Liquid argon benchmarks

Sebastian Micluța-Câmpeanu, Mikhail Vaganov

July 12, 2020

The purpose of these benchmarks is to compare several integrators for use in molecular dynamics simulation. We will use a simulation of liquid argon form the examples of NBodySimulator as test case.

```
using ProgressLogging
using NBodySimulator, OrdinaryDiffEq, StaticArrays
using Plots, DataFrames, StatsPlots

function setup(t)
    T = 120.0 # K
    kb = 1.38e-23 # J/K
    ϵ = T * kb # J
    σ = 3.4e-10 # m
    ρ = 1374 # kg/m^3
    m = 39.95 * 1.6747 * 1e-27 # kg
    N = 350
    L = (m*N/ρ)^(1/3)
    R = 3.5σ
    v_dev = sqrt(kb * T / m) # m/s

    _L = L / σ
    _σ = 1.0
    _ϵ = 1.0
    _m = 1.0
    _v = v_dev / sqrt(ϵ / m)
    _R = R / σ

    bodies = generate_bodies_in_cell_nodes(N, _m, _v, _L)
    lj_parameters = LennardJonesParameters(_ϵ, _σ, _R)
    pbc = CubicPeriodicBoundaryConditions(_L)
    lj_system = PotentialNBodySystem(bodies, Dict(:lennard_jones => lj_parameters));
    simulation = NBodySimulation(lj_system, (0.0, t), pbc, _ϵ/T)

    return simulation
end
```

```
setup (generic function with 1 method)
```

In order to compare different integrating methods we will consider a fixed simulation time and change the timestep (or tolerances in the case of adaptive methods).

```
function benchmark(energyerr, rts, bytes, allocs, nt, nf, t, configs)
    simulation = setup(t)
    prob = SecondOrderODEProblem(simulation)
    for config in configs
        alg = config.alg
        sol, rt, b, gc, memalloc = @timed solve(prob, alg();
            save_everystep=false, progress=true, progress_name="$alg", config...)
        result = NBodySimulator.SimulationResult(sol, simulation)
        ΔE = total_energy(result, t) - total_energy(result, 0)
        energyerr[alg] = ΔE
        rts[alg] = rt
        bytes[alg] = b
        allocs[alg] = memalloc
        nt[alg] = sol.destats.naccept
        nf[alg] = sol.destats.nf + sol.destats.nf2
    end
end

function run_benchmark!(results, t, integrators, tol...; c=ones(length(integrators)))
    @progress "Benchmark at t=$t" for τ in zip(tol...)
        runtime = Dict()
        ΔE = Dict()
        nt = Dict()
        nf = Dict()
        b = Dict()
        allocs = Dict()
        cfg = config(integrators, c, τ...)

        GC.gc()
        benchmark(ΔE, runtime, b, allocs, nt, nf, t, cfg)
        get_tol(idx) = haskey(cfg[idx], :dt) ? cfg[idx].dt : (cfg[idx].abstol,
cfg[idx].rtol)

        for (idx,i) in enumerate(integrators)
            push!(results, [string(i), runtime[i], get_tol(idx)..., abs(ΔE[i]), nt[i],
nf[i], c[idx]])
        end
    end
    return results
end
```

```
run_benchmark! (generic function with 1 method)
```

We will consider symplectic integrators first

```
symplectic_integrators = [
    VelocityVerlet,
    VerletLeapfrog,
    PseudoVerletLeapfrog,
    McAte2,
    CalvoSanz4,
    McAte5,
    Yoshida6,
    KahanLi8,
    SofSpa10
];
```

Since for each method there is a different cost for a timestep, we need to take that into account when choosing the tolerances (`dts` or `abstol&reltol`) for the solvers. This cost was estimated using the commented code below and the results were hardcoded in order to prevent fluctuations in the results between runs due to differences in callibration times.

The calibration is based on running a simulation with equal tolerances for all solvers and then computing the cost as the runtime / number of timesteps. The absolute value of the cost is not very relevant, so the cost was normalized to the cost of one `VelocityVerlet` step.

```
config(integrators, c, τ) = [ (alg=a, dt=τ*c_a) for (a,c_a) in zip(integrators, c)]

t = 35.0
τs = 1e-3

# warmup
c_symplectic = ones(length(symplectic_integrators))
benchmark(Dict(), Dict(), Dict(), Dict(), Dict(), Dict(), 10.,
    config(symplectic_integrators, c_symplectic, τs))

# results = DataFrame(:integrator=>String[], :runtime=>Float64[], :τ=>Float64[],
#     :EnergyError=>Float64[], :timesteps=>Int[], :f_evals=>Int[], :cost=>Float64[]);
# run_benchmark!(results, t, symplectic_integrators, τs)

# c_symplectic .= results[!, :runtime] ./ results[!, :timesteps]
# c_Verlet = c_symplectic[1]
# c_symplectic /= c_Verlet

c_symplectic = [
    1.00,   # VelocityVerlet
    1.05,   # VerletLeapfrog
    0.98,   # PseudoVerletLeapfrog
    1.02,   # McAte2
    2.38,   # CalvoSanz4
    2.92,   # McAte5
    3.74,   # Yoshida6
    8.44,   # KahanLi8
    15.76   # SofSpa10
]
```

```
9-element Array{Float64,1}:
  1.0
  1.05
  0.98
  1.02
  2.38
  2.92
  3.74
  8.44
 15.76
```

Let us now benchmark the solvers for a fixed simulation time and variable timestep

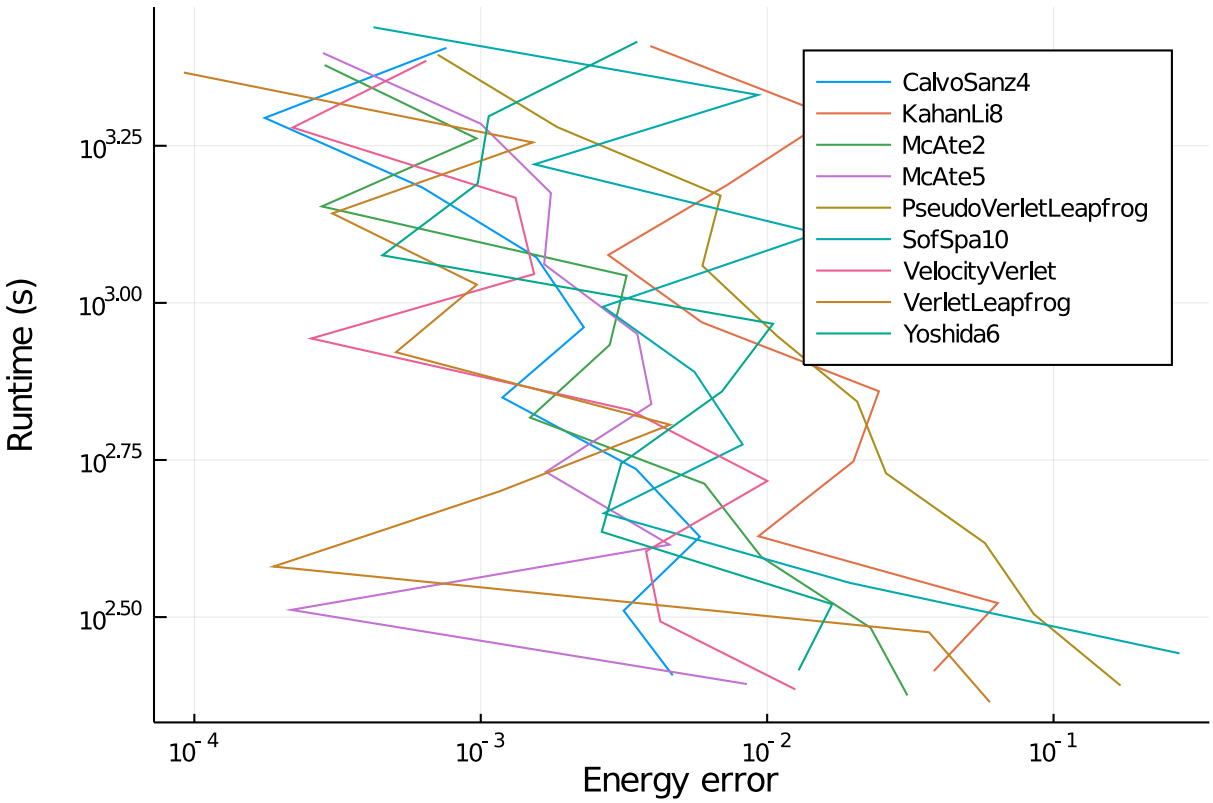
```
t = 40.0
τs = 10 .^range(-4, -3, length=10)
```

```
results = DataFrame(:integrator=>String[], :runtime=>Float64[], :τ=>Float64[],
    :EnergyError=>Float64[], :timesteps=>Int[], :f_evals=>Int[], :cost=>Float64[]);
run_benchmark!(results, t, symplectic_integrators, τs, c=c_symplectic)
```

	integrator	runtime		EnergyError	timesteps	f_evals	cost
	String	Float64	Float64	Float64	Int64	Int64	Float64
1	VelocityVerlet	2427.84	0.0001	0.000645418	400000	800002	1.0
2	VerletLeapfrog	2325.46	0.000105	9.19241e-5	380953	1142861	1.05
3	PseudoVerletLeapfrog	2482.19	9.8e-5	0.000707672	408164	1224494	0.98
4	McAte2	2390.15	0.000102	0.000284938	392157	1176473	1.02
5	CalvoSanz4	2545.22	0.000238	0.00075854	168068	1512614	2.38
6	McAte5	2497.74	0.000292	0.00028132	136987	1506859	2.92
7	Yoshida6	2603.2	0.000374	0.00351961	106952	1604282	3.74
8	KahanLi8	2562.97	0.000844	0.0039062	47394	1658792	8.44
9	SofSpa10	2744.93	0.001576	0.00042212	25381	1802053	15.76
10	VelocityVerlet	1900.88	0.000129155	0.000220459	309706	619414	1.0
11	VerletLeapfrog	1800.14	0.000135613	0.00151847	294958	884876	1.05
12	PseudoVerletLeapfrog	1904.08	0.000126572	0.00184984	316026	948080	0.98
13	McAte2	1826.69	0.000131738	0.000966246	303633	910901	1.02
14	CalvoSanz4	1969.19	0.000307389	0.000176207	130129	1171163	2.38
15	McAte5	1927.83	0.000377133	0.000999696	106064	1166706	2.92
16	Yoshida6	1981.35	0.00048304	0.0010662	82809	1242137	3.74
17	KahanLi8	1977.79	0.00109007	0.0168758	36695	1284327	8.44
18	SofSpa10	2142.3	0.00203548	0.0092905	19652	1395294	15.76
19	VelocityVerlet	1469.92	0.00016681	0.00132258	239794	479590	1.0
20	VerletLeapfrog	1388.0	0.000175151	0.00030329	228375	685127	1.05
21	PseudoVerletLeapfrog	1481.53	0.000163474	0.00687861	244688	734066	0.98
22	McAte2	1423.95	0.000170146	0.000278806	235092	705278	1.02
23	CalvoSanz4	1525.99	0.000397008	0.00062539	100754	906788	2.38
24	McAte5	1495.65	0.000487085	0.00175768	82122	903344	2.92
25	Yoshida6	1547.26	0.00062387	0.000975349	64116	961742	3.74
26	KahanLi8	1538.6	0.00140788	0.00722301	28412	994422	8.44
27	SofSpa10	1661.42	0.00262893	0.00154015	15216	1080338	15.76
28	VelocityVerlet	1111.4	0.000215443	0.00153625	185664	371330	1.0
29	VerletLeapfrog	1069.05	0.000226216	0.000969216	176823	530471	1.05
30	PseudoVerletLeapfrog	1145.51	0.000211135	0.00594357	189453	568361	0.98
31	McAte2	1104.84	0.000219752	0.00323069	182024	546074	1.02
32	CalvoSanz4	1181.05	0.000512755	0.00156593	78010	702092	2.38
33	McAte5	1152.03	0.000629095	0.00166473	63584	699426	2.92
34	Yoshida6	1190.26	0.000805759	0.000453861	49643	744647	3.74
35	KahanLi8	1191.6	0.00181834	0.00278701	21999	769967	8.44
36	SofSpa10	1290.37	0.00339539	0.0149966	11781	836453	15.76
37	VelocityVerlet	877.659	0.000278256	0.000256306	143753	287508	1.0
38	VerletLeapfrog	834.707	0.000292169	0.000505853	136908	410726	1.05
39	PseudoVerletLeapfrog	884.118	0.000272691	0.0108862	146687	440063	0.98
40	McAte2	857.447	0.000283821	0.00282055	140934	422804	1.02
41	CalvoSanz4	914.179	0.000662249	0.0022904	60401	543611	2.38
42	McAte5	892.012	0.000812507	0.00352153	49231	541543	2.92
43	Yoshida6	926.18	0.00104068	0.0104854	38437	576557	3.74
44	KahanLi8	931.236	0.00234848	0.0059155	17033	596157	8.44
45	SofSpa10	986.089	0.00438531	0.00267222	9122	647664	15.76
46	VelocityVerlet	674.915	0.000359381	0.00332968	111303	222608	1.0
47	VerletLeapfrog	639.816	0.00037735	0.00457842	106003	318011	1.05
48	PseudoVerletLeapfrog	696.478	0.000352194	0.0206204	113574	340724	0.98
49	McAte2	656.772	0.000366569	0.00148744	109120	327362	1.02
50	CalvoSanz4	707.021	0.000855328	0.00119014	46766	420896	2.38
51	McAte5	690.261	0.00104939	0.00394261	38118	419300	2.92
52	Yoshida6	722.735	0.00134409	0.00695213	29760	446402	3.74

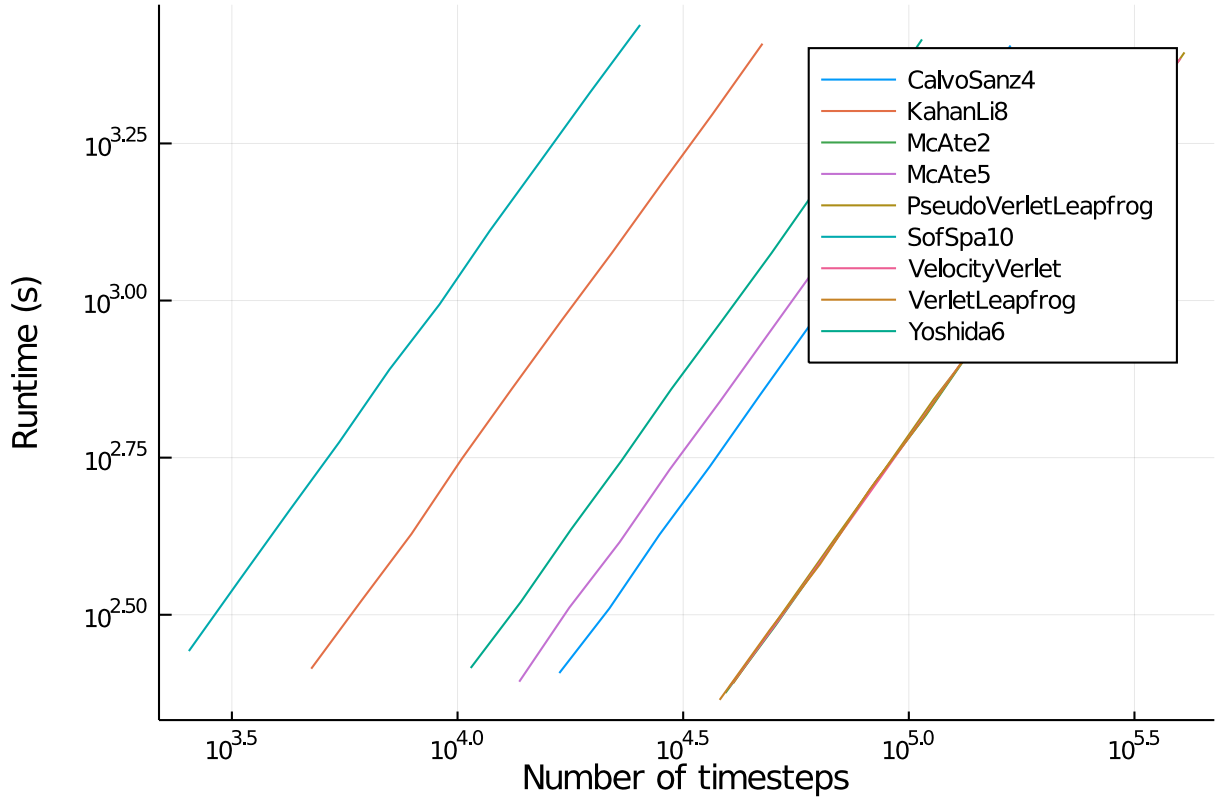
The energy error as a function of runtime is given by

```
@df results plot(:EnergyError, :runtime, group=:integrator,
    xscale=:log10, yscale=:log10, xlabel="Energy error", ylabel="Runtime (s)")
```



Looking at the runtime as a function of timesteps, we can observe that we have a linear dependency for each method, and the slope is the previously computed cost per step.

```
@df results plot(:timesteps, :runtime, group=:integrator,
    xscale=:log10, yscale=:log10, xlabel="Number of timesteps", ylabel="Runtime (s)")
```

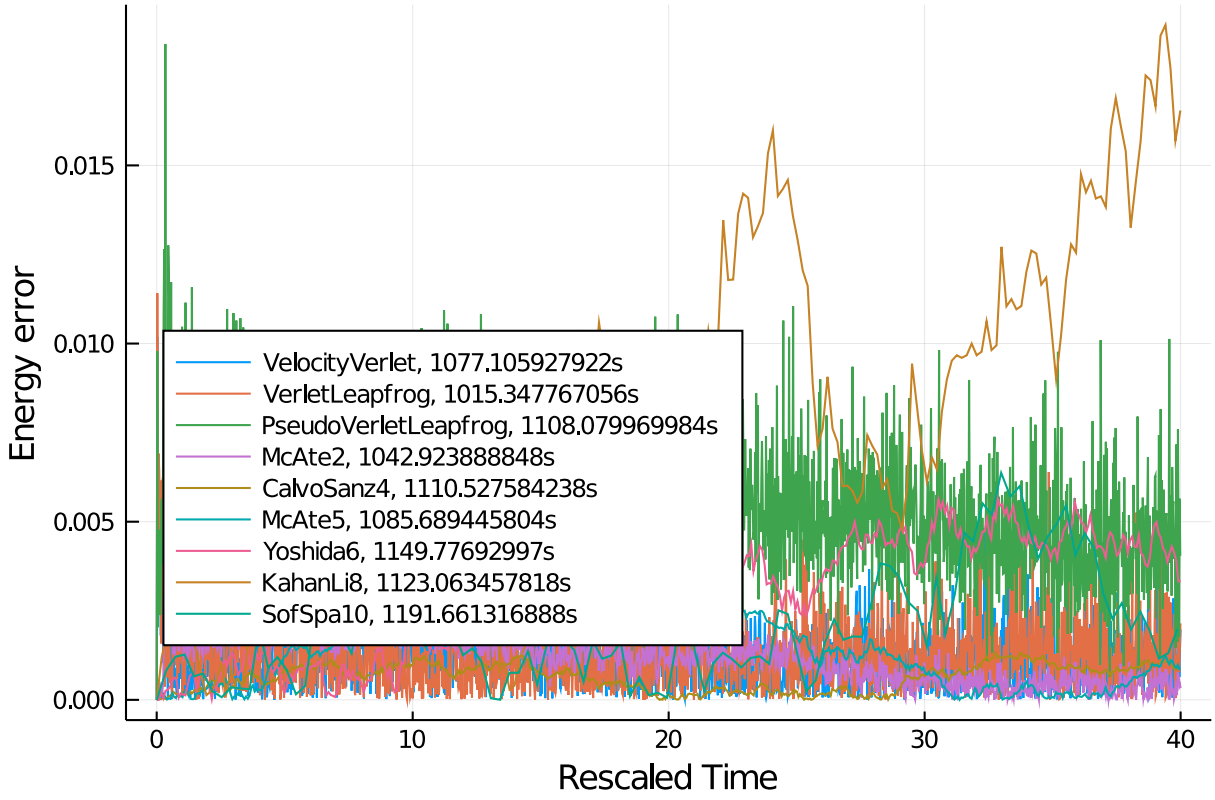


We can also look at the energy error history

```
function benchmark(energyerr, rts, ts, t, configs)
    simulation = setup(t)
    prob = SecondOrderODEProblem(simulation)
    for config in configs
        alg = config.alg
        sol, rt = @timed solve(prob, alg(); progress=true, progress_name="$alg",
config...)
        result = NBodySimulator.SimulationResult(sol, simulation)
        ΔE(t) = total_energy(result, t) - total_energy(result, 0)
        energyerr[alg] = [ΔE(t) for t in sol.t[2:10^2:end]]
        rts[alg] = rt
        ts[alg] = sol.t[2:10^2:end]
    end
end

ΔE = Dict()
rt = Dict()
ts = Dict()
configs = config(symplectic_integrators, c_symplectic, 2.3e-4)
benchmark(ΔE, rt, ts, 40., configs)

plt = plot(xlabel="Rescaled Time", ylabel="Energy error", legend=:bottomleft);
for c in configs
    plot!(plt, ts[c.alg], abs.(ΔE[c.alg]), label="$(c.alg), $(rt[c.alg])s")
end
plt
```



Now, let us compare some adaptive methods

```
adaptive_integrators=[
    # Non-stiff ODE methods
    Tsit5,
    Vern7,
    Vern9,
    # DPRKN
    DPRKN6,
    DPRKN8,
    DPRKN12,
];
```

Similarly to the case of symplectic methods, we will take into account the average cost per timestep in order to have a fair comparison between the solvers.

```
config(integrators, c, at, rt) = [ (alg=a, abstol=at*2^c_a, rtol=rt*2^c_a) for (a,c_a)
in zip(integrators, c)]

t = 35.0
ats = 10 .^range(-14, -4, length=10)
rts = 10 .^range(-14, -4, length=10)

# warmup
c_adaptive = ones(length(adaptive_integrators))
benchmark(Dict(), Dict(), Dict(), Dict(), Dict(), Dict(), 10.,
    config(adaptive_integrators, 1, ats[1], rts[1]))

# results = DataFrame(:integrator=>String[], :runtime=>Float64[], :abstol=>Float64[],
#     :reltol=>Float64[], :EnergyError=>Float64[], :timesteps=>Int[], :f_evals=>Int[],
```

```
:cost=>Float64[]);
# run_benchmark!(results, t, adaptive_integrators, ats[1], rts[1])

# c_adaptive .= results[!, :runtime] ./ results[!, :timesteps]
# c_adaptive /= c_Verlet

c_adaptive = [
    3.55,   # Tsit5,
    7.84,   # Vern7,
    11.38,  # Vern9
    3.56,   # DPRKN6,
    5.10,   # DPRKN8,
    8.85    # DPRKN12,
]
```

```
6-element Array{Float64,1}:
  3.55
  7.84
 11.38
  3.56
  5.1
  8.85
```

Let us now benchmark the solvers for a fixed simulation time and variable timestep

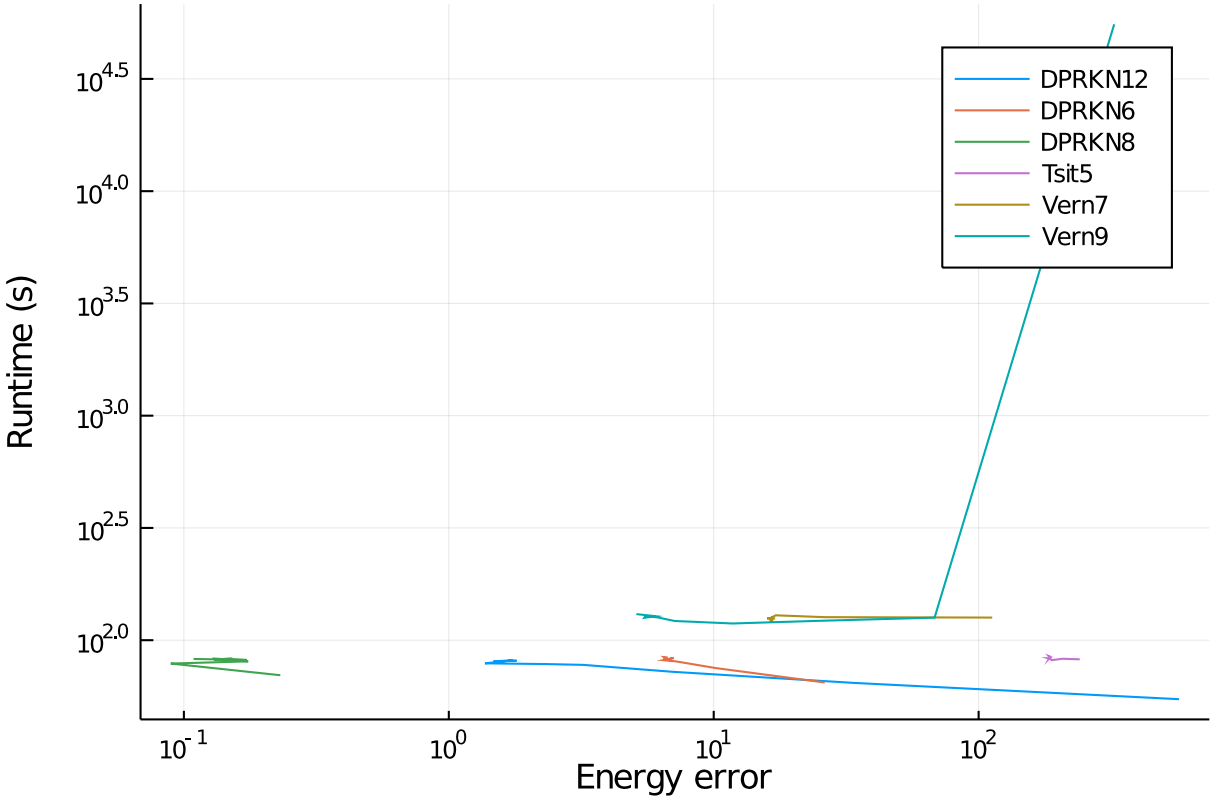
```
t = 40.0

results = DataFrame(:integrator=>String[], :runtime=>Float64[], :abstol=>Float64[],
    :reltol=>Float64[], :EnergyError=>Float64[], :timesteps=>Int[], :f_evals=>Int[],
:cost=>Float64[]);
run_benchmark!(results, t, adaptive_integrators, ats, rts, c=c_adaptive)
```

	integrator	runtime	abstol	reltol	EnergyError	timesteps	f_evals	cost
	String	Float64	Float64	Float64	Float64	Int64	Int64	Float64
1	Tsit5	84.8237	1.17127e-13	1.17127e-13	184.414	4052	27363	3.55
2	Vern7	124.625	2.29126e-12	2.29126e-12	17.0536	3379	40812	7.84
3	Vern9	130.705	2.66515e-11	2.66515e-11	5.10738	2612	42594	11.38
4	DPRKN6	83.806	1.17942e-13	1.17942e-13	6.45146	4426	32008	3.56
5	DPRKN8	82.5273	3.42968e-13	3.42968e-13	0.108784	2919	29644	5.1
6	DPRKN12	81.0723	4.6144e-12	4.6144e-12	1.67223	1536	27814	8.85
7	Tsit5	83.1344	1.51275e-12	1.51275e-12	185.398	4044	27243	3.55
8	Vern7	125.321	2.95928e-11	2.95928e-11	16.7116	3390	41272	7.84
9	Vern9	127.902	3.44217e-10	3.44217e-10	6.01902	2569	41842	11.38
10	DPRKN6	83.0043	1.52327e-12	1.52327e-12	6.57652	4429	32064	3.56
11	DPRKN8	81.6602	4.4296e-12	4.4296e-12	0.172629	2910	29544	5.1
12	DPRKN12	80.9975	5.95973e-11	5.95973e-11	1.801	1532	27670	8.85
13	Tsit5	83.1362	1.95379e-11	1.95379e-11	182.93	4061	27483	3.55
14	Vern7	124.99	3.82206e-10	3.82206e-10	15.9059	3391	41152	7.84
15	Vern9	125.844	4.44574e-9	4.44574e-9	5.46865	2563	41778	11.38
16	DPRKN6	84.0543	1.96738e-11	1.96738e-11	6.48554	4442	32141	3.56
17	DPRKN8	82.6729	5.72104e-11	5.72104e-11	0.128465	2917	29554	5.1
18	DPRKN12	80.6419	7.69729e-10	7.69729e-10	1.47639	1529	27580	8.85
19	Tsit5	82.8809	2.52342e-10	2.52342e-10	184.668	4034	27099	3.55
20	Vern7	125.929	4.93638e-9	4.93638e-9	16.6017	3382	40922	7.84
21	Vern9	127.508	5.74189e-8	5.74189e-8	5.51264	2584	42162	11.38
22	DPRKN6	83.7876	2.54097e-10	2.54097e-10	6.65275	4443	32183	3.56
23	DPRKN8	82.2919	7.38901e-10	7.38901e-10	0.149336	2919	29614	5.1
24	DPRKN12	81.2347	9.94143e-9	9.94143e-9	1.73875	1534	27724	8.85
25	Tsit5	82.6119	3.25912e-9	3.25912e-9	184.352	4063	27447	3.55
26	Vern7	121.172	6.37558e-8	6.37558e-8	16.6453	3354	40452	7.84
27	Vern9	126.554	7.41593e-7	7.41593e-7	6.10006	2556	41586	11.38
28	DPRKN6	82.1777	3.28179e-9	3.28179e-9	6.70031	4418	31840	3.56
29	DPRKN8	81.8722	9.54327e-9	9.54327e-9	0.130281	2918	29584	5.1
30	DPRKN12	78.8974	1.28398e-7	1.28398e-7	1.36911	1521	27490	8.85
31	Tsit5	84.1321	4.20932e-8	4.20932e-8	187.279	4048	27357	3.55
32	Vern7	121.945	8.23438e-7	8.23438e-7	16.3706	3364	40632	7.84
33	Vern9	121.783	9.57805e-6	9.57805e-6	7.11649	2478	40194	11.38
34	DPRKN6	83.3736	4.2386e-8	4.2386e-8	7.05147	4429	31952	3.56
35	DPRKN8	82.647	1.23256e-7	1.23256e-7	0.144106	2923	29734	5.1
36	DPRKN12	78.3065	1.65833e-6	1.65833e-6	2.31288	1505	27112	8.85
37	Tsit5	85.0752	5.43655e-7	5.43655e-7	183.566	4057	27483	3.55
38	Vern7	125.644	1.06351e-5	1.06351e-5	16.4062	3390	41642	7.84
39	Vern9	118.721	0.000123705	0.000123705	11.7815	2334	38882	11.38
40	DPRKN6	81.5022	5.47436e-7	5.47436e-7	6.44867	4391	31469	3.56
41	DPRKN8	81.9509	1.59191e-6	1.59191e-6	0.14134	2905	29364	5.1
42	DPRKN12	77.6623	2.14182e-5	2.14182e-5	3.2172	1460	26392	8.85
43	Tsit5	81.4953	7.02157e-6	7.02157e-6	188.448	4009	27027	3.55
44	Vern7	129.088	0.000137358	0.000137358	17.1583	3378	42592	7.84
45	Vern9	121.786	0.00159771	0.00159771	23.8134	2200	39778	11.38
46	DPRKN6	80.5052	7.07041e-6	7.07041e-6	7.1887	4308	30720	3.56
47	DPRKN8	80.5051	2.05604e-5	2.05604e-5	0.17482	2875	29164	5.1
48	DPRKN12	72.3467	0.000276626	0.000276626	6.97224	1378	24862	8.85
49	Tsit5	82.6817	9.06871e-5	9.06871e-5	207.392	3920	27189	3.55
50	Vern7	126.657	0.00177404	0.00177404	26.2367	3267	41972	7.84
51	Vern9	126.032	0.0206353	0.0206353	68.1922	2080	41378	11.38
52	DPRKN6	75.0982	9.13178e-5	9.13178e-5	10.2122	4087	28893	3.56

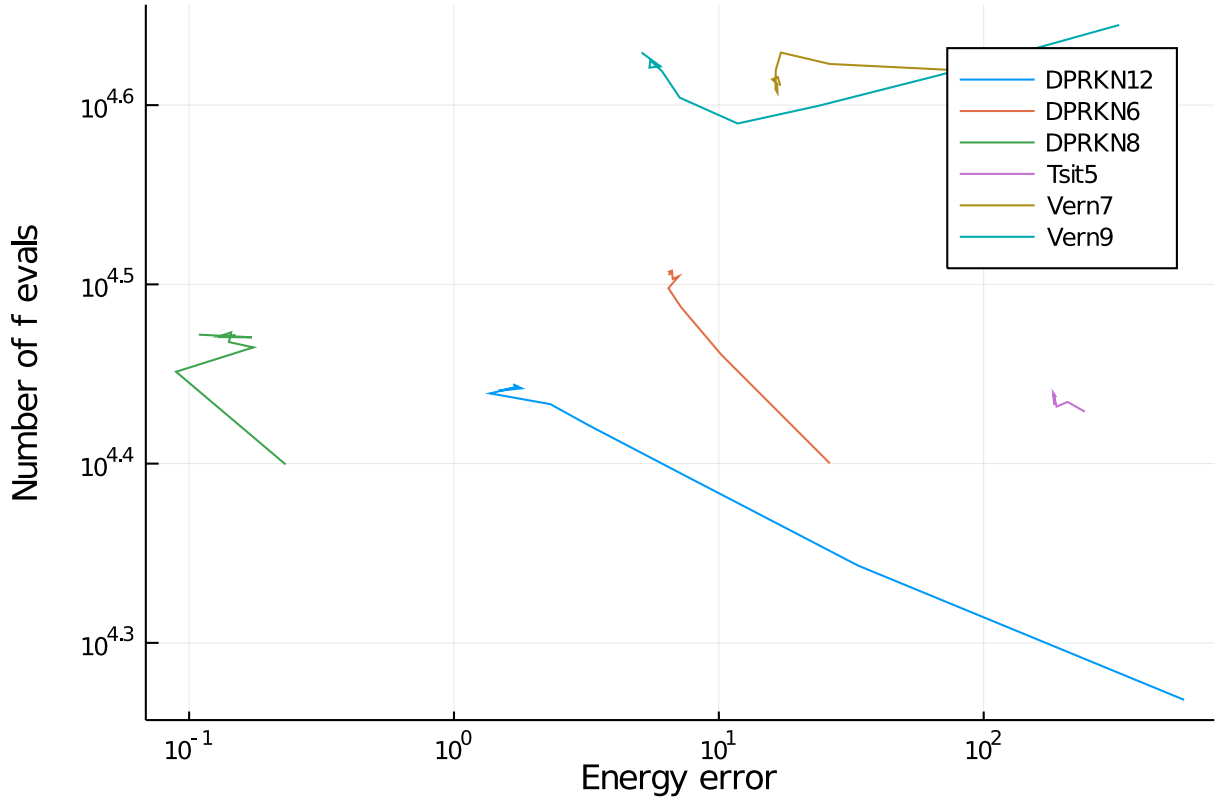
The energy error as a function of runtime is given by

```
@df results plot(:EnergyError, :runtime, group=:integrator,
    xscale=:log10, yscale=:log10, xlabel="Energy error", ylabel="Runtime (s)")
```



If we consider the number of function evaluations instead, we obtain

```
@df results plot(:EnergyError, :f_evals, group=:integrator,
    xscale=:log10, yscale=:log10, xlabel="Energy error", ylabel="Number of f evals")
```



We will now compare the best performing solvers

```
t = 40.0

symplectic_integrators = [
    VelocityVerlet,
    VerletLeapfrog,
    PseudoVerletLeapfrog,
    McAte2,
    CalvoSanz4
]

c_symplectic = [
    1.00,   # VelocityVerlet
    1.05,   # VerletLeapfrog
    0.98,   # PseudoVerletLeapfrog
    1.02,   # McAte2
    2.38,   # CalvoSanz4
]

results1 = DataFrame(:integrator=>String[], :runtime=>Float64[], :τ=>Float64[],
    :EnergyError=>Float64[], :timesteps=>Int[], :f_evals=>Int[], :cost=>Float64[]);
run_benchmark!(results1, t, symplectic_integrators, τs, c=c_symplectic)

adaptive_integrators=[
    DPRKN6,
    DPRKN8,
    DPRKN12,
]

c_adaptive = [
```

```
    3.56,    # DPRKN6,
    5.10,    # DPRKN8,
    8.85     # DPRKN12,
]

results2 = DataFrame(:integrator=>String[], :runtime=>Float64[], :abstol=>Float64[],
    :reltol=>Float64[], :EnergyError=>Float64[], :timesteps=>Int[], :f_evals=>Int[],
:cost=>Float64[]);
run_benchmark!(results2, t, adaptive_integrators, ats, rts, c=c_adaptive)

append!(results1, results2, cols=:union)
results1
```

	integrator	runtime		EnergyError	timesteps	f_evals	cost	
	String	Float64	Float64?	Float64	Int64	Int64	Float64	
1	VelocityVerlet	2431.67	0.0001	0.000645418	400000	800002	1.0	
2	VerletLeapfrog	2328.74	0.000105	9.19241e-5	380953	1142861	1.05	
3	PseudoVerletLeapfrog	2480.47	9.8e-5	0.000707672	408164	1224494	0.98	
4	McAte2	2391.76	0.000102	0.000284938	392157	1176473	1.02	
5	CalvoSanz4	2549.4	0.000238	0.00075854	168068	1512614	2.38	
6	VelocityVerlet	1906.08	0.000129155	0.000220459	309706	619414	1.0	
7	VerletLeapfrog	1798.01	0.000135613	0.00151847	294958	884876	1.05	
8	PseudoVerletLeapfrog	1906.47	0.000126572	0.00184984	316026	948080	0.98	
9	McAte2	1827.51	0.000131738	0.000966246	303633	910901	1.02	
10	CalvoSanz4	1967.89	0.000307389	0.000176207	130129	1171163	2.38	
11	VelocityVerlet	1473.63	0.00016681	0.00132258	239794	479590	1.0	
12	VerletLeapfrog	1390.3	0.000175151	0.00030329	228375	685127	1.05	
13	PseudoVerletLeapfrog	1482.54	0.000163474	0.00687861	244688	734066	0.98	
14	McAte2	1423.76	0.000170146	0.000278806	235092	705278	1.02	
15	CalvoSanz4	1524.02	0.000397008	0.00062539	100754	906788	2.38	
16	VelocityVerlet	1110.66	0.000215443	0.00153625	185664	371330	1.0	
17	VerletLeapfrog	1066.1	0.000226216	0.000969216	176823	530471	1.05	
18	PseudoVerletLeapfrog	1147.52	0.000211135	0.00594357	189453	568361	0.98	
19	McAte2	1107.91	0.000219752	0.00323069	182024	546074	1.02	
20	CalvoSanz4	1184.34	0.000512755	0.00156593	78010	702092	2.38	
21	VelocityVerlet	878.973	0.000278256	0.000256306	143753	287508	1.0	
22	VerletLeapfrog	834.112	0.000292169	0.000505853	136908	410726	1.05	
23	PseudoVerletLeapfrog	884.117	0.000272691	0.0108862	146687	440063	0.98	
24	McAte2	858.192	0.000283821	0.00282055	140934	422804	1.02	
25	CalvoSanz4	914.795	0.000662249	0.0022904	60401	543611	2.38	
26	VelocityVerlet	676.253	0.000359381	0.00332968	111303	222608	1.0	
27	VerletLeapfrog	639.766	0.00037735	0.00457842	106003	318011	1.05	
28	PseudoVerletLeapfrog	696.615	0.000352194	0.0206204	113574	340724	0.98	
29	McAte2	656.951	0.000366569	0.00148744	109120	327362	1.02	
30	CalvoSanz4	707.192	0.000855328	0.00119014	46766	420896	2.38	
31	VelocityVerlet	520.561	0.000464159	0.0100122	86178	172358	1.0	
32	VerletLeapfrog	500.355	0.000487367	0.00116252	82074	246224	1.05	
33	PseudoVerletLeapfrog	536.233	0.000454876	0.0259983	87937	263813	0.98	
34	McAte2	515.348	0.000473442	0.0060436	84488	253466	1.02	
35	CalvoSanz4	545.479	0.0011047	0.00348148	36209	325883	2.38	
36	VelocityVerlet	403.334	0.000599484	0.00377671	66725	133452	1.0	
37	VerletLeapfrog	381.592	0.000629458	0.000189046	63547	190643	1.05	
38	PseudoVerletLeapfrog	414.997	0.000587495	0.0575549	68086	204260	0.98	
39	McAte2	394.584	0.000611474	0.0095366	65416	196250	1.02	
40	CalvoSanz4	424.505	0.00142677	0.00582094	28036	252326	2.38	
41	VelocityVerlet	311.724	0.000774264	0.00423553	51662	103326	1.0	
42	VerletLeapfrog	299.723	0.000812977	0.0367738	49202	147608	1.05	
43	PseudoVerletLeapfrog	319.739	0.000758778	0.0852897	52717	158153	0.98	
44	McAte2	304.158	0.000789749	0.0229745	50650	151952	1.02	
45	CalvoSanz4	324.524	0.00184275	0.00315543	21707	195365	2.38	
46	VelocityVerlet	243.025	0.001	0.0125423	40001	80004	1.0	
47	VerletLeapfrog	231.524	0.00105	0.0599366	38096	114290	1.05	
48	PseudoVerletLeapfrog	245.55	0.00098	0.171365	40817	122453	0.98	
49	McAte2	237.933	0.00102	0.0308871	39216	117650	1.02	
50	CalvoSanz4	255.939	0.00238	0.00468181	16807	151265	2.38	
51	DPRKN6	83.3252	*missing*	6.45146	4426	32008	3.56	1.
52	DPRKN8	82.9884	*missing*	0.108784	2919	29644	5.1	3.

The energy error as a function of runtime is given by

```
@df results1 plot(:EnergyError, :runtime, group=:integrator,
    xscale=:log10, yscale=:log10, xlabel="Energy error", ylabel="Runtime (s)")
```