DDRKAM Reference Manual
Data-Driven Runge-Kutta and Adams Methods

Shyamal Suhana Chandra

2025

Contents

1 Introduction

This manual provides comprehensive documentation for the DDRKAM (Data-Driven Runge-Kutta and Adams Methods) framework. The framework implements numerical methods for solving ordinary differential equations (ODEs) with support for traditional and hierarchical data-driven approaches.

The framework includes:

- Euler's Method (1st order)
- Data-Driven Euler's Method (DDEuler)
- Runge-Kutta 3rd Order Method (RK3)
- Data-Driven Runge-Kutta 3rd Order (DDRK3)
- Adams Methods (AM)
- Data-Driven Adams Methods (DDAM)

2 Euler's Method

2.1 Overview

Euler's Method is the simplest numerical method for solving ODEs. It is a first-order explicit method with local truncation error $O(h^2)$.

2.2 Algorithm

$$y_{n+1} = y_n + h \cdot f(t_n, y_n) \tag{1}$$

where h is the step size, f is the ODE function, and y_n is the state at time t_n.

2.3 API Reference

2.3.1 euler_step

Performs a single integration step using Euler's method.

```
double euler_step(ODEFunction f, double t0, double* y0,
                  size_t n, double h, void* params);
```

Parameters:

- `f`: Function pointer to the ODE system
- `t0`: Current time
- `y0`: Current state vector (modified in-place)
- `n`: Dimension of the system
- `h`: Step size
- `params`: User-defined parameters

Returns: New time value (t0 + h)

2.3.2 euler_solve

Solves an ODE system over a time interval using Euler's method.

```
size_t euler_solve(ODEFunction f, double t0, double t_end,
                   const double* y0, size_t n, double h,
                   void* params, double* t_out, double* y_out);
```

3 Data-Driven Euler's Method

3.1 Overview

Data-Driven Euler's Method (DDEuler) extends standard Euler's method with a hierarchical transformer-inspired architecture that applies adaptive corrections to improve accuracy.

3.2 Algorithm

$$y_{n+1} = y_n + h \cdot f(t_n, y_n) + h \cdot \alpha \cdot \text{Attention}(y_n) \tag{2}$$

where α is a learning rate and $\text{Attention}(y_n)$ is computed through hierarchical transformer layers.

3.3 API Reference

3.3.1 hierarchical_euler_init

Initializes a Data-Driven Euler solver.

```
int hierarchical_euler_init(HierarchicalEulerSolver* solver,
                            size_t num_layers, size_t state_dim,
                            size_t hidden_dim);
```

3.3.2 hierarchical_euler_step

Performs a single integration step using Data-Driven Euler.

```
double hierarchical_euler_step(HierarchicalEulerSolver* solver,
                               ODEFunction f, double t, double* y,
                               double h, void* params);
```

3.3.3 hierarchical_euler_solve

Solves an ODE system using Data-Driven Euler over a time interval.

```
size_t hierarchical_euler_solve(HierarchicalEulerSolver* solver,
                                ODEFunction f, double t0, double t_end,
                                const double* y0, double h, void* params,
                                double* t_out, double* y_out);
```

4 Parallel and Distributed Methods

4.1 Overview

All methods support parallel, distributed, concurrent, hierarchical, and stacked execution modes. This enables:

- Multi-threaded execution (OpenMP, pthreads)
- Distributed computing (MPI)
- Concurrent execution of multiple methods
- Hierarchical/stacked architectures
- Enhanced performance and scalability

4.2 Parallel Runge-Kutta

4.2.1 parallel_rk_init

Initialize parallel RK3 solver.

```
int parallel_rk_init(ParallelRKSolver* solver, size_t state_dim,
                     size_t num_workers, ParallelMode mode,
                     StackedConfig* stacked);
```

4.2.2 parallel_rk_step

Perform parallel RK3 step.

```
double parallel_rk_step(ParallelRKSolver* solver, ODEFunction f,
                        double t, double* y, double h, void* params);
```

4.2.3 stacked_rk_step

Perform stacked/hierarchical RK3 step.

```
double stacked_rk_step(ParallelRKSolver* solver, ODEFunction f,
                       double t, double* y, double h, void* params);
```

4.2.4 concurrent_rk_execute

Execute multiple RK3 instances concurrently.

```
int concurrent_rk_execute(ParallelRKSolver* solvers[], size_t num_solvers,
                          ODEFunction f, double t, const double* y, double h,
                          void* params, double** results);
```

5 Bayesian ODE Solvers with Dynamic Programming

5.1 Forward-Backward Solver

Probabilistic solver computing full posterior distribution.

5.1.1 forward_backward_init

Initialize forward-backward solver with state space discretization and transition matrix.

5.1.2 forward_backward_step

Update forward probabilities with new observation. Complexity: $O(S^2)$ where S is fixed state space size.

5.1.3 forward_backward_get_statistics

Get mean, variance, and full posterior distribution.

5.2 Viterbi Solver

Exact (MAP) solver finding most likely solution path.

5.2.1 viterbi_init

Initialize Viterbi solver.

5.2.2 viterbi_step

Update Viterbi table with new observation.

5.2.3 viterbi_get_map

Get MAP estimate and path probability.

5.3 Randomized Dynamic Programming

Adaptive control via Monte Carlo value estimation.

5.3.1 randomized_dp_init

Initialize randomized DP solver with sampling parameters.

5.3.2 randomized_dp_step

O(1) step: estimate value and choose optimal control.

5.3.3 randomized_dp_solve

Solve ODE using randomized DP with backward induction.

6 O(1) Approximation Solvers

Constant-time approximation methods for hard real-time constraints.

6.1 Lookup Table Solver

Pre-computed solutions with O(1) lookup and bilinear interpolation.

6.1.1 lookup_table_init

Initialize lookup table solver with grid dimensions.

6.1.2 lookup_table_precompute

Pre-compute solution grid offline (can use any method).

6.1.3 lookup_table_solve

O(1) lookup with bilinear interpolation.

6.2 Neural Network Approximator

Neural network for O(1) solution approximation.

6.2.1 neural_approximator_init

Initialize neural network with layer sizes.

6.2.2 neural_approximator_solve

O(1) forward pass to approximate solution.

6.3 Chebyshev Polynomial Approximator

Polynomial approximation with O(k) $\approx$ O(1) evaluation.

6.3.1 chebyshev_approximator_init

Initialize Chebyshev approximator.

6.3.2 chebyshev_approximator_solve

O(1) Chebyshev evaluation using Clenshaw's algorithm.

7 Causal and Granger Causality Solvers

7.1 Causal RK4 Solver

Strictly causal RK4 using only past information.

7.1.1 causal_rk4_init

Initialize causal RK4 solver.

7.1.2 causal_rk4_step

O(1) causal RK4 step.

7.2 Causal Adams Solver

Multi-step Adams method with causal constraints.

7.2.1 causal_adams_init

Initialize causal Adams solver.

7.2.2 causal_adams_step

O(1) causal Adams step.

7.3 Granger Causality Solver

Analyzes causal relationships and adapts solving strategy.

7.3.1 granger_causality_init

Initialize Granger causality solver.

7.3.2 granger_causality_step

O(1) step with adaptive solving based on causality.

7.3.3 granger_causality_get_matrix

Get Granger causality matrix.

8 Quantum ODE Solver

Quantum-inspired nonlinear ODE solver for post-real-time future prediction.

8.1 quantum_ode_init

Initialize quantum ODE solver.

8.2 quantum_ode_step

Quantum-inspired optimization step.

8.3 quantum_ode_predict_future

Predict future states using quantum superposition.

8.4 quantum_ode_refine

Post-real-time refinement for improved accuracy.

9 Reverse Belief Propagation with Lossless Tracing

Backwards uncertainty propagation with lossless state tracing.

9.1 reverse_belief_init

Initialize reverse belief propagation solver with lossless tracing.

9.2 reverse_belief_forward_solve

Solve forward and build lossless trace (stores exact states, derivatives, Jacobians).

9.3 reverse_belief_reverse_solve

Propagate beliefs backwards using lossless trace.

9.4 reverse_belief_smooth

Combine forward and reverse passes for optimal smoothed estimates.

9.5 belief_propagate_forward

Propagate belief forward: $P(t + \Delta t) = J \cdot P(t) \cdot J^T$.

9.6 belief_propagate_backward

Propagate belief backward: $P(t) = J^{-1} \cdot P(t + \Delta t) \cdot (J^{-1})^T$.

10 Real-Time, Online, and Dynamic Methods

10.1 Real-Time Methods

Real-time methods process streaming data with minimal latency.

10.1.1 realtime_rk_init

Initialize real-time RK3 solver.

```
int realtime_rk_init(RealtimeRKSolver* solver, size_t state_dim,
                     double step_size, DataCallback callback,
                     void* callback_data);
```

10.1.2 realtime_rk_step

Perform real-time RK3 step with streaming support.

```
double realtime_rk_step(RealtimeRKSolver* solver, ODEFunction f,
                        double t, double* y, double h, void* params);
```

10.2 Online Methods

Online methods adapt to incoming data with incremental learning.

10.2.1 online_rk_init

Initialize online RK3 solver.

```
int online_rk_init(OnlineRKSolver* solver, size_t state_dim,
                   double initial_step_size, double learning_rate);
```

10.2.2 online_rk_step

Perform online RK3 step with adaptive step size.

```
double online_rk_step(OnlineRKSolver* solver, ODEFunction f,
                      double t, double* y, void* params);
```

10.3 Dynamic Methods

Dynamic methods provide fully adaptive execution.

10.3.1 dynamic_rk_init

Initialize dynamic RK3 solver.

```
int dynamic_rk_init(DynamicRKSolver* solver, size_t state_dim,
                    double initial_step_size, double adaptation_rate);
```

10.3.2 dynamic_rk_step

Perform dynamic RK3 step with adaptive parameters.

```
double dynamic_rk_step(DynamicRKSolver* solver, ODEFunction f,
                       double t, double* y, void* params);
```

11 Nonlinear Programming Solvers

11.1 Karmarkar's Algorithm

Karmarkar's Algorithm is a polynomial-time interior point method for linear programming. It provides guaranteed polynomial-time convergence for linear programming problems.

11.1.1 karmarkar_solver_init

Initialize Karmarkar solver.

```
int karmarkar_solver_init(KarmarkarSolver* solver, size_t state_dim,
                          ADAMSolverType type, double alpha, double beta,
                          double mu, double epsilon, const double* c,
                          const double** A, const double* b,
                          size_t num_constraints);
```

11.1.2 karmarkar_ode_solve

Solve ODE using Karmarkar's algorithm.

```
int karmarkar_ode_solve(KarmarkarSolver* solver, ODEFunction f,
                        double t0, double t_end, const double* y0,
                        void* params, double* y_out);
```

11.2 Nonlinear ODE Solver

11.2.1 nonlinear_ode_init

Initialize nonlinear ODE solver using NLP methods.

```
int nonlinear_ode_init(NonlinearODESolver* solver, size_t state_dim,
                       NLPSolverType solver_type, ObjectiveFunction objective,
                       ConstraintFunction constraints, void* params);
```

11.2.2 nonlinear_ode_solve

Solve ODE using nonlinear programming.

```
int nonlinear_ode_solve(NonlinearODESolver* solver, ODEFunction f,
                        double t0, double t_end, const double* y0,
                        double* y_out);
```

11.3 Nonlinear PDE Solver

11.3.1 nonlinear_pde_init

Initialize nonlinear PDE solver.

```
int nonlinear_pde_init(NonlinearPDESolver* solver, size_t spatial_dim,
                       const size_t* grid_size, NLPSolverType solver_type,
                       PDEFunction pde_func, void* params);
```

11.3.2 nonlinear_pde_solve

Solve PDE using nonlinear programming.

```
int nonlinear_pde_solve(NonlinearPDESolver* solver, double t0, double t_end,
                        const double* u0, double* u_out);
```

12 Additional Distributed, Data-Driven, Online, Real-Time Solvers

12.1 Distributed Data-Driven Solver

Combines distributed computing with data-driven methods.

12.2 Online Data-Driven Solver

Combines online learning with data-driven methods.

12.3 Real-Time Data-Driven Solver

Combines real-time processing with data-driven methods.

12.4 Distributed Online Solver

Combines distributed computing with online learning.

12.5 Distributed Real-Time Solver

Combines distributed computing with real-time processing.

13 Runge-Kutta 3rd Order Method

13.1 Overview

The Runge-Kutta 3rd order method provides a good balance between accuracy and computational efficiency for solving ODEs.

13.2 API Reference

13.2.1 rk3_step

Performs a single integration step using RK3.

```
double rk3_step(ODEFunction f, double t0, double* y0,
                size_t n, double h, void* params);
```

Parameters:

- `f`: Function pointer to the ODE system
- `t0`: Current time
- `y0`: Current state vector (modified in-place)
- `n`: Dimension of the system
- `h`: Step size
- `params`: User-defined parameters

Returns: New time value (t0 + h)

13.2.2 rk3_solve

Solves an ODE system over a time interval.

```
size_t rk3_solve(ODEFunction f, double t0, double t_end,
                 const double* y0, size_t n, double h,
                 void* params, double* t_out, double* y_out);
```

Parameters:

- `f`: Function pointer to the ODE system
- `t0`: Initial time
- `t_end`: Final time
- `y0`: Initial state vector
- `n`: Dimension of the system
- `h`: Step size

- `params`: User-defined parameters
- `t_out`: Output time array (allocated by caller)
- `y_out`: Output state array (n × num_steps, allocated by caller)

Returns: Number of steps taken

13.3 Example

```
void lorenz(double t, const double* y, double* dydt, void* params) {
    double* p = (double*)params;
    double sigma = p[0], rho = p[1], beta = p[2];
    dydt[0] = sigma * (y[1] - y[0]);
    dydt[1] = y[0] * (rho - y[2]) - y[1];
    dydt[2] = y[0] * y[1] - beta * y[2];
}

double params[3] = {10.0, 28.0, 8.0/3.0};
double y0[3] = {1.0, 1.0, 1.0};
double t_out[100];
double y_out[300];
size_t steps = rk3_solve(lorenz, 0.0, 1.0, y0, 3, 0.01,
                         params, t_out, y_out);
```

14 Adams Methods

14.1 Adams-Bashforth 3rd Order

Predictor step for multi-step integration.

```
void adams_bashforth3(ODEFunction f, const double* t,
                      const double* y, size_t n, double h,
                      void* params, double* y_pred);
```

14.2 Adams-Moulton 3rd Order

Corrector step for multi-step integration.

```
void adams_moulton3(ODEFunction f, const double* t,
                    const double* y, size_t n, double h,
                    void* params, const double* y_pred,
                    double* y_corr);
```

15 Hierarchical Runge-Kutta Method

15.1 Overview

The hierarchical RK method uses a transformer-like architecture with multiple processing layers and attention mechanisms.

15.2 API Reference

15.2.1 hierarchical_rk_init

Initializes a hierarchical RK solver.

```
int hierarchical_rk_init(HierarchicalRKSolver* solver,
                         size_t num_layers, size_t state_dim,
                         size_t hidden_dim);
```

Returns: 0 on success, -1 on failure

15.2.2 hierarchical_rk_free

Frees resources allocated by the solver.

```
void hierarchical_rk_free(HierarchicalRKSolver* solver);
```

15.2.3 hierarchical_rk_solve

Solves an ODE using the hierarchical method.

```
size_t hierarchical_rk_solve(HierarchicalRKSolver* solver,
                             ODEFunction f, double t0, double t_end,
                             const double* y0, double h, void* params,
                             double* t_out, double* y_out);
```

16 Objective-C Framework

16.1 DDRKAMSolver

Main solver class for Objective-C applications.

```
DDRKAMSolver* solver = [[DDRKAMSolver alloc]
                        initWithDimension:3];
NSDictionary* result = [solver solveWithFunction:^(double t,
                                                   const double* y,
                                                   double* dydt,
                                                   void* params) {
    // ODE definition
} startTime:0.0 endTime:1.0
initialState:@[@1.0, @1.0, @1.0]
stepSize:0.01 params:NULL];
```

16.2 DDRKAMVisualizer

Visualization component for plotting solutions.

```
DDRKAMVisualizer* viz = [[DDRKAMVisualizer alloc] init];
NSView* view = [viz createVisualizationViewWithTime:timeArray
                                              state:stateArray
                                          dimension:3];
[viz exportToCSV:@"/path/to/output.csv"
            time:timeArray
           state:stateArray];
```

16.3 DDRKAMHierarchicalSolver

Hierarchical solver for Objective-C.

```
DDRKAMHierarchicalSolver* solver =
    [[DDRKAMHierarchicalSolver alloc]
     initWithDimension:3 numLayers:4 hiddenDim:32];
```

17 Map/Reduce Framework

17.1 Overview

The Map/Reduce framework provides distributed ODE solving on commodity hardware with fault tolerance through redundancy. It partitions the state space across mapper nodes, processes derivatives in parallel, and aggregates results through reducer nodes.

17.2 API Reference

17.2.1 mapreduce_ode_init

Initializes a Map/Reduce ODE solver.

```
int mapreduce_ode_init(MapReduceODESolver* solver,
                       size_t state_dim,
                       const MapReduceConfig* config);
```

Parameters:

- `solver`: Pointer to Map/Reduce solver structure
- `state_dim`: Dimension of the ODE system
- `config`: Configuration structure with mapper/reducer counts, redundancy settings, etc.

Returns: 0 on success, -1 on failure

17.2.2 mapreduce_ode_solve

Solves an ODE system using Map/Reduce framework.

```
int mapreduce_ode_solve(MapReduceODESolver* solver,
                        ODEFunction f,
                        double t0, double t_end,
                        const double* y0,
                        double h, void* params,
                        double* y_out);
```

Parameters:

- `solver`: Initialized Map/Reduce solver
- `f`: ODE function pointer
- `t0, t_end`: Time interval
- `y0`: Initial state

- `h`: Step size
- `params`: User-defined parameters
- `y_out`: Output state

Returns: 0 on success, -1 on failure

17.2.3 mapreduce_estimate_cost

Estimates the computational cost for Map/Reduce execution.

```
double mapreduce_estimate_cost(const MapReduceODESolver* solver,
                               double* compute_hours,
                               double* network_cost);
```

17.3 Example

```
#include "mapreduce_solvers.h"

MapReduceODESolver solver;
MapReduceConfig config = {
    .num_mappers = 4,
    .num_reducers = 2,
    .chunk_size = 100,
    .enable_redundancy = 1,
    .redundancy_factor = 3,
    .use_commodity_hardware = 1,
    .network_bandwidth = 100.0,
    .compute_cost_per_hour = 0.10
};

mapreduce_ode_init(&solver, 1000, &config);

double y0[1000] = {1.0, ...};
double y_out[1000];
mapreduce_ode_solve(&solver, my_ode, 0.0, 1.0, y0, 0.01, NULL, y_out);

double cost = mapreduce_estimate_cost(&solver, NULL, NULL);
mapreduce_ode_free(&solver);
```

18 Apache Spark Framework

18.1 Overview

The Apache Spark framework provides distributed ODE solving using Resilient Distributed Datasets (RDDs) for fault-tolerant computation. Spark offers superior performance for iterative algorithms through RDD caching and lineage-based recovery.

18.2 API Reference

18.2.1 spark_ode_init

Initializes a Spark ODE solver.

```
int spark_ode_init(SparkODESolver* solver,
                   size_t state_dim,
                   const SparkConfig* config);
```

Parameters:

- `solver`: Pointer to Spark solver structure
- `state_dim`: Dimension of the ODE system
- `config`: Configuration structure with executor counts, caching settings, etc.

Returns: 0 on success, -1 on failure

18.2.2 spark_ode_solve

Solves an ODE system using Spark framework.

```
int spark_ode_solve(SparkODESolver* solver,
                    ODEFunction f,
                    double t0, double t_end,
                    const double* y0,
                    double h, void* params,
                    double* y_out);
```

Parameters:

- `solver`: Initialized Spark solver
- `f`: ODE function pointer
- `t0, t_end`: Time interval
- `y0`: Initial state
- `h`: Step size
- `params`: User-defined parameters
- `y_out`: Output state

Returns: 0 on success, -1 on failure

18.2.3 spark_estimate_cost

Estimates the computational cost for Spark execution.

```
double spark_estimate_cost(const SparkODESolver* solver,
                           double* compute_hours,
                           double* network_cost,
                           double* storage_cost);
```

18.3 Example

```c
#include "spark_solvers.h"

SparkODESolver solver;
SparkConfig config = {
    .num_executors = 4,
    .cores_per_executor = 2,
    .memory_per_executor = 2048,
    .num_partitions = 8,
    .enable_caching = 1,
    .enable_checkpointing = 1,
    .checkpoint_interval = 1.0,
    .use_commodity_hardware = 1,
    .network_bandwidth = 100.0,
    .compute_cost_per_hour = 0.10,
    .enable_dynamic_allocation = 1
};

spark_ode_init(&solver, 1000, &config);

double y0[1000] = {1.0, ...};
double y_out[1000];
spark_ode_solve(&solver, my_ode, 0.0, 1.0, y0, 0.01, NULL, y_out);

double cost = spark_estimate_cost(&solver, NULL, NULL, NULL);
spark_ode_free(&solver);
```

19 Non-Orthodox Architectures

19.1 Micro-Gas Jet Circuit

Micro-gas jet circuits use fluid dynamics for computation. See `nonorthodox_architectures.h` for API.

19.2 Dataflow (Arvind)

Tagged token dataflow computing for fine-grained parallelism.

19.3 ACE (Turing)

Turing's stored-program computer architecture implementation.

19.4 Systolic Array

Regular array of processing elements with local communication.

19.5 TPU (Patterson)

Google TPU architecture for matrix acceleration.

19.6 GPU Architectures

Support for CUDA, Metal, Vulkan, and AMD GPU acceleration.

19.7 Spiralizer with Chord Algorithm (Chandra, Shyamal)

Spiralizer architecture combining Chord distributed hash tables with Robert Morris collision hashing (MIT) and spiral traversal patterns.

API:

```
SpiralizerChordConfig config = {
    .num_nodes = 256,
    .finger_table_size = 8,
    .hash_table_size = 1024,
    .enable_morris_hashing = 1,
    .enable_spiral_traversal = 1
};
SpiralizerChordSolver solver;
spiralizer_chord_ode_init(&solver, n, &config);
spiralizer_chord_ode_solve(&solver, f, t0, t_end, y0, h, params, y_out);
spiralizer_chord_ode_free(&solver);
```

19.8 Lattice Architecture (Waterfront variation - Chandra, Shyamal)

Variation of Turing's Waterfront architecture, presented by USC alum from HP Labs at MIT event online at Strata. Multi-dimensional lattice with Waterfront buffering.

API:

```
LatticeWaterfrontConfig config = {
    .lattice_dimensions = 4,
    .nodes_per_dimension = 16,
    .waterfront_size = 256,
    .enable_waterfront_buffering = 1,
    .enable_lattice_routing = 1
};
LatticeWaterfrontSolver solver;
lattice_waterfront_ode_init(&solver, n, &config);
lattice_waterfront_ode_solve(&solver, f, t0, t_end, y0, h, params, y_out);
lattice_waterfront_ode_free(&solver);
```

19.9 Standard Parallel Computing Architectures

MPI (Message Passing Interface):

```
MPIConfig config = {
    .num_processes = 8,
    .process_rank = 0,
    .communication_buffer_size = 1024,
    .enable_collective_ops = 1
};
MPISolver solver;
mpi_ode_init(&solver, n, &config);
mpi_ode_solve(&solver, f, t0, t_end, y0, h, params, y_out);
mpi_ode_free(&solver);
```

OpenMP (Open Multi-Processing):

```
OpenMPConfig config = {
    .num_threads = 8,
    .chunk_size = 64,
    .schedule_type = 1,  // dynamic
    .enable_affinity = 1
};
OpenMPSolver solver;
openmp_ode_init(&solver, n, &config);
openmp_ode_solve(&solver, f, t0, t_end, y0, h, params, y_out);
openmp_ode_free(&solver);
```

Pthreads (POSIX Threads):

```
PthreadsConfig config = {
    .num_threads = 8,
    .enable_work_stealing = 1,
    .enable_barrier_sync = 1
};
PthreadsSolver solver;
pthreads_ode_init(&solver, n, &config);
pthreads_ode_solve(&solver, f, t0, t_end, y0, h, params, y_out);
pthreads_ode_free(&solver);
```

19.10 Specialized Hardware Architectures

FPGA AWS F1 (Xilinx UltraScale+):

```
FPGAAWSF1Config config = {
    .num_fpga_devices = 1,
    .num_logic_cells = 2500000,
    .num_dsp_slices = 6840,
    .pcie_bandwidth = 16,  // GB/s
    .enable_hls_acceleration = 1
};
FPGAAWSF1Solver solver;
fpga_aws_f1_ode_init(&solver, n, &config);
fpga_aws_f1_ode_solve(&solver, f, t0, t_end, y0, h, params, y_out);
fpga_aws_f1_ode_free(&solver);
```

TilePU Sunway (SW26010):

```
TilePUSunwayConfig config = {
    .num_core_groups = 4,
    .cores_per_group = 64,
    .num_management_cores = 4,
    .enable_dma = 1,
    .enable_register_communication = 1
};
TilePUSunwaySolver solver;
tilepu_sunway_ode_init(&solver, n, &config);
tilepu_sunway_ode_solve(&solver, f, t0, t_end, y0, h, params, y_out);
tilepu_sunway_ode_free(&solver);
```

Coprocessor Intel Xeon Phi:

```
CoprocessorXeonPhiConfig config = {
    .num_cores = 72,
    .num_threads_per_core = 4,
    .high_bandwidth_memory = 16,  // GB
    .enable_wide_vector = 1,  // 512-bit
    .enable_mic_architecture = 1
};
CoprocessorXeonPhiSolver solver;
coprocessor_xeon_phi_ode_init(&solver, n, &config);
coprocessor_xeon_phi_ode_solve(&solver, f, t0, t_end, y0, h, params, y_out);
coprocessor_xeon_phi_ode_free(&solver);
```

19.11 Additional Architectures

See `nonorthodox_architectures.h` for:

- Standard Parallel Computing: MPI (Message Passing Interface), OpenMP (Open Multi-Processing), Pthreads (POSIX Threads)
- GPGPU (General-Purpose GPU) for platform-agnostic GPU computing
- Vector Processor architecture for SIMD data-parallel operations
- Specialized Hardware: ASIC (Application-Specific Integrated Circuit), FPGA (Field-Programmable Gate Array), FPGA AWS F1 (Xilinx UltraScale+), DSP (Digital Signal Processor)
- Quantum Processing Units: QPU Azure (Microsoft Quantum), QPU Intel Horse Ridge (cryogenic quantum control)
- Specialized Processing Units: TilePU Mellanox (Tile-GX72), TilePU Sunway (SW26010), DPU Microsoft (biological computation), MFPU (Microfluidic Processing Unit), NPU (Neuromorphic Processing Unit), LPU Lightmatter (photonic computing)
- AsAP (Asynchronous Array of Simple Processors) - UC Davis architecture
- Coprocessor: Intel Xeon Phi many-core coprocessor with wide vector units
- Massively-Threaded (Korf) - Frontier search with massive threading
- STARR (Chandra et al.) - Semantic memory architecture - https://github.com/shyamalschandra/STARR
- TrueNorth (IBM), Loihi (Intel), BrainChips - Neuromorphic architectures
- Racetrack (Parkin), Phase Change Memory (IBM Research)
- Lyric (MIT), HW Bayesian Networks (Chandra)
- Semantic Lexographic Binary Search (Chandra & Chandra)
- Kernelized SPS Binary Search (Chandra, Shyamal)
- Multiple-Search Representation Tree Algorithm

19.12 Multiple-Search Representation Tree Algorithm

Uses multiple search strategies (BFS, DFS, A*, Best-First) with tree and graph state representations.

API:

```
MultipleSearchTreeConfig config = {
    .max_tree_depth = 100,
    .max_nodes = 10000,
    .num_search_strategies = 4,
    .enable_bfs = 1,
    .enable_dfs = 1,
    .enable_astar = 1,
    .enable_best_first = 1,
    .heuristic_weight = 1.0
};
MultipleSearchTreeSolver solver;
multiple_search_tree_ode_init(&solver, n, &config);
multiple_search_tree_ode_solve(&solver, f, t0, t_end, y0, h, params, y_out);
multiple_search_tree_ode_free(&solver);
```

20 Platform Support

- macOS 10.13+
- iOS 11.0+
- visionOS 1.0+

21 Copyright