件 f 的状态相位
    // ------------------------------------------------------------------------
---
    cx a[4], f;

    // ------------------------------------------------------------------------
---
    // 逆计算 (UNCOMPUTE) — 与计算阶段绝对对称（顺序相反）
    // ------------------------------------------------------------------------
---
    mcx a[1], a[2], a[3], a[4];    // 撤销步骤 2e
    x a[1]; x a[3];

    x dr[0]; x dr[1];  x dr[2];
    mcx dr[0], dr[1], dr[2], a[3]; // 撤销步骤 2d
    x dr[0]; x dr[1];  x dr[2];

    cx a[0],  a[2];                 // 撤销步骤 2c
    cx dr[2], a[2];

    cx a[0],  a[1];                 // 撤销步骤 2b
    cx xr[2], a[1];

    cx vr[2], a[0];                 // 撤销步骤 2a
    cx mr[2], a[0];
    // a[0..4] = |00000⟩ — 已完全恢复
}

//
==========================================================================
// 子程序：扩散算子 D^ = 2|s⟩⟨s| - I
// H → X → MCZ(12 qubit) → X → H
// 修复 [3]: Uncompute MCZ 阶段不再出现重复的 ccx 循环
//
==========================================================================
def grover_diffusion(qubit[3] xr, qubit[3] vr,
                     qubit[3] mr, qubit[3] dr, qubit[5] a) {

    // 3a: 对全部 12 个量子比特应用 H 门
    h xr[0]; h xr[1]; h xr[2];
    h vr[0]; h vr[1]; h vr[2];
    h mr[0]; h mr[1]; h mr[2];
    h dr[0]; h dr[1]; h dr[2];
```

```
// 3b: 对全部应用 X 门 → 映射 |0...0⟩ → |1...1⟩
x xr[0]; x xr[1]; x xr[2];
x vr[0]; x vr[1]; x vr[2];
x mr[0]; x mr[1]; x mr[2];
x dr[0]; x dr[1]; x dr[2];

// 3c: MCZ(12 qubit) = H(目标位) → MCX(12 qubit) → H(目标位)
// 将 MCX(12) 分解为 Toffoli 链 — 重用已清空的 a[0..4]
h dr[2];

// 第 1 层：两两配对
ccx xr[0], xr[1], a[0];  // a[0] = xr[0] 与 xr[1] 的与运算
ccx xr[2], vr[0], a[1];  // a[1] = xr[2] 与 vr[0] 的与运算
ccx vr[1], vr[2], a[2];  // a[2] = vr[1] 与 vr[2] 的与运算
ccx mr[0], mr[1], a[3];  // a[3] = mr[0] 与 mr[1] 的与运算
ccx mr[2], dr[0], a[4];  // a[4] = mr[2] 与 dr[0] 的与运算

// 第 2 层：合并第 1 层结果
ccx a[0], a[1], a[0];       // a[0] = 前 4 位相与
// 注意：需要中间量子比特 — uncompute 后重用
// 不重叠量子比特的完整分解：
ccx a[0], a[2], a[1];      // a[1] = a[0] 与 a[2] 的与运算
ccx a[3], a[4], a[2];      // a[2] = a[3] 与 a[4] 的与运算
ccx a[1], a[2], a[3];      // a[3] = 全部 10 位
ccx dr[1], a[3], a[4];     // a[4] = 11 位相与
cx  a[4], dr[2];           // MCX: 当 12 位全部为 1 时翻转 dr[2]

// 第 2 层逆计算 (顺序相反，无额外重复 — 修复 [3])
ccx dr[1], a[3], a[4];
ccx a[1], a[2], a[3];
ccx a[3], a[4], a[2];
ccx a[0], a[2], a[1];
ccx a[0], a[1], a[0];

// 第 1 层逆计算
ccx mr[2], dr[0], a[4];
ccx mr[0], mr[1], a[3];
ccx vr[1], vr[2], a[2];
ccx xr[2], vr[0], a[1];
ccx xr[0], xr[1], a[0];

h dr[2];                    // 撤销 H 门 → 完整的 MCZ 完成

// 3d: 撤销 X 门
x xr[0]; x xr[1]; x xr[2];
x vr[0]; x vr[1]; x vr[2];
x mr[0]; x mr[1]; x mr[2];
x dr[0]; x dr[1]; x dr[2];

// 3e: 第二次应用 H 门 — 完成扩散算子 D^
```

```
    h xr[0]; h xr[1]; h xr[2];
    h vr[0]; h vr[1]; h vr[2];
    h mr[0]; h mr[1]; h mr[2];
    h dr[0]; h dr[1]; h dr[2];
}

//
================================================================================
// 主程序
//
================================================================================

// 将 flag 初始化为 |−⟩ 以便相位翻转
x flag;
h flag;

// 第一步：生成均匀叠加态
h x_reg[0]; h x_reg[1]; h x_reg[2];
h v_reg[0]; h v_reg[1]; h v_reg[2];
h m_reg[0]; h m_reg[1]; h m_reg[2];
h d_reg[0]; h d_reg[1]; h d_reg[2];

// Grover 循环 T 轮
// T = floor((π/4) × sqrt(M/k))
// M=4096, k=1 → T≈50 | k=64 → T=6 | k=1024 → T=1
for uint i in [1:1] {
    nktg_oracle(x_reg, v_reg, m_reg, d_reg, anc, flag);
    grover_diffusion(x_reg, v_reg, m_reg, d_reg, anc);
}

// 第五步：测量
c_x = measure x_reg;   // 010 → x=2.0 正 ✓
c_v = measure v_reg;   // 011 → v=3.0 正 ✓
c_m = measure m_reg;   // 001 → m=5.0 正 ✓
c_d = measure d_reg;   // 101 → dm/dt=-0.5 负 ✓
                        // 000 → photon: Oracle 未标记 ✓
```

5. German (Deutsch)

BEKANNTMACHUNG DER SPEZIFIKATIONEN FÜR SOFTWARE UND ALGORITHMEN DES NKTg-GESETZES ÜBER VARIABLE TRÄGHEIT

I. ALLGEMEINE INFORMATIONEN

- **Softwarename:**

  Berechnungssystem für variable Trägheit NKTg (NKTg Dynamics Calculator).

- **Autor:**

Nguyễn Khánh Tùng

- **Programmiersprachen:**

  C++ (ISO/IEC 14882) und Assembly (x64).

II. THEORETISCHE BASIS: NKTg-GESETZ ÜBER VARIABLE TRÄGHEIT

Die Software wird auf der Grundlage der Prinzipien des NKTg-Gesetzes ausgeführt:

- **Grundlegende Beziehung:**

  Die Bewegungstendenz hängt von der Position (x), der Geschwindigkeit (v) und der Masse (m) ab. Formel: NKTg = f(x, v, m).

- **Kernproduktgrößen:**
  - Impuls (p): p = m * v
  - Größe NKTg_1: Produkt aus Position und Impuls (NKTg_1 = x * p).
  - Größe NKTg_2: Produkt aus der Änderungsrate der Masse und dem Impuls (NKTg_2 = (dm/dt) * p).
- **Maßeinheit:**

  NKTm (Einheit der variablen Trägheit).

III. DETAILLIERTE ALGORITHMUS-BESCHREIBUNG

1. Datenverarbeitungsprozess

Der Algorithmus führt die Analyse der Bewegungstendenz über die folgenden logischen Schritte durch:

- **Schritt 1:**

  Empfang der Eingangsparameter: Position (x), Geschwindigkeit (v), Masse (m) und Massenänderungsrate (dm/dt).

- **Schritt 2:**

  Berechnung des linearen Impulses p = m * v.

- **Schritt 3:**

  Berechnung der variablen Trägheitswerte NKTg_1 und NKTg_2.

- **Schritt 4:**

  Klassifizierung der Tendenz (Tendency) basierend auf dem Vorzeichen der Werte:

  - NKTg_1 > 0: Entfernung vom stabilen Zustand.
  - NKTg_1 < 0: Annäherung an den stabilen Zustand.
  - NKTg_2 > 0: Die Massenänderung unterstützt die Bewegung.
  - NKTg_2 < 0: Die Massenänderung behindert die Bewegung.

IV. QUELLCODE DER AUSFÜHRUNG (SOURCE CODE)

1. Hochsprachen-Quellcode (C++)

C++

```
#include <cstdio>

/**
 * Bibliothek zur Berechnung der variablen Trägheit nach dem NKTg-Gesetz
 * Alle Urheberrechte bezüglich der Berechnungslogik und der
Tendenzdefinitionen vorbehalten.
 */
int main() {
   // 1. Deklaration der Parameter: x (Position), v (Geschwindigkeit), m
(Masse), dm_dt (Änderungsrate von m)
   double x = 2.0, v = 3.0, m = 5.0, dm_dt = -0.5;

   // 2. Ausführung der Berechnung nach dem NKTg-Gesetz
   double p = m * v;        // p = m * v
   double n1 = x * p;       // NKTg1 = x * p
   double n2 = dm_dt * p;   // NKTg2 = (dm/dt) * p

   // 3. Anzeige der Ergebnisse im Record-Format
   printf("{ p = %.1f\n  nktg1 = %.1f\n  nktg2 = %.1f\n", p, n1, n2);

   // Klassifizierung der Bewegungstendenz (Tendency Classification)
   printf("  tendency1 = \"%s\"\n",
           n1 > 0 ? "Entfernung vom stabilen Zustand" :
           n1 < 0 ? "Annäherung an den stabilen Zustand" : "Stabiles
Gleichgewicht");

   printf("  tendency2 = \"%s\" }",
           n2 > 0 ? "Massenänderung unterstützt die Bewegung" :
           n2 < 0 ? "Massenänderung behindert die Bewegung" : "Kein Effekt
durch Massenänderung");

   return 0;
}
```

2. Low-Level-Quellcode (Assembly x64)

```
; ------------------------------------------------------------------------
; ALGORITHMUS ZUR ANALYSE DER BEWEGUNGSTENDENZ NKTg (x64 NASM)
; Autor: Nguyễn Khánh Tùng
; © 2026 Nguyễn Khánh Tùng. Alle Rechte vorbehalten.
; ------------------------------------------------------------------------
; Beispiel-Daten (aus dem Originaldokument):
;   x=2.0, v=3.0, m=5.0, dm/dt=-0.5
;   p=15.0 | NKTg1=+30.0 → Entfernung | NKTg2=-7.5 → Behinderung
;
; Funktion analyze_nktg_tendency — System V AMD64 ABI (Linux):
;   Eingabe: xmm0=x, xmm1=v, xmm2=m, xmm3=dm/dt
;   Ausgabe: rax=0 (OK), rax=-1 (NaN-Fehler)
; ------------------------------------------------------------------------

section .data
    fmt_all  db "{ p = %.1f", 10,
             db "  nktg1 = %.1f", 10,
             db "  nktg2 = %.1f", 10,
             db "  tendency1 = ""%s""", 10,
```

```
            db "  tendency2 = ""%s"" }", 10, 0

    str_away    db "Entfernung vom stabilen Zustand", 0
    str_toward  db "Annäherung an den stabilen Zustand", 0
    str_stable  db "Stabiles Gleichgewicht", 0
    str_support db "Massenänderung unterstützt die Bewegung", 0
    str_resist  db "Massenänderung behindert die Bewegung", 0
    str_noeff   db "Kein Effekt durch Massenänderung", 0

    val_x     dq 2.0
    val_v     dq 3.0
    val_m     dq 5.0
    val_dm_dt dq -0.5

section .text
    extern printf
    global analyze_nktg_tendency
    global main

;
================================================================================
; analyze_nktg_tendency
;
================================================================================
analyze_nktg_tendency:
    push rbp
    mov rbp, rsp
    and rsp, -16

    ; --- Schritt 1: Empfang der Eingangsparameter über Register ---
    ; xmm0 = x      (Position)
    ; xmm1 = v      (Geschwindigkeit)
    ; xmm2 = m      (Masse)
    ; xmm3 = dm/dt  (Massenänderungsrate)
    ; Die Register xmm0–xmm3 werden geschützt, kein direktes Überschreiben

    ; --- Schritt 2: Berechnung des Impulses p = m * v ---
    movsd xmm4, xmm2           ; xmm4 = m
    mulsd xmm4, xmm1           ; xmm4 = p = 5.0 * 3.0 = 15.0

    ; --- Schritt 3: Berechnung von NKTg1 und NKTg2 ---
    movsd xmm5, xmm0           ; xmm5 = x
    mulsd xmm5, xmm4           ; xmm5 = NKTg1 = x * p = 2.0 * 15.0 = 30.0

    movsd xmm6, xmm3           ; xmm6 = dm/dt
    mulsd xmm6, xmm4           ; xmm6 = NKTg2 = (dm/dt) * p = (-0.5) * 15.0 =
-7.5

    ; --- Schritt 4: Klassifizierung der Tendenz basierend auf dem Vorzeichen
---
    xorps xmm7, xmm7           ; xmm7 = 0.0

    ; 4.1. Klassifizierung von NKTg1 → temporäre Speicherung in r8
    ucomisd xmm5, xmm7         ; Vergleiche NKTg1 mit 0.0
    jp .error_nan             ; PF=1 → NaN (Not-a-Number)
    lea r8, [rel str_stable]   ; Standard: NKTg1 = 0 → Stabiles Gleichgewicht
    ja .t1_away               ; NKTg1 > 0
    jb .t1_toward             ; NKTg1 < 0
    jmp .check_nktg2
.t1_away:
    lea r8, [rel str_away]     ; "Entfernung vom stabilen Zustand"
    jmp .check_nktg2
.t1_toward:
```

```
    lea r8, [rel str_toward]  ; "Annäherung an den stabilen Zustand"

    ; 4.2. Klassifizierung von NKTg2 → temporäre Speicherung in r9
.check_nktg2:
    ucomisd xmm6, xmm7         ; Vergleiche NKTg2 mit 0.0
    jp .error_nan             ; NaN-Schutz
    lea r9, [rel str_noeff]   ; Standard: NKTg2 = 0 → Kein Effekt durch
Massenänderung
    ja .t2_support            ; NKTg2 > 0
    jb .t2_resist             ; NKTg2 < 0
    jmp .do_printf
.t2_support:
    lea r9, [rel str_support] ; "Massenänderung unterstützt die Bewegung"
    jmp .do_printf
.t2_resist:
    lea r9, [rel str_resist]  ; "Massenänderung behindert die Bewegung"

    ; --- Einmaliger Aufruf von printf ---
.do_printf:
    lea rdi, [rel fmt_all]    ; rdi = Formatstring
    mov rsi, r8               ; rsi = Zeiger tendency1 (aus r8)
    mov rdx, r9               ; rdx = Zeiger tendency2 (aus r9)
    movsd xmm0, xmm4           ; xmm0 = p     = 15.0
    movsd xmm1, xmm5           ; xmm1 = NKTg1 = 30.0
    movsd xmm2, xmm6           ; xmm2 = NKTg2 = -7.5
    mov eax, 3                ; 3 Gleitkommaparameter
    call printf

    xor eax, eax              ; return 0
    leave
    ret

.error_nan:
    mov rax, -1
    leave
    ret

; ========================================================================
; main — Beispieldaten in Register laden, analyze_nktg_tendency aufrufen
; ========================================================================
main:
    push rbp
    mov rbp, rsp
    and rsp, -16

    ; --- Schritt 1: Laden der Eingangsparameter (Daten aus Originaldokument)
---
    movsd xmm0, [rel val_x]     ; xmm0 = x     = 2.0
    movsd xmm1, [rel val_v]     ; xmm1 = v     = 3.0
    movsd xmm2, [rel val_m]     ; xmm2 = m     = 5.0
    movsd xmm3, [rel val_dm_dt] ; xmm3 = dm/dt = -0.5

    call analyze_nktg_tendency

    xor eax, eax
    leave
    ret

; ------------------------------------------------------------------------
; Erwartetes Ergebnis:
; { p = 15.0
;   nktg1 = 30.0
;   nktg2 = -7.5
```

```
;   tendency1 = "Entfernung vom stabilen Zustand"
;   tendency2 = "Massenänderung behindert die Bewegung" }
; ---------------------------------------------------------------------------
```

V. DEFINITION DES STABILEN ZUSTANDS

Der stabile Zustand in dieser Software ist definiert als jener Zustand, in dem Position (x), Geschwindigkeit (v) und Masse (m) so interagieren, dass die Bewegungsstruktur aufrechterhalten wird, Kontrollverlust vermieden wird und das inhärente Bewegungsmuster des Objekts bewahrt bleibt.

VI. QUANTENERWEITERUNG: NKTg-ALGORITHMUS AUF QUANTENCOMPUTER-PLATTFORMEN

1. Quantentheoretische Basis

Die Quantenerweiterung bewahrt die Gesamtheit der fundamentalen Größen des NKTg-Gesetzes:

- **Impuls (p):** p = m * v
- **Größe NKTg_1:** Produkt aus Position und Impuls (NKTg_1 = x * p)
- **Größe NKTg_2:** Produkt aus Massenänderungsrate und Impuls (NKTg_2 = (dm/dt) * p)
- **Maßeinheit:** NKTm (Einheit der variablen Trägheit)

Anstatt jede Kombination (x, v, m, dm/dt) wie im klassischen Modell sequenziell zu berechnen, kodiert das Quantenmodell den gesamten Parameterraum in einem Superpositionszustand und nutzt Quantenparallelität für die Suche.

2. Quantenphysikalische Problemstellung

- **Eingabe:** Parameterraum (x, v, m, dm/dt), wobei jede Variable in N = 2^n Werteebenen diskretisiert und mit n Qubits kodiert wird. Der gesamte Suchraum umfasst M = N^4 = 2^{4n} Kombinationen.
- **Problem:** Für eine gewählte Tendenzbedingung f aus der Menge:

Bedingung f	**Tendenz**
NKTg_1 > 0	Entfernung vom stabilen Zustand
NKTg_1 < 0	Annäherung an den stabilen Zustand
NKTg_2 > 0	Massenänderung unterstützt die Bewegung
NKTg_2 < 0	Massenänderung behindert die Bewegung
NKTg_1 > 0 und NKTg_2 < 0	Kombination beider Bedingungen

Finde alle Kombinationen (x, v, m, dm/dt) ∈{1..N}^4, welche die gewählte Bedingung f erfüllen.

- **Ausgabe:** Die Menge der Parameterkombinationen, die f erfüllen, erhalten nach einer Quantenmessung.

3. Quantenzustandsraum

Der gesamte Parameterraum wird in einem Quantenregister von 4n Qubits kodiert:

$|q\rangle = |x\rangle \otimes |v\rangle \otimes |m\rangle \otimes |dm/dt\rangle$

Der Anfangszustand ist eine gleichförmige Superposition über Hadamard-Gatter:

$|s\rangle = H^{\otimes 4n} |0\rangle^{4n} = (1/\sqrt{M}) \sum |x, v, m, dm/dt\rangle$

Jede Kombination existiert gleichzeitig mit einer anfänglichen Wahrscheinlichkeitsamplitude von $1/\sqrt{M}$.

4. Quantenverarbeitungs-Algorithmus

Der Algorithmus führt die NKTg-Tendenzanalyse über die folgenden logischen Schritte durch:

- **Schritt 1:** Empfang und Kodierung der Eingangsparameter (x, v, m, dm/dt) im Register von 4n Qubits. Anwendung von Hadamard-Gattern zur Erzeugung einer gleichförmigen Superposition über alle M = N^4 Kombinationen.
- **Schritt 2:** Konstruktion eines parametrisierten Orakels $\hat{\hat{O}}$ gemäß der gewählten Tendenzbedingung f. Das Orakel führt drei reversible Operationen aus:
  - **Compute:** Berechnung von p = m * v, NKTg_1 = x * p, NKTg_2 = (dm/dt) * p auf dem Ancilla-Register.
  - **Phase kickback:** Invertierung der Phase der Zustände, die f erfüllen.
  - **Uncompute:** Wiederherstellung des Ancilla-Registers auf $|0\rangle$.

  $\hat{\hat{O}} |x,v,m,dm/dt\rangle = -|x,v,m,dm/dt\rangle$, wenn f(x,v,m,dm/dt) = 1

  $\hat{\hat{O}} |x,v,m,dm/dt\rangle = +|x,v,m,dm/dt\rangle$, wenn f(x,v,m,dm/dt) = 0

- **Schritt 3:** Anwendung des Diffusionsoperators – Verstärkung der Amplitude der Lösungszustände, Unterdrückung der Amplitude nicht erfüllender Zustände:

  $\hat{D} = 2|s\rangle\langle s| - I$

- **Schritt 4:** Wiederholung von Schritt 2 und Schritt 3 für T optimale Zyklen. Mit k als Anzahl der Lösungen im Raum M:

  $T = \lfloor(\pi/4) * \sqrt{M/k}\rfloor$ Zyklen.

  Nach T Zyklen geht die Wahrscheinlichkeit, die korrekte Lösung zu messen, gegen 1.

- **Schritt 5:** Messung des Registers von 4n Qubits – Erhalt der Kombination (x, v, m, dm/dt), die die Bedingung f mit hoher Wahrscheinlichkeit erfüllt. Wiederholung O(k) mal, um die gesamte Menge der k Lösungen zu sammeln.

5. Vergleich zwischen klassischem und Quantenmodell

Kriterien	**Klassisches Modell**	**Quantenmodell**
Suchraum	M = N^4 Kombinationen	M = N^4 = 2^{4n} Zustände
Verarbeitungsmethode	Sequenziell pro Kombination	Quantenparallelität
Komplexität	O(M) = O(N^4)	O(√M) = O(N^2)
Beschleunigung	—	Quadratisch gegenüber klassisch
Ressourcen	Unbegrenzter Speicher	4n + Ancilla Qubits
Suchbedingung	Eine Bedingung f pro Lauf	Eine Bedingung f pro Lauf

6. Wissenschaftliche Bedeutung

Das NKTg-Quantenmodell bewahrt die physikalische Natur des NKTg-Gesetzes – die Größen NKTg_1, NKTg_2 und die Kriterien für die Tendenzklassifizierung – während es den Grover-Algorithmus nutzt, um die Suchkomplexität von O(M) = O(N^4) auf O(√M) = O(N^2) zu reduzieren. Das Orakel ist parametrisiert aufgebaut, was eine flexible Anwendung auf jede Tendenzbedingung innerhalb der NKTg-Menge ermöglicht. Dies stellt einen natürlichen Erweiterungsschritt vom klassischen Modell zur Quanteninformatik dar und eröffnet Perspektiven für die Simulation komplexer physikalischer Systeme sowie die großflächige Parameteroptimierung.

```
//
// ============================================================================
// NKTg-ALGORITHMUS — QUANTEN (OpenQASM 3.0) — Vollständige korrigierte
Version
// Autor: Nguyễn Khánh Tùng
// © 2026 Nguyễn Khánh Tùng. Alle Rechte vorbehalten.
// ----------------------------------------------------------------------------
---
// Korrekturen nach Audit:
//   [1] mcx ersetzt ccx+cx für die dm/dt=0 Erkennung
//   [2] mcx ersetzt ccx+cx für die Bedingung MODE 5
//   [3] Uncompute MCZ wiederholt nicht mehr unnötig die ccx
//   [4] Orakel in einer standardmäßigen OpenQASM 3.0 Subroutine gekapselt
//
// ============================================================================

OPENQASM 3.0;
include "stdgates.inc";
```

```
//
// ============================================================================
// DEKLARATIONEN DER REGISTER
//
// ============================================================================
qubit[3] x_reg;
qubit[3] v_reg;
qubit[3] m_reg;
qubit[3] d_reg;
qubit[5] anc;
qubit    flag;

bit[3] c_x;
bit[3] c_v;
bit[3] c_m;
bit[3] c_d;

//
// ============================================================================
// SUBROUTINE : NKTg-ORAKEL (MODE 5 : NKTg1>0 UND NKTg2<0 UND dm/dt≠0)
// Struktur : Compute → Phase kickback → Uncompute (absolute Symmetrie)
//
// ============================================================================
def nktg_oracle(qubit[3] xr, qubit[3] vr, qubit[3] mr,
                qubit[3] dr, qubit[5] a, qubit f) {

    // ------------------------------------------------------------------------
---
    // COMPUTE
    // ------------------------------------------------------------------------
---

    // Schritt 2a : sign(p) = sign(m) XOR sign(v)  →  a[0]
    cx mr[2], a[0];
    cx vr[2], a[0];         // a[0] = sign(m) XOR sign(v) = sign(p)

    // Schritt 2b : sign(NKTg1) = sign(x) XOR sign(p)  →  a[1]
    //          NKTg1 > 0  ⇔  a[1] = 0
    cx xr[2], a[1];
    cx a[0],  a[1];         // a[1] = sign(NKTg1)

    // Schritt 2c : sign(NKTg2) = sign(d) XOR sign(p)  →  a[2]
    //          NKTg2 < 0  ⇔  a[2] = 1
    cx dr[2], a[2];
    cx a[0],  a[2];         // a[2] = sign(NKTg2)

    // Schritt 2d : Erkennung dm/dt = 0  →  a[3]
    //          dm/dt = 0  ⇔  dr[2]=0 UND dr[1]=0 UND dr[0]=0
    //          Nach X : dr[2]=1 UND dr[1]=1 UND dr[0]=1
    //          FIX [1] : nutzt mcx mit 3 Kontrollen anstatt fehlerhaftem
ccx+cx
    x dr[0]; x dr[1];  x dr[2];
    mcx dr[0], dr[1], dr[2], a[3]; // a[3]=1 wenn dm/dt=0
    x dr[0]; x dr[1];  x dr[2];    // X rückgängig machen

    // Schritt 2e : Vollständige Bedingung MODE 5  →  a[4]
    //          f = NKTg1>0 UND NKTg2<0 UND dm/dt≠0
    //          NKTg1>0  ⇔  a[1]=0 → X(a[1])
    //          NKTg2<0  ⇔  a[2]=1
    //          dm/dt≠0  ⇔  a[3]=0 → X(a[3])
    //          FIX [2] : mcx mit 3 Kontrollen anstatt fehlerhaftem ccx+cx
    x a[1]; x a[3];
    mcx a[1], a[2], a[3], a[4];    // a[4]=1 wenn f=1
```

```
    // ------------------------------------------------------------------------
---
    // PHASE KICKBACK
    // flag=|−⟩ : CX(a[4], flag) → invertiert die Phase aller Zustände, die f
erfüllen
    // ------------------------------------------------------------------------
---
    cx a[4], f;

    // ------------------------------------------------------------------------
---
    // UNCOMPUTE — absolute Symmetrie zu Compute (umgekehrte Reihenfolge)
    // ------------------------------------------------------------------------
---
    mcx a[1], a[2], a[3], a[4];    // Schritt 2e rückgängig
    x a[1]; x a[3];

    x dr[0]; x dr[1];  x dr[2];
    mcx dr[0], dr[1], dr[2], a[3]; // Schritt 2d rückgängig
    x dr[0]; x dr[1];  x dr[2];

    cx a[0],  a[2];                // Schritt 2c rückgängig
    cx dr[2], a[2];

    cx a[0],  a[1];                // Schritt 2b rückgängig
    cx xr[2], a[1];

    cx vr[2], a[0];                // Schritt 2a rückgängig
    cx mr[2], a[0];
    // a[0..4] = |00000⟩ — vollständig wiederhergestellt
}

//
================================================================================
// SUBROUTINE : DIFFUSION D^ = 2|s⟩⟨s| − I
// H → X → MCZ(12 Qubit) → X → H
// FIX [3] : Uncompute MCZ wiederholt nicht mehr unnötig die ccx
//
================================================================================
def grover_diffusion(qubit[3] xr, qubit[3] vr,
                     qubit[3] mr, qubit[3] dr, qubit[5] a) {

    // 3a : H auf allen 12 Qubits
    h xr[0]; h xr[1]; h xr[2];
    h vr[0]; h vr[1]; h vr[2];
    h mr[0]; h mr[1]; h mr[2];
    h dr[0]; h dr[1]; h dr[2];

    // 3b : X überall → mappe |0...0⟩ → |1...1⟩
    x xr[0]; x xr[1]; x xr[2];
    x vr[0]; x vr[1]; x vr[2];
    x mr[0]; x mr[1]; x mr[2];
    x dr[0]; x dr[1]; x dr[2];

    // 3c : MCZ(12 Qubit) = H(Ziel) → MCX(12 Qubit) → H(Ziel)
    // Zerlege MCX(12) in Toffoli-Kette — nutze saubere a[0..4]
    h dr[2];

    // Ebene 1 : Gruppierung paarweise
    ccx xr[0], xr[1], a[0];  // a[0] = xr[0] AND xr[1]
    ccx xr[2], vr[0], a[1];  // a[1] = xr[2] AND vr[0]
    ccx vr[1], vr[2], a[2];  // a[2] = vr[1] AND vr[2]
```

```
    ccx mr[0], mr[1], a[3];  // a[3] = mr[0] AND mr[1]
    ccx mr[2], dr[0], a[4];  // a[4] = mr[2] AND dr[0]

    // Ebene 2 : Kombiniere Ergebnisse von Ebene 1
    ccx a[0], a[1], a[0];      // a[0] = erste 4 Bits UND-verknüpft
    // Hinweis: Zwischen-Qubits benötigt — Wiederverwendung nach Uncompute
    // Vollständige Zerlegung ohne Qubit-Überschneidung:
    ccx a[0], a[2], a[1];     // a[1] = a[0] AND a[2]
    ccx a[3], a[4], a[2];     // a[2] = a[3] AND a[4]
    ccx a[1], a[2], a[3];     // a[3] = alle 10 Bits
    ccx dr[1], a[3], a[4];    // a[4] = alle 11 Bits UND-verknüpft
    cx  a[4], dr[2];          // MCX : kippe dr[2], wenn alle 12 Bits = 1

    // Uncompute Ebene 2 (umgekehrt, keine Wiederholungen — FIX [3])
    ccx dr[1], a[3], a[4];
    ccx a[1], a[2], a[3];
    ccx a[3], a[4], a[2];
    ccx a[0], a[2], a[1];
    ccx a[0], a[1], a[0];

    // Uncompute Ebene 1
    ccx mr[2], dr[0], a[4];
    ccx mr[0], mr[1], a[3];
    ccx vr[1], vr[2], a[2];
    ccx xr[2], vr[0], a[1];
    ccx xr[0], xr[1], a[0];

    h dr[2];                   // H rückgängig → MCZ vollständig
abgeschlossen

    // 3d : X rückgängig
    x xr[0]; x xr[1]; x xr[2];
    x vr[0]; x vr[1]; x vr[2];
    x mr[0]; x mr[1]; x mr[2];
    x dr[0]; x dr[1]; x dr[2];

    // 3e : Zweites H — schließt Dˆ ab
    h xr[0]; h xr[1]; h xr[2];
    h vr[0]; h vr[1]; h vr[2];
    h mr[0]; h mr[1]; h mr[2];
    h dr[0]; h dr[1]; h dr[2];
}

//
================================================================================
// HAUPTPROGRAMM
//
================================================================================

// Initialisierung des Flags auf |-⟩ für Phase Kickback
x flag;
h flag;

// Schritt 1 : Gleichförmige Superposition
h x_reg[0]; h x_reg[1]; h x_reg[2];
h v_reg[0]; h v_reg[1]; h v_reg[2];
h m_reg[0]; h m_reg[1]; h m_reg[2];
h d_reg[0]; h d_reg[1]; h d_reg[2];

// Grover-Schleife T Zyklen
// T = floor((π/4) × sqrt(M/k))
// M=4096, k=1 → T≈50 | k=64 → T=6 | k=1024 → T=1
for uint i in [1:1] {
```

```
    nktg_oracle(x_reg, v_reg, m_reg, d_reg, anc, flag);
    grover_diffusion(x_reg, v_reg, m_reg, d_reg, anc);
}

// Schritt 5 : Messung
c_x = measure x_reg;   // 010 → x=2.0 positiv ✓
c_v = measure v_reg;   // 011 → v=3.0 positiv ✓
c_m = measure m_reg;   // 001 → m=5.0 positiv ✓
c_d = measure d_reg;   // 101 → dm/dt=-0.5 negativ ✓
                        // 000 → Photonen: Orakel nicht markiert ✓
```

6. Japanese (日本語)

NKTg 変位慣性法則に関するソフトウェアおよびアルゴリズム仕様公表

I. 基本情報

- **ソフトウェア名:**

  NKTg 変位慣性計算システム (NKTg Dynamics Calculator)

- **著者:**

  Nguyễn Khánh Tùng

- **プログラミング言語:**

  C++ (ISO/IEC 14882) および Assembly (x64)

II. 理論的基礎：NKTg 変位慣性法則

本ソフトウェアは、NKTg法則の原理に基づいて実行されます。

- **基本関係式:**

  運動の傾向は、位置 (x)、速度 (v)、および質量 (m) に依存します。公式：NKTg = f(x, v, m)

- **核となる積の量:**
  - 運動量 (p): p = m * v

    - NKTg_1 量: 位置と運動量の積 (NKTg_1 = x * p)
    - NKTg_2 量: 質量変化率と運動量の積 (NKTg_2 = (dm/dt) * p)
- **計量単位:**

  NKTm (変位慣性単位)

III. アルゴリズムの詳細説明

1. データ処理プロセス

アルゴリズムは、以下の論理ステップに従って運動傾向分析を行います。

- **ステップ 1:**

  入力パラメータの受信：位置 (x)、速度 (v)、質量 (m)、および質量変化率 (dm/dt)

- **ステップ 2:**

  線運動量 p = m * v の計算

- **ステップ 3:**

  変位慣性値 NKTg_1 および NKTg_2 の計算

- **ステップ 4:**

  値の正負に基づく傾向 (Tendency) の分類：

    - NKTg_1 > 0: 安定状態から離反
    - NKTg_1 < 0: 安定状態への接近
    - NKTg_2 > 0: 質量変化が運動を促進
    - NKTg_2 < 0: 質量変化が運動を阻害

IV. 実行ソースコード (SOURCE CODE)

1. 高水準言語ソースコード (C++)

C++

```
#include <cstdio>

/**
```

```
 * NKTg法則に基づく変位慣性計算ライブラリ
 * 計算ロジックおよび傾向の定義に関するすべての著作権を保持します。
 */
int main() {
   // 1. パラメータの宣言：x (位置), v (速度), m (質量), dm_dt (質量 m の変化率)
   double x = 2.0, v = 3.0, m = 5.0, dm_dt = -0.5;

   // 2. NKTg法則に従った計算の実行
   double p = m * v;         // p = m * v
   double n1 = x * p;        // NKTg1 = x * p
   double n2 = dm_dt * p;    // NKTg2 = (dm/dt) * p

   // 3. レコード形式での結果表示
   printf("{ p = %.1f\n  nktg1 = %.1f\n  nktg2 = %.1f\n", p, n1, n2);

   // 運動傾向の分類 (Tendency Classification)
   printf("  tendency1 = \"%s\"\n",
           n1 > 0 ? "安定状態から離反" :
           n1 < 0 ? "安定状態への接近" : "安定的平衡");

   printf("  tendency2 = \"%s\" }",
           n2 > 0 ? "質量変化が運動を促進" :
           n2 < 0 ? "質量変化が運動を阻害" : "質量変化の影響なし");

   return 0;
}
```

2. 低水準言語ソースコード (Assembly x64)

```
; ------------------------------------------------------------------------
; NKTg 運動傾向分析アルゴリズム (x64 NASM)
; 著者：Nguyễn Khánh Tùng
; © 2026 Nguyễn Khánh Tùng. All rights reserved.
; ------------------------------------------------------------------------
; サンプルデータ（オリジナルドキュメントより）：
;   x=2.0, v=3.0, m=5.0, dm/dt=-0.5
;   p=15.0 | NKTg1=+30.0 → 離反 | NKTg2=-7.5 → 阻害
;
; analyze_nktg_tendency 関数 — System V AMD64 ABI (Linux):
;   入力：xmm0=x, xmm1=v, xmm2=m, xmm3=dm/dt
;   出力：rax=0 (正常), rax=-1 (NaN エラー)
; ------------------------------------------------------------------------

section .data
    fmt_all  db "{ p = %.1f", 10,
             db "  nktg1 = %.1f", 10,
             db "  nktg2 = %.1f", 10,
             db "  tendency1 = ""%s""", 10,
             db "  tendency2 = ""%s"" }", 10, 0

    str_away    db "安定状態から離反", 0
```

```
    str_toward  db "安定状態への接近", 0
    str_stable  db "安定的平衡", 0
    str_support db "質量変化が運動を促進", 0
    str_resist  db "質量変化が運動を阻害", 0
    str_noeff   db "質量変化の影響なし", 0

    val_x     dq 2.0
    val_v     dq 3.0
    val_m     dq 5.0
    val_dm_dt dq -0.5

section .text
    extern printf
    global analyze_nktg_tendency
    global main

;
================================================================================
; analyze_nktg_tendency
;
================================================================================
analyze_nktg_tendency:
    push rbp
    mov rbp, rsp
    and rsp, -16

    ; --- ステップ 1: レジスタ経由での入力パラメータの受信 ---
    ; xmm0 = x      (位置)
    ; xmm1 = v      (速度)
    ; xmm2 = m      (質量)
    ; xmm3 = dm/dt  (質量変化率)
    ; xmm0-xmm3 レジスタは保護され、直接上書きされません

    ; --- ステップ 2: 運動量 p = m * v の計算 ---
    movsd xmm4, xmm2            ; xmm4 = m
    mulsd xmm4, xmm1            ; xmm4 = p = 5.0 * 3.0 = 15.0

    ; --- ステップ 3: NKTg1 および NKTg2 の計算 ---
    movsd xmm5, xmm0            ; xmm5 = x
    mulsd xmm5, xmm4            ; xmm5 = NKTg1 = x * p = 2.0 * 15.0 = 30.0

    movsd xmm6, xmm3            ; xmm6 = dm/dt
    mulsd xmm6, xmm4            ; xmm6 = NKTg2 = (dm/dt) * p = (-0.5) * 15.0 =
-7.5

    ; --- ステップ 4: 正負の符号に基づく傾向分類 ---
    xorps xmm7, xmm7            ; xmm7 = 0.0

    ; 4.1. NKTg1 の分類 → r8 に一時保存
    ucomisd xmm5, xmm7          ; NKTg1 と 0.0 を比較
    jp .error_nan              ; PF=1 → 非数 (NaN)
    lea r8, [rel str_stable]    ; デフォルト：NKTg1 = 0 → 安定的平衡
    ja .t1_away                ; NKTg1 > 0
    jb .t1_toward              ; NKTg1 < 0
```

```
    jmp .check_nktg2
.t1_away:
    lea r8, [rel str_away]    ; "安定状態から離反"
    jmp .check_nktg2
.t1_toward:
    lea r8, [rel str_toward]  ; "安定状態への接近"

    ; 4.2. NKTg2 の分類 → r9 に一時保存
.check_nktg2:
    ucomisd xmm6, xmm7         ; NKTg2 と 0.0 を比較
    jp .error_nan             ; NaN 保護
    lea r9, [rel str_noeff]   ; デフォルト：NKTg2 = 0 → 質量変化の影響なし
    ja .t2_support            ; NKTg2 > 0
    jb .t2_resist             ; NKTg2 < 0
    jmp .do_printf
.t2_support:
    lea r9, [rel str_support] ; "質量変化が運動を促進"
    jmp .do_printf
.t2_resist:
    lea r9, [rel str_resist]  ; "質量変化が運動を阻害"

    ; --- printf を一度だけ呼び出し ---
.do_printf:
    lea rdi, [rel fmt_all]    ; rdi = フォーマット文字列
    mov rsi, r8               ; rsi = tendency1 ポインタ (r8より)
    mov rdx, r9               ; rdx = tendency2 ポインタ (r9より)
    movsd xmm0, xmm4           ; xmm0 = p     = 15.0
    movsd xmm1, xmm5           ; xmm1 = NKTg1 = 30.0
    movsd xmm2, xmm6           ; xmm2 = NKTg2 = -7.5
    mov eax, 3                ; 浮動小数点引数の数 3
    call printf

    xor eax, eax              ; return 0
    leave
    ret

.error_nan:
    mov rax, -1
    leave
    ret

; ========================================================================
; main — サンプルデータをレジスタにロードし、analyze_nktg_tendency を呼び出す
; ========================================================================
main:
    push rbp
    mov rbp, rsp
    and rsp, -16

    ; --- ステップ 1: 入力パラメータのロード（オリジナルドキュメントのデータ） ---
    movsd xmm0, [rel val_x]     ; xmm0 = x     = 2.0
    movsd xmm1, [rel val_v]     ; xmm1 = v     = 3.0
    movsd xmm2, [rel val_m]     ; xmm2 = m     = 5.0
    movsd xmm3, [rel val_dm_dt] ; xmm3 = dm/dt = -0.5

    call analyze_nktg_tendency
```

```
    xor eax, eax
    leave
    ret

; ------------------------------------------------------------------------
; 期待される結果：
; { p = 15.0
;   nktg1 = 30.0
;   nktg2 = -7.5
;   tendency1 = "安定状態から離反"
;   tendency2 = "質量変化が運動を阻害" }
; ------------------------------------------------------------------------
```

V. 安定状態の定義

本ソフトウェアにおける「安定状態」とは、位置 (x)、速度 (v)、および質量 (m)
が相互に作用して運動構造を維持し、制御不能を回避し、物体固有の運動パターンを保持している状態として定義されます。

VI. 量子拡張：量子コンピューティングプラットフォーム上の NKTg アルゴリズム

1. 量子理論的基礎

量子拡張においても、NKTg法則のすべての基本的物理量は保持されます。

- **運動量 (p):** p = m * v
- **NKTg_1 量:** 位置と運動量の積 (NKTg_1 = x * p)
- **NKTg_2 量:** 質量変化率と運動量の積 (NKTg_2 = (dm/dt) * p)
- **計量単位:** NKTm (変位慣性単位)

量子モデルでは、古典モデルのように各組み合わせ (x, v, m, dm/dt)
を逐次計算するのではなく、パラメータ空間全体を重ね合わせ状態にエンコードし、量子並列性を利用して探索を行います。

2. 量子問題の定式化

- **入力:** パラメータ空間 (x, v, m, dm/dt)。各変数は N = 2^n 個の値レベルに離散化され、n 個の量子ビットでエンコードされます。総探索空間は M = N^4 = 2^{4n}
  個の組み合わせを含みます。
- **問題:** 以下の集合から選択された傾向条件 f について：

条件 f	傾向
NKTg_1 > 0	安定状態から離反
NKTg_1 < 0	安定状態への接近
NKTg_2 > 0	質量変化が運動を促進
NKTg_2 < 0	質量変化が運動を阻害
NKTg_1 > 0 かつ NKTg_2 < 0	両条件の組み合わせ

条件 f を満たすすべての組み合わせ (x, v, m, dm/dt) ∈{1..N}^4 を見つけます。

- **出力:** 量子測定後に得られる、条件 f を満たすパラメータ組み合わせの集合。

3. 量子状態空間

パラメータ空間全体は、4n 個の量子ビットを持つ量子レジスタにエンコードされます。

$|q\rangle = |x\rangle \otimes |v\rangle \otimes |m\rangle \otimes |dm/dt\rangle$

初期状態はアダマールゲートによる一様重ね合わせ状態です。

$|s\rangle = H^{\otimes 4n} |0\rangle^{4n} = (1/\sqrt{M}) \sum |x, v, m, dm/dt\rangle$

すべての組み合わせは、初期確率振幅 $1/\sqrt{M}$ を持って同時に存在します。

4. 量子処理アルゴリズム

アルゴリズムは、以下の論理ステップを通じて NKTg 傾向分析を実行します。

- **ステップ 1:** 入力パラメータ (x, v, m, dm/dt) を受信し、4n 量子ビットレジスタにエンコードします。アダマールゲートを適用し、全 M = N^4 個の組み合わせに対して一様重ね合わせを作成します。

- **ステップ 2:** 選択された傾向条件 f に従って、パラメータ化されたオラクル $\hat{O}$ を構築します。オラクルは以下の3つの可逆操作を実行します。
    - **Compute (計算):** 補助 (ancilla) レジスタ上で p = m * v, NKTg_1 = x * p, NKTg_2 = (dm/dt) * p を計算。
    - **Phase kickback (位相キックバック):** 条件 f を満たす状態の位相を反転。
    - **Uncompute (逆計算):** 補助レジスタを |0⟩ に復元。

      f(x,v,m,dm/dt) = 1 の場合 $\hat{O}$ |x,v,m,dm/dt⟩ = -|x,v,m,dm/dt⟩

      f(x,v,m,dm/dt) = 0 の場合 $\hat{O}$ |x,v,m,dm/dt⟩ = +|x,v,m,dm/dt⟩

- **ステップ 3:** 拡散演算子 (Diffusion operator) を適用し、解となる状態の振幅を増幅し、条件を満たさない状態の振幅を抑制します。

  $\hat{D} = 2|s\rangle\langle s| - I$

- **ステップ 4:** ステップ 2 とステップ 3 を T 回の最適サイクル繰り返します。空間 M 内の解の数を k とすると：

  T = ⌊(π/4) * √{M/k}⌋ サイクル。

  T サイクル後、正しい解を測定する確率は1に近づきます。

- **ステップ 5:** 4n 量子ビットレジスタの測定を行い、高い確率で条件 f を満たす組み合わせ (x, v, m, dm/dt) を得ます。O(k) 回繰り返すことで、k 個の解の集合をすべて収集します。

5. 古典モデルと量子モデルの比較

評価基準	**古典モデル**	**量子モデル**
探索空間	M = N^4 組み合わせ	M = N^4 = 2^{4n} 状態
処理方法	組み合わせごとの逐次処理	量子並列処理
計算量 (複雑性)	O(M) = O(N^4)	O(√M) = O(N^2)

評価基準	古典モデル	量子モデル
加速性能	—	古典に対し2次（平方根）加速
リソース	無制限のメモリ	4n + ancilla 量子ビット
探索条件	1実行につき1条件 f	1実行につき1条件 f

6. 科学的意義

NKTg量子モデルは、NKTg法則の物理的本質（NKTg_1, NKTg_2
量および傾向分類基準）を完全に保持しつつ、グローバーのアルゴリズムを利用して探索の複雑さを O(M) = O(N^4) から O(√M) = O(N^2)
へと劇的に低減します。オラクルはパラメータ化された構造として設計されており、NKTg集合内のいかなる傾向条件にも柔軟に適用可能です。これは古典モデルから量子コンピューティングへの自然な拡張ステップであり、複雑な物理システムのシミュレーションや大規模なパラメータ最適化への応用の道を切り拓くものです。

```
//
================================================================================
// NKTg アルゴリズム — 量子版 (OpenQASM 3.0) — 完全修正版
// 著者：Nguyễn Khánh Tùng
// © 2026 Nguyễn Khánh Tùng. All rights reserved.
// ------------------------------------------------------------------------
---
// 監査に基づく修正点：
//   [1] dm/dt=0 検出に ccx+cx の代わりに mcx を使用
//   [2] MODE 5 条件に ccx+cx の代わりに mcx を使用
//   [3] Uncompute MCZ における不要な ccx の繰り返しを削除
//   [4] オラクルを標準的な OpenQASM 3.0 サブルーチンとしてカプセル化
//
================================================================================

OPENQASM 3.0;
include "stdgates.inc";

//
================================================================================
// レジスタ宣言
//
================================================================================
```

```
qubit[3] x_reg;
qubit[3] v_reg;
qubit[3] m_reg;
qubit[3] d_reg;
qubit[5] anc;
qubit    flag;

bit[3] c_x;
bit[3] c_v;
bit[3] c_m;
bit[3] c_d;

//
================================================================================
// サブルーチン : NKTg オラクル (MODE 5 : NKTg1>0 かつ NKTg2<0 かつ dm/dt≠0)
// 構造 : Compute (計算) → Phase kickback (位相反転) → Uncompute (逆計算)
[絶対対称]
//
================================================================================
def nktg_oracle(qubit[3] xr, qubit[3] vr, qubit[3] mr,
                qubit[3] dr, qubit[5] a, qubit f) {

    // ------------------------------------------------------------------------
---
    // COMPUTE (計算)
    // ------------------------------------------------------------------------
---

    // ステップ 2a : sign(p) = sign(m) XOR sign(v)  →  a[0]
    cx mr[2], a[0];
    cx vr[2], a[0];         // a[0] = sign(m) XOR sign(v) = sign(p)

    // ステップ 2b : sign(NKTg1) = sign(x) XOR sign(p)  →  a[1]
    //          NKTg1 > 0  ⇔  a[1] = 0
    cx xr[2], a[1];
    cx a[0],  a[1];         // a[1] = sign(NKTg1)

    // ステップ 2c : sign(NKTg2) = sign(d) XOR sign(p)  →  a[2]
    //          NKTg2 < 0  ⇔  a[2] = 1
    cx dr[2], a[2];
    cx a[0],  a[2];         // a[2] = sign(NKTg2)

    // ステップ 2d : dm/dt = 0 の検出  →  a[3]
    //          dm/dt = 0  ⇔  dr[2]=0 かつ dr[1]=0 かつ dr[0]=0
    //          X ゲート後 : dr[2]=1 かつ dr[1]=1 かつ dr[0]=1
    //          修正 [1] : 誤った ccx+cx の代わりに 3制御 mcx を使用
    x dr[0]; x dr[1];  x dr[2];
    mcx dr[0], dr[1], dr[2], a[3]; // dm/dt=0 のとき a[3]=1
    x dr[0]; x dr[1];  x dr[2];    // X を解除

    // ステップ 2e : MODE 5 の完全な条件  →  a[4]
    //          f = NKTg1>0 かつ NKTg2<0 かつ dm/dt≠0
    //          NKTg1>0  ⇔  a[1]=0 → X(a[1])
    //          NKTg2<0  ⇔  a[2]=1
    //          dm/dt≠0  ⇔  a[3]=0 → X(a[3])
```

```
    //           修正 [2] : 誤った ccx+cx の代わりに 3制御 mcx を使用
    x a[1]; x a[3];
    mcx a[1], a[2], a[3], a[4];     // f=1 のとき a[4]=1

    // ------------------------------------------------------------------------
---
    // PHASE KICKBACK (位相反転)
    // flag=|−⟩ : CX(a[4], flag) → 条件 f を満たすすべての状態の位相を反転
    // ------------------------------------------------------------------------
---
    cx a[4], f;

    // ------------------------------------------------------------------------
---
    // UNCOMPUTE (逆計算) — 計算プロセスと絶対に対称（逆順実行）
    // ------------------------------------------------------------------------
---
    mcx a[1], a[2], a[3], a[4];     // ステップ 2e の解除
    x a[1]; x a[3];

    x dr[0]; x dr[1];  x dr[2];
    mcx dr[0], dr[1], dr[2], a[3]; // ステップ 2d の解除
    x dr[0]; x dr[1];  x dr[2];

    cx a[0],  a[2];                  // ステップ 2c の解除
    cx dr[2], a[2];

    cx a[0],  a[1];                  // ステップ 2b の解除
    cx xr[2], a[1];

    cx vr[2], a[0];                  // ステップ 2a の解除
    cx mr[2], a[0];
    // a[0..4] = |00000⟩ — 完全に復元
}

//
// ============================================================================
// サブルーチン : 拡散演算子 D^ = 2|s⟩⟨s| − I
// H → X → MCZ(12 qubit) → X → H
// 修正 [3] : Uncompute MCZ における不要な ccx の繰り返しを削除
//
// ============================================================================
def grover_diffusion(qubit[3] xr, qubit[3] vr,
                     qubit[3] mr, qubit[3] dr, qubit[5] a) {

    // 3a : 全 12 量子ビットに H を適用
    h xr[0]; h xr[1]; h xr[2];
    h vr[0]; h vr[1]; h vr[2];
    h mr[0]; h mr[1]; h mr[2];
    h dr[0]; h dr[1]; h dr[2];

    // 3b : 全てに X を適用 → |0...0⟩ → |1...1⟩ へ写像
    x xr[0]; x xr[1]; x xr[2];
    x vr[0]; x vr[1]; x vr[2];
    x mr[0]; x mr[1]; x mr[2];
    x dr[0]; x dr[1]; x dr[2];
```

```
    // 3c : MCZ(12 qubit) = H(標的) → MCX(12 qubit) → H(標的)
    // MCX(12) をトフォリ・チェーンに分解 — 空の a[0..4] を再利用
    h dr[2];

    // レベル 1 : ペアごとにグループ化
    ccx xr[0], xr[1], a[0];  // a[0] = xr[0] AND xr[1]
    ccx xr[2], vr[0], a[1];  // a[1] = xr[2] AND vr[0]
    ccx vr[1], vr[2], a[2];  // a[2] = vr[1] AND vr[2]
    ccx mr[0], mr[1], a[3];  // a[3] = mr[0] AND mr[1]
    ccx mr[2], dr[0], a[4];  // a[4] = mr[2] AND dr[0]

    // レベル 2 : レベル 1 の結果を結合
    ccx a[0], a[1], a[0];       // a[0] = 最初の 4 ビットの AND
    // 注意：中間ビットが必要 — uncompute 後に再利用
    // 量子ビットの重複なしの完全分解：
    ccx a[0], a[2], a[1];      // a[1] = a[0] AND a[2]
    ccx a[3], a[4], a[2];      // a[2] = a[3] AND a[4]
    ccx a[1], a[2], a[3];      // a[3] = 10 ビットすべての AND
    ccx dr[1], a[3], a[4];     // a[4] = 11 ビットすべての AND
    cx  a[4], dr[2];           // MCX : 12 ビットすべてが 1 のときに dr[2] を反転

    // レベル 2 の逆計算（逆順、繰り返しなし — 修正 [3]）
    ccx dr[1], a[3], a[4];
    ccx a[1], a[2], a[3];
    ccx a[3], a[4], a[2];
    ccx a[0], a[2], a[1];
    ccx a[0], a[1], a[0];

    // レベル 1 の逆計算
    ccx mr[2], dr[0], a[4];
    ccx mr[0], mr[1], a[3];
    ccx vr[1], vr[2], a[2];
    ccx xr[2], vr[0], a[1];
    ccx xr[0], xr[1], a[0];

    h dr[2];                    // H を解除 → MCZ の完了

    // 3d : X を解除
    x xr[0]; x xr[1]; x xr[2];
    x vr[0]; x vr[1]; x vr[2];
    x mr[0]; x mr[1]; x mr[2];
    x dr[0]; x dr[1]; x dr[2];

    // 3e : 2度目の H — D^ の終了
    h xr[0]; h xr[1]; h xr[2];
    h vr[0]; h vr[1]; h vr[2];
    h mr[0]; h mr[1]; h mr[2];
    h dr[0]; h dr[1]; h dr[2];
}

//
================================================================================
// メインプログラム
```

```
//
================================================================================

// 位相反転のため flag を |−⟩ に初期化
x flag;
h flag;

// ステップ 1 : 一様重ね合わせ
h x_reg[0]; h x_reg[1]; h x_reg[2];
h v_reg[0]; h v_reg[1]; h v_reg[2];
h m_reg[0]; h m_reg[1]; h m_reg[2];
h d_reg[0]; h d_reg[1]; h d_reg[2];

// グローバーのループを T サイクル実行
// T = floor((π/4) × sqrt(M/k))
// M=4096, k=1 → T≈50 | k=64 → T=6 | k=1024 → T=1
for uint i in [1:1] {
    nktg_oracle(x_reg, v_reg, m_reg, d_reg, anc, flag);
    grover_diffusion(x_reg, v_reg, m_reg, d_reg, anc);
}

// ステップ 5 : 測定
c_x = measure x_reg;   // 010 → x=2.0 正 ✓
c_v = measure v_reg;   // 011 → v=3.0 正 ✓
c_m = measure m_reg;   // 001 → m=5.0 正 ✓
c_d = measure d_reg;   // 101 → dm/dt=-0.5 負 ✓
                        // 000 → photon : オラクルでマークされなかった場合 ✓
```

7. Spanish (Español)

ANUNCIO DE ESPECIFICACIONES DE SOFTWARE Y ALGORITMOS DE LA LEY NKTg DE INERCIA VARIABLE

I. INFORMACIÓN BÁSICA

- **Nombre del Software:**

  Sistema de Cálculo de Dinámica de Inercia Variable NKTg (NKTg Dynamics Calculator).

- **Autor:**

  Nguyễn Khánh Tùng

- **Lenguajes de Programación:**

  C++ (ISO/IEC 14882) y Assembly (x64).

II. BASE TEÓRICA: LEY NKTg DE INERCIA VARIABLE

El software se ejecuta basado en los principios de la Ley NKTg:

- **Relación Fundamental:**

  La tendencia del movimiento depende de la posición (x), la velocidad (v) y la masa (m). Fórmula: NKTg = f(x, v, m).

- **Magnitudes de Producto Principales:**
  - Momento lineal (p): p = m * v
  - Magnitud NKTg_1: Producto de la posición por el momento (NKTg_1 = x * p).
  - Magnitud NKTg_2: Producto de la tasa de cambio de la masa por el momento (NKTg_2 = (dm/dt) * p).
- **Unidad de Medida:**

  NKTm (Unidad de Inercia Variable).

III. DESCRIPCIÓN DETALLADA DEL ALGORITMO

1. Proceso de Procesamiento de Datos

El algoritmo realiza el análisis de la tendencia del movimiento a través de los siguientes pasos lógicos:

- **Paso 1:**

  Recepción de parámetros de entrada: posición (x), velocidad (v), masa (m) y tasa de cambio de masa (dm/dt).

- **Paso 2:**

  Cálculo del momento lineal p = m * v.

- **Paso 3:**

  Cálculo de los valores de inercia variable NKTg_1 y NKTg_2.

- **Paso 4:**

  Clasificación de la tendencia (Tendency) basada en el signo de los valores:

  - NKTg_1 > 0: Alejamiento del estado estable.
  - NKTg_1 < 0: Acercamiento al estado estable.
  - NKTg_2 > 0: El cambio de masa favorece el movimiento.
  - NKTg_2 < 0: El cambio de masa dificulta el movimiento.

IV. CÓDIGO FUENTE DE EJECUCIÓN (SOURCE CODE)

1. Código Fuente en Lenguaje de Alto Nivel (C++)

C++

```
#include <cstdio>
```

```
/**
 * Biblioteca de cálculo de inercia variable según la Ley NKTg
 * Se reservan todos los derechos de autor con respecto a la lógica de
cálculo y definiciones de tendencia.
 */
int main() {
   // 1. Declaración de parámetros: x (posición), v (velocidad), m (masa),
dm_dt (tasa de cambio de m)
   double x = 2.0, v = 3.0, m = 5.0, dm_dt = -0.5;

   // 2. Ejecución del cálculo según la Ley NKTg
   double p = m * v;        // p = m * v
   double n1 = x * p;       // NKTg1 = x * p
   double n2 = dm_dt * p;   // NKTg2 = (dm/dt) * p

   // 3. Visualización de resultados en formato Record
   printf("{ p = %.1f\n  nktg1 = %.1f\n  nktg2 = %.1f\n", p, n1, n2);

   // Clasificación de la tendencia del movimiento (Tendency Classification)
   printf("  tendency1 = \"%s\"\n",
           n1 > 0 ? "Alejamiento del estado estable" :
           n1 < 0 ? "Acercamiento al estado estable" : "Equilibrio estable");

   printf("  tendency2 = \"%s\" }",
           n2 > 0 ? "El cambio de masa favorece el movimiento" :
           n2 < 0 ? "El cambio de masa dificulta el movimiento" : "Sin efecto
por cambio de masa");

   return 0;
}
```

2. Código Fuente en Lenguaje de Bajo Nivel (Assembly x64)

```
; ------------------------------------------------------------------------
; ALGORITMO DE ANÁLISIS DE TENDENCIA DE MOVIMIENTO NKTg (x64 NASM)
; Autor: Nguyễn Khánh Tùng
; © 2026 Nguyễn Khánh Tùng. Todos los derechos reservados.
; ------------------------------------------------------------------------
; Datos de ejemplo (del documento original):
;   x=2.0, v=3.0, m=5.0, dm/dt=-0.5
;   p=15.0 | NKTg1=+30.0 → Alejamiento | NKTg2=-7.5 → Dificultad
;
; Función analyze_nktg_tendency — System V AMD64 ABI (Linux):
;   Entrada: xmm0=x, xmm1=v, xmm2=m, xmm3=dm/dt
;   Salida: rax=0 (OK), rax=-1 (Error NaN)
; ------------------------------------------------------------------------

section .data
    fmt_all  db "{ p = %.1f", 10,
             db "  nktg1 = %.1f", 10,
             db "  nktg2 = %.1f", 10,
             db "  tendency1 = ""%s""", 10,
             db "  tendency2 = ""%s"" }", 10, 0

    str_away    db "Alejamiento del estado estable", 0
    str_toward  db "Acercamiento al estado estable", 0
    str_stable  db "Equilibrio estable", 0
    str_support db "El cambio de masa favorece el movimiento", 0
    str_resist  db "El cambio de masa dificulta el movimiento", 0
    str_noeff   db "Sin efecto por cambio de masa", 0
```

```
    val_x     dq 2.0
    val_v     dq 3.0
    val_m     dq 5.0
    val_dm_dt dq -0.5

section .text
    extern printf
    global analyze_nktg_tendency
    global main

;
================================================================================
; analyze_nktg_tendency
;
================================================================================
analyze_nktg_tendency:
    push rbp
    mov rbp, rsp
    and rsp, -16

    ; --- Paso 1: Recepción de parámetros de entrada vía registros ---
    ; xmm0 = x      (posición)
    ; xmm1 = v      (velocidad)
    ; xmm2 = m      (masa)
    ; xmm3 = dm/dt  (tasa de cambio de masa)
    ; Los registros xmm0–xmm3 están protegidos, no se sobrescriben
directamente

    ; --- Paso 2: Cálculo del momento p = m * v ---
    movsd xmm4, xmm2           ; xmm4 = m
    mulsd xmm4, xmm1           ; xmm4 = p = 5.0 * 3.0 = 15.0

    ; --- Paso 3: Cálculo de NKTg1 y NKTg2 ---
    movsd xmm5, xmm0           ; xmm5 = x
    mulsd xmm5, xmm4           ; xmm5 = NKTg1 = x * p = 2.0 * 15.0 = 30.0

    movsd xmm6, xmm3           ; xmm6 = dm/dt
    mulsd xmm6, xmm4           ; xmm6 = NKTg2 = (dm/dt) * p = (-0.5) * 15.0 =
-7.5

    ; --- Paso 4: Clasificación de tendencia basada en el signo ---
    xorps xmm7, xmm7           ; xmm7 = 0.0

    ; 4.1. Clasificación de NKTg1 → Almacenamiento temporal en r8
    ucomisd xmm5, xmm7         ; Compara NKTg1 con 0.0
    jp .error_nan             ; PF=1 → NaN (Not-a-Number)
    lea r8, [rel str_stable]   ; Por defecto: NKTg1 = 0 → Equilibrio estable
    ja .t1_away               ; NKTg1 > 0
    jb .t1_toward             ; NKTg1 < 0
    jmp .check_nktg2
.t1_away:
    lea r8, [rel str_away]    ; "Alejamiento del estado estable"
    jmp .check_nktg2
.t1_toward:
    lea r8, [rel str_toward]  ; "Acercamiento al estado estable"

    ; 4.2. Clasificación de NKTg2 → Almacenamiento temporal en r9
.check_nktg2:
    ucomisd xmm6, xmm7         ; Compara NKTg2 con 0.0
    jp .error_nan             ; Protección NaN
    lea r9, [rel str_noeff]   ; Por defecto: NKTg2 = 0 → Sin efecto por
cambio de masa
    ja .t2_support            ; NKTg2 > 0
```

```
    jb .t2_resist             ; NKTg2 < 0
    jmp .do_printf
.t2_support:
    lea r9, [rel str_support] ; "El cambio de masa favorece el movimiento"
    jmp .do_printf
.t2_resist:
    lea r9, [rel str_resist]  ; "El cambio de masa dificulta el movimiento"

    ; --- Llamada única a printf ---
.do_printf:
    lea rdi, [rel fmt_all]    ; rdi = cadena de formato
    mov rsi, r8               ; rsi = puntero tendency1 (desde r8)
    mov rdx, r9               ; rdx = puntero tendency2 (desde r9)
    movsd xmm0, xmm4           ; xmm0 = p     = 15.0
    movsd xmm1, xmm5           ; xmm1 = NKTg1 = 30.0
    movsd xmm2, xmm6           ; xmm2 = NKTg2 = -7.5
    mov eax, 3                ; 3 parámetros de punto flotante
    call printf

    xor eax, eax              ; return 0
    leave
    ret

.error_nan:
    mov rax, -1
    leave
    ret

; ========================================================================
; main — Carga datos de ejemplo en registros, llama a analyze_nktg_tendency
; ========================================================================
main:
    push rbp
    mov rbp, rsp
    and rsp, -16

    ; --- Paso 1: Carga de parámetros de entrada (datos del documento
original) ---
    movsd xmm0, [rel val_x]     ; xmm0 = x     = 2.0
    movsd xmm1, [rel val_v]     ; xmm1 = v     = 3.0
    movsd xmm2, [rel val_m]     ; xmm2 = m     = 5.0
    movsd xmm3, [rel val_dm_dt] ; xmm3 = dm/dt = -0.5

    call analyze_nktg_tendency

    xor eax, eax
    leave
    ret

; ------------------------------------------------------------------------
; Resultado esperado:
; { p = 15.0
;   nktg1 = 30.0
;   nktg2 = -7.5
;   tendency1 = "Alejamiento del estado estable"
;   tendency2 = "El cambio de masa dificulta el movimiento" }
; ------------------------------------------------------------------------
```

V. DEFINICIÓN DEL ESTADO ESTABLE

El estado estable en este software se define como el estado en el que la posición (x), la velocidad (v) y la masa (m) interactúan para mantener la estructura del movimiento, evitando la pérdida de control y preservando el patrón de movimiento inherente al objeto.

VI. EXTENSIÓN CUÁNTICA: ALGORITMO NKTg EN PLATAFORMAS DE COMPUTACIÓN CUÁNTICA

1. Base Teórica Cuántica

La extensión cuántica preserva la totalidad de las magnitudes fundamentales de la Ley NKTg:

- **Momento (p):** p = m * v
- **Magnitud NKTg_1:** Producto de la posición por el momento (NKTg_1 = x * p)
- **Magnitud NKTg_2:** Producto de la tasa de cambio de la masa por el momento (NKTg_2 = (dm/dt) * p)
- **Unidad de Medida:** NKTm (Unidad de Inercia Variable)

En lugar de calcular secuencialmente cada combinación (x, v, m, dm/dt) como en el modelo clásico, el modelo cuántico codifica todo el espacio de parámetros en un estado de superposición y utiliza el paralelismo cuántico para la búsqueda.

2. Planteamiento del Problema Cuántico

- **Entrada:** Espacio de parámetros (x, v, m, dm/dt), donde cada variable se discretiza en N = 2^n niveles de valor y se codifica con n qubits. El espacio de búsqueda total contiene M = N^4 = 2^{4n} combinaciones.
- **Problema:** Para una condición de tendencia f elegida del conjunto:

Condición f	**Tendencia**
NKTg_1 > 0	Alejamiento del estado estable
NKTg_1 < 0	Acercamiento al estado estable
NKTg_2 > 0	El cambio de masa favorece el movimiento
NKTg_2 < 0	El cambio de masa dificulta el movimiento
NKTg_1 > 0 y NKTg_2 < 0	Combinación de ambas condiciones

Encontrar todas las combinaciones (x, v, m, dm/dt) ∈{1..N}^4 que satisfagan la condición f elegida.

- **Salida:** El conjunto de combinaciones de parámetros que satisfacen f, obtenido tras una medición cuántica.

3. Espacio de Estados Cuánticos

Todo el espacio de parámetros se codifica en un registro cuántico de 4n qubits:

$|q\rangle = |x\rangle \otimes |v\rangle \otimes |m\rangle \otimes |dm/dt\rangle$

El estado inicial es una superposición uniforme mediante puertas Hadamard:

$|s\rangle = H^{\otimes 4n} |0\rangle^{4n} = (1/\sqrt{M}) \sum |x, v, m, dm/dt\rangle$

Cada combinación existe simultáneamente con una amplitud de probabilidad inicial de $1/\sqrt{M}$.

4. Algoritmo de Procesamiento Cuántico

El algoritmo realiza el análisis de tendencia NKTg a través de los siguientes pasos lógicos:

- **Paso 1:** Recibe y codifica los parámetros de entrada (x, v, m, dm/dt) en el registro de 4n qubits. Aplica puertas Hadamard para crear una superposición uniforme sobre las M = N^4 combinaciones.
- **Paso 2:** Construye un Oráculo parametrizado $\hat{O}$ según la condición de tendencia f elegida. El Oráculo realiza tres operaciones reversibles:
    - **Compute (Calcular):** Calcula p = m * v, NKTg_1 = x * p, NKTg_2 = (dm/dt) * p en el registro auxiliar (ancilla).
    - **Phase kickback (Retroceso de fase):** Invierte la fase de los estados que satisfacen f.
    - **Uncompute (Deshacer cálculo):** Restaura el registro auxiliar a $|0\rangle$.

      $\hat{O} |x,v,m,dm/dt\rangle = -|x,v,m,dm/dt\rangle$ si f(x,v,m,dm/dt) = 1

      $\hat{O} |x,v,m,dm/dt\rangle = +|x,v,m,dm/dt\rangle$ si f(x,v,m,dm/dt) = 0

- **Paso 3:** Aplica el operador de Difusión (Diffusion operator) – amplifica la amplitud de los estados solución y suprime la amplitud de los estados que no cumplen la condición:

  $\hat{D} = 2|s\rangle\langle s| - I$

- **Paso 4:** Repite el Paso 2 y el Paso 3 durante T ciclos óptimos. Siendo k el número de soluciones en el espacio M:

  $T = \lfloor (\pi/4) * \sqrt{M/k} \rfloor$ ciclos.

  Tras T ciclos, la probabilidad de medir la solución correcta tiende a 1.

- **Paso 5:** Medición del registro de 4n qubits – se obtiene la combinación (x, v, m, dm/dt) que satisface la condición f con alta probabilidad. Se repite O(k) veces para recolectar todo el conjunto de k soluciones.

5. Comparación entre el Modelo Clásico y el Modelo Cuántico

Criterios	Modelo Clásico	Modelo Cuántico
Espacio de búsqueda	M = N^4 combinaciones	M = N^4 = 2^{4n} estados
Método de procesamiento	Secuencial por combinación	Paralelismo cuántico
Complejidad	O(M) = O(N^4)	O(√M) = O(N^2)
Aceleración	—	Cuadrática respecto al clásico
Recursos	Memoria ilimitada	4n + ancilla qubits
Condición de búsqueda	Una condición f por ejecución	Una condición f por ejecución

6. Significado Científico

El modelo cuántico NKTg preserva la naturaleza física de la Ley NKTg —las magnitudes NKTg_1, NKTg_2 y los criterios de clasificación de tendencias— mientras utiliza el algoritmo de Grover para reducir la complejidad de la búsqueda de O(M) = O(N^4) a O(√M) = O(N^2). El Oráculo está diseñado de forma parametrizada, permitiendo una aplicación flexible a cualquier condición de tendencia dentro del conjunto NKTg. Este es un paso de extensión natural del modelo clásico hacia la computación cuántica, abriendo perspectivas para la simulación de sistemas físicos complejos y la optimización de parámetros a gran escala.

```
//
================================================================================
// ALGORITMO NKTg — CUÁNTICO (OpenQASM 3.0) — Versión Corregida Completa
// Autor: Nguyễn Khánh Tùng
// © 2026 Nguyễn Khánh Tùng. Todos los derechos reservados.
// ------------------------------------------------------------------------------
---
// Correcciones tras la auditoría:
//   [1] Uso de mcx en lugar de ccx+cx para la detección de dm/dt=0
//   [2] Uso de mcx en lugar de ccx+cx para la condición MODE 5
//   [3] Uncompute MCZ no repite innecesariamente los ciclos ccx
//   [4] Oráculo encapsulado en una subrutina estándar de OpenQASM 3.0
//
================================================================================

OPENQASM 3.0;
include "stdgates.inc";

//
================================================================================
// DECLARACIÓN DE REGISTROS
//
================================================================================
qubit[3] x_reg;
```

```
qubit[3] v_reg;
qubit[3] m_reg;
qubit[3] d_reg;
qubit[5] anc;
qubit    flag;

bit[3] c_x;
bit[3] c_v;
bit[3] c_m;
bit[3] c_d;

//
================================================================================
// SUBRUTINA: ORÁCULO NKTg (MODO 5: NKTg1>0 Y NKTg2<0 Y dm/dt≠0)
// Estructura: Compute → Phase kickback → Uncompute (Simetría absoluta)
//
================================================================================
def nktg_oracle(qubit[3] xr, qubit[3] vr, qubit[3] mr,
                qubit[3] dr, qubit[5] a, qubit f) {

    // ------------------------------------------------------------------------
---
    // COMPUTE (CÁLCULO)
    // ------------------------------------------------------------------------
---

    // Paso 2a: sign(p) = sign(m) XOR sign(v)  →  a[0]
    cx mr[2], a[0];
    cx vr[2], a[0];         // a[0] = sign(m) XOR sign(v) = sign(p)

    // Paso 2b: sign(NKTg1) = sign(x) XOR sign(p)  →  a[1]
    //          NKTg1 > 0  ⇔  a[1] = 0
    cx xr[2], a[1];
    cx a[0],  a[1];         // a[1] = sign(NKTg1)

    // Paso 2c: sign(NKTg2) = sign(d) XOR sign(p)  →  a[2]
    //          NKTg2 < 0  ⇔  a[2] = 1
    cx dr[2], a[2];
    cx a[0],  a[2];         // a[2] = sign(NKTg2)

    // Paso 2d: Detección dm/dt = 0  →  a[3]
    //          dm/dt = 0  ⇔  dr[2]=0 Y dr[1]=0 Y dr[0]=0
    //          Tras X: dr[2]=1 Y dr[1]=1 Y dr[0]=1
    //          FIX [1]: usa mcx con 3 controles en lugar de ccx+cx erróneo
    x dr[0]; x dr[1];  x dr[2];
    mcx dr[0], dr[1], dr[2], a[3]; // a[3]=1 cuando dm/dt=0
    x dr[0]; x dr[1];  x dr[2];    // Deshacer X

    // Paso 2e: Condición completa MODO 5  →  a[4]
    //          f = NKTg1>0 Y NKTg2<0 Y dm/dt≠0
    //          NKTg1>0  ⇔  a[1]=0 → X(a[1])
    //          NKTg2<0  ⇔  a[2]=1
    //          dm/dt≠0  ⇔  a[3]=0 → X(a[3])
    //          FIX [2]: mcx con 3 controles en lugar de ccx+cx erróneo
    x a[1]; x a[3];
    mcx a[1], a[2], a[3], a[4];    // a[4]=1 cuando f=1

    // ------------------------------------------------------------------------
---
    // PHASE KICKBACK (RETROCESO DE FASE)
    // flag=|-⟩: CX(a[4], flag) → invierte fase de todos los estados que
cumplen f
```

```
    // ------------------------------------------------------------------------
---
    cx a[4], f;

    // ------------------------------------------------------------------------
---
    // UNCOMPUTE (DESHACER CÁLCULO) — simetría absoluta con Compute (orden
inverso)
    // ------------------------------------------------------------------------
---
    mcx a[1], a[2], a[3], a[4];    // Deshacer paso 2e
    x a[1]; x a[3];

    x dr[0]; x dr[1];  x dr[2];
    mcx dr[0], dr[1], dr[2], a[3]; // Deshacer paso 2d
    x dr[0]; x dr[1];  x dr[2];

    cx a[0],  a[2];                // Deshacer paso 2c
    cx dr[2], a[2];

    cx a[0],  a[1];                // Deshacer paso 2b
    cx xr[2], a[1];

    cx vr[2], a[0];                // Deshacer paso 2a
    cx mr[2], a[0];
    // a[0..4] = |00000⟩ — completamente restaurados
}

//
================================================================================
// SUBRUTINA: DIFUSIÓN D^ = 2|s⟩⟨s| - I
// H → X → MCZ(12 qubit) → X → H
// FIX [3]: Uncompute MCZ no repite innecesariamente los ciclos ccx
//
================================================================================
def grover_diffusion(qubit[3] xr, qubit[3] vr,
                     qubit[3] mr, qubit[3] dr, qubit[5] a) {

    // 3a: H en todos los 12 qubits
    h xr[0]; h xr[1]; h xr[2];
    h vr[0]; h vr[1]; h vr[2];
    h mr[0]; h mr[1]; h mr[2];
    h dr[0]; h dr[1]; h dr[2];

    // 3b: X en todos → mapeo |0...0⟩ → |1...1⟩
    x xr[0]; x xr[1]; x xr[2];
    x vr[0]; x vr[1]; x vr[2];
    x mr[0]; x mr[1]; x mr[2];
    x dr[0]; x dr[1]; x dr[2];

    // 3c: MCZ(12 qubit) = H(objetivo) → MCX(12 qubit) → H(objetivo)
    // Descomponer MCX(12) en cadena Toffoli — reutiliza a[0..4] limpios
    h dr[2];

    // Nivel 1: Agrupación por pares
    ccx xr[0], xr[1], a[0];  // a[0] = xr[0] AND xr[1]
    ccx xr[2], vr[0], a[1];  // a[1] = xr[2] AND vr[0]
    ccx vr[1], vr[2], a[2];  // a[2] = vr[1] AND vr[2]
    ccx mr[0], mr[1], a[3];  // a[3] = mr[0] AND mr[1]
    ccx mr[2], dr[0], a[4];  // a[4] = mr[2] AND dr[0]

    // Nivel 2: Combina resultados del Nivel 1
    ccx a[0], a[1], a[0];      // a[0] = AND de los primeros 4 bits
```

```
    // Nota: qubits intermedios necesarios — reutilización tras uncompute
    // Descomposición completa sin solapamiento de qubits:
    ccx a[0], a[2], a[1];     // a[1] = a[0] AND a[2]
    ccx a[3], a[4], a[2];     // a[2] = a[3] AND a[4]
    ccx a[1], a[2], a[3];     // a[3] = los 10 bits
    ccx dr[1], a[3], a[4];    // a[4] = AND de los 11 bits
    cx  a[4], dr[2];          // MCX: invierte dr[2] si los 12 bits = 1

    // Uncompute Nivel 2 (inverso, sin repeticiones — FIX [3])
    ccx dr[1], a[3], a[4];
    ccx a[1], a[2], a[3];
    ccx a[3], a[4], a[2];
    ccx a[0], a[2], a[1];
    ccx a[0], a[1], a[0];

    // Uncompute Nivel 1
    ccx mr[2], dr[0], a[4];
    ccx mr[0], mr[1], a[3];
    ccx vr[1], vr[2], a[2];
    ccx xr[2], vr[0], a[1];
    ccx xr[0], xr[1], a[0];

    h dr[2];                   // Deshacer H → MCZ completo finalizado

    // 3d: Deshacer X
    x xr[0]; x xr[1]; x xr[2];
    x vr[0]; x vr[1]; x vr[2];
    x mr[0]; x mr[1]; x mr[2];
    x dr[0]; x dr[1]; x dr[2];

    // 3e: Segundo H — finaliza D^
    h xr[0]; h xr[1]; h xr[2];
    h vr[0]; h vr[1]; h vr[2];
    h mr[0]; h mr[1]; h mr[2];
    h dr[0]; h dr[1]; h dr[2];
}

//
// ============================================================================
// PROGRAMA PRINCIPAL
//
// ============================================================================

// Inicialización de flag a |−⟩ para el Phase Kickback
x flag;
h flag;

// Paso 1: Generación de superposición uniforme
h x_reg[0]; h x_reg[1]; h x_reg[2];
h v_reg[0]; h v_reg[1]; h v_reg[2];
h m_reg[0]; h m_reg[1]; h m_reg[2];
h d_reg[0]; h d_reg[1]; h d_reg[2];

// Bucle de Grover de T ciclos
// T = floor((π/4) × sqrt(M/k))
// M=4096, k=1 → T≈50 | k=64 → T=6 | k=1024 → T=1
for uint i in [1:1] {
    nktg_oracle(x_reg, v_reg, m_reg, d_reg, anc, flag);
    grover_diffusion(x_reg, v_reg, m_reg, d_reg, anc);
}

// Paso 5: Medición
c_x = measure x_reg;   // 010 → x=2.0 positivo ✓
```

```
c_v = measure v_reg;   // 011 → v=3.0 positivo ✓
c_m = measure m_reg;   // 001 → m=5.0 positivo ✓
c_d = measure d_reg;   // 101 → dm/dt=-0.5 negativo ✓
                        // 000 → photon: Oráculo no marcado ✓
```

8. Russian (Русский)

ПУБЛИКАЦИЯ СПЕЦИФИКАЦИЙ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ И АЛГОРИТМОВ ЗАКОНА NKTg О ПЕРЕМЕННОЙ ИНЕРЦИИ

I. ОБЩАЯ ИНФОРМАЦИЯ

- **Название ПО:**

  Система расчета динамики переменной инерции NKTg (NKTg Dynamics Calculator).

- **Автор:**

  Нгуен Кхань Тунг

- **Языки программирования:**

  C++ (ISO/IEC 14882) и Assembly (x64).

II. ТЕОРЕТИЧЕСКАЯ БАЗА: ЗАКОН NKTg О ПЕРЕМЕННОЙ ИНЕРЦИИ

Программное обеспечение работает на основе принципов закона NKTg:

- **Фундаментальная зависимость:**

  Тенденция движения зависит от положения (x), скорости (v) и массы (m). Формула: NKTg = f(x, v, m).

- **Основные производные величины:**
  - Импульс (p): p = m * v
  - Величина NKTg_1: Произведение положения на импульс (NKTg_1 = x * p).
  - Величина NKTg_2: Произведение скорости изменения массы на импульс (NKTg_2 = (dm/dt) * p).
- **Единица измерения:**

  NKTm (единица переменной инерции).

III. ПОДРОБНОЕ ОПИСАНИЕ АЛГОРИТМА

1. Процесс обработки данных

Алгоритм выполняет анализ тенденции движения через следующие логические шаги:

- **Шаг 1:**

  Получение входных параметров: положение (x), скорость (v), масса (m) и скорость изменения массы (dm/dt).

- **Шаг 2:**

  Расчет линейного импульса p = m * v.

- **Шаг 3:**

  Расчет значений переменной инерции NKTg_1 и NKTg_2.

- **Шаг 4:**

  Классификация тенденции (Tendency) на основе знака значений:

  - NKTg_1 > 0: Удаление от стабильного состояния.
  - NKTg_1 < 0: Приближение к стабильному состоянию.
  - NKTg_2 > 0: Изменение массы способствует движению.
  - NKTg_2 < 0: Изменение массы препятствует движению.

IV. ИСХОДНЫЙ КОД (SOURCE CODE)

1. Исходный код на языке высокого уровня (C++)

C++

```
#include <cstdio>

/**
 * Библиотека расчета переменной инерции согласно закону NKTg
 * Все авторские права на логику расчетов и определения тенденций защищены.
 */
int main() {
   // 1. Объявление параметров: x (положение), v (скорость), m (масса), dm_dt
(скорость изменения m)
   double x = 2.0, v = 3.0, m = 5.0, dm_dt = -0.5;

   // 2. Выполнение расчета согласно закону NKTg
   double p = m * v;        // p = m * v
   double n1 = x * p;       // NKTg1 = x * p
   double n2 = dm_dt * p;   // NKTg2 = (dm/dt) * p

   // 3. Вывод результатов в формате записи (Record)
   printf("{ p = %.1f\n  nktg1 = %.1f\n  nktg2 = %.1f\n", p, n1, n2);

   // Классификация тенденции движения (Tendency Classification)
   printf("  tendency1 = \"%s\"\n",
           n1 > 0 ? "Удаление от стабильного состояния" :
           n1 < 0 ? "Приближение к стабильному состоянию" : "Стабильное
равновесие");

   printf("  tendency2 = \"%s\" }",
           n2 > 0 ? "Изменение массы способствует движению" :
```

```
           n2 < 0 ? "Изменение массы препятствует движению" : "Нет эффекта от
изменения массы");

   return 0;
}
```

2. Исходный код на низком уровне (Assembly x64)

```
; ------------------------------------------------------------------------
; АЛГОРИТМ АНАЛИЗА ТЕНДЕНЦИИ ДВИЖЕНИЯ NKTg (x64 NASM)
; Автор: Нгуен Кхань Тунг
; © 2026 Нгуен Кхань Тунг. Все права защищены.
; ------------------------------------------------------------------------
; Пример данных (из оригинального документа):
;   x=2.0, v=3.0, m=5.0, dm/dt=-0.5
;   p=15.0 | NKTg1=+30.0 → Удаление | NKTg2=-7.5 → Препятствие
;
; Функция analyze_nktg_tendency — System V AMD64 ABI (Linux):
;   Вход: xmm0=x, xmm1=v, xmm2=m, xmm3=dm/dt
;   Выход: rax=0 (OK), rax=-1 (Ошибка NaN)
; ------------------------------------------------------------------------

section .data
    fmt_all  db "{ p = %.1f", 10,
             db "  nktg1 = %.1f", 10,
             db "  nktg2 = %.1f", 10,
             db "  tendency1 = ""%s""", 10,
             db "  tendency2 = ""%s"" }", 10, 0

    str_away    db "Удаление от стабильного состояния", 0
    str_toward  db "Приближение к стабильному состоянию", 0
    str_stable  db "Стабильное равновесие", 0
    str_support db "Изменение массы способствует движению", 0
    str_resist  db "Изменение массы препятствует движению", 0
    str_noeff   db "Нет эффекта от изменения массы", 0

    val_x     dq 2.0
    val_v     dq 3.0
    val_m     dq 5.0
    val_dm_dt dq -0.5

section .text
    extern printf
    global analyze_nktg_tendency
    global main

;
============================================================================
; analyze_nktg_tendency
;
============================================================================
analyze_nktg_tendency:
    push rbp
    mov rbp, rsp
    and rsp, -16

    ; --- Шаг 1: Получение входных параметров через регистры ---
    ; xmm0 = x      (положение)
    ; xmm1 = v      (скорость)
    ; xmm2 = m      (масса)
    ; xmm3 = dm/dt  (скорость изменения массы)
```

```
    ; Регистры xmm0-xmm3 защищены, прямая перезапись не производится

    ; --- Шаг 2: Расчет импульса p = m * v ---
    movsd xmm4, xmm2           ; xmm4 = m
    mulsd xmm4, xmm1           ; xmm4 = p = 5.0 * 3.0 = 15.0

    ; --- Шаг 3: Расчет NKTg1 и NKTg2 ---
    movsd xmm5, xmm0           ; xmm5 = x
    mulsd xmm5, xmm4           ; xmm5 = NKTg1 = x * p = 2.0 * 15.0 = 30.0

    movsd xmm6, xmm3           ; xmm6 = dm/dt
    mulsd xmm6, xmm4           ; xmm6 = NKTg2 = (dm/dt) * p = (-0.5) * 15.0 =
-7.5

    ; --- Шаг 4: Классификация тенденции на основе знака ---
    xorps xmm7, xmm7           ; xmm7 = 0.0

    ; 4.1. Классификация NKTg1 → Временное хранение в r8
    ucomisd xmm5, xmm7         ; Сравнение NKTg1 с 0.0
    jp .error_nan             ; PF=1 → NaN (не число)
    lea r8, [rel str_stable]   ; По умолчанию: NKTg1 = 0 → Стабильное
равновесие
    ja .t1_away               ; NKTg1 > 0
    jb .t1_toward             ; NKTg1 < 0
    jmp .check_nktg2
.t1_away:
    lea r8, [rel str_away]    ; "Удаление от стабильного состояния"
    jmp .check_nktg2
.t1_toward:
    lea r8, [rel str_toward]  ; "Приближение к стабильному состоянию"

    ; 4.2. Классификация NKTg2 → Временное хранение в r9
.check_nktg2:
    ucomisd xmm6, xmm7         ; Сравнение NKTg2 с 0.0
    jp .error_nan             ; Защита от NaN
    lea r9, [rel str_noeff]   ; По умолчанию: NKTg2 = 0 → Нет эффекта от
изменения массы
    ja .t2_support            ; NKTg2 > 0
    jb .t2_resist             ; NKTg2 < 0
    jmp .do_printf
.t2_support:
    lea r9, [rel str_support] ; "Изменение массы способствует движению"
    jmp .do_printf
.t2_resist:
    lea r9, [rel str_resist]  ; "Изменение массы препятствует движению"

    ; --- Единый вызов printf ---
.do_printf:
    lea rdi, [rel fmt_all]    ; rdi = строка формата
    mov rsi, r8               ; rsi = указатель tendency1 (из r8)
    mov rdx, r9               ; rdx = указатель tendency2 (из r9)
    movsd xmm0, xmm4           ; xmm0 = p     = 15.0
    movsd xmm1, xmm5           ; xmm1 = NKTg1 = 30.0
    movsd xmm2, xmm6           ; xmm2 = NKTg2 = -7.5
    mov eax, 3                ; 3 параметра с плавающей запятой
    call printf

    xor eax, eax              ; return 0
    leave
    ret

.error_nan:
    mov rax, -1
```

```
    leave
    ret

; ========================================================================
; main — Загрузка примеров в регистры, вызов analyze_nktg_tendency
; ========================================================================
main:
    push rbp
    mov rbp, rsp
    and rsp, -16

    ; --- Шаг 1: Загрузка входных параметров (данные из оригинала) ---
    movsd xmm0, [rel val_x]     ; xmm0 = x     = 2.0
    movsd xmm1, [rel val_v]     ; xmm1 = v     = 3.0
    movsd xmm2, [rel val_m]     ; xmm2 = m     = 5.0
    movsd xmm3, [rel val_dm_dt] ; xmm3 = dm/dt = -0.5

    call analyze_nktg_tendency

    xor eax, eax
    leave
    ret

; ------------------------------------------------------------------------
; Ожидаемый результат:
; { p = 15.0
;   nktg1 = 30.0
;   nktg2 = -7.5
;   tendency1 = "Удаление от стабильного состояния"
;   tendency2 = "Изменение массы препятствует движению" }
; ------------------------------------------------------------------------
```

V. ОПРЕДЕЛЕНИЕ СТАБИЛЬНОГО СОСТОЯНИЯ

Стабильное состояние в данном ПО определяется как состояние, при котором положение (x), скорость (v) и масса (m) взаимодействуют таким образом, чтобы поддерживать структуру движения, избегать потери контроля и сохранять присущий объекту паттерн движения.

VI. КВАНТОВОЕ РАСШИРЕНИЕ: АЛГОРИТМ NKTg НА ПЛАТФОРМАХ КВАНТОВЫХ ВЫЧИСЛЕНИЙ

1. Квантовая теоретическая база

Квантовое расширение сохраняет все фундаментальные величины закона NKTg:

- **Импульс (p):** p = m * v
- **Величина NKTg_1:** Произведение положения на импульс (NKTg_1 = x * p)
- **Величина NKTg_2:** Произведение скорости изменения массы на импульс (NKTg_2 = (dm/dt) * p)
- **Единица измерения:** NKTm (единица переменной инерции)

Вместо последовательного расчета каждой комбинации (x, v, m, dm/dt), как в классической модели, квантовая модель кодирует все пространство параметров в состояние суперпозиции и использует квантовый параллелизм для поиска.

2. Постановка квантовой задачи

- **Вход:** Пространство параметров (x, v, m, dm/dt), где каждая переменная дискретизируется на N = 2^n уровней значений и кодируется n кубитами. Общее пространство поиска содержит M = N^4 = 2^{4n} комбинаций.
- **Задача:** Для выбранного условия тенденции f из набора:

Условие f	**Тенденция**
NKTg_1 > 0	Удаление от стабильного состояния
NKTg_1 < 0	Приближение к стабильному состоянию
NKTg_2 > 0	Изменение массы способствует движению
NKTg_2 < 0	Изменение массы препятствует движению
NKTg_1 > 0 и NKTg_2 < 0	Сочетание обоих условий

Найти все комбинации (x, v, m, dm/dt) ∈{1..N}^4, удовлетворяющие выбранному условию f.

- **Выход:** Множество комбинаций параметров, удовлетворяющих f, полученное после квантового измерения.

3. Пространство квантовых состояний

Все пространство параметров кодируется в квантовый регистр из 4n кубитов:

$|q\rangle = |x\rangle \otimes |v\rangle \otimes |m\rangle \otimes |dm/dt\rangle$

Начальное состояние — равномерная суперпозиция через гейты Адамара:

$|s\rangle = H^{\otimes 4n} |0\rangle^{4n} = (1/\sqrt{M}) \sum |x, v, m, dm/dt\rangle$

Каждая комбинация существует одновременно с начальной амплитудой вероятности $1/\sqrt{M}$.

4. Алгоритм квантовой обработки

Алгоритм выполняет анализ тенденций NKTg через следующие логические шаги:

- **Шаг 1:** Получение и кодирование входных параметров (x, v, m, dm/dt) в регистр из 4n кубитов. Применение гейтов Адамара для создания равномерной суперпозиции по всем M = N^4 комбинациям.
- **Шаг 2:** Построение параметризованного Оракула $\hat{O}$ в соответствии с выбранным условием тенденции f. Оракул выполняет три обратимые операции:

- **Compute (Вычисление):** Расчет p = m * v, NKTg_1 = x * p, NKTg_2 = (dm/dt) * p в дополнительном (ancilla) регистре.
- **Phase kickback (Фазовый откат):** Инверсия фазы состояний, удовлетворяющих f.
- **Uncompute (Обратное вычисление):** Восстановление дополнительного регистра в состояние |0⟩.

  $\hat{O}$ |x,v,m,dm/dt⟩ = -|x,v,m,dm/dt⟩, если f(x,v,m,dm/dt) = 1

  $\hat{O}$ |x,v,m,dm/dt⟩ = +|x,v,m,dm/dt⟩, если f(x,v,m,dm/dt) = 0

- **Шаг 3:** Применение оператора диффузии (Diffusion operator) — усиление амплитуды состояний-решений и подавление амплитуды состояний, не удовлетворяющих условию:

  $\hat{D}$ = 2|s⟩⟨ s| - I

- **Шаг 4:** Повторение Шага 2 и Шага 3 в течение T оптимальных циклов. При k решениях в пространстве M:

  T = ⌊(π/4) * √{M/k}⌋ циклов.

  После T циклов вероятность измерения правильного решения стремится к 1.

- **Шаг 5:** Измерение регистра из 4n кубитов — получение комбинации (x, v, m, dm/dt), которая с высокой вероятностью удовлетворяет условию f. Повторение O(k) раз для сбора всего множества из k решений.

5. Сравнение классической и квантовой моделей

Критерии	**Классическая модель**	**Квантовая модель**
Пространство поиска	M = N^4 комбинаций	M = N^4 = 2^{4n} состояний
Метод обработки	Последовательно для каждой комбинации	Квантовый параллелизм
Сложность	O(M) = O(N^4)	O(√M) = O(N^2)
Ускорение	—	Квадратичное по сравнению с классикой
Ресурсы	Неограниченная память	4n + ancilla кубитов

Критерии	Классическая модель	Квантовая модель
Условие поиска	Одно условие f за один запуск	Одно условие f за один запуск

6. Научная значимость

Квантовая модель NKTg сохраняет физическую сущность закона NKTg — величины NKTg_1, NKTg_2 и критерии классификации тенденций — используя при этом алгоритм Гровера для снижения сложности поиска с O(M) = O(N^4) до O(√M) = O(N^2). Оракул спроектирован параметризованным, что позволяет гибко применять его к любому условию тенденции из набора NKTg. Это естественный этап расширения классической модели в сторону квантовых вычислений, открывающий перспективы для моделирования сложных физических систем и крупномасштабной оптимизации параметров.

```
//
================================================================================
// АЛГОРИТМ NKTg — КВАНТОВЫЙ (OpenQASM 3.0) — Полная исправленная версия
// Автор: Нгуен Кхань Тунг
// © 2026 Нгуен Кхань Тунг. Все права защищены.
// ------------------------------------------------------------------------
---
// Исправления после аудита:
//   [1] Использование mcx вместо ccx+cx для обнаружения dm/dt=0
//   [2] Использование mcx вместо ccx+cx для условия MODE 5
//   [3] Uncompute MCZ не повторяет лишние циклы ccx
//   [4] Оракул инкапсулирован в стандартную подпрограмму OpenQASM 3.0
//
================================================================================

OPENQASM 3.0;
include "stdgates.inc";

//
================================================================================
// ОБЪЯВЛЕНИЕ РЕГИСТРОВ
//
================================================================================
qubit[3] x_reg;
qubit[3] v_reg;
qubit[3] m_reg;
qubit[3] d_reg;
qubit[5] anc;
qubit    flag;

bit[3] c_x;
bit[3] c_v;
bit[3] c_m;
bit[3] c_d;

//
================================================================================
// ПОДПРОГРАММА: ОРАКУЛ NKTg (MODE 5: NKTg1>0 И NKTg2<0 И dm/dt≠0)
// Структура: Compute → Phase kickback → Uncompute (Абсолютная симметрия)
//
================================================================================
```

```
def nktg_oracle(qubit[3] xr, qubit[3] vr, qubit[3] mr,
                qubit[3] dr, qubit[5] a, qubit f) {

    // ------------------------------------------------------------------------
---
    // COMPUTE (ВЫЧИСЛЕНИЕ)
    // ------------------------------------------------------------------------
---

    // Шаг 2a: sign(p) = sign(m) XOR sign(v)  →  a[0]
    cx mr[2], a[0];
    cx vr[2], a[0];         // a[0] = sign(m) XOR sign(v) = sign(p)

    // Шаг 2b: sign(NKTg1) = sign(x) XOR sign(p)  →  a[1]
    //          NKTg1 > 0  ⇔  a[1] = 0
    cx xr[2], a[1];
    cx a[0],  a[1];         // a[1] = sign(NKTg1)

    // Шаг 2c: sign(NKTg2) = sign(d) XOR sign(p)  →  a[2]
    //          NKTg2 < 0  ⇔  a[2] = 1
    cx dr[2], a[2];
    cx a[0],  a[2];         // a[2] = sign(NKTg2)

    // Шаг 2d: Обнаружение dm/dt = 0  →  a[3]
    //          dm/dt = 0  ⇔  dr[2]=0 И dr[1]=0 И dr[0]=0
    //          После X: dr[2]=1 И dr[1]=1 И dr[0]=1
    //          FIX [1]: использование mcx с 3 контролями вместо неверного
ccx+cx
    x dr[0]; x dr[1];  x dr[2];
    mcx dr[0], dr[1], dr[2], a[3]; // a[3]=1 если dm/dt=0
    x dr[0]; x dr[1];  x dr[2];    // Отмена X

    // Шаг 2e: Полное условие MODE 5  →  a[4]
    //          f = NKTg1>0 И NKTg2<0 И dm/dt≠0
    //          NKTg1>0  ⇔  a[1]=0 → X(a[1])
    //          NKTg2<0  ⇔  a[2]=1
    //          dm/dt≠0  ⇔  a[3]=0 → X(a[3])
    //          FIX [2]: mcx с 3 контролями вместо неверного ccx+cx
    x a[1]; x a[3];
    mcx a[1], a[2], a[3], a[4];    // a[4]=1 если f=1

    // ------------------------------------------------------------------------
---
    // PHASE KICKBACK (ФАЗОВЫЙ ОТКАТ)
    // flag=|−⟩: CX(a[4], flag) → инвертирует фазу всех состояний,
удовлетворяющих f
    // ------------------------------------------------------------------------
---
    cx a[4], f;

    // ------------------------------------------------------------------------
---
    // UNCOMPUTE (ОБРАТНОЕ ВЫЧИСЛЕНИЕ) — абсолютная симметрия (обратный
порядок)
    // ------------------------------------------------------------------------
---
    mcx a[1], a[2], a[3], a[4];    // Отмена шага 2e
    x a[1]; x a[3];

    x dr[0]; x dr[1];  x dr[2];
    mcx dr[0], dr[1], dr[2], a[3]; // Отмена шага 2d
    x dr[0]; x dr[1];  x dr[2];
```

```
    cx a[0],  a[2];                 // Отмена шага 2c
    cx dr[2], a[2];

    cx a[0],  a[1];                 // Отмена шага 2b
    cx xr[2], a[1];

    cx vr[2], a[0];                 // Отмена шага 2a
    cx mr[2], a[0];
    // a[0..4] = |00000⟩ — полностью восстановлены
}

//
================================================================================
// ПОДПРОГРАММА: ДИФФУЗИЯ D̂ = 2|s⟩⟨s| − I
// H → X → MCZ(12 кубитов) → X → H
// FIX [3]: Uncompute MCZ не повторяет лишние циклы ccx
//
================================================================================
def grover_diffusion(qubit[3] xr, qubit[3] vr,
                     qubit[3] mr, qubit[3] dr, qubit[5] a) {

    // 3a: H на всех 12 кубитах
    h xr[0]; h xr[1]; h xr[2];
    h vr[0]; h vr[1]; h vr[2];
    h mr[0]; h mr[1]; h mr[2];
    h dr[0]; h dr[1]; h dr[2];

    // 3b: X на всех → отображение |0...0⟩ → |1...1⟩
    x xr[0]; x xr[1]; x xr[2];
    x vr[0]; x vr[1]; x vr[2];
    x mr[0]; x mr[1]; x mr[2];
    x dr[0]; x dr[1]; x dr[2];

    // 3c: MCZ(12 кубитов) = H(цель) → MCX(12 кубитов) → H(цель)
    // Разложение MCX(12) в цепь Тоффоли — повторное использование a[0..4]
    h dr[2];

    // Уровень 1: Группировка по парам
    ccx xr[0], xr[1], a[0];  // a[0] = xr[0] AND xr[1]
    ccx xr[2], vr[0], a[1];  // a[1] = xr[2] AND vr[0]
    ccx vr[1], vr[2], a[2];  // a[2] = vr[1] AND vr[2]
    ccx mr[0], mr[1], a[3];  // a[3] = mr[0] AND mr[1]
    ccx mr[2], dr[0], a[4];  // a[4] = mr[2] AND dr[0]

    // Уровень 2: Комбинирование результатов Уровня 1
    ccx a[0], a[1], a[0];      // a[0] = первые 4 бита через AND
    // Примечание: промежуточные кубиты необходимы — использование после
uncompute
    // Полное разложение без перекрытия кубитов:
    ccx a[0], a[2], a[1];     // a[1] = a[0] AND a[2]
    ccx a[3], a[4], a[2];     // a[2] = a[3] AND a[4]
    ccx a[1], a[2], a[3];     // a[3] = все 10 бит
    ccx dr[1], a[3], a[4];    // a[4] = все 11 бит через AND
    cx  a[4], dr[2];          // MCX: инвертирует dr[2], если все 12 бит = 1

    // Uncompute Уровня 2 (инверсия, без повторов — FIX [3])
    ccx dr[1], a[3], a[4];
    ccx a[1], a[2], a[3];
    ccx a[3], a[4], a[2];
    ccx a[0], a[2], a[1];
    ccx a[0], a[1], a[0];

    // Uncompute Уровня 1
```

```
    ccx mr[2], dr[0], a[4];
    ccx mr[0], mr[1], a[3];
    ccx vr[1], vr[2], a[2];
    ccx xr[2], vr[0], a[1];
    ccx xr[0], xr[1], a[0];

    h dr[2];                    // Отмена H → MCZ полностью завершен

    // 3d: Отмена X
    x xr[0]; x xr[1]; x xr[2];
    x vr[0]; x vr[1]; x vr[2];
    x mr[0]; x mr[1]; x mr[2];
    x dr[0]; x dr[1]; x dr[2];

    // 3e: Второе H — завершает D^
    h xr[0]; h xr[1]; h xr[2];
    h vr[0]; h vr[1]; h vr[2];
    h mr[0]; h mr[1]; h mr[2];
    h dr[0]; h dr[1]; h dr[2];
}

//
// ============================================================================
// ГЛАВНАЯ ПРОГРАММА
//
// ============================================================================

// Инициализация флага в состояние |−⟩ для Phase Kickback
x flag;
h flag;

// Шаг 1: Равномерная суперпозиция
h x_reg[0]; h x_reg[1]; h x_reg[2];
h v_reg[0]; h v_reg[1]; h v_reg[2];
h m_reg[0]; h m_reg[1]; h m_reg[2];
h d_reg[0]; h d_reg[1]; h d_reg[2];

// Цикл Гровера T раз
// T = floor((π/4) × sqrt(M/k))
// M=4096, k=1 → T≈50 | k=64 → T=6 | k=1024 → T=1
for uint i in [1:1] {
    nktg_oracle(x_reg, v_reg, m_reg, d_reg, anc, flag);
    grover_diffusion(x_reg, v_reg, m_reg, d_reg, anc);
}

// Шаг 5: Измерение
c_x = measure x_reg;   // 010 → x=2.0 положительно ✓
c_v = measure v_reg;   // 011 → v=3.0 положительно ✓
c_m = measure m_reg;   // 001 → m=5.0 положительно ✓
c_d = measure d_reg;   // 101 → dm/dt=-0.5 отрицательно ✓
                       // 000 → фотон: оракул не пометил ✓
```