

DDRKAM Reference Manual

Data-Driven Runge-Kutta and Adams Methods

Shyamal Suhana Chandra

2025

Contents

1	Introduction	3
2	Euler’s Method	3
2.1	Overview	3
2.2	Algorithm	3
2.3	API Reference	3
2.3.1	euler_step	3
2.3.2	euler_solve	4
3	Data-Driven Euler’s Method	4
3.1	Overview	4
3.2	Algorithm	4
3.3	API Reference	5
3.3.1	hierarchical_euler_init	5
3.3.2	hierarchical_euler_step	5
3.3.3	hierarchical_euler_solve	5
4	Parallel and Distributed Methods	6
4.1	Overview	6
4.2	Parallel Runge-Kutta	6
4.2.1	parallel_rk_init	6
4.2.2	parallel_rk_step	6

4.2.3	stacked_rk_step	7
4.2.4	concurrent_rk_execute	7
5	Runge-Kutta 3rd Order Method	7
5.1	Overview	7
5.2	API Reference	7
5.2.1	rk3_step	7
5.2.2	rk3_solve	8
5.3	Example	9
6	Adams Methods	9
6.1	Adams-Bashforth 3rd Order	9
6.2	Adams-Moulton 3rd Order	9
7	Hierarchical Runge-Kutta Method	10
7.1	Overview	10
7.2	API Reference	10
7.2.1	hierarchical_rk_init	10
7.2.2	hierarchical_rk_free	10
7.2.3	hierarchical_rk_solve	10
8	Objective-C Framework	11
8.1	DDRKAMSolver	11
8.2	DDRKAMVisualizer	11
8.3	DDRKAMHierarchicalSolver	12
9	Platform Support	12
10	Copyright	12

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.

```
1 double euler_step(ODEFunction f, double t0, double* y0,  
2                   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 ($t_0 + h$)

2.3.2 euler_solve

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

```
1 size_t euler_solve(ODEFunction f, double t0, double
    t_end,
2                     const double* y0, size_t n, double h,
3                     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

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

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.

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

3.3.2 hierarchical_euler_step

Performs a single integration step using Data-Driven Euler.

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

3.3.3 hierarchical_euler_solve

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

```
1 size_t hierarchical_euler_solve(HierarchicalEulerSolver*
    solver,
2                               ODEFunction f, double
    t0, double t_end,
3                               const double* y0,
    double h, void*
    params,
4                               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.

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

4.2.2 `parallel_rk_step`

Perform parallel RK3 step.

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

4.2.3 stacked_rk_step

Perform stacked/hierarchical RK3 step.

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

4.2.4 concurrent_rk_execute

Execute multiple RK3 instances concurrently.

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

5 Real-Time, Online, and Dynamic Methods

5.1 Real-Time Methods

Real-time methods process streaming data with minimal latency.

5.1.1 realtime_rk_init

Initialize real-time RK3 solver.

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

5.1.2 realtime_rk_step

Perform real-time RK3 step with streaming support.

```

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

```

5.2 Online Methods

Online methods adapt to incoming data with incremental learning.

5.2.1 online_rk_init

Initialize online RK3 solver.

```

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

```

5.2.2 online_rk_step

Perform online RK3 step with adaptive step size.

```

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

```

5.3 Dynamic Methods

Dynamic methods provide fully adaptive execution.

5.3.1 dynamic_rk_init

Initialize dynamic RK3 solver.

```

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

```


5.3.2 dynamic_rk_step

Perform dynamic RK3 step with adaptive parameters.

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

6 Runge-Kutta 3rd Order Method

6.1 Overview

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

6.2 API Reference

6.2.1 rk3_step

Performs a single integration step using RK3.

```
1 double rk3_step(ODEFunction f, double t0, double* y0,
2                 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 ($t_0 + h$)

6.2.2 rk3_solve

Solves an ODE system over a time interval.

```
1 size_t rk3_solve(ODEFunction f, double t0, double t_end,  
2                 const double* y0, size_t n, double h,  
3                 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 ($n \times \text{num_steps}$, allocated by caller)

Returns: Number of steps taken

6.3 Example

```
1 void lorenz(double t, const double* y, double* dydt,  
   void* params) {  
2     double* p = (double*)params;  
3     double sigma = p[0], rho = p[1], beta = p[2];  
4     dydt[0] = sigma * (y[1] - y[0]);  
5     dydt[1] = y[0] * (rho - y[2]) - y[1];  
6     dydt[2] = y[0] * y[1] - beta * y[2];  
7 }  
8  
9 double params[3] = {10.0, 28.0, 8.0/3.0};
```

```

10 double y0[3] = {1.0, 1.0, 1.0};
11 double t_out[100];
12 double y_out[300];
13 size_t steps = rk3_solve(lorenz, 0.0, 1.0, y0, 3, 0.01,
14                          params, t_out, y_out);

```

7 Adams Methods

7.1 Adams-Bashforth 3rd Order

Predictor step for multi-step integration.

```

1 void adams_bashforth3(ODEFunction f, const double* t,
2                        const double* y, size_t n, double
3                        h,
4                        void* params, double* y_pred);

```

7.2 Adams-Moulton 3rd Order

Corrector step for multi-step integration.

```

1 void adams_moulton3(ODEFunction f, const double* t,
2                    const double* y, size_t n, double h,
3                    void* params, const double* y_pred,
4                    double* y_corr);

```

8 Hierarchical Runge-Kutta Method

8.1 Overview

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

8.2 API Reference

8.2.1 hierarchical_rk_init

Initializes a hierarchical RK solver.

```

1 int hierarchical_rk_init(HierarchicalRKSolver* solver,
2                          size_t num_layers, size_t
3                          state_dim,
4                          size_t hidden_dim);

```

Returns: 0 on success, -1 on failure

8.2.2 hierarchical_rk_free

Frees resources allocated by the solver.

```

1 void hierarchical_rk_free(HierarchicalRKSolver* solver);

```

8.2.3 hierarchical_rk_solve

Solves an ODE using the hierarchical method.

```

1 size_t hierarchical_rk_solve(HierarchicalRKSolver*
2                             solver,
3                             ODEFunction f, double t0,
4                             double t_end,
5                             const double* y0, double h,
6                             void* params,
7                             double* t_out, double*
8                             y_out);

```

9 Objective-C Framework

9.1 DDRKAMSolver

Main solver class for Objective-C applications.

```

1 DDRKAMSolver* solver = [[DDRKAMSolver alloc]
2                          initWithDimension:3];
3 NSDictionary* result = [solver solveWithFunction:^(
4                          double t,
5
6                          const
7
8                          double
9                          *)

```

```

5         y
        ,
        double
        *

        dydt
        ,
6        void
        *

        params
    )

    {
7        // ODE definition
8    } startTime:0.0 endTime:1.0
9    initialState:@[@1.0, @1.0, @1.0]
10   stepSize:0.01 params:NULL];

```

9.2 DDRKAMVisualizer

Visualization component for plotting solutions.

```

1 DDRKAMVisualizer* viz = [[DDRKAMVisualizer alloc] init];
2 NSString* view = [viz createVisualizationViewWithTime:
    timeArray
3
                                state:
                                stateArray
4
                                dimension:3];
5 [viz exportToCSV:@"/path/to/output.csv"
6     time:timeArray
7     state:stateArray];

```

9.3 DDRKAMHierarchicalSolver

Hierarchical solver for Objective-C.

```

1 DDRKAMHierarchicalSolver* solver =
2     [[DDRKAMHierarchicalSolver alloc]

```

```
3         initWithDimension:3 numLayers:4 hiddenDim:32];
```

10 Platform Support

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

11 Copyright

Copyright (C) 2025, Shyamal Suhana Chandra
All rights reserved.