Multiscalorithms

Integrating Multiscale Dynamical Systems

Numerical Algorithms - Final Assignment

Martijn van Beest, Jaro Camphuijsen, Rahiel Kasim
(11059184) (6042473) (1044539)

December 22, 2016

Contents

1 Introduction

The model by Ford, Kac and Zwanzig [1] describes a dynamical system with a multiscale character. We have one distinguished particle that moves in a heatbath with many other N particles. These N particles don't interact with each other, only with the distinguished particle. The particles in the heatbath behave as harmonic oscillators. Finally, the distinguished particle moves in the potential $V(q) = \frac{1}{4}(q^2 - 1)^2$. This potential is plotted in figure 1. A particle in the potential would be pushed to the middle, where the potential is lowest.

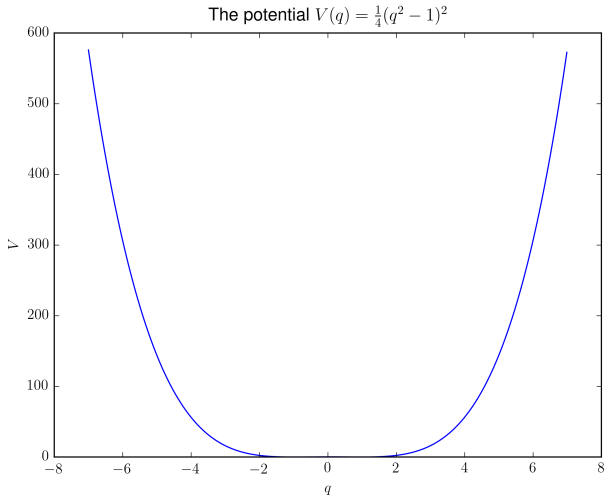

Figure 1: The potential of the system.

2 Problem Description

The problem at hand is that of solving the following system of ordinary differential equations (ODE):

$$\begin{aligned} \dot{q} &= p \\ \dot{p} &= -V\prime(q) + \gamma^2 \sum_{j=1} (u_j - q) \\ \dot{u}_j &= v_j \\ \dot{v}_j &= -j^2(u_j - q) \end{aligned} \tag{1}$$

This system represents a particle in a heat bath, where q and p are respectively the position and momentum of the single distinguished particle and u_j and v_j are respectively the positions and momenta of each of the N harmonic oscillators in the heat bath.
It was shown by Kac and Ford [1] that the distinguished particle in this model exhibits motion similar to Brownian motion under influence of the heat bath's interactions in the limit of $N \to \infty$. However since $\dot{v}_j$ depends on the index j of harmonic oscillators, the model becomes very stiff for large N.

3 Methods

When solving ODE's we use so called numerical integrators to solve an initial value problem (IVP). We calculate the values of the function at the nth time step from the values at previous step(s). This gives us an approximation of a single unique solution to the ODE depending on the initial conditions. Although we do not necessarily find a true solution to the ODE, we track the solution by extrapolating the current solution track at each point to the next time step by integration. In the limit of step size $h \to \infty$ (also implying infinite computation time), this gives the unique solution to the initial value problem.

We've looked at and analyzed several numerical integrators, as we shall see now.

3.1 Forward Euler

Starting at the simplest of integration schemes, the implementation of the Forward Euler scheme can be found in Appendix B. We can replace the value of some function $y(t)$ at $(t + h)$ by the Taylor series:

$$\mathbf{y}(t + h) = \mathbf{y}(t) + h\mathbf{y}'(t) + \frac{1}{2}h^2\mathbf{y}''(t) + \dots \tag{2}$$

The Forward Euler method is a first order explicit integration scheme, where first order means that we drop second and higher order terms in the Taylor series approximation (first order also means that the local error at every step is of size $O(h^2)$), so we end up with:

$$\mathbf{y}(t+h) = \mathbf{y}(t) + h\mathbf{y}'(t), \tag{3}$$

Here $\mathbf{y}'(t)$ is simply the system of expressions for the first derivatives from the differential equation as shown in (1) evaluated at the current time step, while $\mathbf{y}(t)$ are simply the values at the current time step. Adding the size of the time step times the derivative is the simplest form of integration.
The forward Euler method has a very limited stability, especially for stiff equations like this problem. This means that even while the exact solution converges quickly to a steady state solution, integration with forward Euler results in very rapid divergence.

3.2 Backward Euler

The forward Euler method only uses information from the current time step to calculate the next step. This makes the method very unstable. We can also derive the implicit backward Euler method in the same way as the forward version. By taking the first two terms of the backward Taylor series at time $(t-h)$:

$$\begin{aligned} \mathbf{y}(t-h) &= \mathbf{y}(t) - h\mathbf{y}'(t) \\ \mathbf{y}(t) &= \mathbf{y}(t-h) + h\mathbf{y}'(t) \\ \mathbf{y}(t+h) &= \mathbf{y}(t) + h\mathbf{y}'(t+h) \end{aligned} \tag{4}$$

However this requires to evaluate the system at a time of which there are no function values yet. We can solve the resulting set of non-linear equations using Newton's method, of which the implementation can also be found in appendix B.
Newton's method is used to solve the system of backward Euler equations:

$$\mathbf{g}(\mathbf{y}(t+h)) = \mathbf{y}(t+h) - \mathbf{y}(t) - h\mathbf{f}(\mathbf{y}(t+h)) = 0 \tag{5}$$

where $\mathbf{y}'(t+h)$ in (4) is replaced by $\mathbf{f}(\mathbf{y}(t+h))$ with $\mathbf{f}(\mathbf{y})$ the differential equations evaluated with the values $\mathbf{y}$. Since $\mathbf{y}(t)$ is known, the only variable vector in the equation is $\mathbf{y}(t+h)$ and thus we can solve this system $\mathbf{g}$ of non-linear equations with Newton's method. The system of non-linear equations is replaced with the truncated Taylor series of the system which is a system of linear equations:

$$\mathbf{g}(\mathbf{y}+\mathbf{s}) \approx \mathbf{g}(\mathbf{y}) + \mathbf{J}_g(\mathbf{y})\mathbf{s} \tag{6}$$

Here $\mathbf{J}_g$ is the Jacobian matrix of the system g. The solution to this linear system is taken as an approximation to the solution of the non-linear system, so it has to be solved iteratively to converge towards the true solution up until some specified tolerance. Newton's method needs an initial guess, for which we can use either the previous step in the integration process or the next step calculated using an explicit integration method. Calculating the Jacobian of the system is computationally intensive, as it consists of $(2N+2)^2$ components, and it has to be performed many times per time step. This makes backward Euler only feasible when it is possible to make h very big. This is the case for the current problem of which the stiffness increases with the number of harmonic oscillators N. Backward Euler has the inverse stability region of forward Euler and is therefore especially suited for stiff equations as it is unconditionally stable. It will reach the stable solution for arbitrary step size h.
While unconditionally stable, backward Euler is still of first order and will therefore have a local error comparable to forward Euler. In this sense it is not necessary to set a tolerance on Newton's method lower than order $\mathcal{O}(h^2)$, since the error in the integration will always be $\mathcal{O}(h^2)$. To improve this local error we need to increase the order of the integration scheme.

3.3 Runge-Kutta 4

Integration schemes from the explicit Runge-Kutte family are of the form

$$\mathbf{y}_{k+1} = \mathbf{y}_k + \sum_{i=1}^{s} b_i \mathbf{k}_i \tag{7}$$

where $\mathbf{k}_i$ are the intermediate evaluations of the system of derivatives at the values of the current time step adjusted by the previous intermediate result. The two best known explicit Runge-Kutta methods are the forward Euler method, which is of order 1 and explained in section 3.1, and Runge-Kutta of fourth order (RK4) which is given below:

$$\mathbf{y}_{k+1} = \mathbf{y}_k + \frac{h}{6}(\mathbf{k}_1 + 2\mathbf{k}_2 + 2\mathbf{k}_3 + \mathbf{k}_4) \tag{8}$$

$$\begin{aligned} \mathbf{k}_1 &= \mathbf{f}(t_k, \mathbf{y}_k) \\ \mathbf{k}_2 &= \mathbf{f}(t_k + h/2, \mathbf{y}_k + (h/2)\mathbf{k}_1) \\ \mathbf{k}_3 &= \mathbf{f}(t_k + h/2, \mathbf{y}_k + (h/2)\mathbf{k}_2) \\ \mathbf{k}_4 &= \mathbf{f}(t_k + h, \mathbf{y}_k + h\mathbf{k}_3) \end{aligned} \tag{9}$$

The implementation of this method is straightforward and the local error is $\mathcal{O}(h^5)$. This is a major improvement on forward Euler, however since it is still an explicit integration scheme, it is not unconditionally stable and will in the end diverge. The implementation of RK4 can be found in appendix B.

4 Implementation

The methods described in section 3 are implemented using Python 3. The source code can be found in Appendices A to C. The code is separated in three files. The file `main.py` contains the general setup and system initialization. The file `calc.py` contains the different integrator functions and finally the file `sym.py` contains symbolic functions to generate the Jacobian for the Backward Euler function.

All numerical integration is done vectorised, i.e. all vector operations are done at the same time. This implies that given big vectors or matrices, the matrix operations are done transparently over multiple CPU cores. The vector with the values of all variables ($\boldsymbol{y}$) is of the form

$$\begin{pmatrix} q \\ p \\ u_1 \\ \vdots \\ u_N \\ v_1 \\ \vdots \\ v_N \end{pmatrix}$$

. For large N the size of $\boldsymbol{y}$ and even more so of the Jacobian J become large. If we make new arrays of these sizes every time when we evaluate the system and Jacobian, then we spend a lot of time allocating these matrices in memory every time step. So as an optimization, we allocate the memory for the matrices once during initialization and then reuse it during the integration.
Forward Euler and the RK4 functions are pretty straightforward implementations of the schemes as described in section 3.

The implementation of the backward Euler is a bit more extensive since we also need to implement Newton's method to solve the nonlinear system at each time step as well as the Jacobian matrix. As stated before, the Jacobian matrix is computationally expensive. We therefore compute the Jacobian matrix once beforehand in `sym.py`, and evaluate only the values that change for each time step. If we don't use this optimization, the computation time will exists for a major part of Jacobian computations. We use the `sympy` package to compute the Jacobian symbolically. Another optimisation for the backward Euler scheme is in the choice of initial guess. Every time step we first do a step of Forward Euler and use its result as initial guess for Newton's method.
The implementation is also available online on GitHub [2].

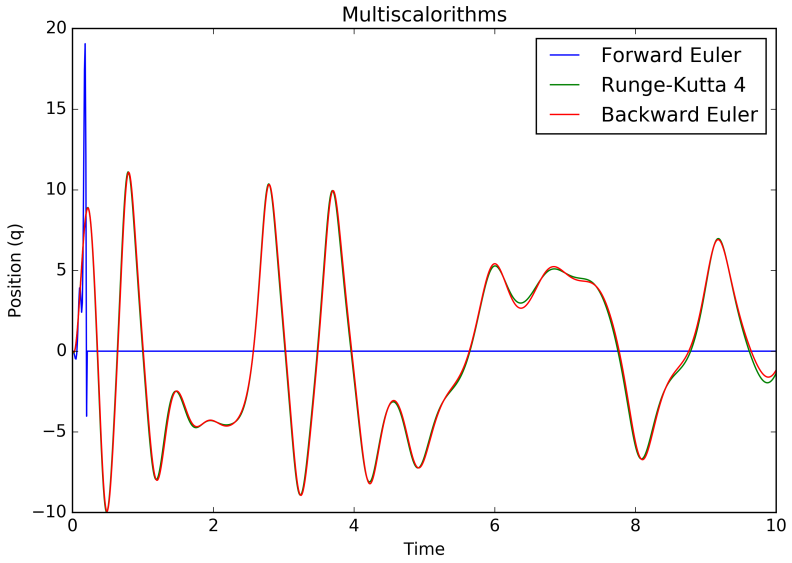

Figure 2: The position of the distinguished particle over a small time period as calculated by the different integration schemes. The value of the Forward Euler is set to zero after its value exceeds 20 (because it diverges).

5 Results

We've used $\gamma = 1$ and $N = 100$. The distinguished particle starts at $q = 0$ with $p = 0.1$ and the N particles have $v_j(0) = 0$ and $u_j(0) = 100r_j$ for all j, where r_j are randomly sampled from a normal distribution with mean 0 and variance 1.

We integrated the model with the three different integration methods. To see how they perform, we look at the value of q integrated from $t = 0$ to $t = 10$ with a step size of $h = 0.01$ and a tolerance for Newton's method of tolerance=$h^2 = 0.001$. The error from the Forward Euler method diverges, so we set those values to zero after its absolute value exceeds 20. The result can be seen in figure 2. We see that the Forward Euler method quickly diverges, it's value is cut off and set to zero so the other graphs stay visible in the same figure. The values for Runge-Kutta 4 and the backward Euler are mostly the same throughout the whole time period. From this we decided to continue comparing the explicit Runge-Kutta 4 with the implicit backward Euler. In this plot the integration time for RK4 was 0.59 CPU and wallclock seconds and backward Euler took 16.3 CPU and 4.1 wallclock seconds. Here we have not taken advantage of the fact that the Backward Euler scheme can handle bigger time steps than RK4.

Taking another look at the potential in figure 1 and the position of the distinguished particle in figure 2, we can interpret what happens. The distinguished particle is pushed out by the heatbath to the sides, but the potential pushes the distinguished particle back to the middle ($q = 0$). This explains the oscillation in q.

To optimize the backward Euler method the initial guess of the Newton method can be chosen in an intelligent way. The simulation was run for increasing step size with constant number of steps for the last integration step as initial guess and with the next integration step using forward Euler as initial guess and CPU times were measured and shown in figure 3

In measuring computational cost we make a distinction between CPU time and wallclock time. CPU time is the actual time a CPU has spent on the integration. This also means that two CPU's computing at the same time for 1 s would give a CPU time of 2 s. The other time we measure is the real time, or wall clock time.

Figure 4 and 5 show comparisons in computation time between the RK4 method and backward Euler for $T = 1000$ and $h = 0.01$. Since the backward Euler method uses parallelization the cpu times in Figure 4 are in fact the cpu times of 8 cores combined. If one is only interested in how much time a computation takes in total, figure 5 shows the elapsed wall time for both integration methods.

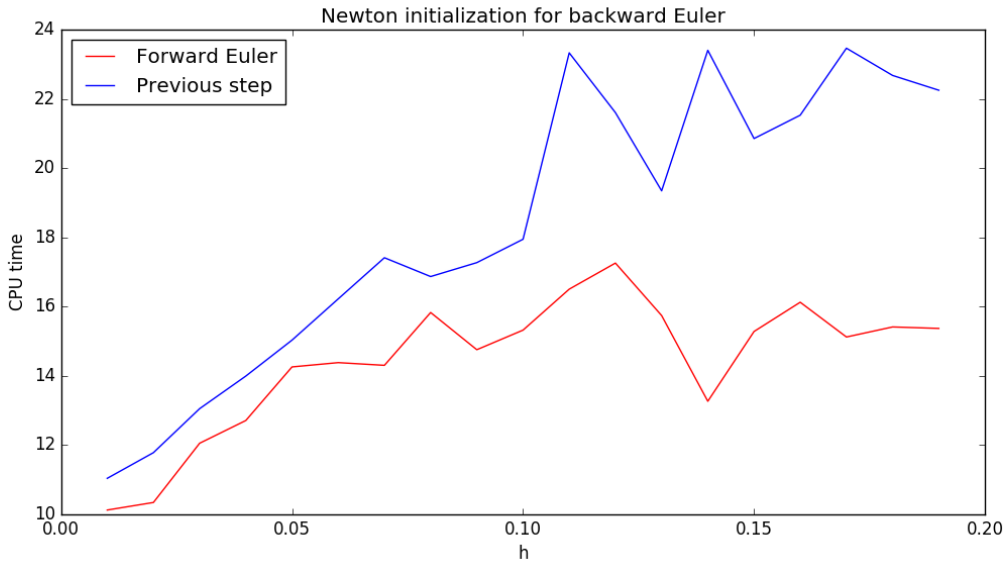

Figure 3: Two methods were used for initializing the Newton method for solving the non-linear equations in backward Euler. Using the previous integration step as initial guess and using the explicit forward Euler method to compute the initial guess. Simulations were run for 1000 steps with increasing step size.

6 Conclusion

In figure 2 it can be seen that the forward Euler method is least stable. After very short time the integration diverges and the integration is killed by our safety test. Both Runge-Kutta 4 and backward Euler remain stable for the full duration of the simulation, however we see a small difference which is probably due to backward Euler being of first order. However for this short simulation Euler took a CPU time of 16.3 while RK4 took 0.59. Also in figure 4 and 5 the CPU and wallclock times are shown for increasing N. This shows the high computation costs for backward Euler due to the iterative solving of the system of equations as opposed to the simple value substitution in RK4. For small time steps and short simulation times RK4 outperforms all, but for backward Euler the step size can be set arbitrarily large while RK4 is bound to small step size to remain stable. And we could in principle choose our backward Euler step size for backward Euler such that its wall time is shorter than Runge-Kutta.

From figure 3 we see that when using the combination of backward Euler with forward Euler as initial guess for the Newton method, for large step sizes the CPU time per step stabilizes, while with the naive initial guess of taking the previous integration step, the computation time keeps growing. As we increase the step size, the error induced by taking the initial guess of the previous integration step keeps growing while the more intelligent guess of the next forward Euler step is much closer to the solution so fewer Newton steps are needed.

We can conclude that the backward Euler method and Runge-Kutta 4 method are both useful in different simulation regimes. While backward Euler is unconditionally stable and can therefore solve stiff systems like these for large time steps, Runge-Kutta 4 has a local error of order h^5 and is therefore more useful for the smaller time scale simulations. We can compute from the CPU times of both methods how much larger the step size of backward Euler should be to outperform Runge-Kutta 4:

$$\frac{t_{\text{bwEuler}}}{t_{\text{RK4}}} = \frac{16.3}{0.59} \approx 28 \tag{10}$$

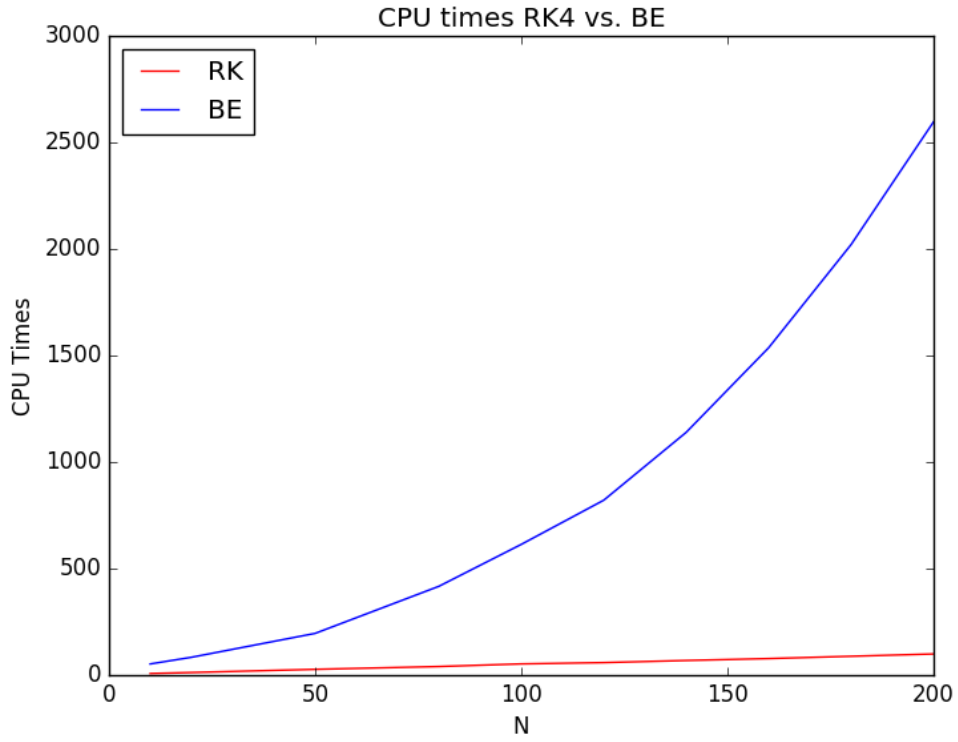

Figure 4: Comparison of CPU times (in seconds) between RK4 and BE for increasing values of N.

7 Recommendations

During implementation we realized that the Jacobian used in Newton's method mostly contains zero's. In Newton's method we use a procedure to solve $\boldsymbol{J_f}(\boldsymbol{x})\boldsymbol{s}_k = -\boldsymbol{f}(\boldsymbol{x_k}$ for $\boldsymbol{s_k}$. This would surely be faster if our Jacobian was implemented as a sparse matrix.

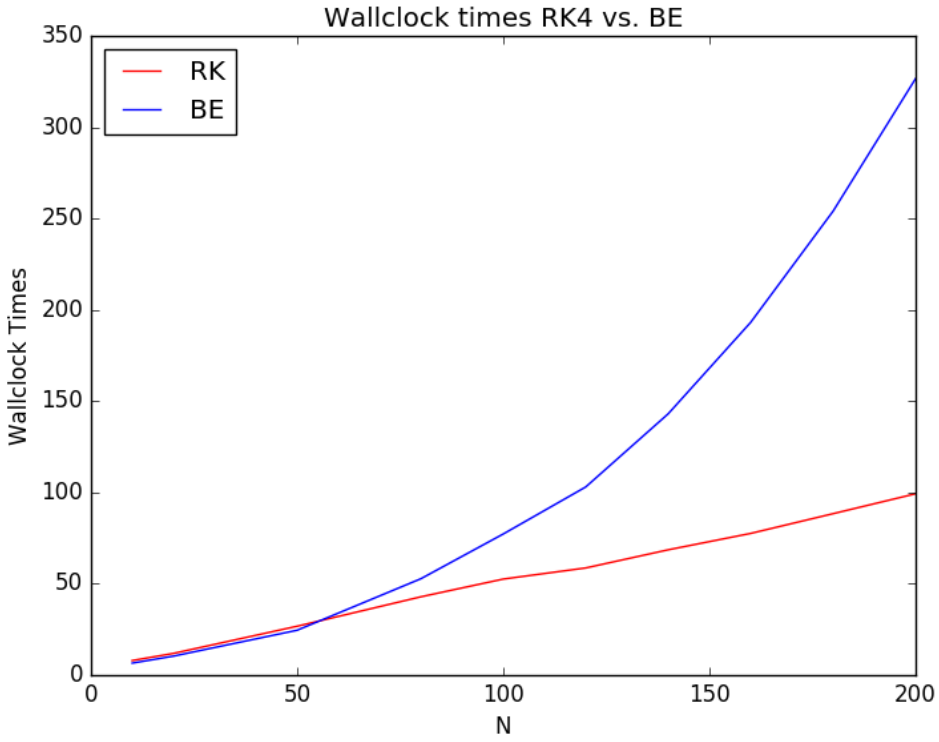

Figure 5: Comparison of Wallclock times (in seconds) between RK4 and BE for increasing values of N.

Appendices

A main.py

```
import argparse
import gc
from random import normalvariate, seed
from time import process_time, perf_counter, time

import numpy as np
import matplotlib
import matplotlib.pyplot as plt
from scipy.integrate import odeint
from sympy import Matrix

from calc import forward_euler, runge_kutta_4, backward_euler
from sym import generate_jacobian


V = lambda q: 1/4 * (q**2 - 1)**2    # double-well potential
dV = lambda q: (q**2 - 1) * q
gamma = 1
SEED = 42


def init(num_steps, N, water=None):
    """Set the system's initial conditions."""
    seed(water)
    x = np.zeros([num_steps + 1, 2 + N + N])  # steps, variables
    x[0, 0] = 0.
    x[0, 1] = 0.1
    for i in range(2, N+2):
        x[0, i] = 100 * normalvariate(0, 1)
        x[0, i+N] = 0
    return x

def f(x):
    """Computes the value of the system at x.
    x Is a vector with as components: q, p, u_1 to u_N, v_1 to v_n
    """
    f.vect[0] = x[1]
    f.vect[1] = -dV(x[0]) + gamma**2 * sum([x[2+i] - x[0] for i in range(2, N+2)])
    for i in range(2, N+2):
        f.vect[i] = x[i+N]
    for j, i in enumerate(range(N+2, 2*N + 2)):
        f.vect[i] = -j**2 * (x[i-N] - x[0])
    return f.vect

def integrate_explicit(method, h, num_steps, N):
    x = init(num_steps, N, SEED)
    start = perf_counter()
    cpu_start = process_time()
    x = method(x, f, h, num_steps)
```

```
    cpu_end = process_time()
    end = perf_counter()
    return x, cpu_end - cpu_start, end - start

def integrate_implicit(h, num_steps, N, tolerance):
    x = init(num_steps, N, SEED)
    j = generate_jacobian(gamma, h, N)
    start = perf_counter()
    cpu_start = process_time()
    x = backward_euler(x, f, j, h, num_steps, tolerance)
    cpu_end = process_time()
    end = perf_counter()
    return x, cpu_end - cpu_start, end - start

def integrate_scipy(h, times, N):
    x = init(1, N, SEED)[0]
    j = generate_jacobian(gamma, h, N)
    start = perf_counter()
    cpu_start = process_time()
    x = odeint(lambda y, t: f(y), x, times, Dfun=lambda y, t: j(y))
    cpu_end = process_time()
    end = perf_counter()
    return x, cpu_end - cpu_start, end - start

def suppress_diverge(x, vect_length):
    """Replaces diverging values with zero's."""
    diverged = False
    for i in range(len(x)):
        if abs(x[i][0]) > 20:
            diverged = True
        if diverged:
            x[i] = np.zeros(vect_length)
    return x


def main():
    global N
    parser = argparse.ArgumentParser(description="Numerically Integrate the model by Ford, Kac and Zwanzig
                                     epilog="Homepage: https://github.com/sunsistemo/multiscalorithms")
    parser.add_argument("-T", "--time", help="integration time period", type=int, default=10)
    parser.add_argument("-dt", "--time-step", help="integration time step", type=float, default=0.01)
    parser.add_argument("-N", type=int, default=100)
    parser.add_argument("-m", "--method", help="integration method", type=str, default="rk4")
    parser.add_argument("-tol", "--tolerance", help="Newton's method convergence tolerance", type=float, d
    parser.add_argument("--compare-explicit-implicit", help=compare_explicit_implicit.__doc__, action="sto
    parser.add_argument("--plot-methods", help=plot_methods.__doc__, action="store_true")
    parser.add_argument("--plot-potential", help=plot_potential.__doc__, action="store_true")
    args = parser.parse_args()

    t_end = args.time
    h = args.time_step
    num_steps = int(t_end / h)
    times = h * np.array(range(num_steps + 1))
```

```
    N = args.N
    vect_length = 2 + N + N
    f.vect = np.empty(vect_length)

    if args.compare_explicit_implicit:
        return compare_explicit_implicit()
    if args.plot_methods:
        return plot_methods()
    if args.plot_potential:
        return plot_potential()

    tolerance = args.tolerance
    methods = {"fe": forward_euler, "rk4": runge_kutta_4, "be": backward_euler, "scipy": odeint}
    method = methods.get(args.method)
    if method is None:
        raise ValueError("Available methods are: {}".format(", ".join(methods.keys())))

    if method.__name__ in ["forward_euler", "runge_kutta_4"]:
        x, cpu_t, t = integrate_explicit(method, h, num_steps, N)
    elif method.__name__ == "backward_euler":
        x, cpu_t, t = integrate_implicit(h, num_steps, N, tolerance)
    elif method.__name__ == "odeint":
        x, cpu_t, t = integrate_scipy(h, times, N)
    return x, cpu_t, t

def compare_explicit_implicit():
    """Compare the CPU time needed to integrate the system with the Runge-Kutta 4
    method vs. the Backward Euler method.

    All other script flags are ignored except N.
    """
    t_end = 1000
    # First we'll do explicit
    h = 0.01
    num_steps = int(t_end / h)
    times = h * np.array(range(num_steps + 1))
    gc.disable()                # don't measure garbage-collection
    x1, cpu_t1, t1 = integrate_explicit(runge_kutta_4, h, num_steps, N)
    del(x1)                     # de-allocate this massive array
    gc.collect()

    # And now implicit
    h2 = 0.1
    tolerance = h**2            # because local truncation error is O(h^2) for Backward Euler
    num_steps2 = int(t_end / h2)
    times2 = h2 * np.array(range(num_steps2 + 1))
    x2, cpu_t2, t2 = integrate_implicit(h2, num_steps2, N, tolerance)
    del(x2)
    gc.enable()
    print("Methods: RK4, Backward Euler")
    print("CPU times: ", cpu_t1, cpu_t2)
    print("Wallclock times: ", t1, t2)
    with open("explicit_implicit_N={}_{}.txt".format(N, int(time())), "w") as f:
        f.writelines(["Method\t CPU time\t Wallclock time\n",
```

```
                    "RK4:\t {:<25}\t\t {}\n".format(cpu_t1, t1),
                    "BE: \t {:<25}\t\t {}\n".format(cpu_t2, t2)])

def plot_methods():
    h = 0.01
    t_end = 10
    num_steps = int(t_end / h)
    times = h * np.array(range(num_steps + 1))
    x1, cpu_t1, t1 = integrate_explicit(forward_euler, h, num_steps, N)
    x2, cpu_t2, t2 = integrate_explicit(runge_kutta_4, h, num_steps, N)

    tolerance = h**2
    x3, cpu_t3, t3 = integrate_implicit(h, num_steps, N, tolerance)

    # suppress Forward Euler divergence
    x1 = suppress_diverge(x1, 2 + N + N)

    plt.figure(figsize=(9, 6))
    plt.plot(times, x1[:, 0], 'b', label="Forward Euler")
    plt.plot(times, x2[:, 0], 'g', label="Runge-Kutta 4")
    plt.plot(times, x3[:, 0], 'r', label="Backward Euler")
    plt.title("Multiscalorithms")
    plt.xlabel("Time")
    plt.ylabel("Position (q)")
    plt.legend()
    # plt.show()
    print("Methods: Forward Euler, RK4, Backward Euler")
    print("CPU times: ", cpu_t1, cpu_t2, cpu_t3)
    print("Wallclock times: ", t1, t2, t3)
    plt.show()
    # plt.savefig("methods_position_comparison", dpi=400)

def plot_potential():
    matplotlib.rc("text", usetex=True)
    l = 7
    x = np.arange(-l, l, 0.01)
    y = [V(i) for i in x]
    plt.plot(x, y)
    plt.title(r"The potential $V(q) = \frac{1}{4} (q^2 - 1)^2$")
    plt.xlabel("$q$")
    plt.ylabel("$V$")
    plt.savefig("potential", dpi=400)

if __name__ == "__main__":
    main()
```

B calc.py

```
import numpy as np
from numpy import inf
from numpy.linalg import solve, norm
from tqdm import trange

# range = trange
```

```
def forward_euler(x, f, h, num_steps):
    for step in range(num_steps):
        x[step + 1] = x[step] + h * f(x[step])
    return x

def runge_kutta_4(x, f, h, num_steps):
    for step in range(num_steps):
        k1 = f(x[step])
        k2 = f(x[step] + h / 2. * k1)
        k3 = f(x[step] + h / 2. * k2)
        k4 = f(x[step] + h*k3)
        x[step + 1] = x[step] + h / 6. * (k1 + 2*k2 + 2*k3 + k4)
    return x

def newton(v, system, jacobian, tolerance):
    s = np.full(len(v), inf)    # initial difference

    while norm(s, ord=inf) > tolerance:
        f = system(v)
        j = jacobian(v)

        s = solve(j, -f)        # solving J(v) s = -f(v) for s
        v = v + s
    return v

def backward_euler(x, f, j, h, num_steps, tolerance):
    for step in range(num_steps):
        system = lambda v: v - h * f(v) - x[step]
        # Initial guess for Newton's method is 1 step Forward Euler
        x0 = x[step] + h * f(x[step])
        x[step + 1] = newton(x0, system, j, tolerance)
    return x
```

C sym.py

```
from numpy import array
from sympy import latex, Matrix, symbols, Sum, Indexed


def jacobian(gamma, h, N):
    V = lambda q: 1/4 * (q**2 - 1)**2
    dV = lambda q: (q**2 - 1) * q
    vect_length = 2 + N + N

    q, p = symbols("q p")
    u = symbols(["u" + str(i) for i in range(1, N+1)])
    v = symbols(["v" + str(i) for i in range(1, N+1)])
    j = symbols("j")

    y = Matrix([q, p, *u, *v])

    f = [p,
         -dV(q) + gamma**2 * Sum(Indexed("u", j) - q, (j, 1, N))]
```

```
    g = [-j**2 * (u[j-1] - q) for j in range(1, N+1)]
    f = Matrix(f + v + g)

    g = y - h*f                 # Backward Euler equation
    jacob = g.jacobian(y)
    return jacob


def generate_jacobian(gamma, h, N):

    jacob = jacobian(gamma, h, N)
    j10 = str(jacob[1, 0])
    jacob[1, 0] = 0
    jacob = jacob.tolist()
    jacob = [[float(x) for x in row] for row in jacob]

    def j(x):
        q = x[0]
        j.m[1, 0] = eval(j10)
        return j.m
    j.m = array(jacob)

    return j
```

References

[1] G. W. Ford, M. Kac, and P. Mazur, "Statistical mechanics of assemblies of coupled oscillators," *Journal of Mathematical Physics*, vol. 6, no. 4, pp. 504–515, 1965.

[2] Martijn van Beest, Jaro Camphuijsen, Rahiel Kasim, "Multiscalorithms." [`https://github.com/sunsistemo/multiscalorithms`; accessed 22-December-2016].