Lie group integrators and efficient integration of gradient flow

Alexei Bazavov
Michigan State University

Based on:
Bazavov, 2007.04225
Bazavov, Chuna, 2101.05320

Lattice 2021
MIT July 26 — 30 2021

Classical explicit Runge-Kutta methods

$$\frac{dy}{dt} = f(t, y)$$

Algorithm 1 Explicit classical s-stage Runge-Kutta method

1: **for** i=1,...,s **do**
2: $\quad y_i = y_t + h \sum_{j=1}^{i-1} a_{ij} k_j$ $\qquad \triangleright\ a_{i,j \geqslant i} = 0$
3: $\quad k_i = f(t + hc_i, y_i)$ $\qquad \triangleright\ c_1 = 0$
4: **end for**
5: $y_{t+h} = y_t + h \sum_{i=1}^{s} b_i k_i$

$$\begin{array}{c|ccc} c_2 & a_{21} & & \\ c_3 & a_{31} & a_{32} & \\ \hline & b_1 & b_2 & b_3 \end{array}$$

Classical 2N-storage Runge-Kutta methods

Algorithm 2 $2N$-storage explicit classical s-stage Runge-Kutta method

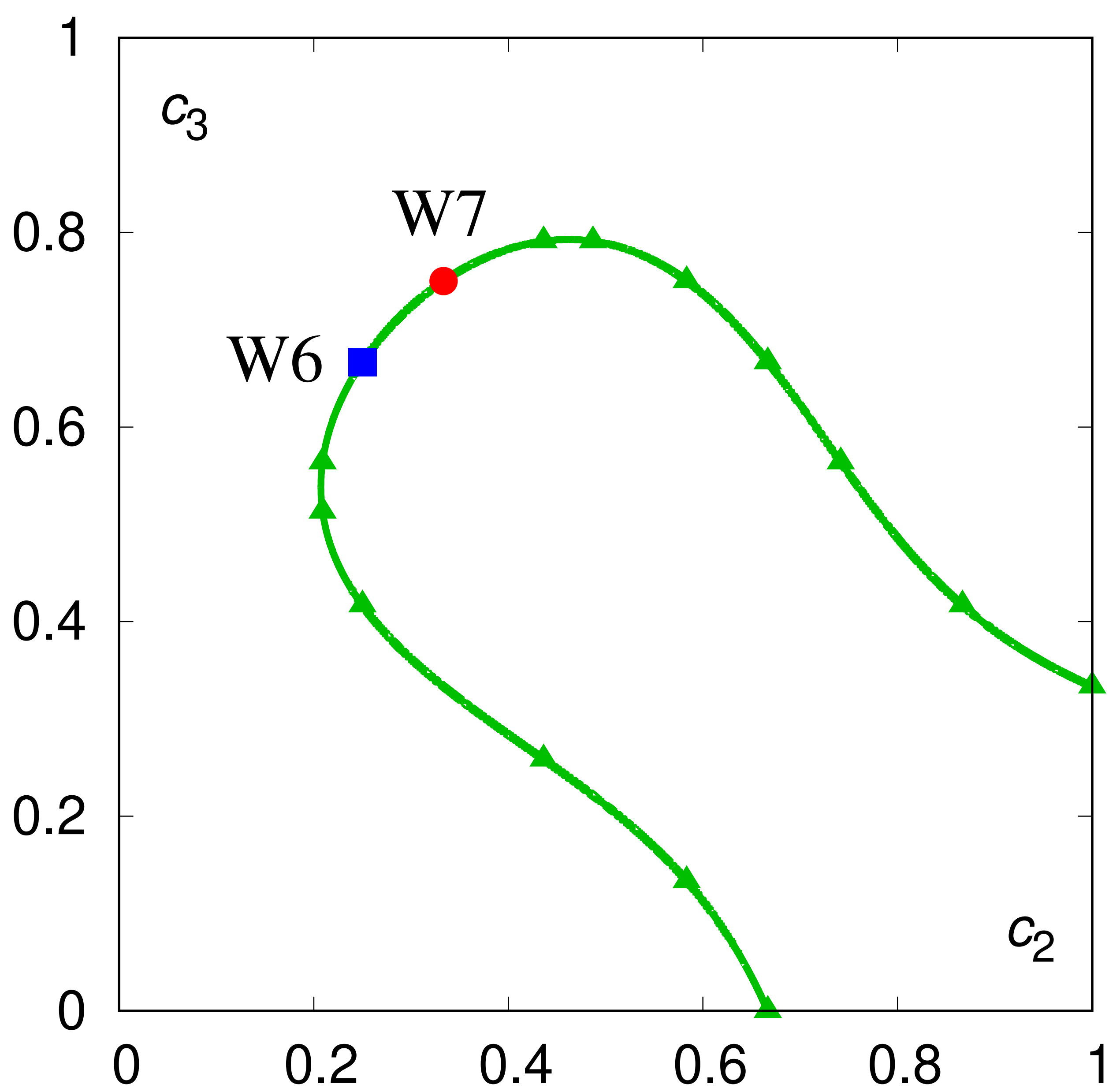
$y_0 = y_t$
for i=1,…,s **do**
    $\Delta y_i = A_i \Delta y_{i-1} + hf(y_{i-1})$ $\triangleright\ A_1 = 0$
    $y_i = y_{i-1} + B_i \Delta y_i$
end for
$y_{t+h} = y_s$

$$a_{ij} = \begin{cases} A_{j+1} a_{i,j+1} + B_j, & j < i-1, \\ B_j, & j = i-1, \\ 0, & \text{otherwise}, \end{cases}$$

$$b_i = \begin{cases} A_{i+1} b_{i+1} + B_i, & i < s, \\ B_i, & i = s, \end{cases}$$

Williamson, 1980

The Williamson curve



- Extra order condition for three-stage third-order 2N-storage Runge-Kutta methods (Williamson, 1980)

$$c_3^2(1 - c_2) + c_3\left(c_2^2 + \frac{1}{2}c_2 - 1\right) + \left(\frac{1}{3} - \frac{1}{2}c_2\right) = 0$$

Structure-preserving integration

$$\frac{dY}{dt} = F(Y)Y$$

- Instead of $y \to y + hf$ we want $Y \to \exp(hF)Y$
- How do we interpret e.g. $y_3 = y_t + h(a_{31}k_1 + a_{32}k_2)$?
  - $Y_3 = \exp(h(a_{31}K_1 + a_{32}K_2))Y_t$?
  - $Y_3 = \exp(ha_{31}K_1)\exp(ha_{32}K_2)Y_t$?
  - $Y_3 = \exp(ha_{32}K_2)\exp(ha_{31}K_1)Y_t$?

Keep single exponential per stage, but add commutators, e.g. $\tilde{c}[K_1, K_2]$

Munthe-Kaas, 1995, 1998

Multiple exponentials with multiple terms but no commutators, e.g.

$\exp(h(\alpha_{2;31}K_1 + \alpha_{2;32}K_2))\exp(h(\alpha_{1;31}K_1 + \alpha_{1;32}K_2))$

Celledoni, Marthinsen, Owren, 2006

Third-order integrator of Lüscher, 1006.4518

$$\dot{V}_t = Z(V_t)V_t,$$

$$W_0 = V_t,$$

$$W_1 = \exp\{\tfrac{1}{4}Z_0\}W_0,$$

$$W_2 = \exp\{\tfrac{8}{9}Z_1 - \tfrac{17}{36}Z_0\}W_1,$$

$$V_{t+\epsilon} = \exp\{\tfrac{3}{4}Z_2 - \tfrac{8}{9}Z_1 + \tfrac{17}{36}Z_0\}W_2,$$

$$Z_i = \epsilon Z(W_i), \qquad i = 0, 1, 2.$$

If you are curious about the derivation:
Bazavov, Chuna, 2101.05320

- Is this coefficient scheme unique? NO
- Are there higher order ones? YES

Higher order integrators?

- Most codes seem to use the original Lüscher, 1006.4518 integrator
- Variable step size scheme based on Lüscher's, e.g. Fritzsch, Ramos, 1301.4388 are also in use — not covered here
- Some evidence of fourth-order, e.g. Cè et al., 1506.06052

> Alternative RK methods for integrating (B.1) are given by the Crouch–Grossman integrators [41, 42]. They are a special case of so-called *commutator-free* Lie group methods [43]. The third order algorithm described in Ref. [12] belongs to this class. The conditions which the coefficients need to satisfy, order by order, are computable up to arbitrary order [44]. They are given by the order conditions for a classical RK method, plus specific extra conditions. At fourth order, however, we did not find a coefficient scheme with the useful properties of the Lüscher's integrator in terms of exponential reusing.

2N-storage commutator-free Lie group method

- It has been recently shown that 2N-storage classical Runge-Kutta schemes of Williamson type are automatically structure-preserving integrators of the same order, 2007.04225:
  - proved at third order
  - conjectured for higher order
- The coefficient scheme of Lüscher, 1006.4518 is equivalent to the classical scheme #6 in Williamson, 1980

Algorithm 6 $2N$-storage s-stage commutator-free Runge-Kutta Lie group method

$Y_0 = Y_t$
for i=1,…,s **do**
  $\Delta Y_i = A_i \Delta Y_{i-1} + hF(Y_{i-1})$ ▹ $A_1 = 0$
  $Y_i = \exp(B_i \Delta Y_i) Y_{i-1}$
end for
$Y_{t+h} = Y_s$

Gradient flow in Grid: qcd/smearing/WilsonFlow.h

```
void WilsonFlow<Gimpl>::evolve_step(typename Gimpl::GaugeField &U) const{
  GaugeField Z(U.Grid());
  GaugeField tmp(U.Grid());
  SG.deriv(U, Z);
  Z *= 0.25;                                     // Z0 = 1/4 * F(U)
  Gimpl::update_field(Z, U, -2.0*epsilon);      // U = W1 = exp(ep*Z0)*W0

  Z *= -17.0/8.0;
  SG.deriv(U, tmp); Z += tmp;                    // -17/32*Z0 +Z1
  Z *= 8.0/9.0;                                  // Z = -17/36*Z0 +8/9*Z1
  Gimpl::update_field(Z, U, -2.0*epsilon);      // U_= W2 = exp(ep*Z)*W1

  Z *= -4.0/3.0;
  SG.deriv(U, tmp); Z += tmp;                    // 4/3*(17/36*Z0 -8/9*Z1) +Z2
  Z *= 3.0/4.0;                                  // Z = 17/36*Z0 -8/9*Z1 +3/4*Z2
  Gimpl::update_field(Z, U, -2.0*epsilon);      // V(t+e) = exp(ep*Z)*W2
}
```

Gradient flow in Grid: qcd/smearing/WilsonFlow.h

```
void WilsonFlow<Gimpl>::evolve_step(typename Gimpl::GaugeField &U) const{
  GaugeField Z(U.Grid());
  GaugeField tmp(U.Grid());




  Z *= 0.0;
  SG.deriv(U, tmp); Z += tmp;
  tmp = Z;
  tmp *= 0.25;                                  // Z0 = 1/4 * F(U)
  Gimpl::update_field(tmp, U, -2.0*epsilon);    // U = W1 = exp(ep*Z0)*W0

  Z *= -17.0/32.0;
  SG.deriv(U, tmp); Z += tmp;                  // -17/32*Z0 +Z1
  tmp = Z;
  tmp *= 8.0/9.0;                               // Z = -17/36*Z0 +8/9*Z1
  Gimpl::update_field(tmp, U, -2.0*epsilon);    // U_= W2 = exp(ep*Z)*W1

  Z *= -32.0/27.0;
  SG.deriv(U, tmp); Z += tmp;                  // 4/3*(17/36*Z0 -8/9*Z1) +Z2
  tmp = Z;
  tmp *= 3.0/4.0;                               // Z = 17/36*Z0 -8/9*Z1 +3/4*Z2
  Gimpl::update_field(tmp, U, -2.0*epsilon);    // V(t+e) = exp(ep*Z)*W2
}
```

Gradient flow in Grid: qcd/smearing/WilsonFlow.h

```
void WilsonFlow<Gimpl>::evolve_step(typename Gimpl::GaugeField &U) const{
  GaugeField Z(U.Grid());
  GaugeField tmp(U.Grid());

#define RK_STAGES 3
  double A[RK_STAGES] = {0,-17/32.,-32/27.};
  double B[RK_STAGES] = {1/4.,8/9.,3/4.};

  Z *= A[0];
  SG.deriv(U, tmp); Z += tmp;
  tmp = Z;
  tmp *= B[0];                                   // Z0 = 1/4 * F(U)
  Gimpl::update_field(tmp, U, -2.0*epsilon);     // U = W1 = exp(ep*Z0)*W0

  Z *= A[1];
  SG.deriv(U, tmp); Z += tmp;                   // -17/32*Z0 +Z1
  tmp = Z;
  tmp *= B[1];                                 // Z = -17/36*Z0 +8/9*Z1
  Gimpl::update_field(tmp, U, -2.0*epsilon);     // U_= W2 = exp(ep*Z)*W1

  Z *= A[2];
  SG.deriv(U, tmp); Z += tmp;                   // 4/3*(17/36*Z0 -8/9*Z1) +Z2
  tmp = Z;
  tmp *= B[2];                                 // Z = 17/36*Z0 -8/9*Z1 +3/4*Z2
  Gimpl::update_field(tmp, U, -2.0*epsilon);     // V(t+e) = exp(ep*Z)*W2
}
```

Gradient flow in Grid: qcd/smearing/WilsonFlow.h

```
void WilsonFlow<Gimpl>::evolve_step(typename Gimpl::GaugeField &U) const{
  GaugeField Z(U.Grid());
  GaugeField tmp(U.Grid());

#define RK_STAGES 3
  double A[RK_STAGES] = {0,-17/32.,-32/27.};
  double B[RK_STAGES] = {1/4.,8/9.,3/4.};

  for( int i=0; i<RK_STAGES; i++ ) {
    Z *= A[i];
    SG.deriv(U, tmp); Z += tmp;
    tmp = Z;
    tmp *= B[i];
    Gimpl::update_field(tmp, U, -2.0*epsilon);
  }

}
```

Gradient flow in Grid: qcd/smearing/WilsonFlow.h

```
void WilsonFlow<Gimpl>::evolve_step(typename Gimpl::GaugeField &U) const{
  GaugeField Z(U.Grid());
  GaugeField tmp(U.Grid());

#define RK_STAGES 3
  double A[RK_STAGES] = {0,-5/9.,-153/128.};
  double B[RK_STAGES] = {1/3.,15/16.,8/15.};

  for( int i=0; i<RK_STAGES; i++ ) {
    Z *= A[i];
    SG.deriv(U, tmp); Z += tmp;
    tmp = Z;
    tmp *= B[i];
    Gimpl::update_field(tmp, U, -2.0*epsilon);
  }

}
```

Scheme #7 of Williamson, 1980

Gradient flow in Grid: qcd/smearing/WilsonFlow.h

```
void WilsonFlow<Gimpl>::evolve_step(typename Gimpl::GaugeField &U) const{
  GaugeField Z(U.Grid());
  GaugeField tmp(U.Grid());

#define RK_STAGES 5
  double A[RK_STAGES] = {0,
                         -567301805773/1357537059087.,
                         -2404267990393/2016746695238.,
                         -3550918686646/2091501179385.,
                         -1275806237668/842570457699.};
  double B[RK_STAGES] = {1432997174477/9575080441755.,
                          5161836677717/13612068292357.,
                          1720146321549/2090206949498.,
                          3134564353537/4481467310338.,
                          2277821191437/14882151754819.};

  for( int i=0; i<RK_STAGES; i++ ) {
    Z *= A[i];
    SG.deriv(U, tmp); Z += tmp;
    tmp = Z;
    tmp *= B[i];
    Gimpl::update_field(tmp, U, -2.0*epsilon);
  }

}
```

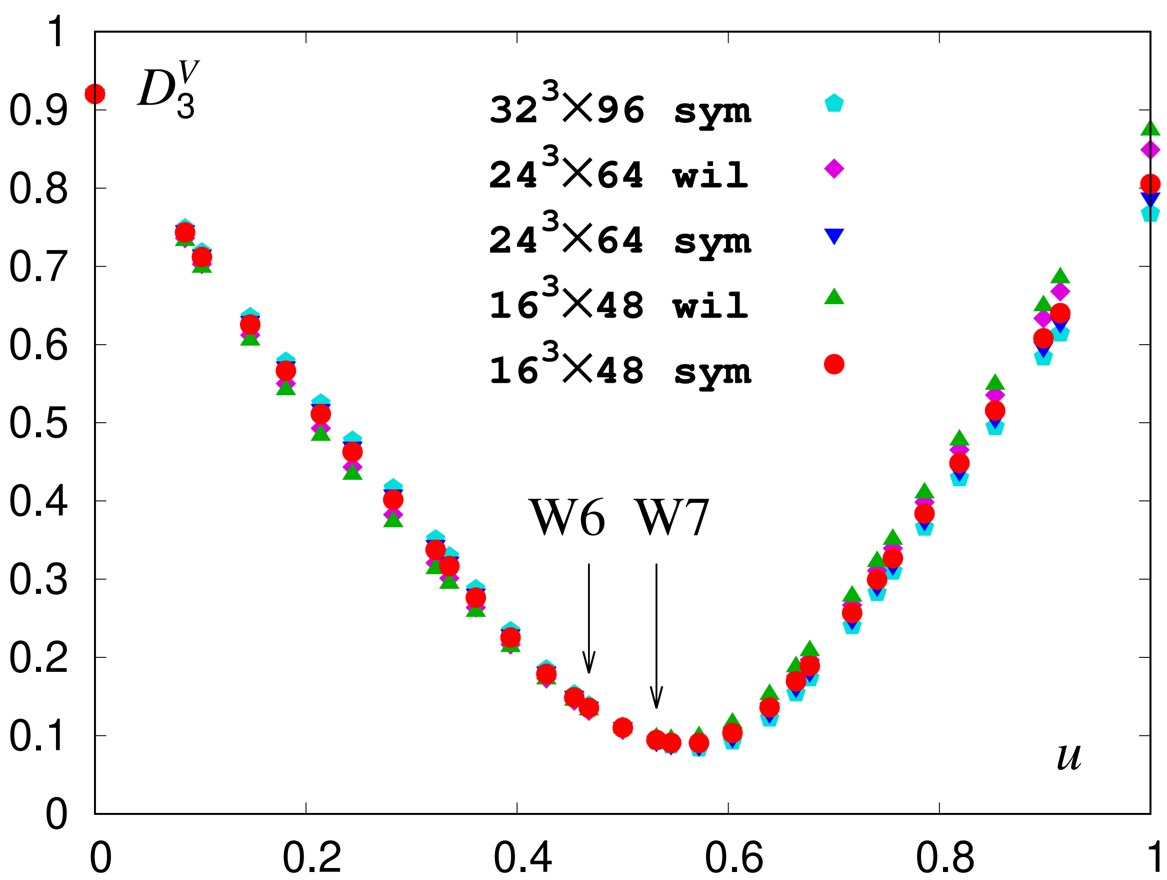
Carpenter, Kennedy, 1994, 4th order

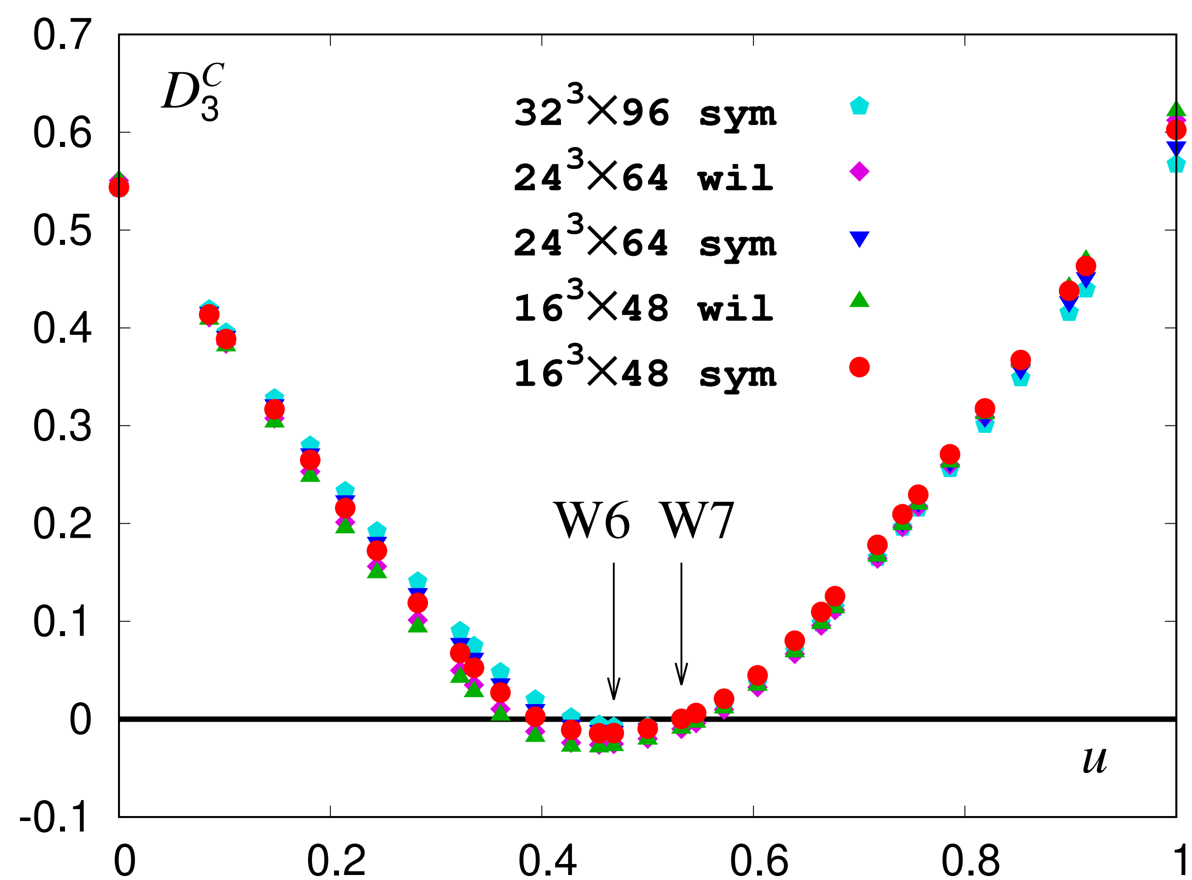
Gradient flow in Grid: qcd/smearing/WilsonFlow.h

```
void WilsonFlow<Gimpl>::evolve_step(typename Gimpl::GaugeField &U) const{
  GaugeField Z(U.Grid());
  GaugeField tmp(U.Grid());

#define RK_STAGES 6
  double A[RK_STAGES] = {0,-0.737101392796,-1.634740794341,
                         -0.744739003780,-1.469897351522,-2.813971388035};
  double B[RK_STAGES] = {0.032918605146,0.823256998200,0.381530948900,
                          0.200092213184,1.718581042715,0.27};

  for( int i=0; i<RK_STAGES; i++ ) {
    Z *= A[i];
    SG.deriv(U, tmp); Z += tmp;
    tmp = Z;
    tmp *= B[i];
    Gimpl::update_field(tmp, U, -2.0*epsilon);
  }

}
```

Berland, Bogey, Bailly, 2006, 4th order

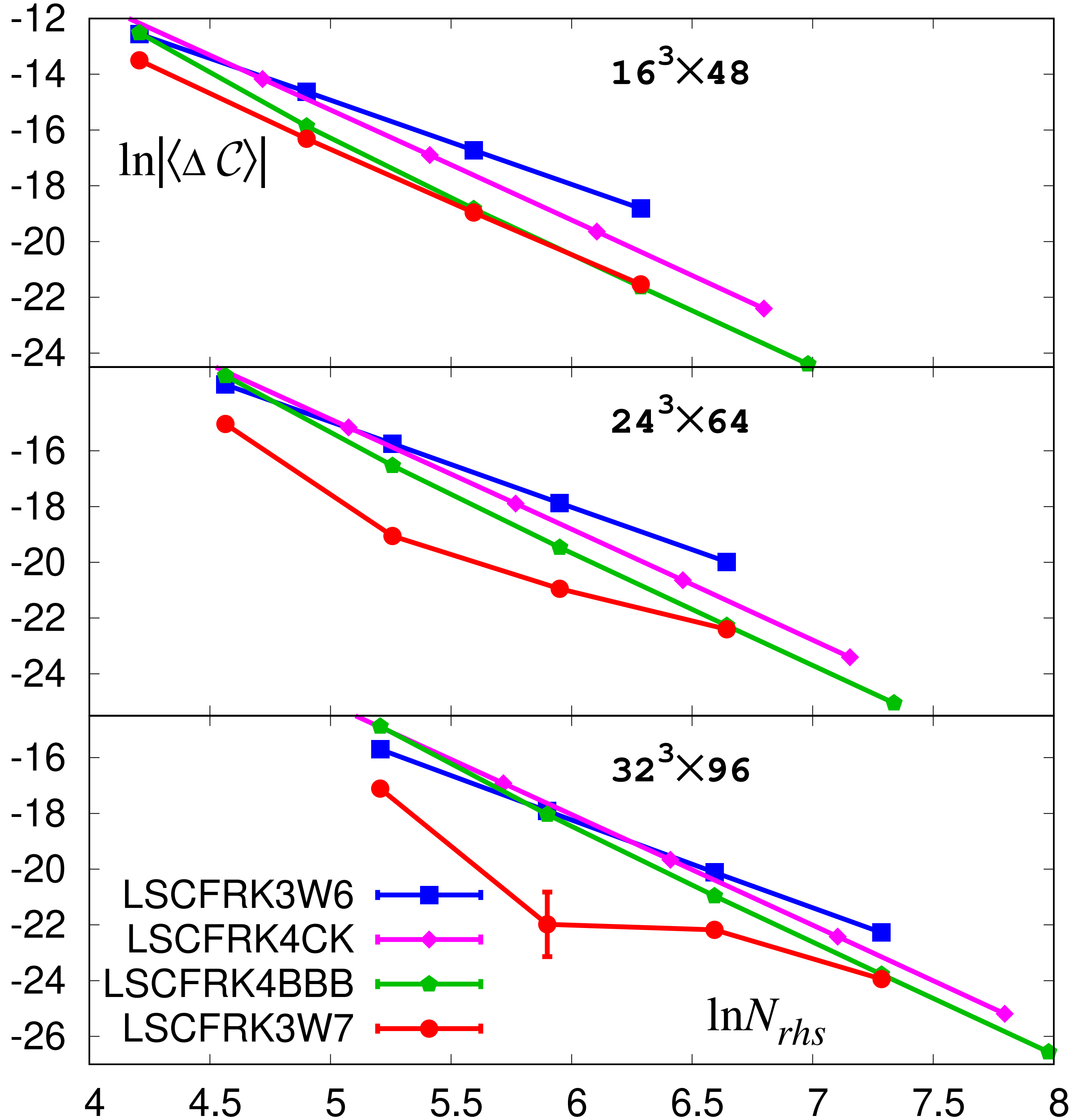
LO error for third-order schemes





- Leading order, $O(h^3)$, coefficient of the integration error in the norm of the gauge field itself vs the coefficient scheme
- Leading order, $O(h^3)$, coefficient of the integration error in the energy density vs the coefficient scheme

Error in the energy density vs computational cost



- Ensembles with $a = 0.15$, 0.12 and 0.09 fm
- Two third-order and two fourth-order schemes
- Non-monotonicity due to zero crossing of the error (i.e. some schemes approach the solution from above and some from below)

Bazavov, Chuna, 2101.05320

Conclusion

- 2N-storage classical Runge-Kutta methods of Williamson type are automatically structure-preserving integrators of the same order
- Must be easy to introduce this type of integrators into existing codes, e.g. Grid
- Implemented in MILC:
  - RKMK (with commutators): third, fourth, fifth and eighth order
  - Low-storage: third order with arbitrary coefficients, fourth order with Carpenter, Kennedy and Berland, Bogey, Bailly coefficients
  - Variable steps size third(second) order pairs with arbitrary coefficients and Bogacki-Shampine type
- Possible gains depend on the application, for scale setting on the MILC ensembles low-storage fourth order works best