TECHNISCHE UNIVERSITÄT WIEN

Diplomarbeit

Exponential integrators for time-dependent multi-particle Schrödinger equations

zur Erlangung des akademischen Grades
Diplom-Ingenieur

im Rahmen des Studiums
Masterstudium Technische Mathematik (066 394)

eingereicht von
Alexander Josef Grosz
Matrikelnummer 01525490

ausgeführt am Institut für Analysis und Scientific Computing,
Fakultät für Mathematik und Geoinformation,
Technische Universität Wien

Betreuer: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Winfried Auzinger

Wien, März 2020

(Unterschrift Verfasser) (Unterschrift Betreuer)

Abstract

We compare different time stepping methods for the multi-configuration time-dependent Hartree-Fock (MCTDHF) method for the Schrödinger equation. Especially we focus on exponential integrators, where the differential equation is first transformed using the variation-of-constants formula or via the Lawson transformation and then solved numerically.

First we compare the methods on a cubic Schrödinger equation with an exact solution to verify the expected convergence behaviour of our implementation and to numerically compare properties such as the error constant.

Then we use two model problems – a helium atom and a quantum dot that we irradiate by an external potential (laser pulse) – which we discretize using the MCTDHF for further evaluation. On these problems we will additionally evaluate adaptive multi-step methods (again using an exponential transformation) and observe the change in the size of the time step.

We find that although the exponential one-step methods (using either transformation) provide excellent stability results, the Adams-Lawson multi-step methods with a predictor-corrector step are the most efficient methods due to the ability to increase the convergence order arbitrarily at virtually no extra computational cost.

The efficiency is further increased using time step adaptivity, where we observe that the time step prediction reacts to local extrema of the external potential which seem to pose a stability requirement for the methods.

Contents

1. Discretisation **4**
- 1.1. MCTDHF . . . 4
- 1.2. Spectral methods . . . 5
- 1.3. Calculation of the ground state . . . 6

2. Time stepping methods **7**
- 2.1. One-step methods . . . 7
  - 2.1.1. Runge-Kutta methods (RK4) . . . 7
  - 2.1.2. Splitting methods . . . 8
  - 2.1.3. Exponential Runge-Kutta methods . . . 9
  - 2.1.4. Lawson transformation . . . 10
- 2.2. Multi-step methods . . . 11
  - 2.2.1. Predictor-corrector methods and time step adaptivity . . . 12

3. Numerical experiments **13**
- 3.1. Comparison with an explicit reference solution . . . 14
  - 3.1.1. One-step methods . . . 15
  - 3.1.2. Multi-step Methods . . . 20
  - 3.1.3. Conclusion . . . 23
- 3.2. Irradiated helium atom . . . 25
  - 3.2.1. Stability . . . 25
  - 3.2.2. Convergence . . . 29
  - 3.2.3. Adaptive methods . . . 31
- 3.3. Quantum dot . . . 37
  - 3.3.1. Stability . . . 37
  - 3.3.2. Convergence . . . 38
  - 3.3.3. Adaptive methods . . . 38

A. Additional figures **47**
- A.1. Cubic Schrödinger equation . . . 47
- A.2. Helium experiment . . . 55
- A.3. Quantum dot experiment . . . 58

B. Code listings **66**
- B.1. Cubic Schrödinger equation . . . 66
- B.2. Helium experiment . . . 70
- B.3. Quantum dot experiment . . . 79

1. Discretisation

In this chapter, we are going to briefly discuss the underlying problem and its discretisation using the MCTDHF method.

1.1. MCTDHF

First, we present the multiconfiguration time-dependent Hartree-Fock method according to [3] and [8].

We want to solve the time-dependent Schrödinger equation

$$\mathrm{i}\frac{\partial \Psi}{\partial t} = H(t)\Psi = A\Psi + B(t)\Psi,$$

where the wave function $\Psi(q_1, \ldots, q_f, t)$ depends on f electrons with coordinates $q_i = (s_i, \vec{r}_i)$ with spin $s_i = \pm\frac{1}{2}$ and spatial coordinates $\vec{r}_i$ and on time t.

In Hartree methods for the steady Schrödinger equation, the full wave function Ψ is approximated by products of single-electron orbital functions $\varphi_j(q_i)$. The Pauli principle can be used to derive anti-symmetry of the solution regarding the exchange of electron coordinates q_i, q_j. With this restriction our model reduction is referred to as Hartree-Fock method. Note that we have disregarded the spin in the model reduction because the Hamiltonian is independent of it as well. The spin can therefore be set as an initial condition and the number of electrons with a given spin is preserved.

The multiconfiguration Hartree-Fock method uses the following ansatz with N linearly independent orbitals φ_j

$$\Psi_{\mathrm{MCHF}}(q_1, \ldots, q_f) = \frac{1}{\sqrt{f!}} \sum_{j_1=1}^{N} \cdots \sum_{j_f=1}^{N} b_{j_1 \cdots j_f} \varphi_{j_1}(q_1) \cdots \varphi_{j_f}(q_f)$$

where as mentioned above for the electronic Schrödinger equation we can consider the coefficients $b_{j_1 \cdots j_f}$ to be antisymmetric, leaving only $\binom{N}{f}$ independent coefficients for computation. We omit the index and denote the MCTDHF approximation by Ψ.

Now we introduce time-dependency for every orbital function and also the coefficients and therefore write $\varphi_{j_i}(q_i; t)$ as well as $b_{j_1 \cdots j_f}(t)$. Using the Frenkel-Dirac variational principle, we can derive differential equations for the coordinates in the manifold

$$\mathcal{M} = \left\{ \Psi : \Psi(\vec{q}, t) = \frac{1}{\sqrt{f!}} \sum_{j_1=1}^{N} \cdots \sum_{j_f=1}^{N} b_{j_1 \cdots j_f}(t) \varphi_{j_1}(q_1, t) \cdots \varphi_{j_f}(q_f, t) \right\}$$

of our ansatz functions by requiring

$$\left\langle \delta\Psi \left| \mathrm{i}\frac{\partial}{\partial t} - H(t) \right| \Psi \right\rangle = 0 \quad \forall t,$$

where $\delta\Psi$ varies in the tangent space of $\mathcal{M}$. Thus, the residual $\left\|\mathrm{i}\frac{\partial\Psi}{\partial t} - H\Psi\right\|$ is minimized in $\mathcal{M}$.

For uniqueness, we introduce additional constraints

$$\begin{aligned} \langle\varphi_j|\varphi_k\rangle &= \delta_{j,k}, \quad \forall t = 0, \\ \left\langle \varphi_j \left| \frac{\partial\varphi_k}{\partial t} \right\rangle \right. &= -\mathrm{i}\langle\varphi_j|A|\varphi_k\rangle, \end{aligned}$$

which provide orthonormality of the φ_i for all $t > 0$ using that the operator A is self-adjoint and derive the following "working equations" for the motion of the coefficients and the single particle functions:

$$\begin{aligned} \mathrm{i}\frac{\partial b_{j_1\cdots j_f}}{\partial t} &= \sum_{k_1\cdots k_f} \langle\varphi_{j_1}\cdots\varphi_{j_f}|B|\varphi_{k_1}\cdots\varphi_{k_f}\rangle b_{k_1\cdots k_f}, \quad \forall j_1\cdots j_f, \\ \mathrm{i}\frac{\partial\varphi_j}{\partial t} &= A\varphi_j + (I-P)\sum_{k=1}^{N}\sum_{\ell=1}^{N}\rho^{-1}_{j,\ell}\bar{b}_{\ell,k}\varphi_k, \quad j = 1,\ldots,N, \end{aligned}$$

where P is the orthogonal projector onto the space spanned by φ_j and

$$\begin{aligned} \Psi_j &= \langle\varphi_j|\Psi\rangle, \\ \rho_{j,\ell} &= \langle\Psi_j|\Psi_\ell\rangle, \\ \bar{b}_{j,\ell} &= \langle\Psi_j|B|\Psi_\ell\rangle. \end{aligned}$$

It can be assumed that the so-called density matrix $(\rho_{j,\ell})$ is non-singular in practical use.

Note that the anti-symmetry of the solution does not need to be enforced but instead is preserved because of the symmetry of H.

1.2. Spectral methods

Now we are going to discuss how we represent the equations from the previous chapter on a discrete grid (standard method, see e.g. [11]).

We assume a one-dimensional setting and would like to approximate a complex-valued, smooth and periodic function $u(x)$ on an interval $[-L, L]$. We specify n equidistant grid points (where for simplicity, n is chosen as even) on the interval by

$$X = \left(x_j = j \cdot \frac{2L}{n},\ j = -\frac{n}{2},\ldots,\frac{n}{2}-1 \right)$$

and associate a grid function with $u(x)$ by evaluating at these grid points

$$U = \left(U_j = u(x_j),\ j = -\frac{n}{2}, \ldots, \frac{n}{2} - 1\right).$$

We define an orthonormal basis on the complex linear space formed by these grid functions with the inner product

$$\langle U, V \rangle = \frac{1}{n} \sum_{j=-\frac{n}{2}}^{\frac{n}{2}-1} \overline{U}_j V_j$$

yielding

$$\varphi_k(x) = \mathrm{e}^{\frac{\mathrm{i}k\pi x}{L}} \quad k = -\frac{n}{2}, \ldots, \frac{n}{2} - 1.$$

This basis allows a fast transformation (FFT) from the above defined function space to the coefficient space of this basis.

Since this basis consists of eigenvectors for the differentiation operator, we can very efficiently calculate an approximation of the derivative of a function. This allows us to quickly evaluate a matrix function to the discretized Laplace operator, e.g. a matrix exponential, by utilising that it can be represented as a diagonal matrix. This evaluation occurs in the propagation of a differential equation in the MCTDHF.

Note that instead of the usually proposed complex absorbing potential we only use periodic boundary conditions for simplicity. In the numerical experiments, the locality of the solution was observed and the spatial region was chosen accordingly, such as not to run into significant interaction with the boundary.

1.3. Calculation of the ground state

We use an approach explained in [2] (note that the theoretical results there do not apply in our general case, but the method is nevertheless practically used) to calculate the ground state of our problem using an adaptive method and a spectral spatial discretisation.

The imaginary time propagation method (also known as “gradient flow with discrete normalisation”/GFDN) can be thought of as an application of the gradient descent method to the energy of the wave function (minimization problem). After each step of the descent, we enforce the normalisation $\|\Psi\|_2 = 1$ by scaling the resulting vector back to the unit sphere.

The notion “imaginary time” is based on the fact that the equation for the ground state calculation can equivalently be obtained from the underlying Schrödinger equation using the transformation $t \mapsto \mathrm{i}t$.

2. Time stepping methods

In this chapter we are going to discuss the various time stepping methods and approaches which are going to be compared in the experiments. Generally in this chapter, we discuss the solution of a (sufficiently smooth) initial value problem

$$x'(t) = f(t, x), \quad x(t_0) = x_0,$$

where x and f are vector-valued. Note that any presented implicit methods are going to be used without solving the corresponding (non-linear system of) equations. Instead, they are applied in the predictor-corrector setting as explained below.

2.1. One-step methods

The most straight-forward approach to time propagation are one-step methods. E.g. the theory of classical Runge-Kutta methods is usually content of beginner's courses.

As the name implies, in one-step methods we do not use any additional information from previous steps, so in the explanation of the methods we can always consider the first step only. In the convergence analysis (which can be found in the literature), of course, we have to consider the application of multiple iterations of the method to estimate the cumulative error. All presented convergence orders will correspond not to the local (consistency) error, but to the global cumulative error.

Our approximation of the function value at time t can be described as

$$x(t) \approx \Phi^{t,t_0} x_0,$$

where the propagation function Φ depends on the initial value x_0, the times t_0 and t and on the function $f(t, x)$.

2.1.1. Runge-Kutta methods (RK4)

First we will briefly describe general Runge-Kutta methods, where we base our presentation on [4], and then provide the classical Runge-Kutta method of order 4 (RK4) as an example. We write $\tau = t - t_0$. An s-stage Runge-Kutta method is a propagation function given by

$$\Phi^{t+\tau,t} x = x + \tau \Psi(t, x, \tau) = x + \tau \sum_{i=1}^{s} b_i k_i,$$

$$k_i = f(t + c_i \tau, x + \tau \sum_{j=1}^{s} a_{ij} k_j).$$

Because we want to deal with explicit methods only, we restrict ourselves to $a_{ij} = 0$ for $j \geq i$. Usually, the coefficients are collected in vectors b, c and a square matrix $A \in \mathbb{R}^{s \times s}$ and we display the parameters of a method as a so called Butcher tableau, i.e. (with the RK4 array shown as an explicit example)

$$\begin{array}{c|c} c & A \\ \hline & b^T \end{array} = \begin{array}{c|cccc} c_1 & 0 & & & \\ c_2 & a_{21} & 0 & & \\ \vdots & \vdots & \ddots & \ddots & \\ c_s & a_{s1} & a_{s2} & \cdots & 0 \\ \hline & b_1 & b_2 & \cdots & b_s \end{array} = \begin{array}{c|cccc} 0 & & & & \\ 1/2 & 1/2 & & & \\ 1/2 & 0 & 1/2 & & \\ 1 & 0 & 0 & 1 & \\ \hline & 1/6 & 1/3 & 1/3 & 1/6 \end{array}$$

For the classical RK4 method, we therefore specify the full propagation function as

$$\begin{aligned} k_1 &= f(t_0, x_0) \\ k_2 &= f\left(t_0 + \frac{\tau}{2}, x_0 + \frac{\tau}{2} k_1\right) \\ k_3 &= f\left(t_0 + \frac{\tau}{2}, x_0 + \frac{\tau}{2} k_2\right) \\ k_4 &= f\left(t_0 + \tau, x_0 + \tau k_3\right) \\ \Psi^{t,t_0} x_0 &= x_0 + \tau \left(\frac{k_1}{6} + \frac{k_2}{3} + \frac{k_3}{3} + \frac{k_4}{6}\right). \end{aligned}$$

The convergence properties of Runge-Kutta methods can be proved for sufficiently smooth problems by Taylor expansion and a stability argument to control the propagation of the step-wise errors, as explained in [4].

The RK4 method is a 4-stage method $(k_1, \ldots, k_4)$ and therefore needs 4 evaluations of f in each time step. By Taylor expansion it can be seen that this method is optimal in the sense that there cannot be a 4th order Runge-Kutta method using fewer stages. Statements of this type exist generalised for s-stage methods.

2.1.2. Splitting methods

As a reference for splitting methods, we cite [10]. Splitting methods are applicable to partitioned systems, where we can *split* f into a sum of operators:

$$x'(t) = f(t, x) = A(x) + B(x, t).$$

We can additionally also assume B as time-independent if we autonomise the equation: Adding $s(t) = t$ with $s'(t) = 1$ to the system of equations as a replacement for the time variable, we achieve

$$\begin{pmatrix} x(t) \\ s(t) \end{pmatrix}' = \begin{pmatrix} A(x) \\ 1 \end{pmatrix} + \begin{pmatrix} B(s, x) \\ 0 \end{pmatrix}.$$

We now construct an integrator by combining intermediate integrators for the summands A, B according to the following (generally non-symmetric) scheme

$$e^{\tau(A+B)} = e^{a_m \tau A} e^{b_m \tau B} \cdots e^{a_1 \tau A} e^{b_1 \tau B} e^{a_0 \tau A} e^{b_0 \tau B}.$$

The integration is thereby realised by propagating the ODE using only A or B and advancing in time by $a_k\tau$ or $b_k\tau$, choosing the coefficients such as to achieve the desired order of convergence. We will specifically use two methods, proposed by Yoshida (4 stage) and Suzuki (5 stage), where both are 4th order methods constructed by composition (see [5]).

The use of splitting methods is advantageous where A and B themselves can be (more) easily propagated than $A+B$. In our case, we will be able to integrate A easily and the propagation of the B-part will be more challenging. In our experiments, we will compare the use of different one-step methods for the propagation of the B-part in order to see which ones are suitable to lead to splitting methods that show the desired order.

2.1.3. Exponential Runge-Kutta methods

We present this class of methods following [6]. Let us assume an autonomous problem $x' = f(x)$. First we linearise the equation at w and consider

$$v'(t) + Av(t) = g(v(t)), \quad v(0) = v_0 := x_0 - w$$

with $A = -Df(w)$ (note the sign of this operator because we moved the linear term to the left-hand side) and $v(t) = x(t) - w$.

We rewrite the reformulation using a variation-of-constants formula as

$$v(t) = \mathrm{e}^{-tA}v_0 + \int_0^t \mathrm{e}^{-(t-\tau)A}g(v(\tau))\ d\tau$$

with $A = -Df(0)$: Differentiation with respect to t yields

$$\begin{aligned} v'(t) &= -A\mathrm{e}^{-tA}v_0 - A\mathrm{e}^{-tA}\int_0^t \mathrm{e}^{-(t-\tau)A}g(v(\tau))\ d\tau + \mathrm{e}^{-(t-t)A}g(v(t)) \\ &= -Av(t) + g(v(t)) \end{aligned}$$

and thus the linearised equation above. Exponential methods use this representation to calculate a solution by approximating the value of the integral.

We write for the general case

$$v(t_0+\tau) = \mathrm{e}^{-\tau A}v(t_0) + \int_0^\tau \mathrm{e}^{-(\tau-\delta)A}g(t_0+\delta, v(t_0+\delta))\ d\delta.$$

As the value of g depends on the unknown function v, we can try to approximate it in a similar manner as we have seen with RK4. An s-step exponential Runge-Kutta method is defined by

$$\begin{aligned} v_1 &= \chi(-\tau A)v_0 + \tau\sum_{i=1}^{s} b_i(-\tau A)G_i, \\ G_j &= g(t_0 + c_j\tau, U_j) \\ U_i &= \chi_i(-\tau A)v_0 + \tau\sum_{j=1}^{s} a_{ij}(-\tau A)G_j, \end{aligned}$$

where the coefficient (functions) χ, χ_i, a_{ij} and b_i are exponential functions (or approximations), usually

$$\chi(z) = \mathrm{e}^z, \quad \chi_i(z) = \mathrm{e}^{c_i z}.$$

Considering the (formal) limit $A \to 0$ we can call the Runge-Kutta method defined by $b_i = b_i(0)$, $a_{ij} = a_{ij}(0)$ and $\chi(0) = 1$ the *underlying Runge-Kutta method* of an exponential Runge-Kutta method.

Again, we can write the coefficient (functions) as a butcher tableau and consider explicit methods only:

$$\begin{array}{c|cccc|c}
c_1 & 0 & & & & \chi_1(-\tau A) \\
c_2 & a_{21}(-\tau A) & 0 & & & \chi_2(-\tau A) \\
\vdots & \vdots & \ddots & \ddots & & \vdots \\
c_s & a_{s1}(-\tau A) & a_{s2}(-\tau A) & \cdots & 0 & \chi_s(-\tau A) \\
\hline
 & b_1(-\tau A) & b_2(-\tau A) & \cdots & b_s(-\tau A) & \chi(-\tau A)
\end{array}$$

In the numerical experiments, we will use the method proposed by Krogstad (and we have observed that other proposed 4th order exponential Runge-Kutta methods perform very similarly), given by (function arguments have been omitted)

$$\begin{array}{c|cccc}
0 & 0 & & & \\
\frac{1}{2} & \frac{1}{2}\varphi_{1,2} & 0 & & \\
\frac{1}{2} & \frac{1}{2}\varphi_{1,3} - \varphi_{2,3} & \varphi_{2,3} & 0 & \\
1 & \varphi_{1,4} - 2\varphi_{2,4} & 0 & 2\varphi_{2,4} & 0 \\
\hline
 & \varphi_1 - 3\varphi_2 + 4\varphi_3 & 2\varphi_2 - 4\varphi_3 & 2\varphi_2 - 4\varphi_3 & -\varphi_2 + 4\varphi_3
\end{array},$$

where

$$\varphi_k(z) = \int_0^1 \mathrm{e}^{(1-\theta)z} \frac{\theta^{k-1}}{(k-1)!}\, d\theta, \quad \varphi_{k,j} = \varphi_k(-c_j \tau A).$$

The functions φ_k provide a basis for the functions used with exponential quadrature rules derived from Lagrange interpolation polynomials.

2.1.4. Lawson transformation

Lawson proposed in [9] a transformation for the ODE targeted at the solution of stiff problems with large Lipschitz constants. We define $z(t) = \exp(-tA)x(t)$ which yields

$$\begin{aligned}
z'(t) &= \exp(-tA)x'(t) - A\exp(-tA)x(t) \\
&= \exp(-tA)\left(f(t, \exp(tA)z(t)) - A\exp(tA)z(t)\right) = g(t,z), \\
z(0) &= x_0.
\end{aligned}$$

We want to choose the matrix A such that the Jacobian

$$\frac{\partial g}{\partial z} = e^{-tA}\frac{\partial f}{\partial x}e^{tA} - A = e^{-tA}\left(\frac{\partial f}{\partial x} - A\right)e^{tA}$$

has small eigenvalues (where we used that A commutes with e^{tA}) in order to achieve a less demanding problem for methods, which do not deal well with stiff problems e.g. RK4.

If we consider a splitting approach as in Section 2.1.2 and additionally assume A to be a constant matrix, i.e.

$$x'(t) = f(x) = Ax + B(x)$$

the resulting equation is

$$z(t) = e^{-tA}B(e^{tA}z(t)) = g(t,z), \quad z(0) = x_0.$$

2.2. Multi-step methods

Again, we refer to standard literature [4] for the basic concepts. As the name already indicates, we now not only use a single previous step to calculate the next function value, but rather rely on multiple previous values. We define a k-step linear multi-step method by a recursion

$$\begin{aligned}\alpha_k x_\tau(t_{j+k}) + \alpha_{k-1}x_\tau(t_{j+k-1}) + \cdots + \alpha_0 x_\tau(t_j) \\ = \tau\left(\beta_k f_\tau(t_{j+k}) + \beta_{k-1}f_\tau(t_{j+k-1}) + \cdots + \beta_0 f_\tau(t_j)\right),\end{aligned}$$

where we assume an equidistant grid $t_0 < t_1 < \cdots$ with $t_{i+1} - t_i = \tau$. The function x_τ approximates the solution and f_τ approximates the value of f at a given point, e.g. $f_\tau(t_i) = f(t_i, x_\tau(t_i))$.

For explicit methods, we set $\beta_k = 0$, $\alpha_k = 1$ w.l.o.g. and thereby get

$$\begin{aligned}x_\tau(t_{j+k}) = &-\left(\alpha_{k-1}x_\tau(t_{j+k-1}) + \cdots + \alpha_0 x_\tau(t_j)\right) \\ &+ \tau\left(\beta_{k-1}f_\tau(t_{j+k-1}) + \cdots + \beta_0 f_\tau(t_j)\right).\end{aligned}$$

Moving one step ahead to $x_\tau(t_{j+k+1})$, we recognize that we have already used (and therefore calculated) the values for $f_\tau(t_{j+1}), \ldots, f_\tau(t_{j+k-1})$. Thus, in each step, we only have to evaluate f once, i.e.

$$f_\tau(t_{j+k}) = f(t_{j+k}, x_\tau(t_{j+k})).$$

As we can construct high order methods with increasing step numbers, we expect higher efficiency of multi-step methods in comparison to one-step methods where the evaluation of f is expensive.

In explicit exponential multistep integrators, the integrand in the variation-of-constants formula is interpolated at the known backward points and this is used for approximating the integral. Analogously the implicit case. Lawson-multistep integrators are based on applying conventional multistep methods after Lawson transformation.

2.2.1. Predictor-corrector methods and time step adaptivity

The predictor-corrector schemes combine explicit and implicit methods. The latter often provide better stability behaviour at the cost of the solution of a usually non-linear system of equations. Predictor-corrector methods try to incorporate implicit methods without the expensive solution of such a system:

1. Calculate the function value $\tilde{x}_n$ at the next time step t_n using an explicit method.
2. In order to solve the equations for the implicit method to calculate x_n, we now use the value $\tilde{x}_n$ as a starting value and use only a few steps of an iterative solution method (fixed-point iteration) to approximate the solution.

In our case, we will use only one iteration for the implicit method and thereby increase the number of f evaluations by 1 (thus doubling the amount).

Additionally to having benefits of an implicit method, we now also have two approximations of our solution, $\tilde{x}_n$ and x_n. If we can expect the solution approximation $\tilde{x}_n$ to be of order k and x_n of order $k+1$, we can estimate the error in the solution. This error estimate can then also be used to suggest a change in step size.

This step size adaptivity comes at the cost of more involved implementation, as we need to transform the coefficients α, β to fit the current configuration $t_j, \ldots, t_{j+k}$. However, for certain problems, the step size necessary to achieve precise results changes over time; using an adaptive method, we can increase the step size locally and thereby reduce computation time.

3. Numerical experiments

In this chapter we present the numerical results for several experiments comparing different time stepping methods as presented in the previous chapter regarding

- convergence order and error constant for the error measured in discrete L^2-norm,
- error to a reference solution compared to the size of the time step/number of steps on a given interval,
- error to a reference solution compared to the number of B calls, which is the most expensive part of the calculation and therefore representative of the total computation time up to a factor and
- stability of a given ground state without any external potential.

Additionally, we evaluate adaptive methods regarding

- how the precision of the solution corresponds to the prescribed tolerance parameter and
- the change of the automatically chosen step size over time.

All the experiments will be in a 1D setting, where x is the space and t the time variable. The MCTDHF parameters are N, the number of orbitals, f, the number of electrons and n_x, the number of space points in the interval I. For equidistant methods we use the denotation dt for the step size in the interval $[t_0, t_1]$.

The experiments used the "Time splitting spectral methods" package for Julia [7].

Due to the large number of figures in this chapter, many have been moved to the appendix to improve readability.

3.1. Comparison with an explicit reference solution

First of all, we compare all the methods in our scope when applied to a nonlinear Schrödinger equation, which is not associated with the MCTDHF approximation. For the cubic Schrödinger equation

$$i\frac{\partial}{\partial t}\psi(x,t) = -\frac{1}{2}\Delta\psi(x,t) + V_\alpha(x,t)\psi(x,t) - \alpha\left|\psi(x,t)\right|^2\psi(x,t), \quad \psi(x,0) = \psi_0(x)$$

we have an explicit solution (with the corresponding initial value at $t = 0$) given by

$$\psi_{\text{ex}}(x,t) = \frac{2\exp(i(\frac{3}{2}t - x))}{\cosh(2(t+x))},$$

where the time dependent potential is

$$V_\alpha(x,t) = -(1-\alpha)\left|\psi_{\text{ex}}(x,t)\right|^2,$$

and we set $\alpha = \frac{1}{2}$. The spatial discretization is based on $n_x = 2^{12}$ points in the interval $I = [-64, 64]$ using spectral methods (see Chapter 1.2) and periodic boundary conditions are imposed.

The step size dt is generally chosen as $\lfloor 2^6 \cdot 1.3^k \rfloor^{-1}$ with $k \in \{0, \dots, 10\}$ and all of the methods in this chapter use a constant step size on an interval $[0, t_1]$. For these step sizes we observe convergence of most methods and thus a systematic and representative behaviour is expected. t_1 is varied from $1, 2, \dots, 10$ in general and fixed at $t_1 = 1$ if only one value of t_1 is being observed.

However, for the classical RK4 method, only much smaller step sizes of $dt \approx 5.6 \cdot 10^{-4}$ yield a converging result because of stability requirements as can be observed in Figure A.1.1. Note that the smallest step size for the other methods is $dt \approx 1.13 \cdot 10^{-3}$. This "convergence" interval only depicts the transition to stable step sizes. The result is then already very precise, so we cannot directly analyse convergence parameters for this method.

Also we do not analyse the well known Strang splitting in detail, as it will be obvious that the lower order makes this method uncompetitive.

Derivation of the explicit solution with time dependent potential An explicit solution for the equation

$$i\frac{\partial w}{\partial t} = -\frac{\partial^2 w}{\partial x^2} - |w|^2 w$$

is given in [1] as

$$w(x,t) = \pm A\sqrt{2}\frac{\exp\left(i(Bx + (A^2 - B^2)t)\right)}{\cosh(Ax - 2ABt)}$$

for arbitrary real constants A, B.

Using the transformation $\psi(x,t) = w\left(x\sqrt{2}, t\right)$ results in a solution for

$$i\frac{\partial \psi}{\partial t} = -\frac{1}{2}\frac{\partial^2 \psi}{\partial x^2} - |\psi|^2 \psi,$$

where in our case we set $A = \sqrt{2}$ and $B = -\frac{1}{\sqrt{2}}$. We can additionally split the cubic part in the equation by a parameter α into

$$\mathrm{i}\frac{\partial\psi}{\partial t} = -\frac{1}{2}\frac{\partial^2\psi}{\partial x^2} - |\psi|^2\psi = -\frac{1}{2}\frac{\partial^2\psi}{\partial x^2} - (1-\alpha)|\psi|^2\psi - \alpha|\psi|^2\psi$$

and then substitute one of the $|\psi|^2$ terms by the explicit solution itself to construct a test problem with a truly time dependent potential and known solution.

3.1.1. One-step methods

3.1.1.1. Comparison of different B-part propagation methods within splitting methods

The Suzuki and Yoshida splitting methods have been implemented using different methods for the propagation of the B-part. We denote a reference method by "Suzuki" which propagates B exactly as explained below. For the other methods we specify the propagation method as "RK4" where the classical Runge-Kutta-method of order 4 is used and "MP" where an "implicit" mid-point rule is used. The mid-point rule has been implemented as an explicit method using fixed-point iteration with different numbers of iteration steps (e.g. "MP3" indicates 3 iteration steps).

We can conclude from Figure 3.1, which shows the convergence of all methods for $dt \to 0$ at $t_1 = 1$, that we should use exactly 3 iterations for the mid-point rule, as we cannot realize a 4th order method with only 2 iterations, but observe less numerical precision with 4 iterations in spite of higher computational cost. Furthermore, using the lower order Heun (RK2) method is also not viable because its insufficient convergence order also leads to insufficient convergence behaviour of the full splitting method.

Regarding the number of B calls, both methods, RK4 and the mid-point rule with 3 iterations, have exactly the same number of right-hand side evaluations and therefore approximately equal computational cost (regarding our targeted usage in MCTDHF).

The results for the Yoshida splitting method are shown in Figure 3.1 below and are very similar to Suzuki splitting. All the mentioned observations apply to this method as well.

Exact propagation of B-part By combining splitting of the differential equation with autonomisation by adding the equation $s(t) = t$, after spatial discretization we obtain

$$\mathrm{i}\frac{\partial}{\partial t}\begin{pmatrix} u(t) \\ s(t) \end{pmatrix} = \begin{pmatrix} Au \\ 1 \end{pmatrix} + \begin{pmatrix} B(s,u) \\ 0 \end{pmatrix},$$

where the new A-part (the left summand) is still a linear ODE (discrete Laplacian in the first and trivial in the second component) and the new B-part (right summand) is fixed in time at each propagation step. In this example, the B-part is

$$\mathrm{i}\frac{\partial u}{\partial t} = \left((1-\alpha)\psi_{\mathrm{ex}}(s) - \alpha|u|^2\right)u,$$

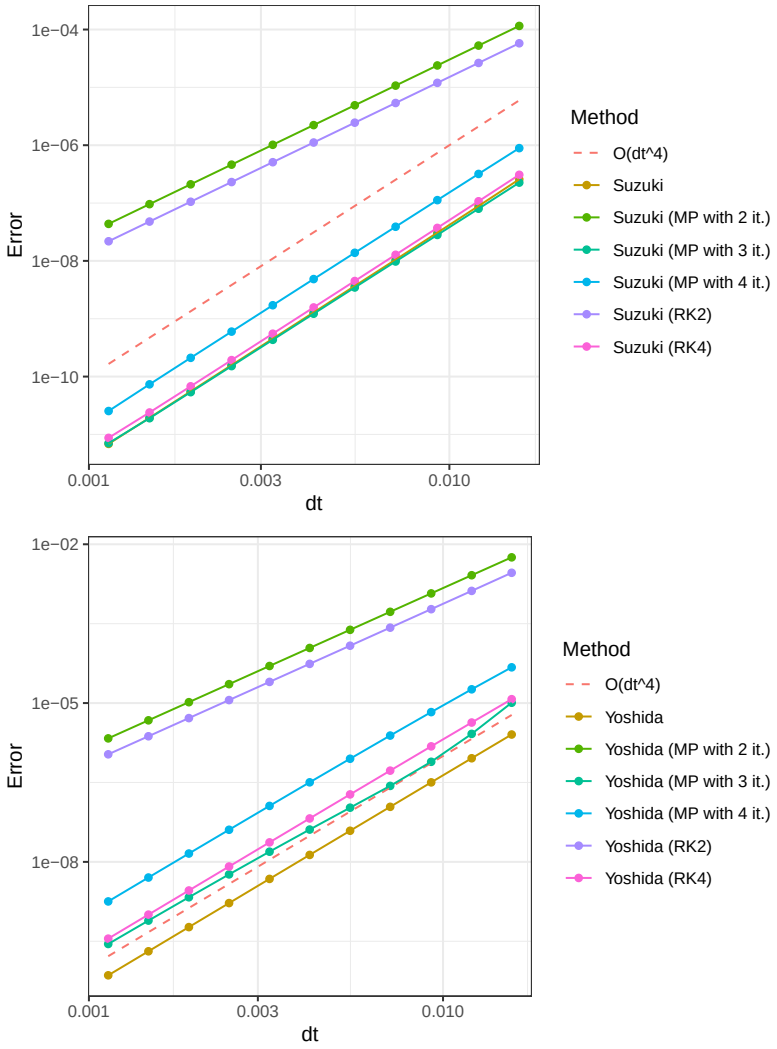

Figure 3.1.: Convergence behaviour of different Suzuki (above) and Yoshida (below) splitting method implementations (regarding the B-part) at $t_1 = 1$ with reference line $\mathcal{O}(dt^4) = 100 \cdot dt^4$ for the cubic Schrödinger equation. We can observe that RK2 or the mid-point rule with 2 iterations are not sufficient to achieve the targeted order of 4. For 4 iterations, we achieve the order but observe a loss of precision compared to 3 iterations.

which results in an ODE at each space point. Therefore, the system now consists of independent scalar equations. We can immediately solve this equation for each component with

$$u(t) = \exp\left(\mathrm{i}t\left((1-\alpha)\psi_{\mathrm{ex}}(s) - \alpha\left|u(0)\right|^2\right)\right)$$

by using

$$\begin{aligned}\frac{\partial u}{\partial t}\overline{u} &= (-\mathrm{i})\left((1-\alpha)\psi_{\mathrm{ex}}(s) - \alpha\left|u\right|^2\right)u\overline{u} \\ &= -\overline{(-\mathrm{i})\left((1-\alpha)\psi_{\mathrm{ex}}(s) - \alpha\left|u\right|^2\right)u\overline{u}} = -u\frac{\partial \overline{u}}{\partial t},\end{aligned}$$

which leads to

$$\frac{\partial}{\partial t}\left|u\right|^2 = \frac{\partial u}{\partial t}\overline{u} + u\frac{\partial \overline{u}}{\partial t} = 0,$$

thus $|u|^2 = |u(0)|^2$ is constant. Therefore we have an explicit solution for each B-part propagation.

3.1.1.2. Convergence Order and Error Constant of Splitting Methods

In this section we discuss the calculated convergence order and error constant from the error results for each dt by using the error of the next larger dt value: For time step sizes dt_1, dt_2 and corresponding errors e_1, e_2 we can calculate the numerical convergence properties by

$$\text{Order} = \frac{\log(\frac{e_1}{e_2})}{\log(\frac{dt_1}{dt_2})}, \qquad \text{Constant} = \frac{e_1}{dt_1^{\text{Order}}}.$$

In all plots displaying an experimentally evaluated error constant, the value is only represented if the corresponding convergence order is in the interval $[3.5, 4.5]$ for 4th order methods and similarly for methods of different orders.

B-part with RK4 In Figure A.1.2 there are several results shown for Suzuki and Yoshida methods using RK4 for the propagation of the B-part. We can observe that both methods achieve the expected convergence order of 4. For higher values of t_1, we observe a slight decrease in convergence order for intermediate time step sizes and an increase in the error constant, which is of magnitude ≈ 10 for Suzuki splitting and between 200 to slightly under 1250 for Yoshida splitting varying strongly.

However, for the smallest time step sizes we observe a drastic reduction in convergence order of the Suzuki method for increasing t_1. At such precise accuracy to the reference solution, this may be attributed to the accumulated numerical noise that occurs for the increasing number of operations during the integration.

B-part with 3 iteration mid-point rule Again, Figure A.1.3 shows the results for the Suzuki and Yoshida methods using the midpoint-rule with 3 iterations for the B-part. We can observe in the first pair of plots that the change of error over time is not as predictable as with RK4 methods. There are two distinguishable intervals in the time

scale, where in the later part (until the end of the observation) the error seems to grow much faster than in the earlier interval. We notice a shift of the time value that splits these intervals to later times for decreasing stepsizes.

The observed convergence orders are as expected for $t_1 = 1$, however for later times, the convergence is rather unsystematic. For this reason, we cannot properly interpret the change of error constant over time.

3.1.1.3. Exponential Runge-Kutta

As stated above, the different proposed fourth order exponential Runge-Kutta methods show a very similar behaviour. Therefore, we examine the Krogstad method in detail. Figure A.1.4 reflects very stable behaviour for the convergence order and error constant for decreasing dt. However, regarding larger time intervals, the error constant increases faster than linearly in t_1, which starts with small constants for $t_1 = 1$ of ≈ 20 which increase to ≈ 500 (however, the convergence behaviour for larger t_1 is increasingly unsystematic for large step sizes).

3.1.1.4. Lawson Runge-Kutta

Regarding our implementation of the Lawson Runge-Kutta method, the convergence order is very stable and the error constant increases linearly for increasing t_1 from 50 to 500 as can be seen in A.1.4. This smaller proportional increase of the error constant is an indicator that this method may be more viable for long-term integration problems than the Krogstad method.

3.1.1.5. Efficiency Comparison by B-calls for One-step Methods

As stated before, in an MCTDHF computation the computation time is approximately proportional to the number of B evaluations. We can therefore compare the precision of the different methods according to this number. Evidently, the number of B-calls in a single step is fixed for each method. We now use only the data for $t_1 = 1$ from the experiment.

We can observe in Figure 3.2, that the popular Strang splitting method (not analysed above) is not viable in our investigated case, because it simply does not provide an order high enough to be efficient for the targeted precisions. We also recognize that the number of B-calls in Suzuki and Yoshida splitting methods is very high due to the high number of evaluations in each B-part. Recall the results from Section 3.1.1.1 that we cannot use a method with fewer stages because of the order reduction.

From our experiments so far, it appears that we should prefer the Lawson Runge-Kutta method for long-time integrations, where the non-linear change of Krogstad's error constant might be a pitfall. This might even be an indicator for stability issues, as we are going to demonstrate in further experiments.

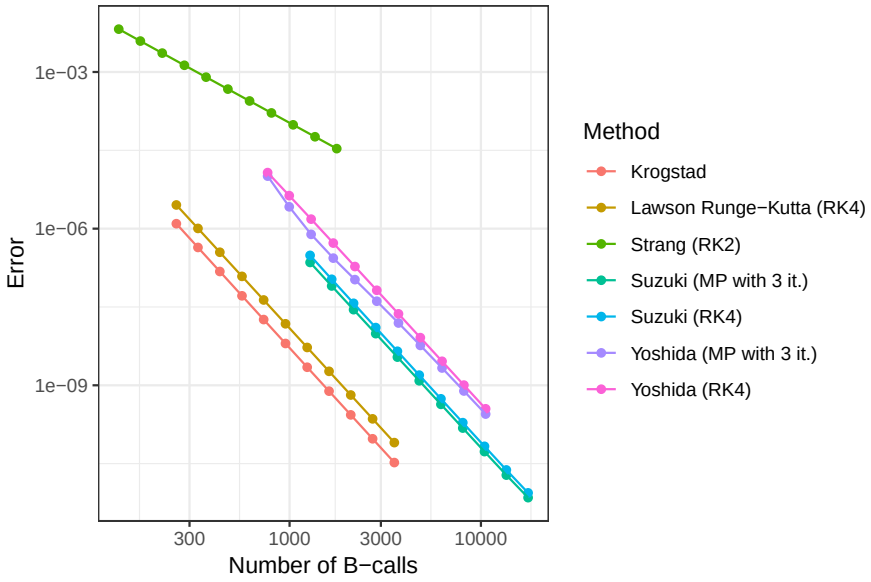

Figure 3.2.: Comparison of viable one-step methods by number of B-calls for the cubic Schrödinger equation. The splitting methods are computationally more expensive than the Krogstad and Lawson Runge-Kutta methods.

3.1.2. Multi-step Methods

3.1.2.1. Comparison of Parameter Choices for Exponential Multi-step Methods and Adams-Lawson Methods

In Figures 3.3 and 3.4 we display the convergence of exponential multi-step and Adams-Lawson methods for different numbers of steps with and without an additional corrector (P/C) step at $t_1 = 1$. We want to specifically compare the P/C methods of a certain step number to the conventional method using an additional step, where we expect the same convergence order for both of these methods (e.g. we expect that the Adams-Lawson method with 3 steps using P/C and the Adams-Lawson method with 4 steps without P/C achieve order 4).

We observe that for exponential multi-step methods the P/C method is slightly more precise than the corresponding method with an additional step. In contrast, for Adams-Lawson methods the P/C method is of substantially higher precision. The expected convergence orders seem to be achieved quite precisely for both methods and all parameter choices.

From now on, we are going to restrict our analysis to methods of orders 4 and 6.

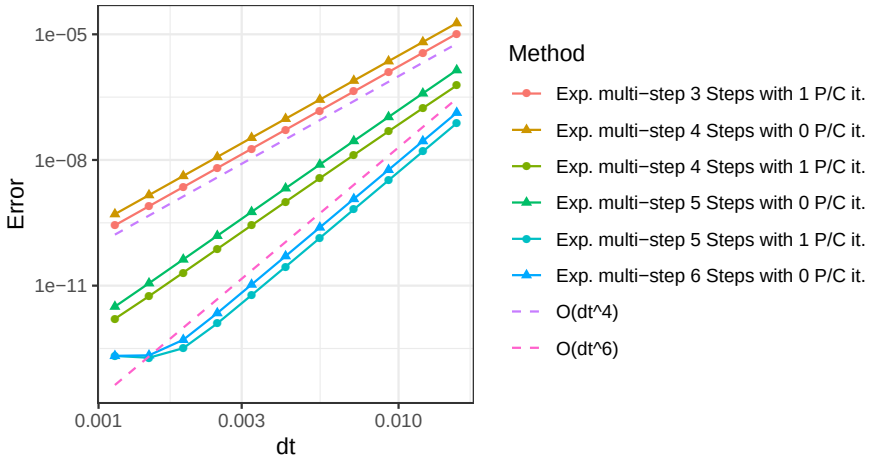

Figure 3.3.: Comparison of exponential multi-step methods with different parameters for the cubic Schrödinger equation. $\mathcal{O}(dt^4) = 100 \cdot dt^4$, $\mathcal{O}(dt^6) = 2 \cdot 10^4 \cdot dt^6$. The predicted orders are achieved very well; the higher order methods achieve higher precision in the observed interval.

3.1.2.2. Convergence Order and Error Constant

Adams-Lawson Methods We can observe the convergence order of 4 for both 4th order Adams-Lawson methods in Figure A.1.5 with slightly increasing order for larger t_1. The error constant is very large for both, the P/C method and the by one step

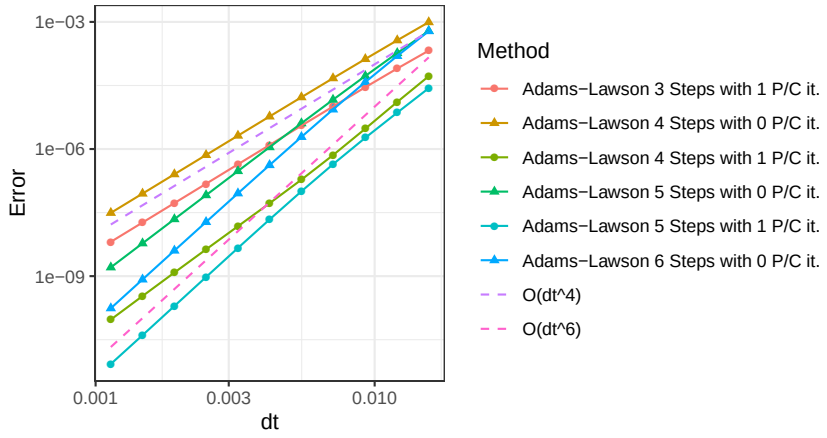

Figure 3.4.: Comparison of Adams-Lawson methods with different parameters for the cubic Schrödinger equation. $\mathcal{O}(dt^4) = 10^4 \cdot dt^4$, $\mathcal{O}(dt^6) = 10^7 \cdot dt^6$. The predicted orders are achieved very well; a significant increase in the error constant for high order methods can already be inferred.

elongated method, and as we have concluded before the P/C method achieves higher precision. Also the growth of the error constant for longer time intervals is comparable.

The results for the 6th order methods can be seen in Figure A.1.6 and are qualitatively similar to those of the 4th order methods.

Exponential Multi-step Methods We can observe the results for the 4th order exponential multi-step methods in Figure A.1.7 and for 6th order methods in Figure A.1.8. Again we observe a slight increase of the convergence order of 4th order methods for larger t_1. Compared to the Adams-Lawson methods, the error constant is considerably smaller. For small step sizes, the precision for the 6th order methods does not improve further as we reach the limit of calculations without dominant numerical noise.

3.1.2.3. Efficiency Comparison by B-calls for Multi-step Methods

Now we compare the multi-step methods by their efficiency regarding the number of B-calls in Figure 3.5. The previously observed increase of precision by using an additional P/C step comes at a cost: because the additional step in this case doubles the number of B-calls in every step, they are less efficient than the respective method of the same order without P/C. For Adams-Lawson methods, the advantage of the higher order of longer methods is less pronounced because of their fast growing error constant.

These observations are as expected: the use of a P/C method has been chosen to increase stability of the methods and most importantly as a time step size estimator for adaptive methods; both considerations do not apply in this experiment.

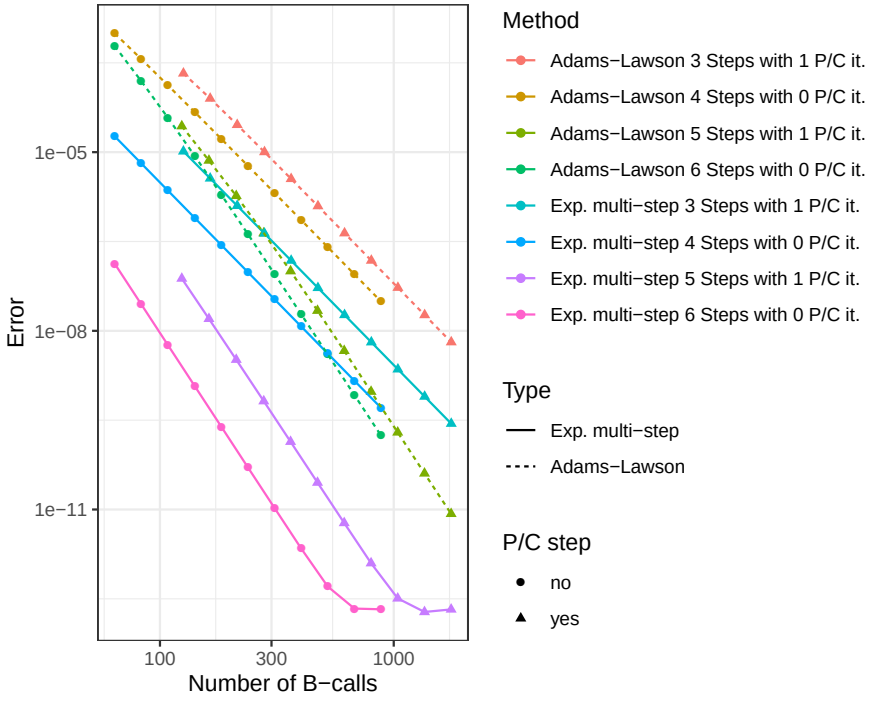

Figure 3.5.: Comparison of different multi-step methods of orders 4 and 6 by number of B-calls for the cubic Schrödinger equation. We can conclude that methods of higher order are preferable and exponential multi-step methods are more efficient.

3.1.3. Conclusion

Based on our analysis in this experiment, we have been able to choose a number of iteration steps when using the mid-point rule for the B-part in splitting methods and have observed that the necessity of high order methods in each B-part of a splitting method results in an inefficient method compared to e.g. an exponential or Lawson Runge-Kutta method.

Even more efficient regarding the number of B-calls, however, are multi-step methods. Because we can choose the order by the number of steps we use for the computation, which does not rely on further B-calls, we can achieve much cheaper methods. These results have been summarized in Figure 3.6, comparing the precision of both one-step and multi-step methods by number of B-calls.

So far, stability considerations have only eliminated the classical RK4 method and did not pose limitations on the other methods. Therefore, we are next going to experiment on a problem with higher stability requirements by applying the time propagation to the MCTDHF-method.

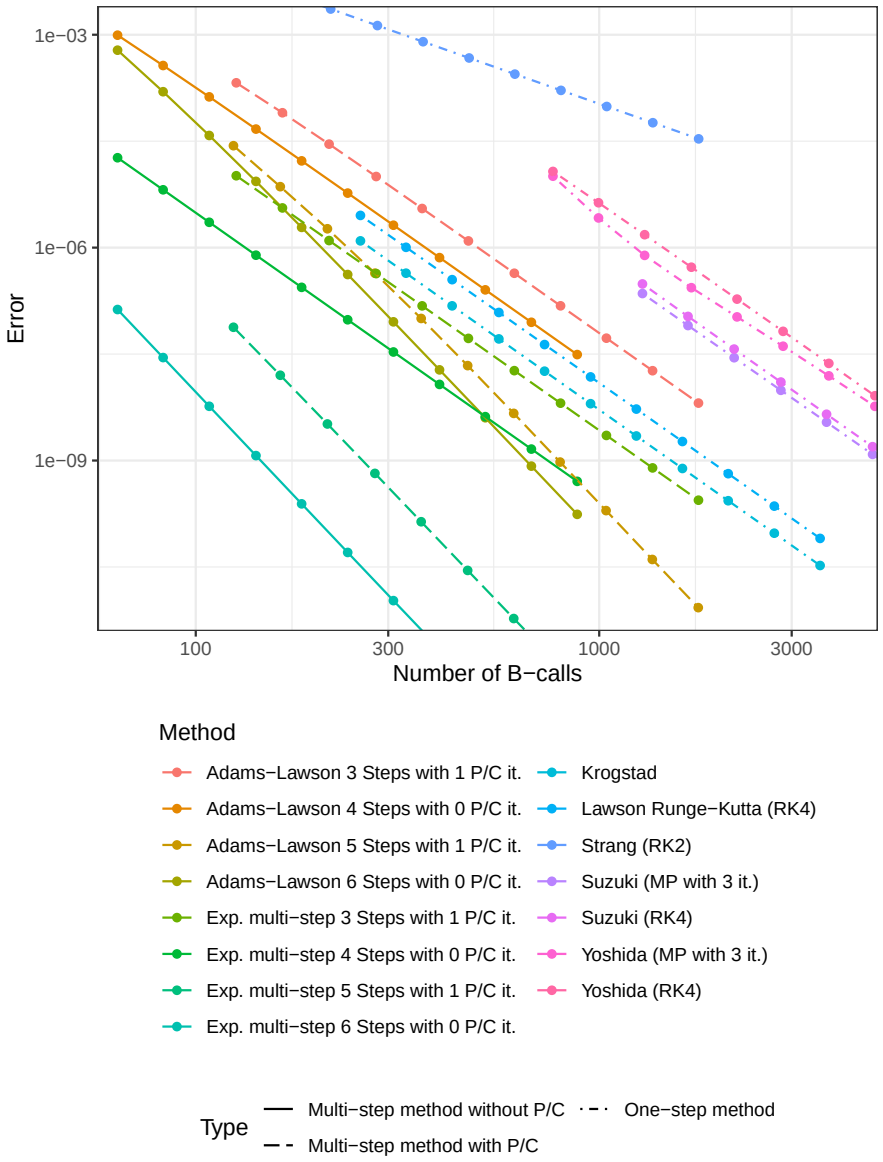

Figure 3.6.: Comparison of all (viable) methods by number of B-calls for the cubic Schrödinger equation. Krogstad and Lawson Runge-Kutta are one-step methods that can compete with multi-step methods. The high order at low cost leads to efficient high precision methods.

3.2. Irradiated helium atom

In this section, we now compare different methods using the MCTDHF-method on a 1D helium atom model as described in [12]. The Hamiltonian in our formulation is now

$$H(x,y;t) = -\frac{1}{2}\Delta - \frac{2}{\sqrt{x^2+b^2}} - \frac{2}{\sqrt{y^2+b^2}} + \frac{1}{\sqrt{(x-y)^2+b^2}} + (x+y)\mathcal{E}(t),$$

where we choose $b = 0.7408$ and $\mathcal{E}(t)$ is an external laser field, by which the helium atom is irradiated. For $\mathcal{E}(t)$, use a frequency $\omega = 0.1837$ and set

$$\mathcal{E}(t) = 0.1894 \cdot \sin(\omega t) \cdot 1.2 \cdot \exp\left(-\frac{(t - 6\frac{\pi}{\omega})^2}{2000}\right),$$

where we modulate the oscillation by a Gaussian envelope to simulate only a short laser burst while still conserving smoothness of the resulting problem in a mathematical sense, in order not to interfere with our (theoretical) convergence results, as opposed to e.g. modulating using a trapezoid curve.

A resulting energy curve of the full experiment (until after the laser burst has ended) can be seen in Figure 3.7, which has been calculated using an adaptive Adams-Lawson method (see Section 3.2.3.2).

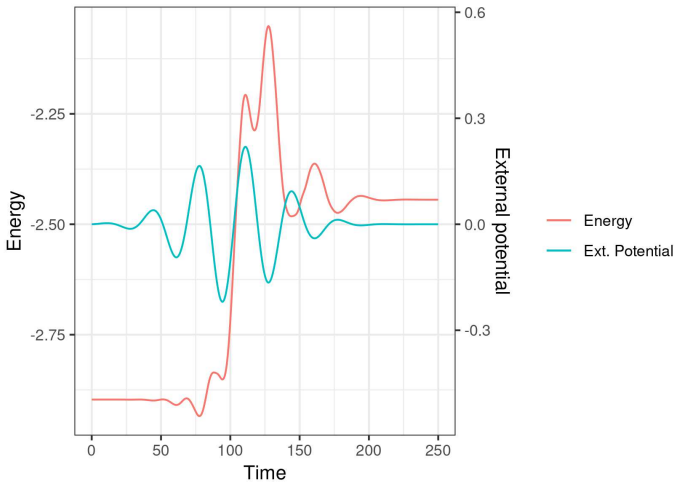

Figure 3.7.: Visualisation of the energy change in the helium experiment.

3.2.1. Stability

In this section, we do not use the external laser field and instead investigate the behaviour of the ground state propagated over time. We draw conclusions about the

stability of the methods by observing the deviation of the norm of the solution which should be conserved as $\|\psi(.,t)\|_{L^2} = 1 \; \forall t$.

The discretization uses $n_x = 512$ points on $[-16, 16]$ and we choose our time steps by calculating $2000, 3000, \ldots, 12000$ steps on $[0, 4\frac{\pi}{\omega}]$. First we choose $N = 6$, the results for $N = 4$ are presented below. For RK4 and the 6-step exponential multi-step methods without P/C all of these step sizes did not yield any useful results.

Results for N = 6 We can observe the stability behaviour in Figures 3.8 and 3.9. Adams-Lawson, Lawson Runge-Kutta and all of the splitting methods show a very stable behaviour, where specifically the Suzuki splitting is of very high precision. Amongst the splitting methods, using RK4 for the B-part results in better stability compared to using the midpoint rule with 3 iterations. Strang splitting is stable, however because of its lower precision the change in the norm is more pronounced. The 6th order Adams-Lawson method without P/C shows unstable behaviour for a very low number of steps.

On the other hand, the exponential multi-step methods, Krogstad and RK4 (results omitted) are not stable for any of these step sizes. As can be clearly seen with Krogstad and the exponential multi-step method using a P/C step, the solution loses its stable behaviour at later times for decreasing step sizes. Therefore we can expect to achieve a converged solution for short time spans or small step sizes.

Generally, the stability of multi-step methods improves when using a P/C step compared to using an additional step. Note that the multi-step methods all require starting values, which have been provided here by using the Lawson Runge-Kutta method (because of its good stability behaviour) with a step size of $\frac{1}{50}$ of the method's step size. For the comparison by the number B-calls, the calculation of the starting values is *not* included.

Results for N = 4 The results for the stability experiment with $N = 4$ are displayed in Figures A.2.9 and A.2.10. The results are very similar to those with $N = 6$ above. The only notable difference occurs for the method Krogstad, where in this case we *cannot* observe any unstable behaviour. With all exponential multi-step methods, the blow-up in the norm occurs at a later time. We can conclude that the stability requirement decreases for $N = 4$ in comparison to $N = 6$, but we generally obtain the same qualitative results.

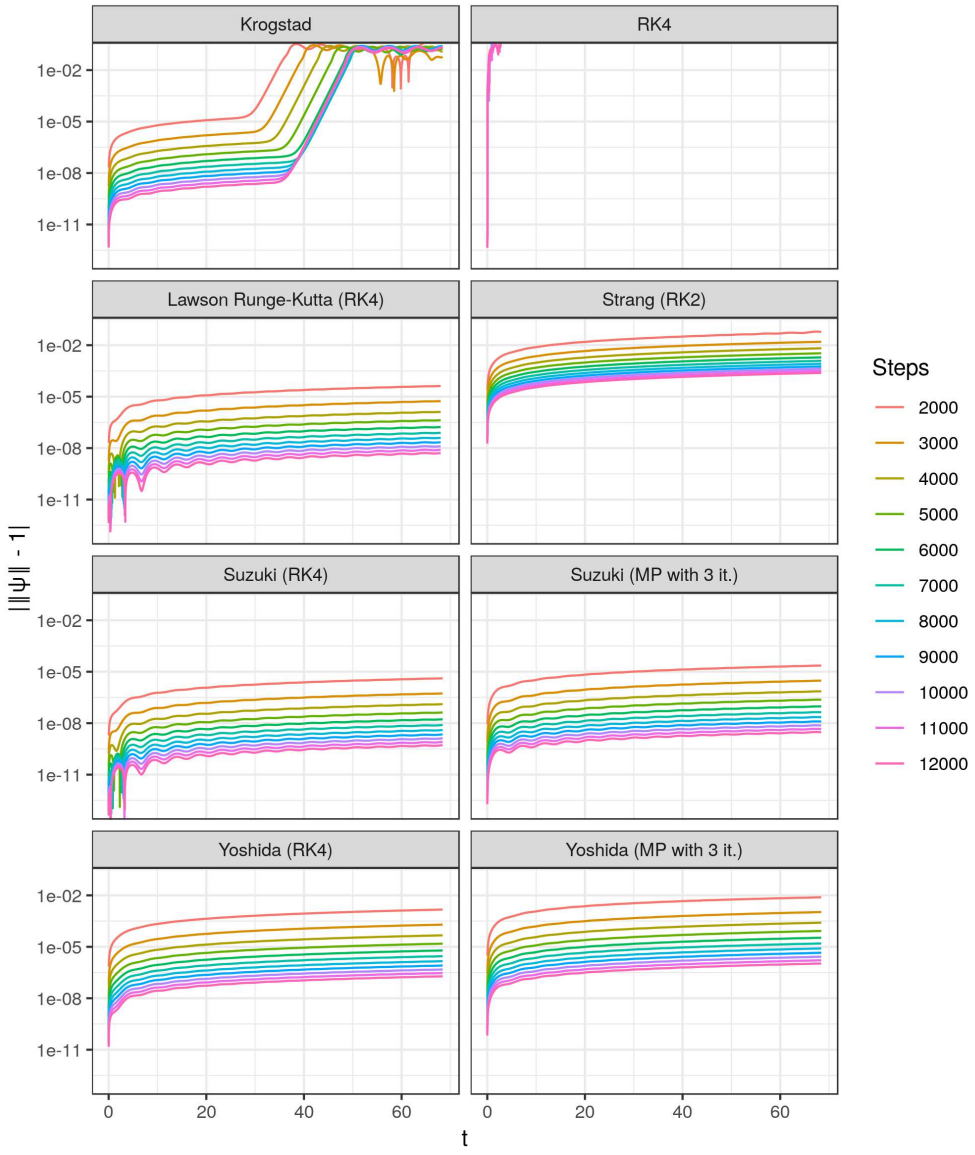

Figure 3.8.: Stability comparison of the different one-step methods in the helium experiment for $N = 6$. Methods Krogstad and RK4 show instabilities.

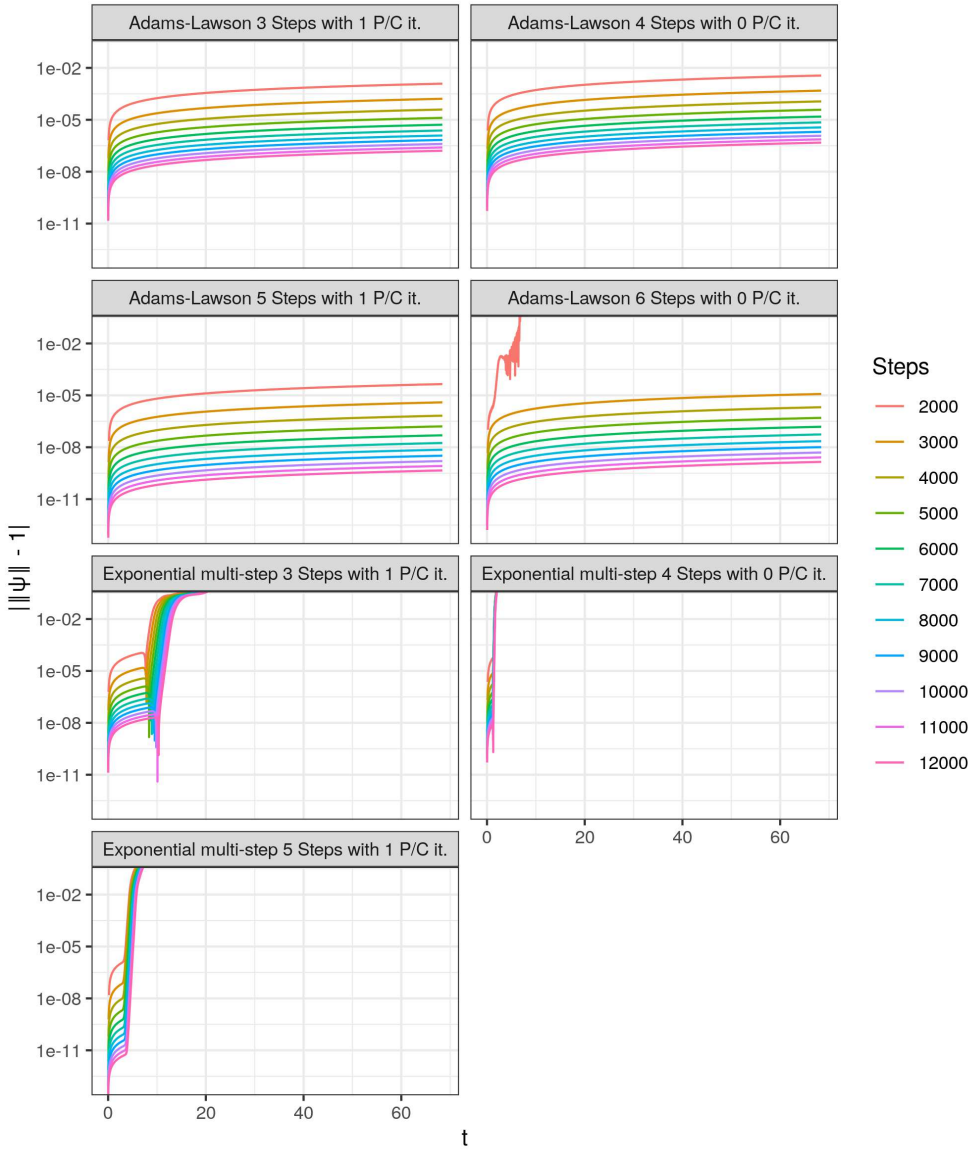

Figure 3.9.: Stability comparison of the different multi-step methods in the helium experiment for $N = 6$. The Adams-Lawson Methods perform much better than the exponential multi-step methods. The exponential multi-step method using 6 steps without P/C is not displayed, because it did not yield any useful results for these step sizes.

3.2.2. Convergence

We now compare the precision of different methods again by the size of the timestep and by the number of B-calls. Because of the structure of the solution, we have to calculate on a larger space interval using $n_x = 8192$ points on $[-256, 256]$, which is the same point density as in the stability experiment (again using a uniform grid). The choice of the space domain results from experiments: we could not observe significant interference of the solution with the boundary of the domain on this interval until $t_1 = 80$. The results in this section have only been calculated for $N = 4$ because of the already quite high calculation runtime.

Due to the lack of an explicit reference solution, we numerically compute a solution using a much smaller time step with a method previously observed as stable (Adams-Lawson 5 Steps with 1 P/C using $dt = 0.5^{11}$) where we expect to achieve a very small error as compared to the exact solution and also a comparatively smaller error in comparison to the tested methods and time steps. Therefore, we cannot state reliable results for the methods, where the solution already has a very small error as compared to our "reference solution", although the general convergence behaviour should be well observable.

As time stepping parameters we choose $t_1 = 80$ as mentioned above and vary $dt = 0.5^k$ where $k = 6, \ldots, 10$. For several methods, most notably the Suzuki and Yoshida splitting methods, we did not calculate the solution for very small time steps because the run time was already significant in this experiment. However, the observed results are sufficient to compare these methods as well.

In Figure A.2.11 we observe the convergence behaviour of different methods compared by step size. There are only few results (if any) for the methods that were previously observed as unstable. We can again clearly observe the convergence orders of the different methods. The exponential multi-step methods, Krogstad and RK4 only contribute data for small stepsizes with high precision due to stability issues as observed above.

The more important observation is the comparison by the number of B-calls. In Figure 3.10 we observe essentially the same general pattern as in the first experiment: the splitting methods are very expensive and multi-step methods have a clear advantage over one-step methods.

Conclusion We can conclude from this experiment that only the Adams-Lawson multi-step methods, especially as a P/C method, and the Lawson Runge-Kutta method are viable as soon as stability issues come into play. The former are preferable due to their higher efficiency.

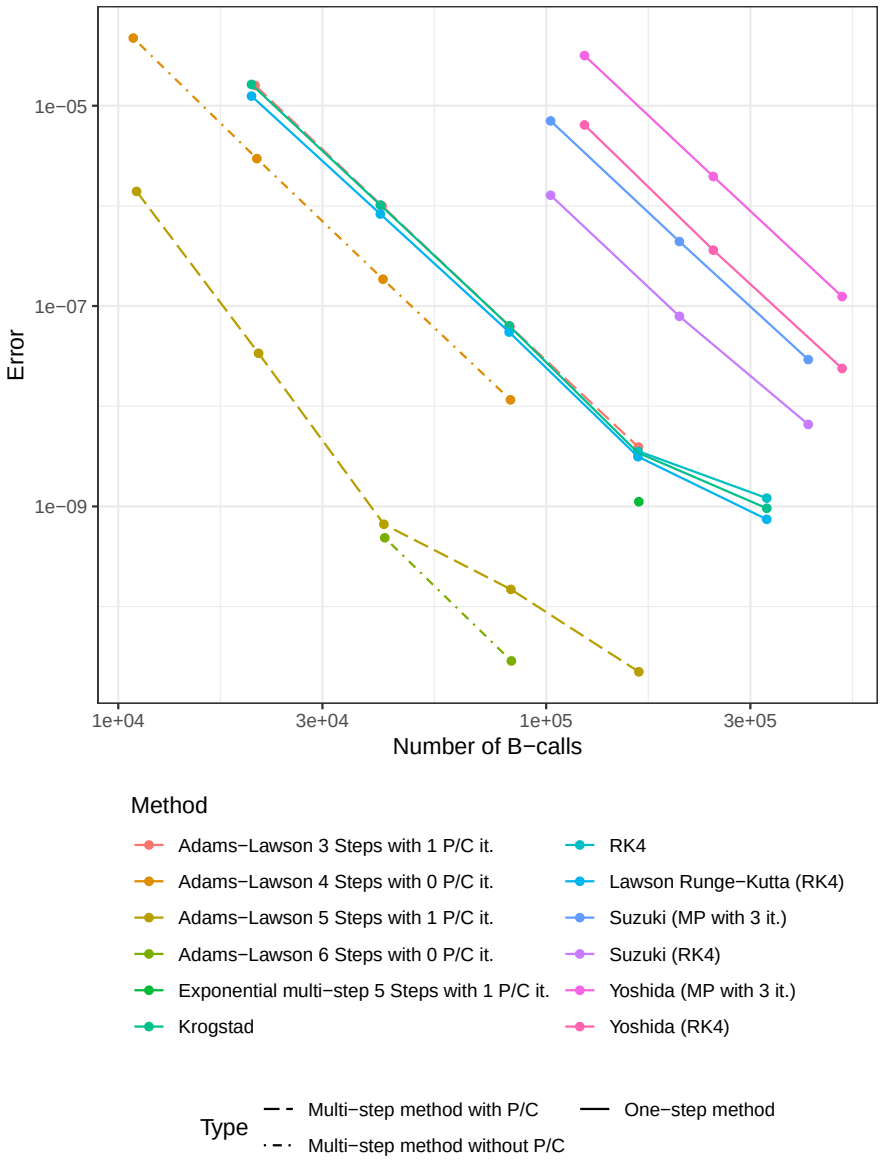

Figure 3.10.: Comparison of all (viable) methods by number of B-calls in the helium experiment ($N = 4$). The general pattern observed for the cubic Schrödinger equation is repeated where the multi-step methods are at a clear advantage and the splitting methods are very expensive.

3.2.3. Adaptive methods

We use the same experiment as for the equidistant methods (of which we are also going to include selected reference results in the comparison). Instead of observing the convergence behaviour we are now going to compare the error (again compared to the same "reference solution" as above at $t_1 = 80$) to the prescribed tolerance. Additionally, we will observe how the step size is being adapted throughout the experiment.

Note that for the calculation of the starting values we do not use a one-step method this time, but rather start with a lower number of steps and increase the number of steps until we have the necessary number of points to continue with our full length method.

We compare the Adams-Lawson methods using $3, \ldots, 7$ steps and exponential multi-step methods using 3, 5 or 7 steps each with an additional P/C step which now not only improves the order and the stability, but is also being used for the estimation of the appropriate time step size.

In Figure 3.11 we display the error compared to the number of B-calls using tolerance prescriptions of $10^{-3}, \ldots, 10^{-9}$ for the adaptive methods. For reference, two equidistant methods have been included as well. We can clearly observe by the number of B-calls that for the exponential multi-step methods the step size adaptivity does not change the total number of steps significantly for stricter tolerances. This is in accordance with our stability results, where we required small step sizes for this method in order to achieve a reasonable solution.

On the other hand, the adaptive Adams-Lawson methods with 5 or more steps were able to lower the total number of steps according to the tolerance while maintaining the precision and we can see an improvement over the equidistant method.

In Figure 3.12 the observed errors are displayed relative to the prescribed tolerance. We can observe that the exponential multi-step methods are less precise than the tolerance, where the Adams-Lawson methods have the tendency to be more precise for an increasing number of steps using the same tolerance.

3.2.3.1. Step size adaptivity

We have observed that the step size adaptivity improves the results for some methods and are now going to investigate how the adaptivity changes the step size of the method over time. First we have displayed the step sizes for all the methods together with a solution graph and the external potential in Figure 3.13, where all the results have been calculated using the tolerance 10^{-6}. There we can clearly observe that the step size control chooses a more or less constant time step size for the Adams-Lawson method, whereas the step size is being decreased significantly for the exponential multi-step methods over time. The step size increases with the number of steps for the Adams-Lawson methods, which is a result of the increasing order. A notable observation is the small dip at the end of the time interval, where the step size control seems to adapt to a local extremum of the external potential by decreasing the step size.

This observation has led to a more detailed visualisation in Figure 3.14, where the change in step size has been displayed for more than just one prescribed tolerance. We

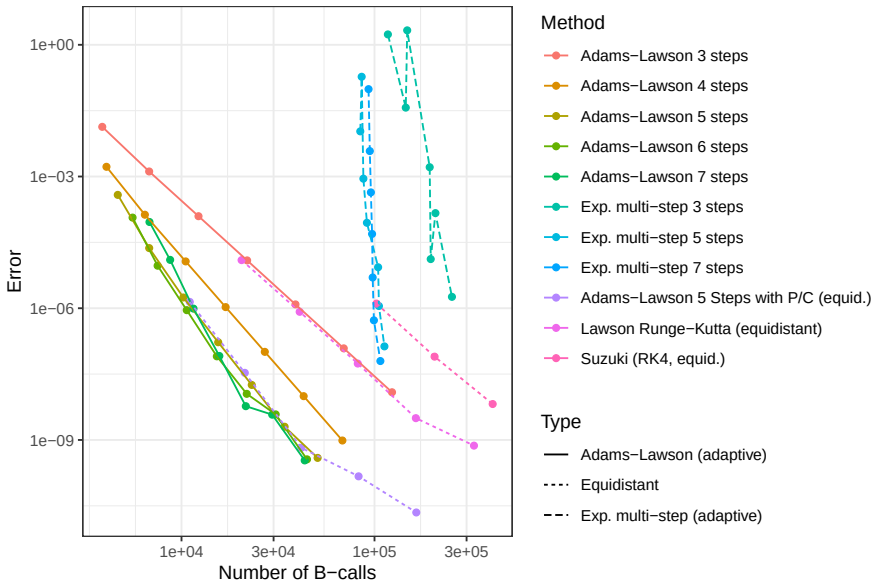

Figure 3.11.: Comparison of adaptive methods by number of B-calls in the helium experiment ($N = 4$). The data points have been created by prescribing a tolerance of $10^{-3}, \ldots, 10^{-9}$. The stability requirement enforces a high number of steps for the exponential multi-step methods regardless of the prescribed tolerance.

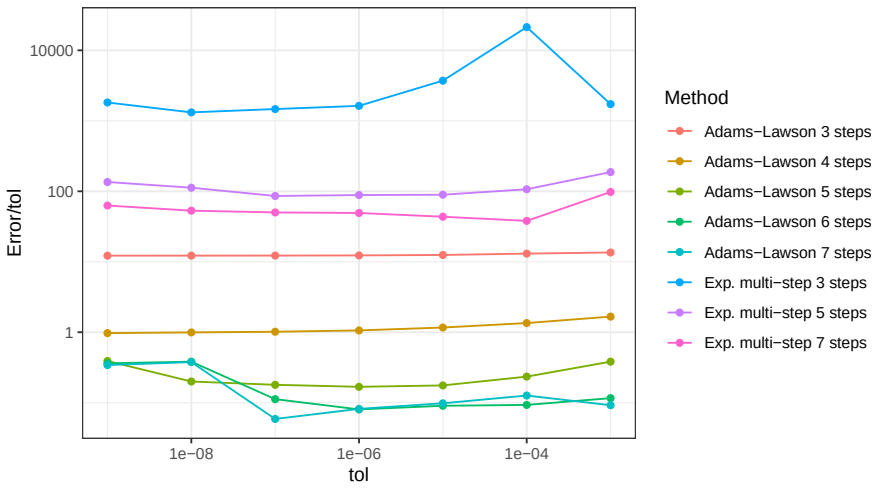

Figure 3.12.: Comparison of adaptive methods using the quotient of $\frac{\text{Error}}{\text{tolerance}}$ in the helium experiment ($N = 4$). We can see that the adaptive step size control manages to achieve the prescribed error tolerance only for the Adams-Lawson method where the methods tend to be more precise for a higher number of steps when using the same tolerance.

can see that the step size control adapts to the local extrema of the external potential for small prescribed tolerances where the "global" step size (which the adaptivity does not change for $t \in [1, 50]$ for example) is larger than a certain value, which seems to depend on the number of steps but *not* on the tolerance. The occurring oscillation for one method can be attributed to its low accuracy and resulting unwanted interaction with the artificial boundary and is therefore neglected.

We can now also discuss the change in this "global" step size for different parameters: we observe a clear improvement when increasing the step count from 3 to 4, however for higher step counts the step size is decreasing slightly again (note that this observation does not indicate the actually achieved efficiency, but only e.g. the expected runtime of an experiment).

This behaviour may indicate a local stability requirement or a local change of convergence behaviour, which the step size adaptivity can overcome.

3.2.3.2. Long time integration

Concluding this chapter, we would like to present some results of calculations where we do not stop at an early stage of the experiment. Instead, we calculate until after the laser has shut off again (or in our model has declined to negligible size). For this experiment, only the adaptive Adams-Lawson methods were able to provide results in reasonable time. We again choose $N = 4$ but extend the space interval to $[-512, 512]$ using $n_x = 16384$ and a tolerance of 10^{-5}.

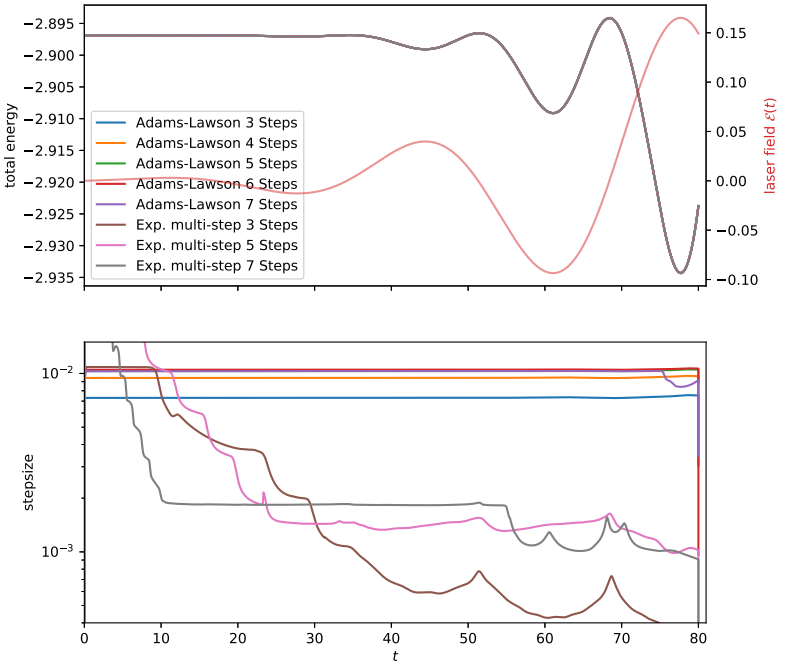

Figure 3.13.: Visualisation of the step size adaptivity for tol $= 10^{-6}$ in the helium experiment. The step size for the exponential multi-step methods is significantly smaller compared to the Adams-Lawson methods, although the latter actually provide more precise results. The step size increases for a higher number of steps as a result of the higher order.

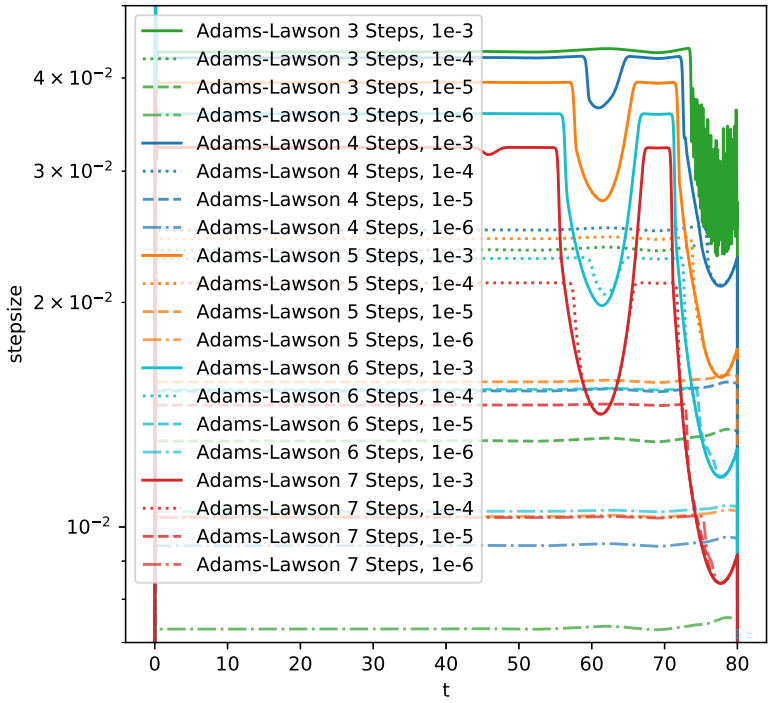

Figure 3.14.: Visualisation of the step size adaptivity for the Adams-Lawson methods using different step numbers and tolerances in the helium experiment ($N = 4$). We can observe that methods using the same step number all adapt the step size to approximately the same local minimum at local extrema of the external potential (see Figure 3.7 or Figure 3.13), whereas during the rest of the experiment, the step size is being adapted as expected (larger step size for higher tolerances).

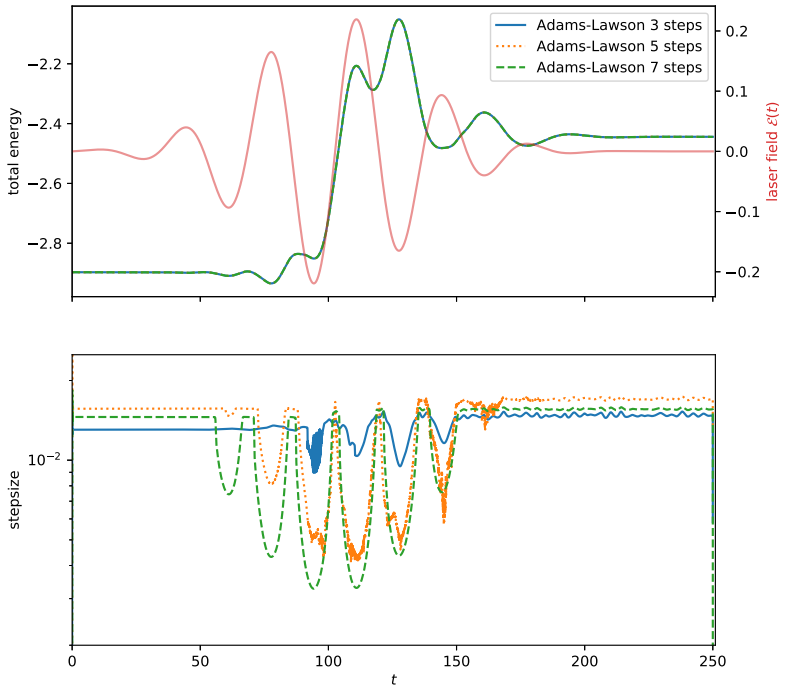

Figure 3.15.: Visualisation of the step size adaptivity for the Adams-Lawson methods in the helium experiment ($N = 4$) until $t_1 = 250$ ($n_x = 16384$, tolerance of 10^{-5}). We can observe that the step sizes return to approximately their starting value when the external potential disappears.

In Figure 3.15 we observe a similar behaviour as above. The 5 step method has been able to use the largest step size "globally" whereas the step size necessary in the local extrema poses tighter restrictions (smaller step sizes) for a higher number of steps.

3.3. Quantum dot

This experiment tests the MCTDHF method on two electrons in a harmonic oscillator potential, another experiment from [12]. We model the electron interaction by a "smoothed Coulomb" potential resulting in the Hamiltonian

$$H(x,y;t) = -\frac{1}{2}\Delta + \frac{x^2+y^2}{32} + \frac{1}{\sqrt{(x-y)^2+\frac{1}{4}}} + (x+y)\mathcal{E}(t),$$

where $\mathcal{E}(t) = \sin(2t)$ models an external laser field.

Because of the localised solution, we can calculate on the interval $[-20, 20]$ using $n_x = 1024$ spatial points for all following experiments. In this experiment, we again compare $N = 4$ and $N = 6$ which is now also tractable for the convergence results. A resulting energy curve can be seen in Figure 3.16, which has been calculated using an equidistant Adams-Lawson method with P/C.

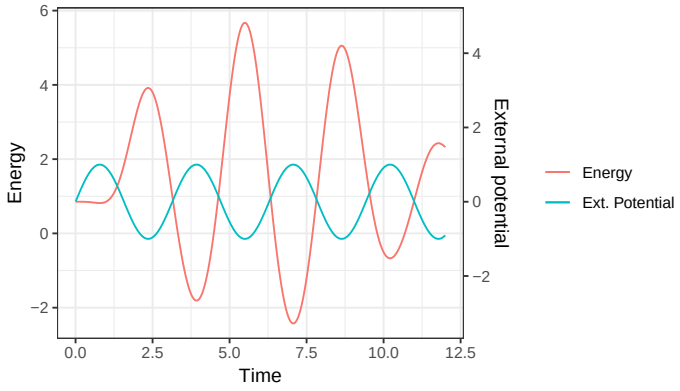

Figure 3.16.: Visualisation of the energy change in the quantum dot experiment.

3.3.1. Stability

As before, we compare the stability of all methods by setting $\mathcal{E}(t) = 0$ and observing the change of the ground state (which is only a result of numerical inaccuracies) as the deviation of the norm of the discrete wave function from the starting value 1.

We calculate using $2000, \ldots, 12000$ steps on the time interval $[0, 70]$.[1]

The results in Figures 3.17 and 3.18 for $N = 4$ (and similarly for $N = 6$ in Figures A.3.12 and A.3.13 in the appendix) show the same general picture as in the helium experiment: except for RK4, all one-step methods are stable. For multi-step methods,

[1] Note that this interval is much larger than the time interval in the convergence experiment of this section.

the exponential multi-step methods are not stable regardless of P/C, the Adams-Lawson methods with P/C are stable however the Adams-Lawson method with 6 steps and without P/C is not stable.

Note the very small scale of error for the 6th order Adams-Lawson methods at approximately 10^{-12}.

3.3.2. Convergence

We now compare the convergence behaviour of all methods in the quantum dot experiment by calculations using a different number of steps in the time interval $[0, 12]$. We choose the number of steps from $\lfloor 600 \cdot 1.3^i \rfloor, i = 0, \ldots, 10$ because we could observe consistent convergence behaviour in this interval and have a sufficient number of data points to visualize it. (Note that any data points with an error $> 5 \cdot 10^{-4}$ have been omitted in the figures.)

We compare the resulting wave functions at $t_1 = 12$ with a numerical "reference solution" using the 5 step P/C Adams-Lawson method and 12000 steps (note that $\lfloor 600 \cdot 1.3^{10} \rfloor = 8271$). The convergence behaviour can be seen in Figure 3.19 ($N = 4$, results for $N = 6$ in appendix Figure A.3.16) where we displayed the error to the number of B-calls. The convergence is very regular and the convergence order can clearly be seen. Again we observe the same pattern as in the previous experiments: the splitting methods are the most expensive, the Adams-Lawson methods are the most efficient and Krogstad and Lawson Runge-Kutta are in between and provide essentially the same results.

There is no data for the exponential multi-step methods displayed, because none of them yielded a converging result for any of the chosen step numbers. In further experiments we observed that e.g. the exponential multi-step method with 3 steps and P/C only started its convergence at over 22000 steps.

Additionally, it is noteworthy that also the Adams-Lawson method with 6 steps only provides converging results for large step numbers. These observations are all in accordance with the previously observed stability behaviour.

As above, we calculate the starting values for the multi step methods using the Lawson Runge-Kutta method with 1/50 of the step size which is excessively precise and should not interfere with our results. As the number of B-calls involved in this calculation is in some cases higher than for the actual method, the data in the figures does *not* include those calls.[2]

3.3.3. Adaptive methods

As before, we use the P/C methods to construct multi step methods with time step adaptivity. We compare Adams-Lawson and exponential multi-step methods using 3, 5 or 7 steps. Because of the observed precision in the convergence experiment, we choose the tolerance from $10^{-3}, \ldots, 10^{-11}$.

[2] For reference, e.g. the Adams-Lawson method with P/C and 5 steps needs 4 additional starting values. Therefore we have $4 \cdot 4 \cdot 50 = 800$ additional B-calls, which is 4 per Lawson Runge-Kutta step for $4 \cdot 50$ steps because of the reduced step size.

First we visualise the precision and efficiency of the adaptive methods for $N = 4$ and compare them to two equidistant methods in Figure 3.20. The general observation is repeated, where we conclude that the adaptivity further increases the efficiency of the Adams-Lawson method, whereas the exponential multi-step methods require very small stepsizes to achieve converging results, which leads to them being expensive in our comparison.

This time, we can observe the stability behaviour of the exponential multi-step methods also for the Adams-Lawson methods with 5 and 7 steps: for high tolerances the number of steps changes marginally compared to the increase of precision. This again indicates a stability requirement for the step size. Note, however, that the required step size is quite large and does not restrict the application of the method.

We also display the precision in comparison to the prescribed tolerance in Figure 3.21. Again, the methods improve the precision over the different tolerance choices. The Adams-Lawson methods with 3 and 5 steps achieve a result marginally worse than the prescribed tolerances in comparison to the much worse exponential multi-step methods. Only the Adams-Lawson methods with 7 steps achieves results better than the targeted tolerance.

The corresponding results for $N = 6$ can be found in Figures A.3.17 and A.3.18 in the appendix and are similar to the results discussed above.

Step size adaptivity For more clarity in the visualisations, we restrict ourselves to the Adams-Lawson methods in this section.

In Figure 3.22 the change of the step size of different adaptive methods can be observed, together with the corresponding energy change using a tolerance of 10^{-5} and $N = 4$ (results for $N = 6$ can be found in Figure A.3.19) It is again clearly visible, that the step size is being adapted to the local extrema of the energy curve. We can also see the improved (smaller) step size of Adams-Lawson methods using more steps.

In Figure 3.23 (A.3.20 for $N = 6$) the above observed behaviour is repeated: The Adams-Lawson method using 5 steps does not increase the step size notably for high tolerances, where from 10^{-6} on we can see distinct increases in the number of steps.

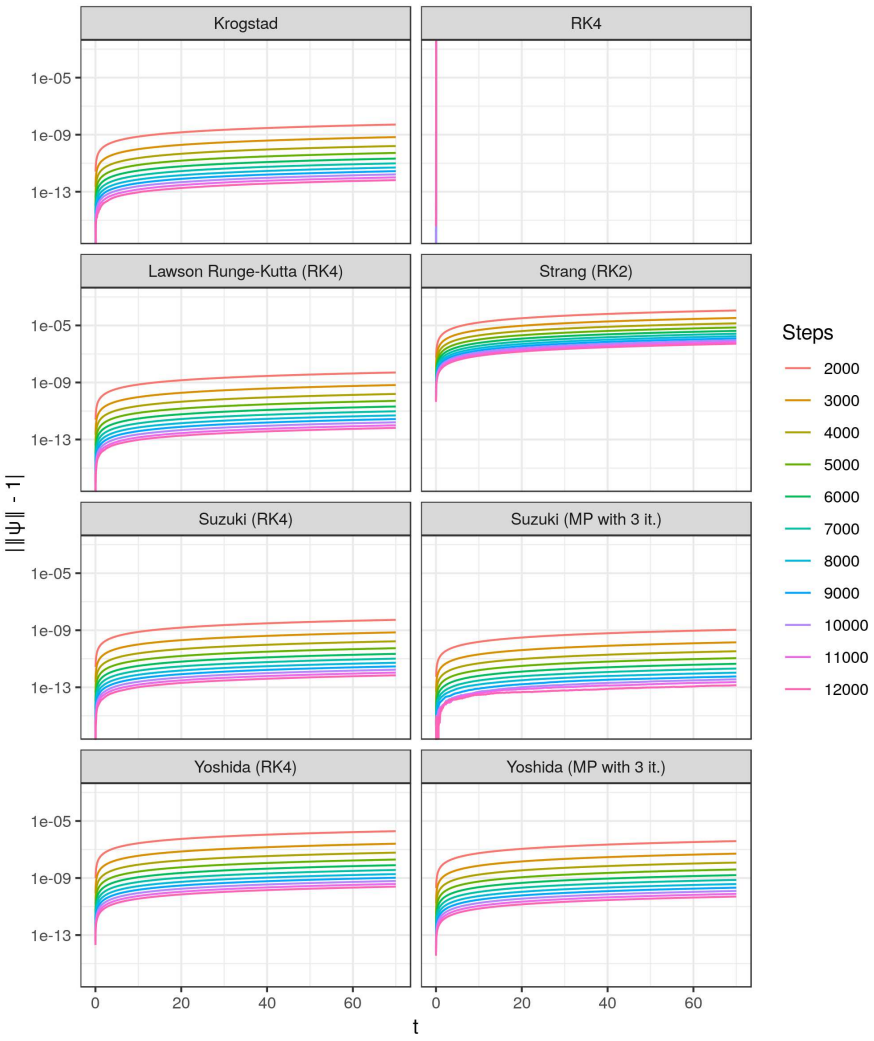

Figure 3.17.: Stability behaviour of all one-step methods in the quantum dot experiment ($N = 4$).

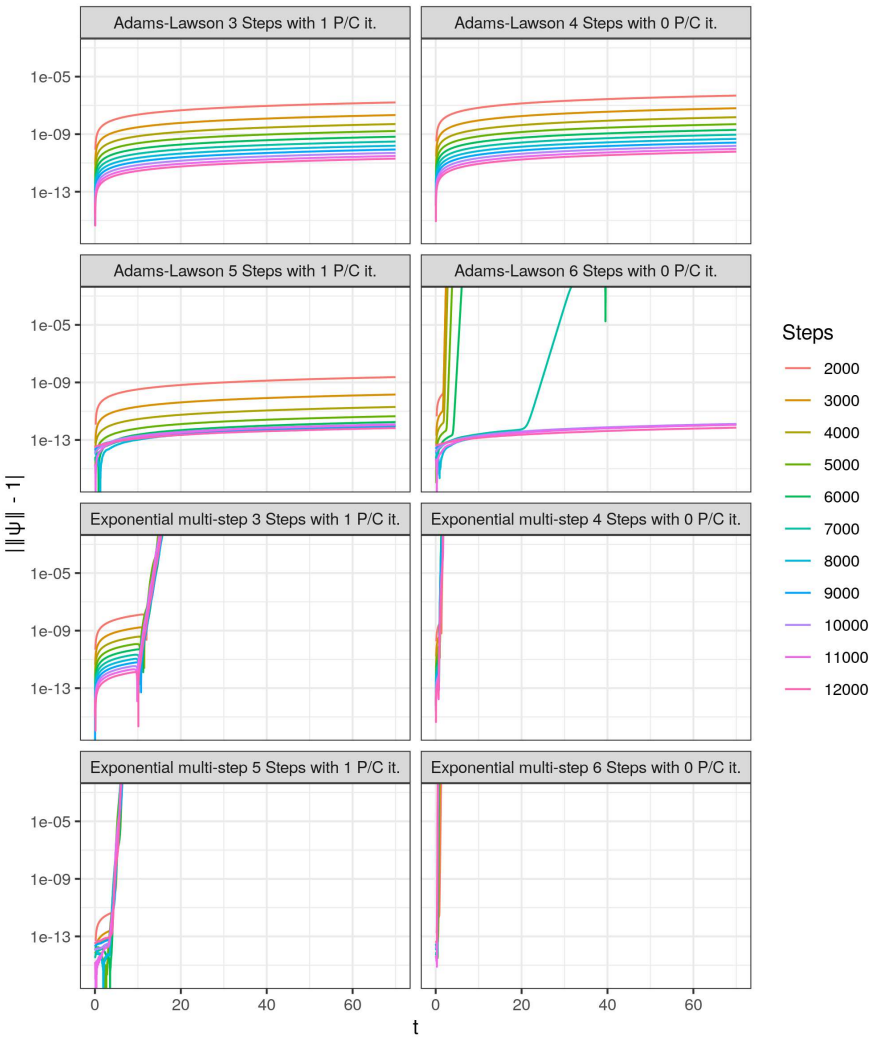

Figure 3.18.: Stability behaviour of all multi-step methods in the quantum dot experiment ($N = 4$).

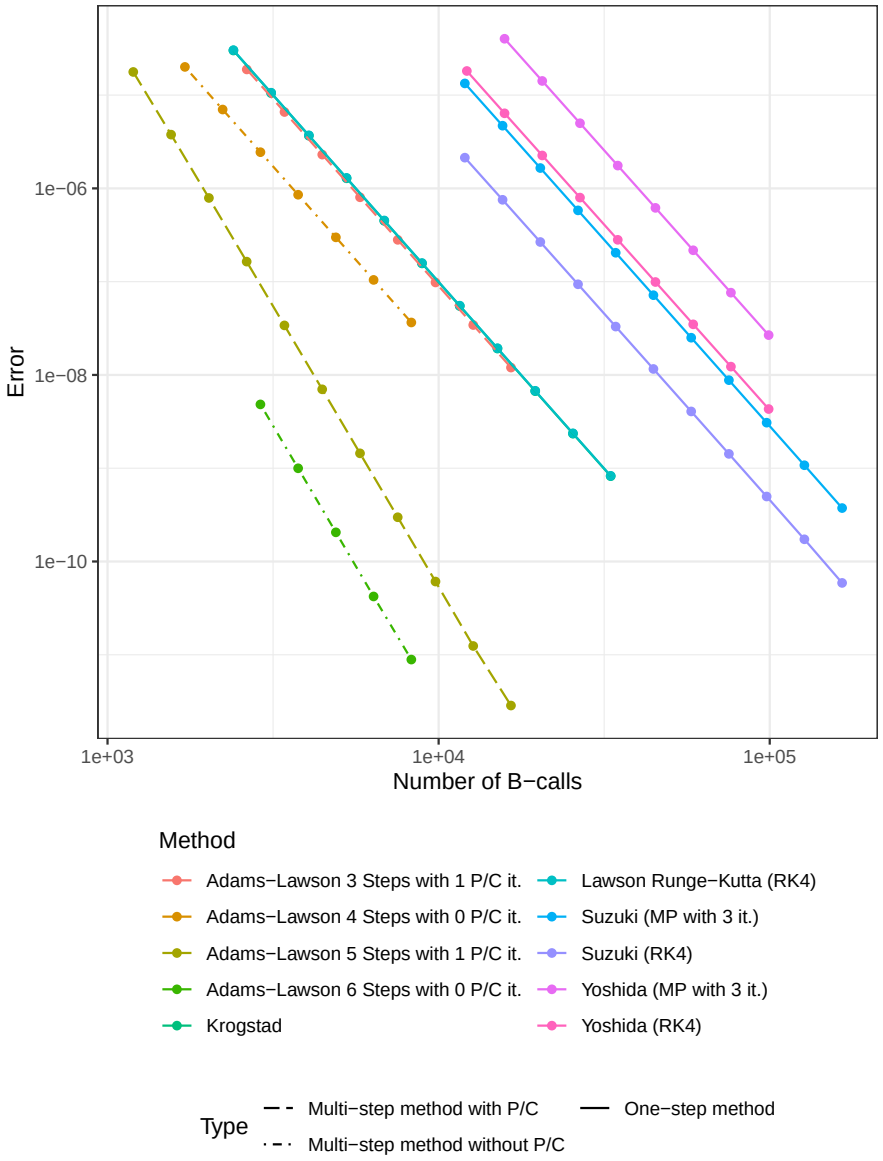

Figure 3.19.: Comparison of all (viable) methods by number of B-calls in the quantum dot experiment ($N = 4$). As with the previous experiments, the higher order Adams-Lawson methods are much more efficient. Note the very precise correspondence of the Krogstad and Lawson Runge-Kutta data points.

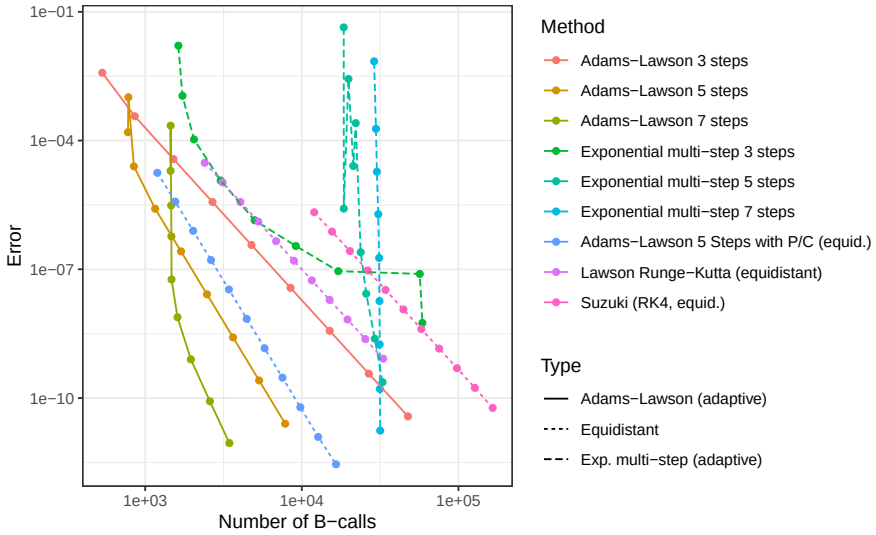

Figure 3.20.: Comparison of adaptive methods by number of B-calls in the quantum dot experiment ($N = 4$). The data points have been created by prescribing a tolerance of $10^{-3}, \ldots, 10^{-11}$. The stability requirement enforces a high number of steps (small step sizes) for the exponential multi-step methods using 5 or 7 steps, regardless of the prescribed tolerance.

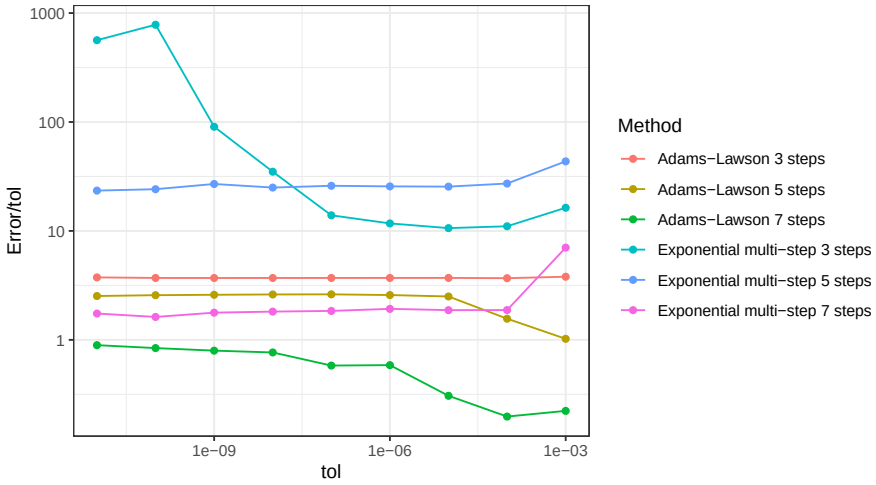

Figure 3.21.: Comparison of adaptive methods using the quotient of $\frac{\text{Error}}{\text{tolerance}}$ in the quantum dot experiment ($N = 4$). We can see that the adaptive step size control manages to achieve the prescribed error tolerance only for the Adams-Lawson method where the methods tend to be more precise for a higher number of steps when using the same tolerance.

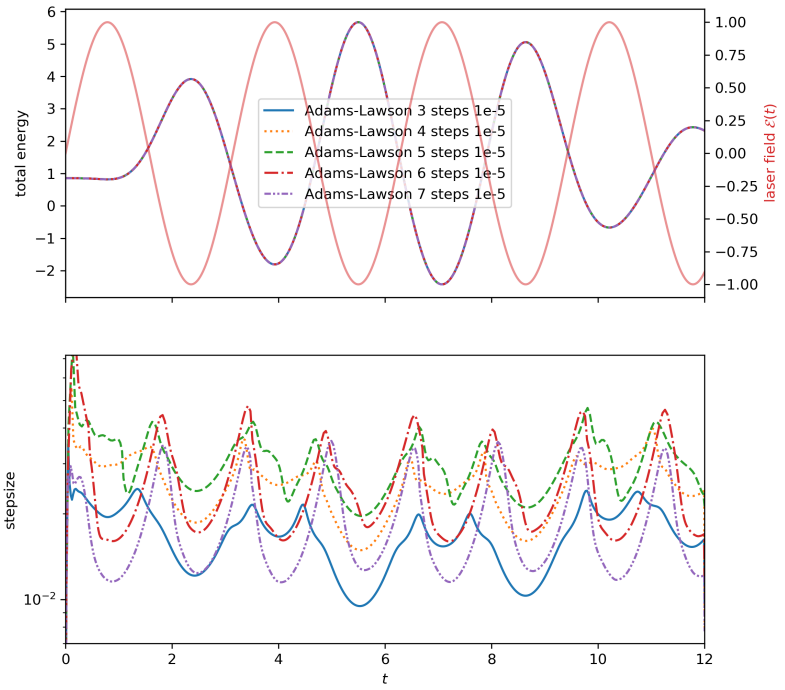

Figure 3.22.: Visualisation of the step size change for the adaptive Adams-Lawson method using between 3 and 7 steps at tolerance 10^{-5} for the quantum dot experiment with $N = 4$. The local minima of the step sizes correspond to local extrema of the energy curve.

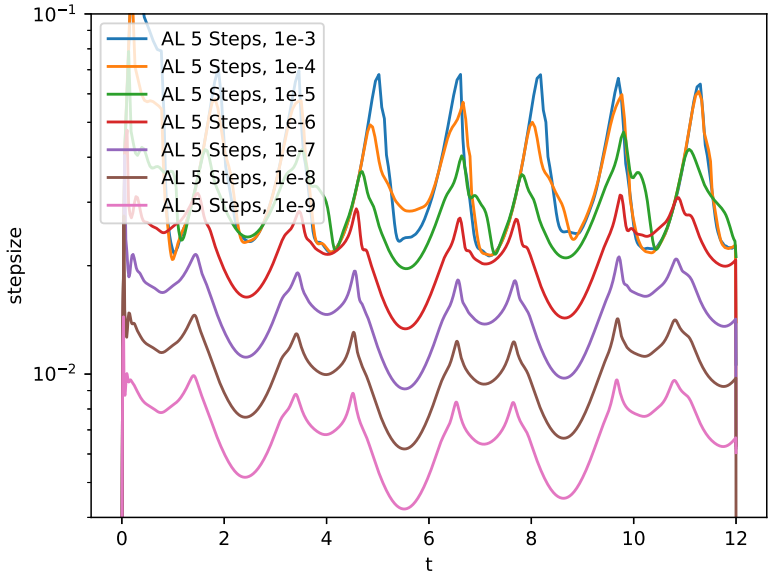

Figure 3.23.: Visualisation of the step size change for different tolerances for the quantum dot experiment with $N = 4$. There is not much change in step size between high tolerances. From 10^{-6} and below we can see clear increments in the number of steps.

A. Additional figures

A.1. Cubic Schrödinger equation

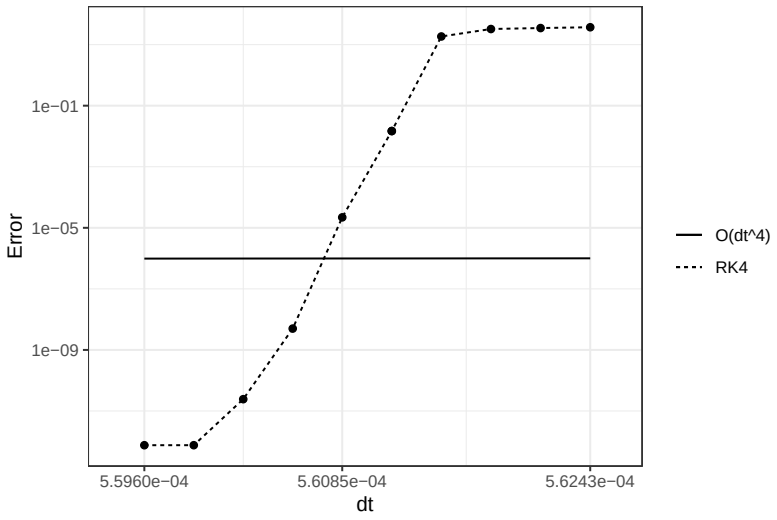

Figure A.1.1.: Convergence of RK4 at time $t_1 = 1$ for the cubic Schrödinger equation. Note the very small change of the timestep size compared to the large change in precision displayed here. A reference line of order $\mathcal{O}(dt^4)$ is plotted to emphasise that this is *not* classical convergence behaviour, but rather the shift from the instability region to a stable method. Still, this shift appears to increase its precision gradually.

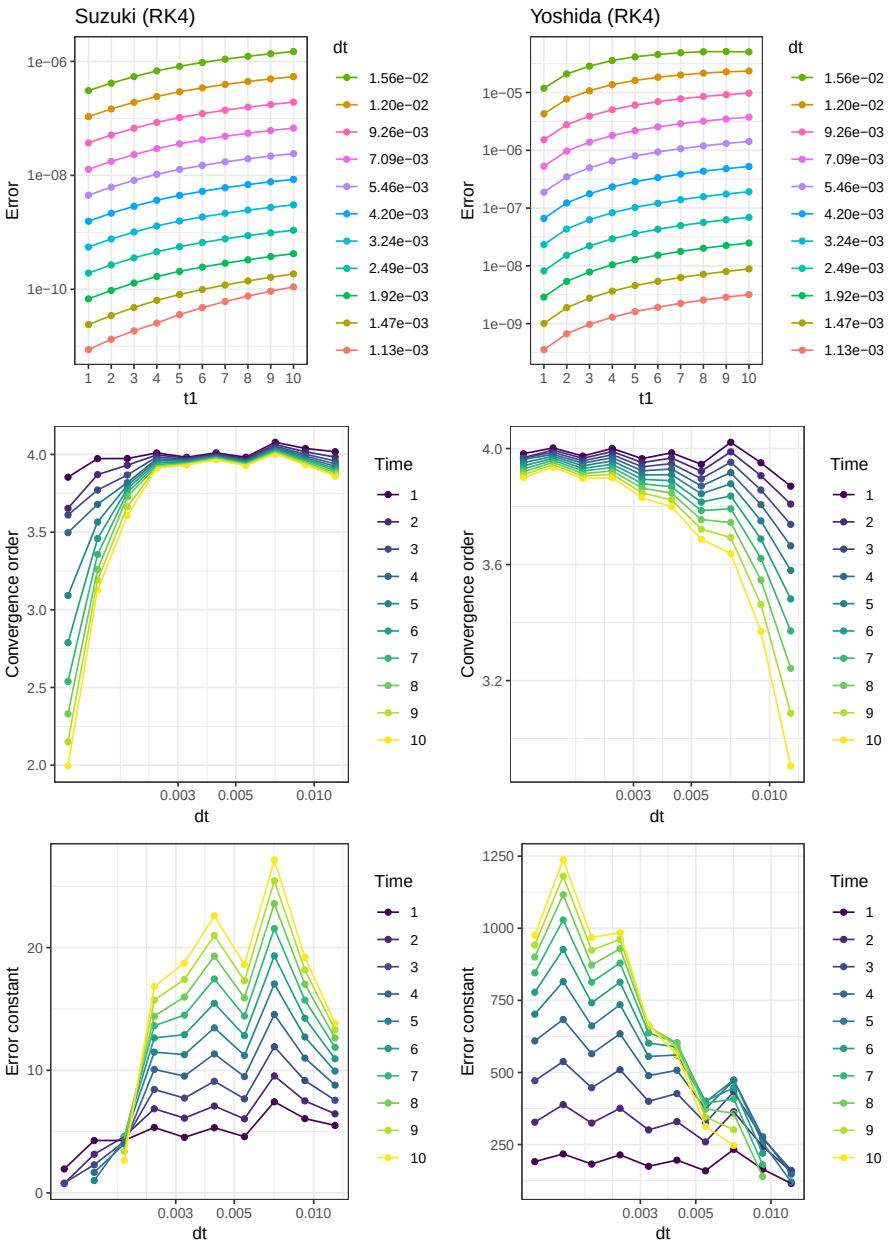

Figure A.1.2.: Change of observed convergence parameters for different dt and t_1 for Suzuki and Yoshida methods with RK4 for B propagation for the cubic Schrödinger equation.

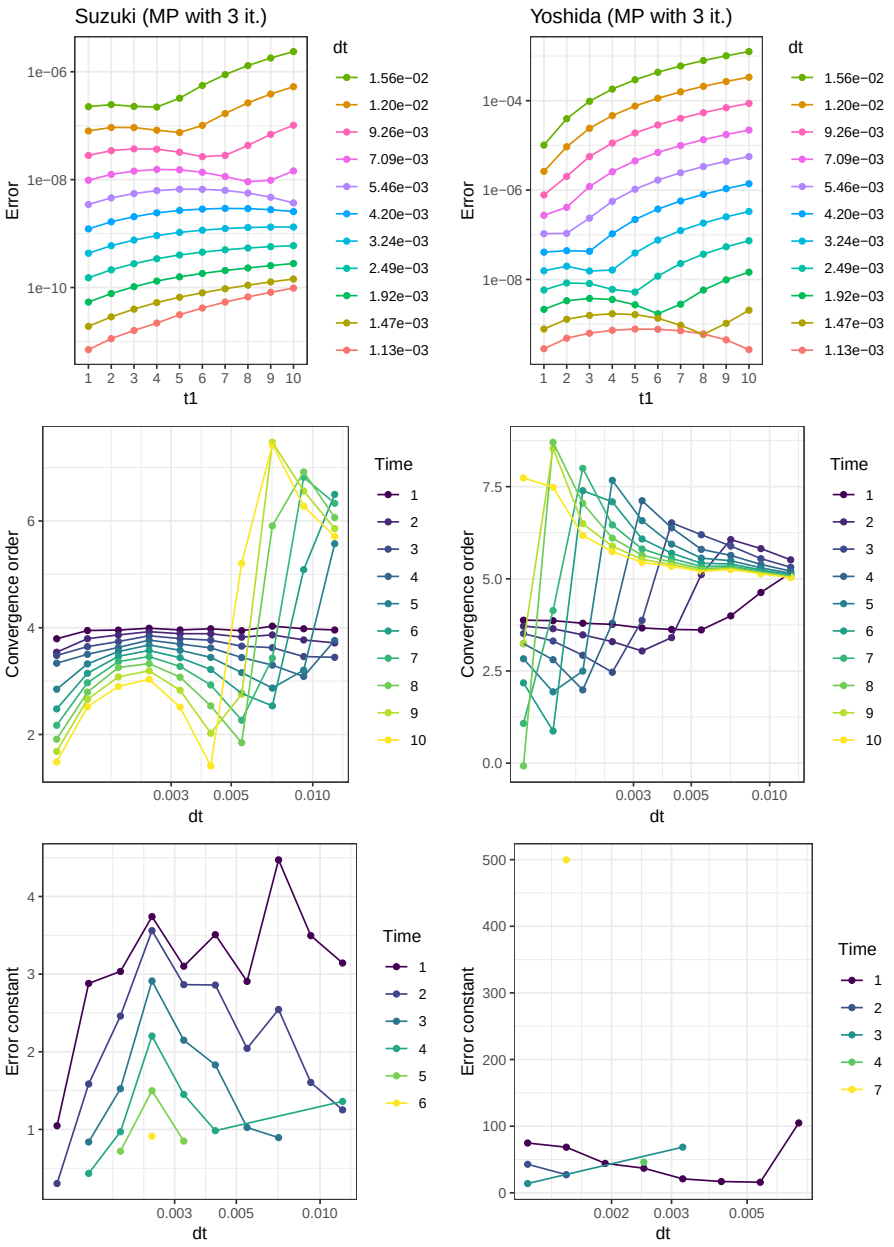

Figure A.1.3.: Change of observed convergence parameters for different dt and t_1 for Suzuki and Yoshida methods using mid-point rule with 3 iterations for B propagation for the cubic Schrödinger equation.

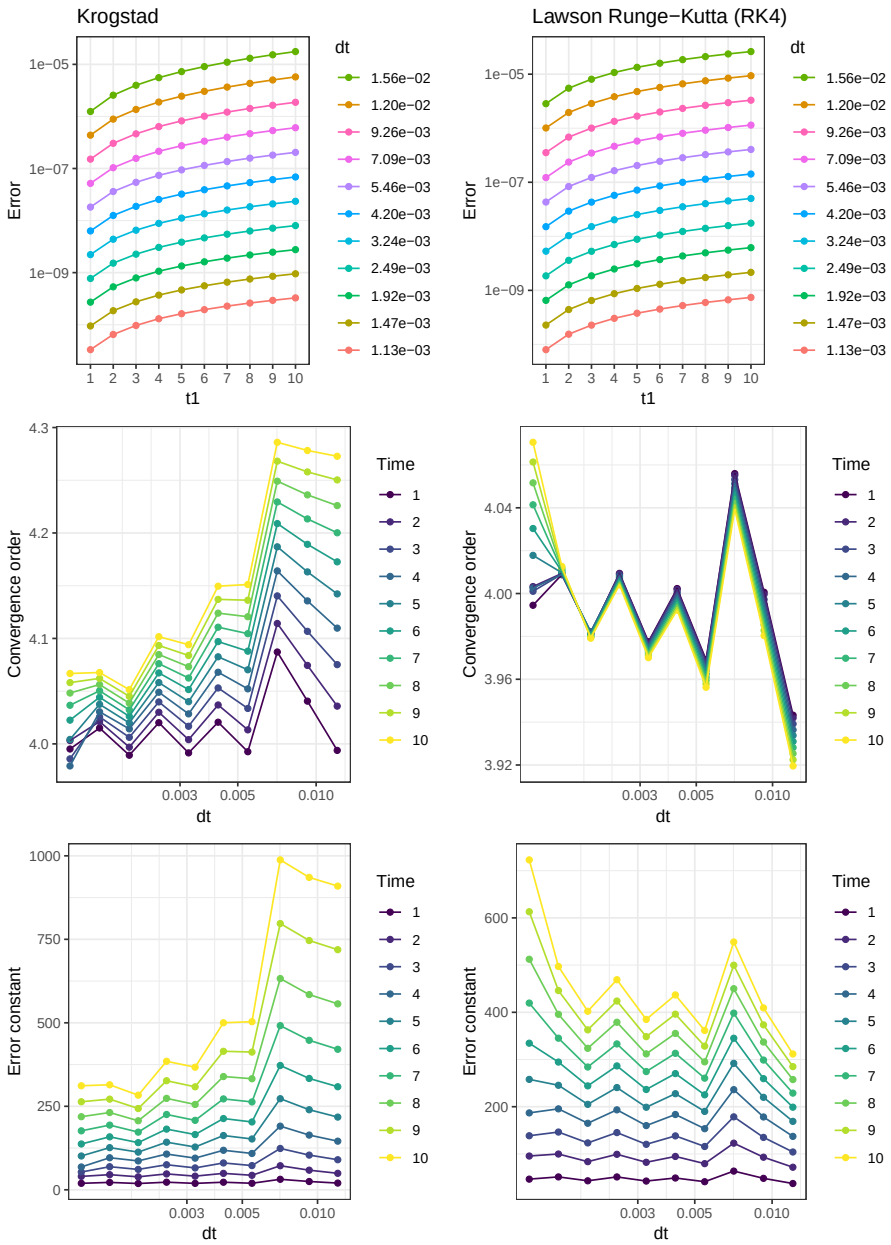

Figure A.1.4.: Change of observed convergence parameters for different dt and t_1 for Krogstad (exponential Runge-Kutta) and Lawson Runge-Kutta (RK4) methods for the cubic Schrödinger equation. Note the very small y-scale for the convergence order of the latter.

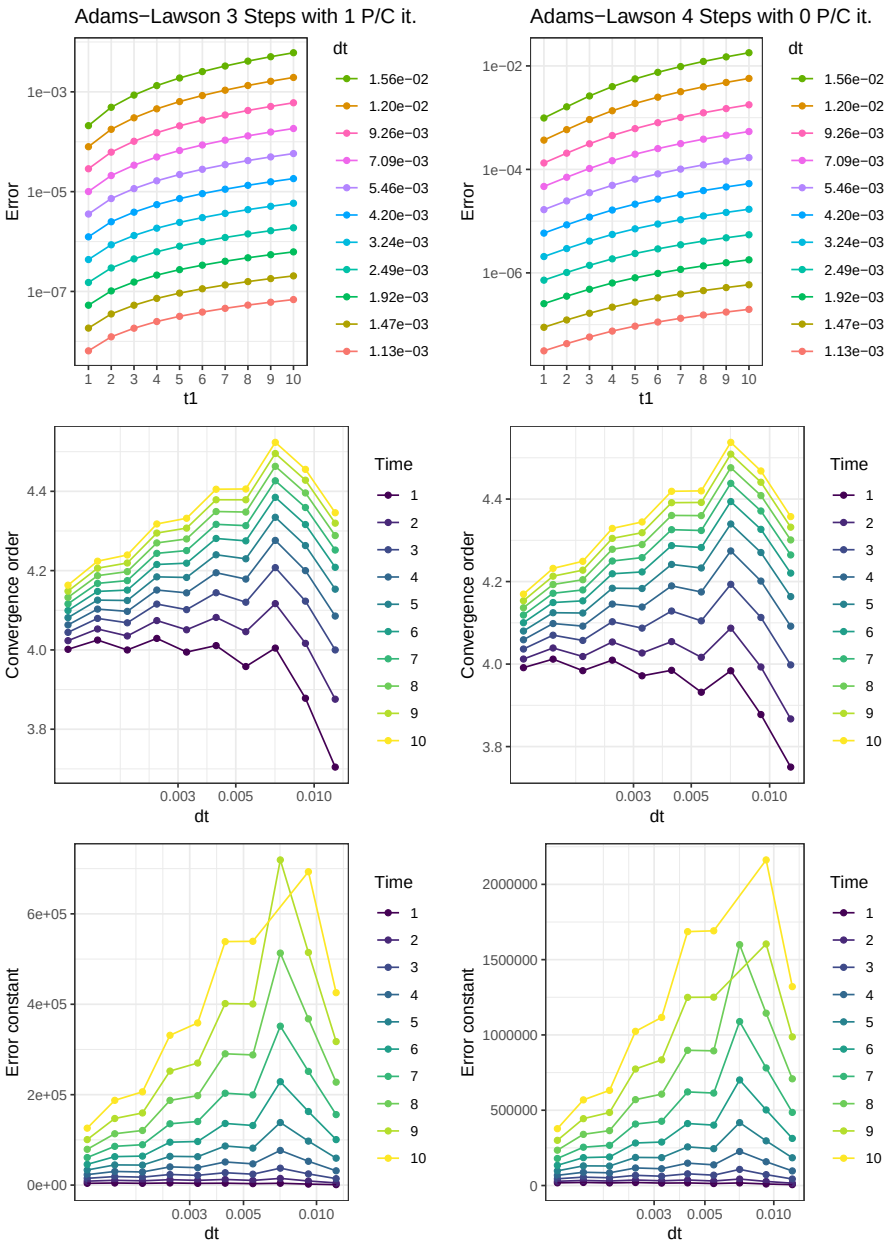

Figure A.1.5.: Change of observed convergence parameters for different dt and t_1 for 4th order Adams-Lawson methods for the cubic Schrödinger equation.

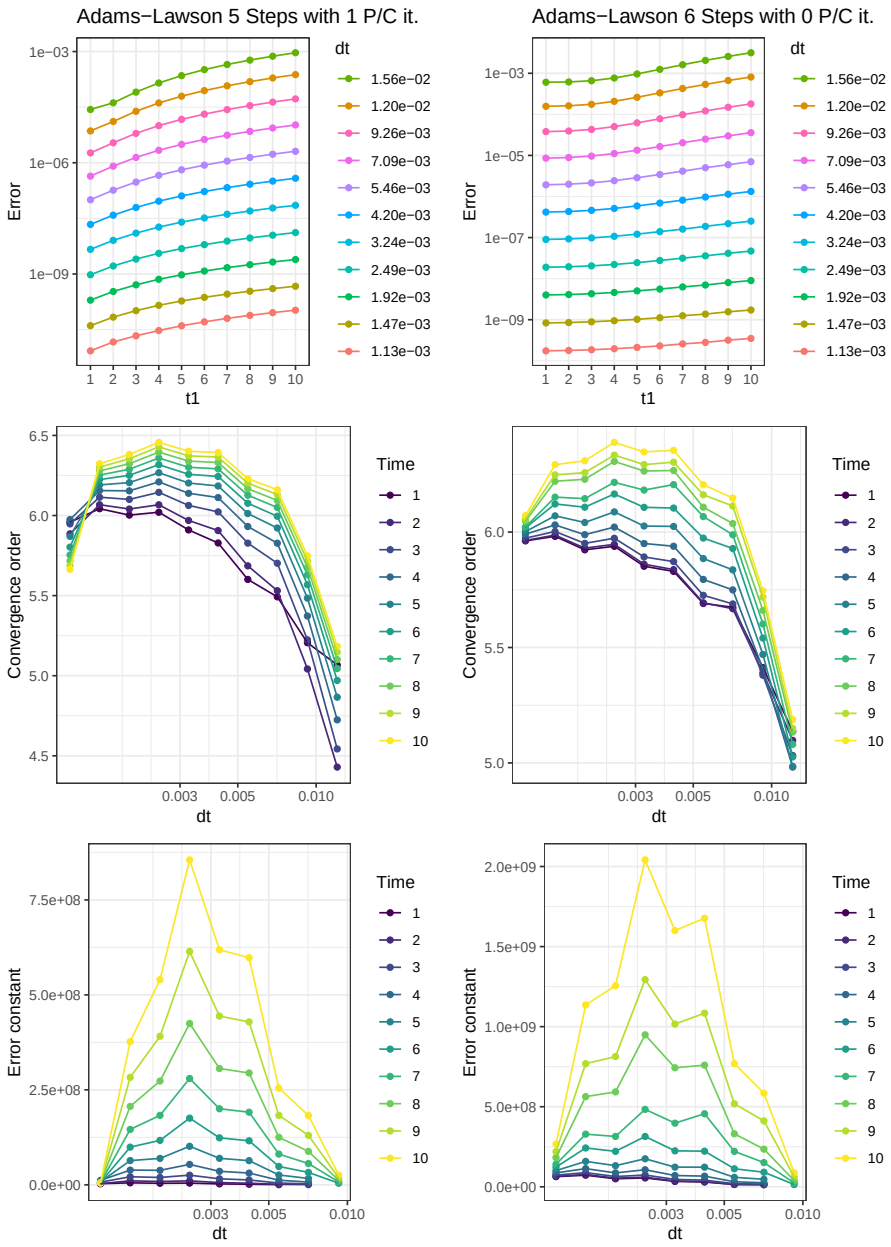

Figure A.1.6.: Change of observed convergence parameters for different dt and t_1 for 6th order Adams-Lawson methods for the cubic Schrödinger equation.

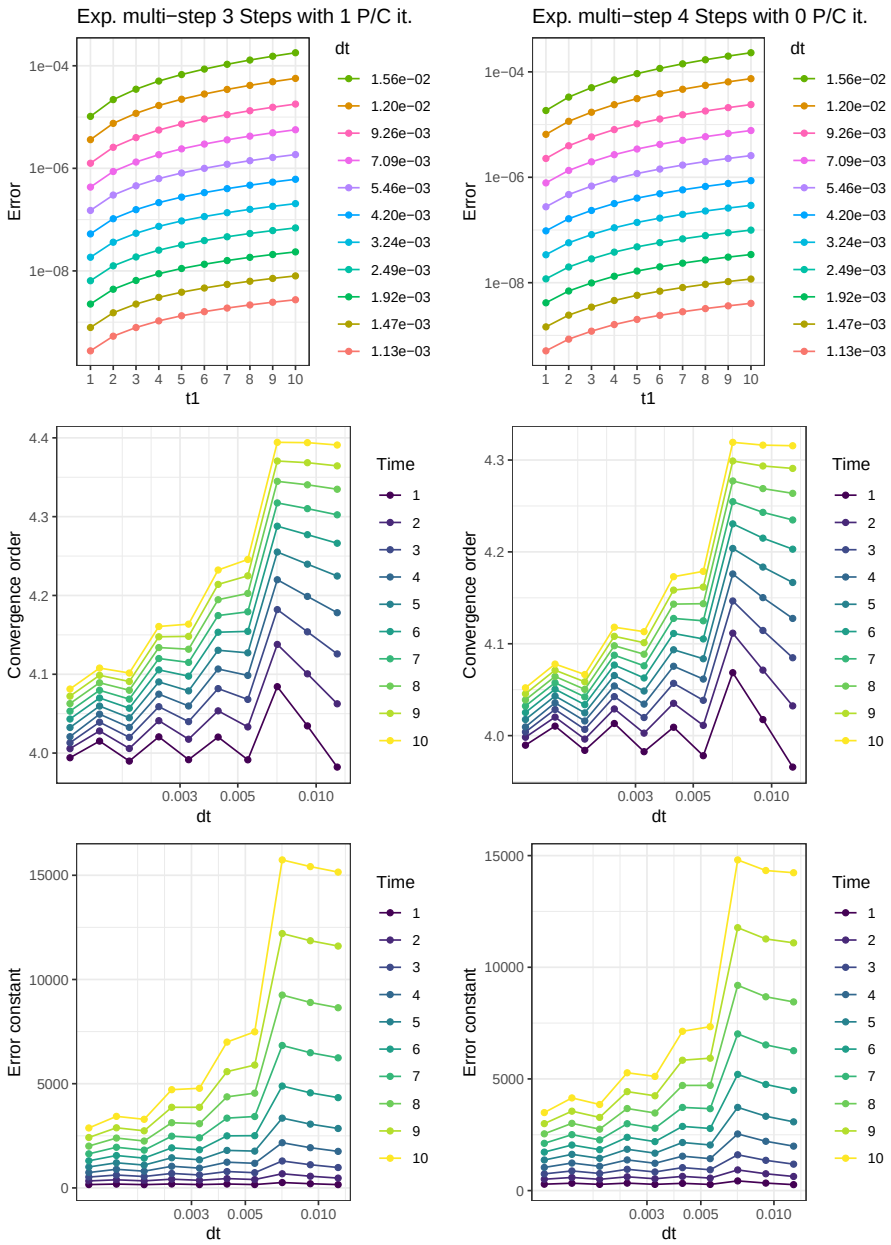

Figure A.1.7.: Change of observed convergence parameters for different dt and t_1 for 4th order Exponential multi-step methods for the cubic Schrödinger equation.

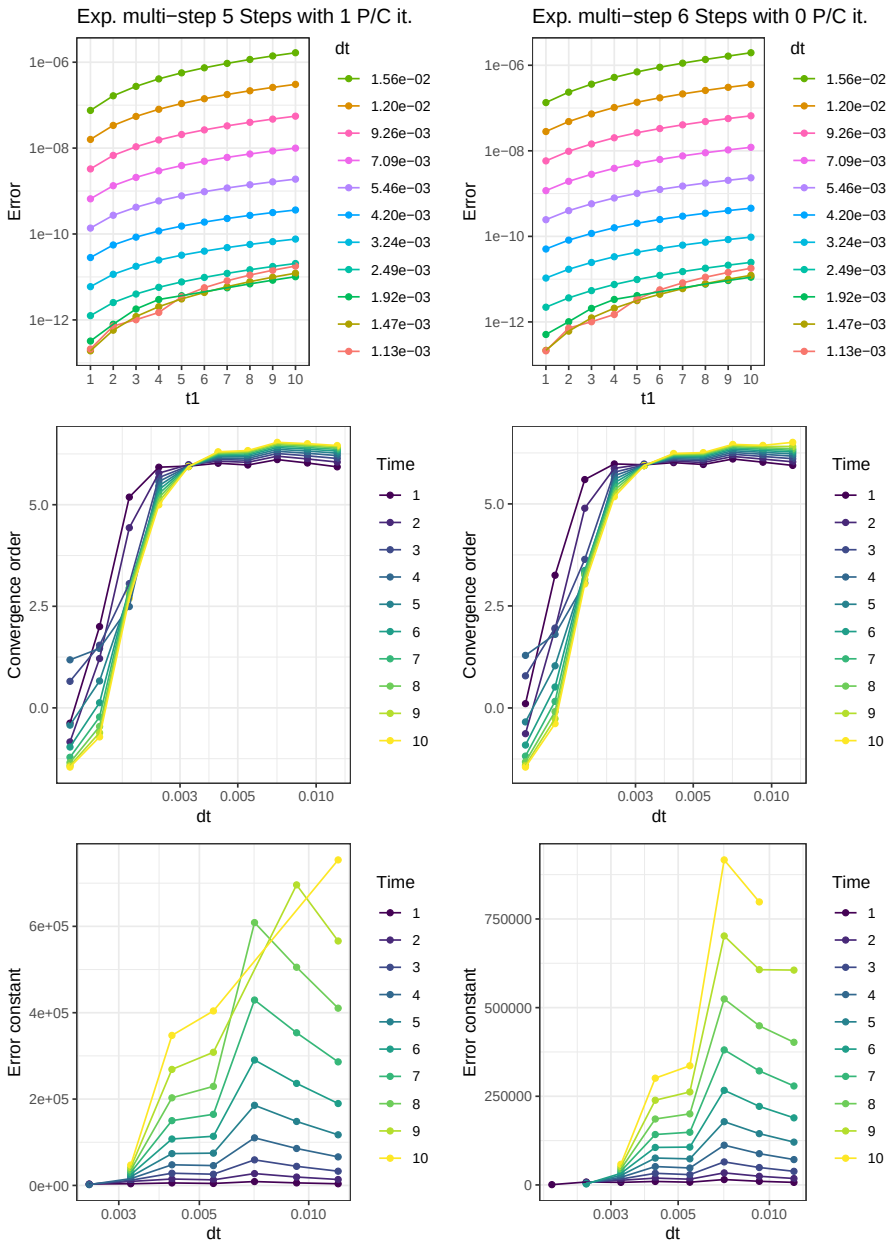

Figure A.1.8.: Change of observed convergence parameters for different dt and t_1 for 6th order Exponential multi-step methods for the cubic Schrödinger equation.

A.2. Helium experiment

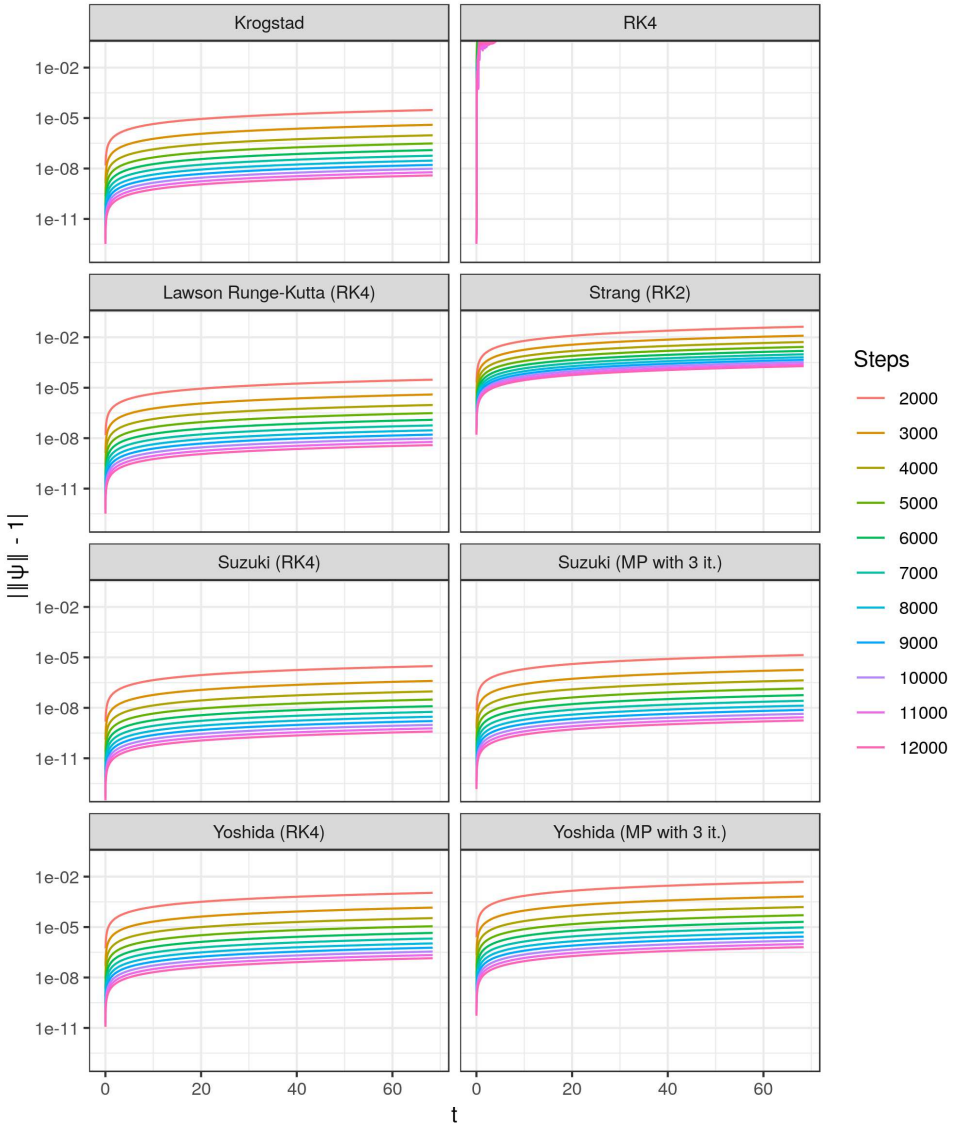

Figure A.2.9.: Stability comparison of the different one-step methods in the helium experiment ($N = 4$). A significant difference to the $N = 6$ experiment is the now stable behaviour of the Krogstad method.

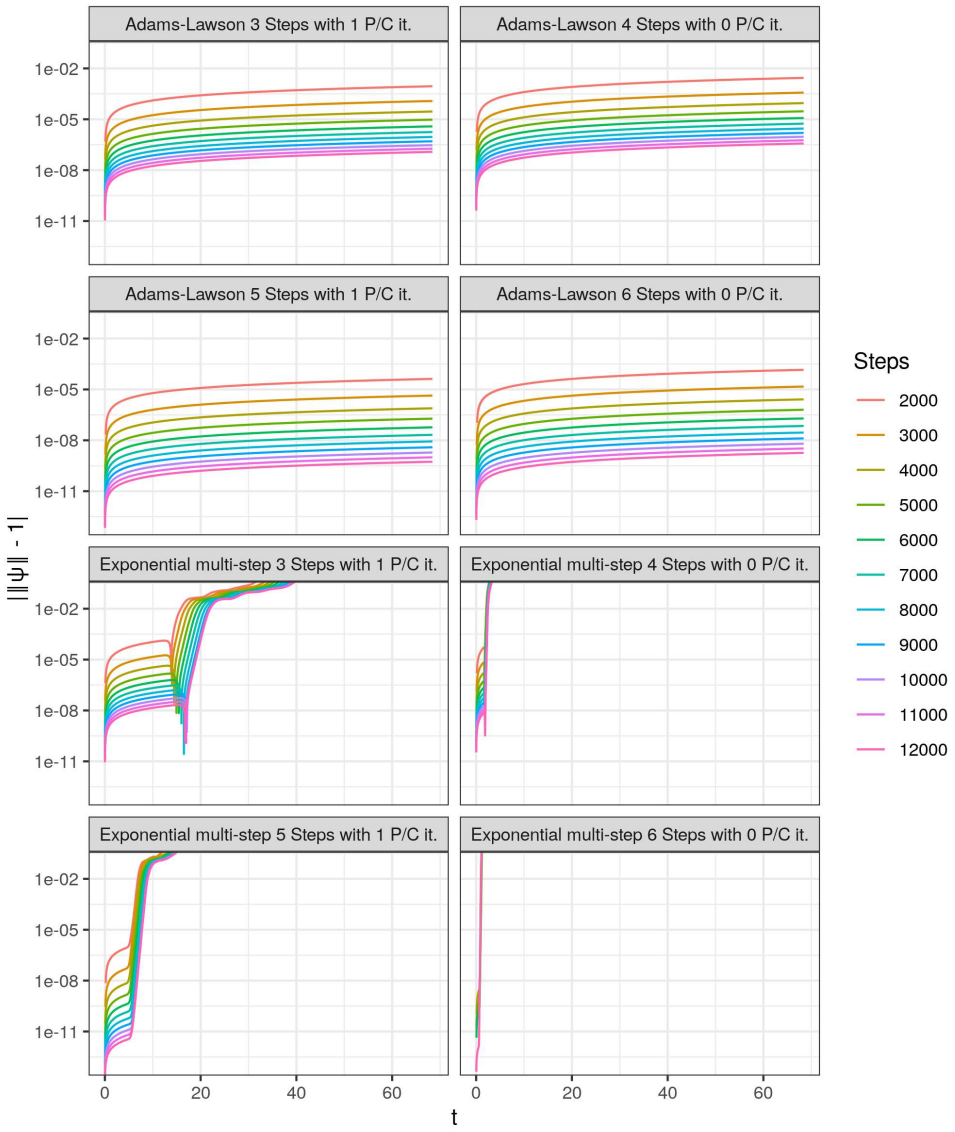

Figure A.2.10.: Stability comparison of the different multi-step methods in the helium experiment ($N = 4$). Compared to the $N = 6$ experiment, the only notable difference is the longer time interval until the blow-up for the exponential multi-step methods.

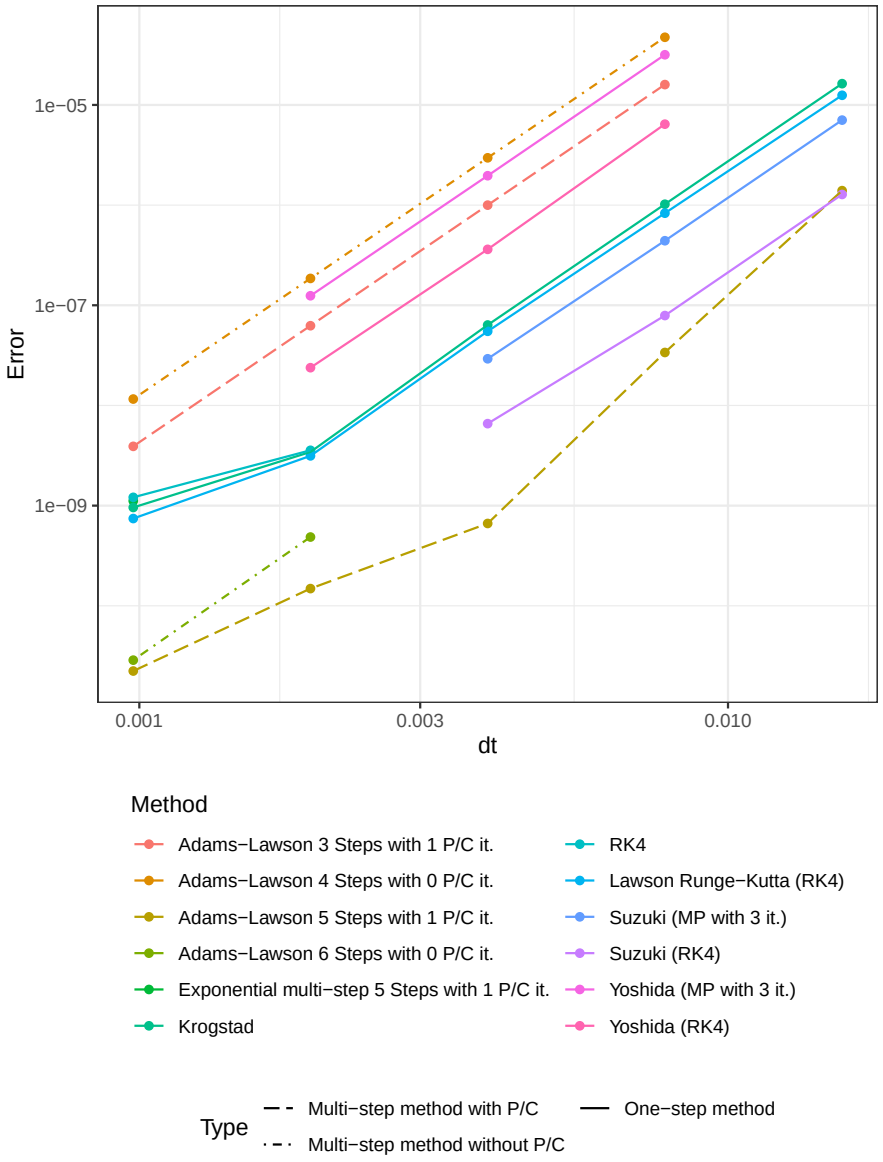

Figure A.2.11.: Precision comparison of all (viable) methods by step size in the helium experiment for $N = 4$. The convergence behaviour of the different methods is very systematic. The expected convergence order for the 6th order methods can only be observed for the Adams-Lawson method with 5 steps and P/C.

A.3. Quantum dot experiment

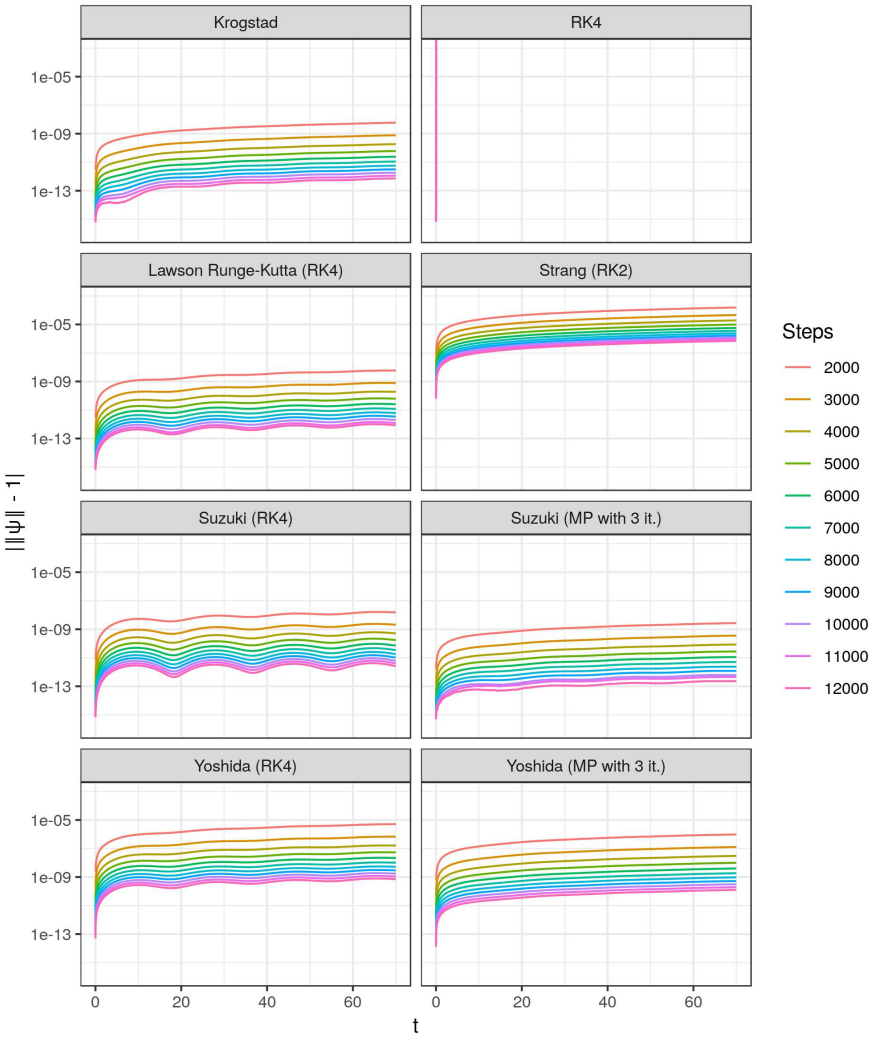

Figure A.3.12.: Stability behaviour of all one-step methods in the quantum dot experiment ($N = 6$).

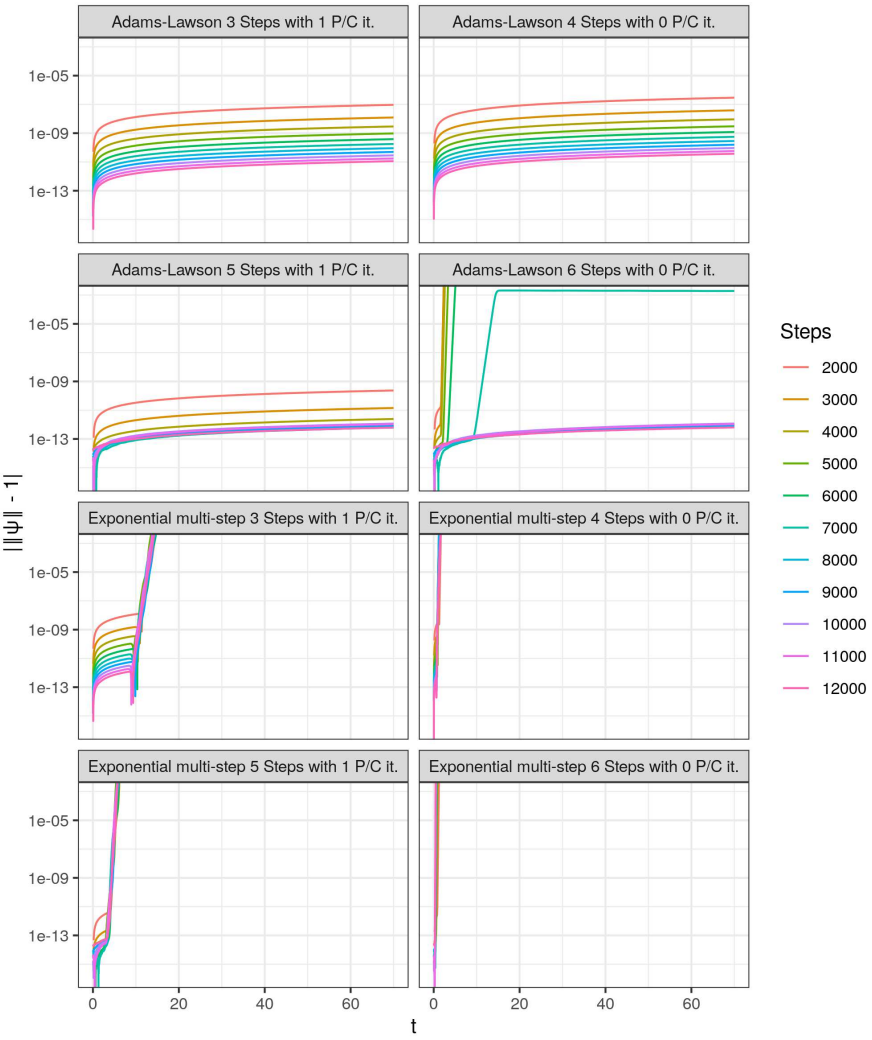

Figure A.3.13.: Stability behaviour of all multi step methods in the quantum dot experiment ($N = 6$).

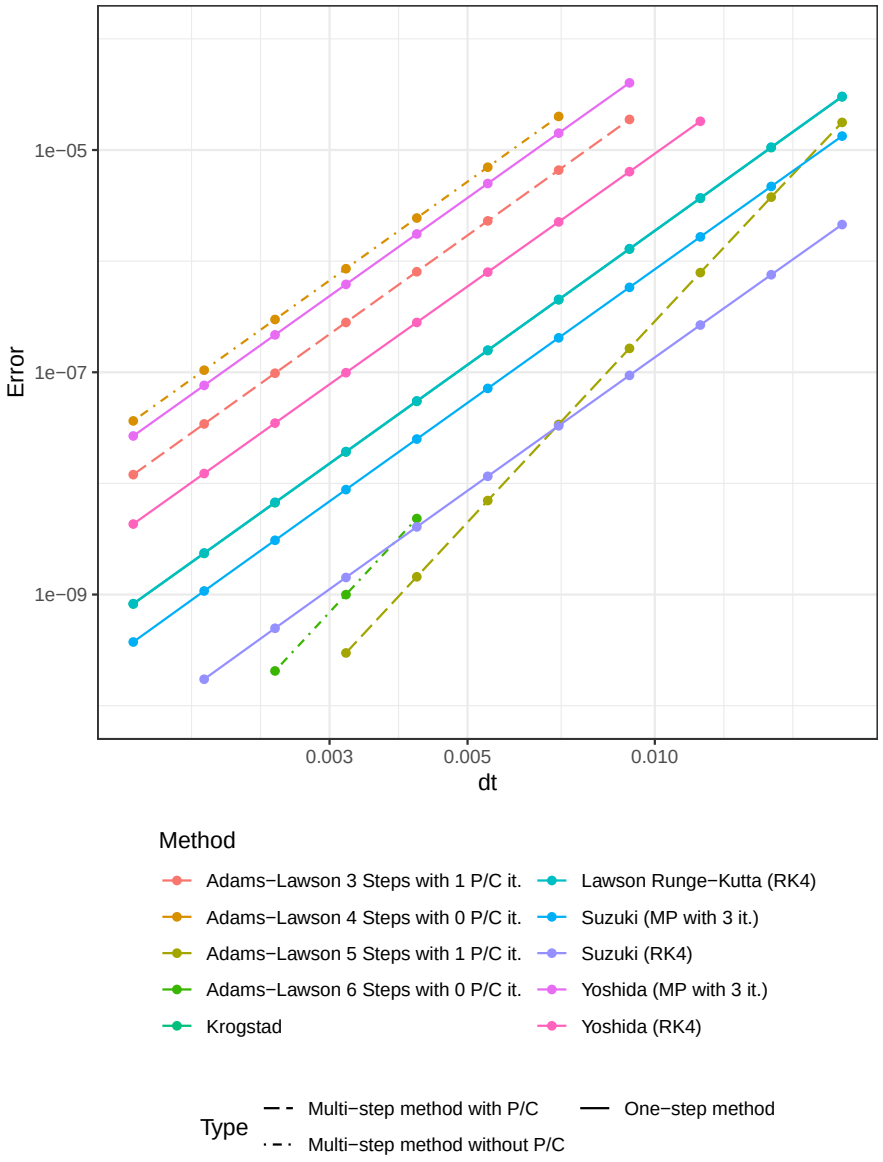

Figure A.3.14.: Precision comparison of all (viable) methods by step size in the quantum dot experiment for $N = 4$. The convergence behaviour of all displayed methods is very systematic and their order according to the expectation.

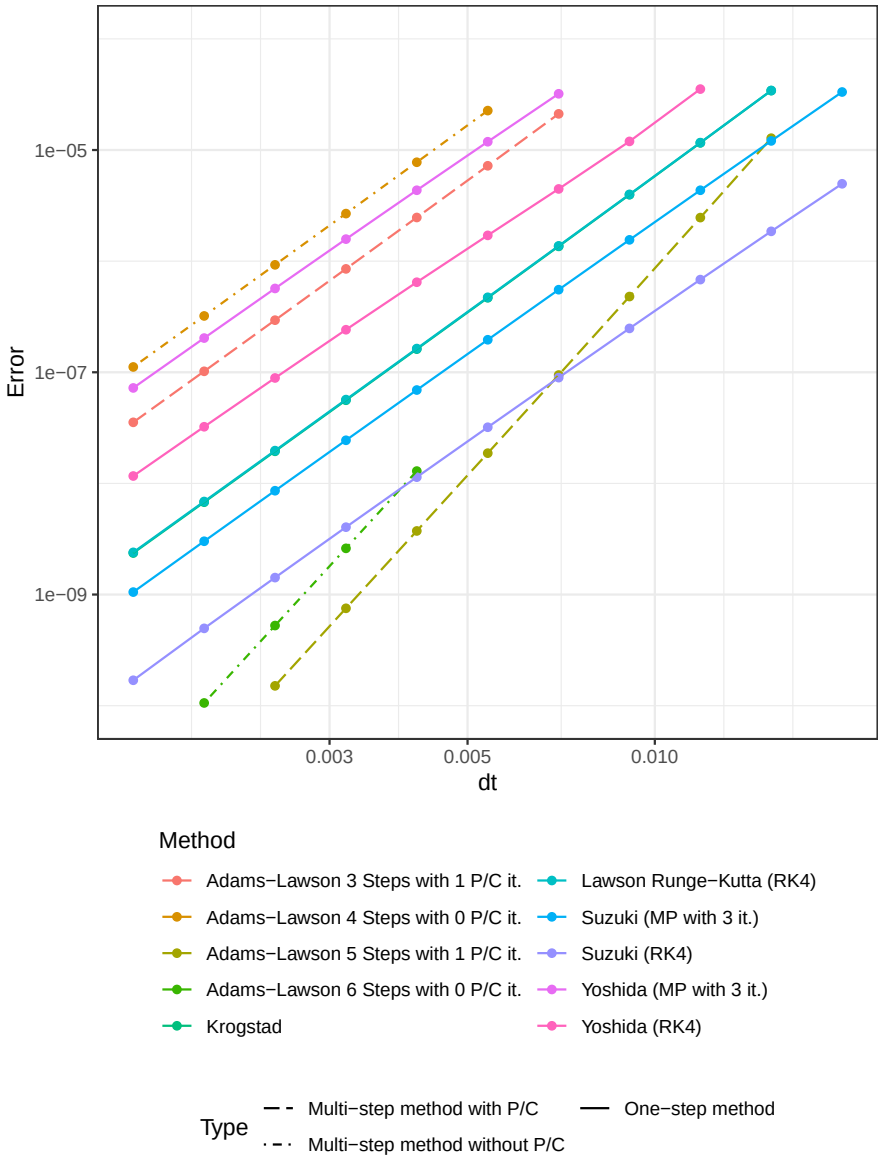

Figure A.3.15.: Precision comparison of all (viable) methods by step size in the quantum dot experiment for $N = 6$. The convergence behaviour of all displayed methods is very systematic and their order according to the expectation.

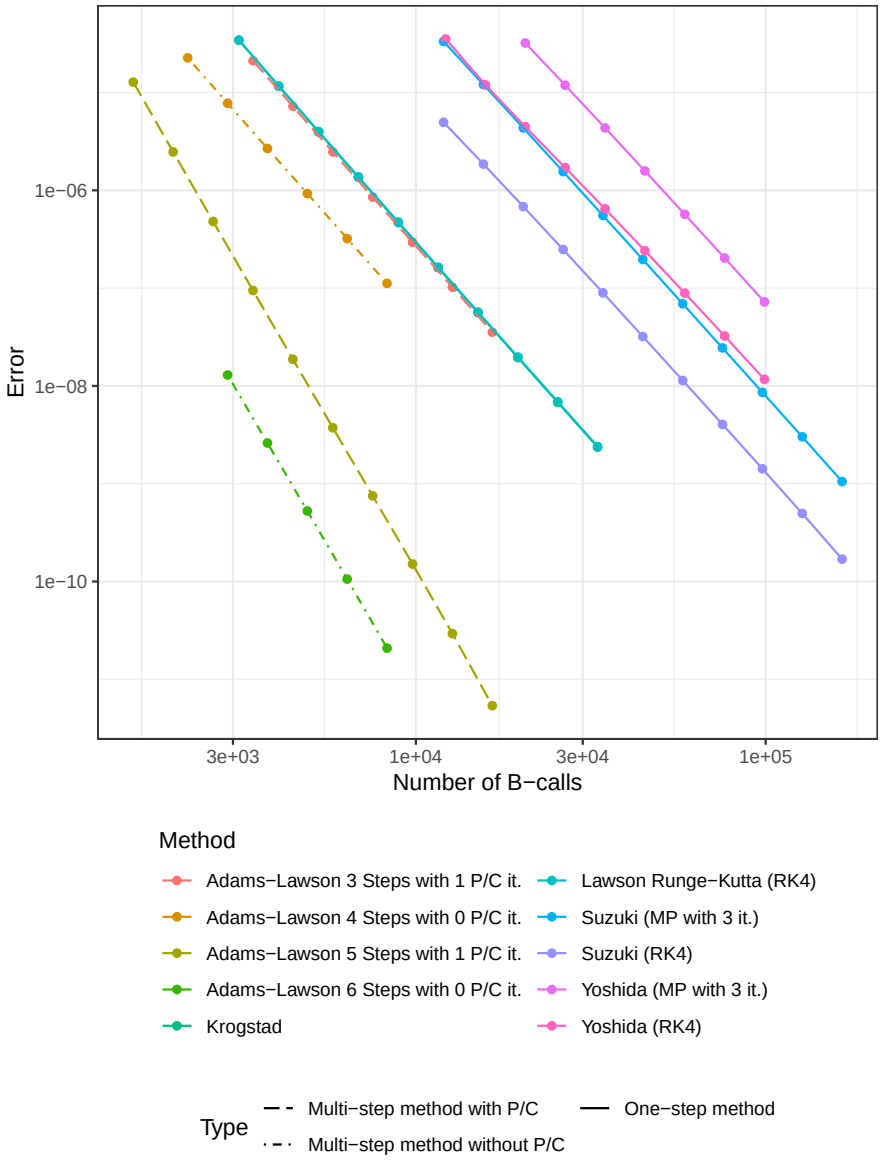

Figure A.3.16.: Comparison of all (viable) methods by number of B-calls in the quantum dot experiment ($N = 6$). As with the previous experiments, the higher order Adams-Lawson methods are much more efficient. Note the very precise overlay of the Krogstad and Lawson Runge-Kutta data points.

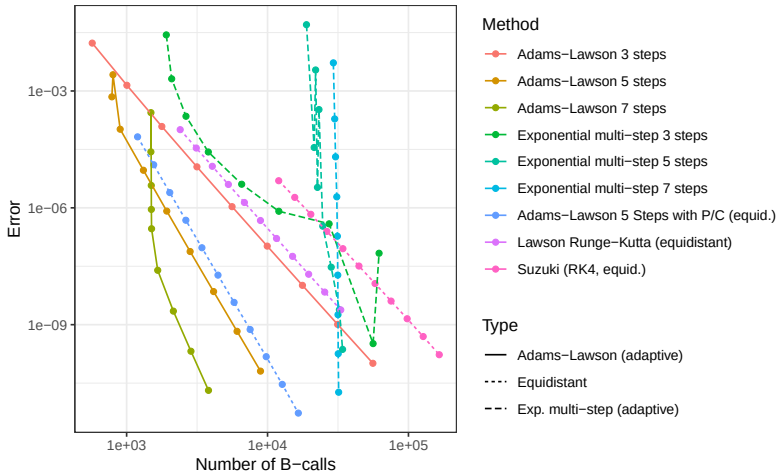

Figure A.3.17.: Comparison of adaptive methods by number of B-calls in the quantum dot experiment ($N = 6$). The data points have been created by prescribing a tolerance of $10^{-3}, \ldots, 10^{-11}$. The step size adaptivity produces small step sizes for the exponential multi-step methods using 5 or 7 steps, regardless of the tolerance.

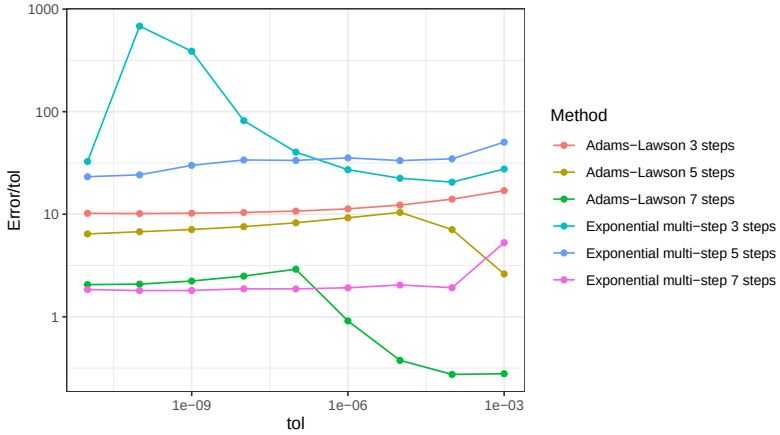

Figure A.3.18.: Comparison of adaptive methods using the quotient of $\frac{\text{Error}}{\text{tolerance}}$ in the quantum dot experiment ($N = 6$). We can see that the adaptive step size control manages to achieve the prescribed error tolerance only for the Adams-Lawson method where the methods tend to be more precise for a higher number of steps when using the same tolerance.

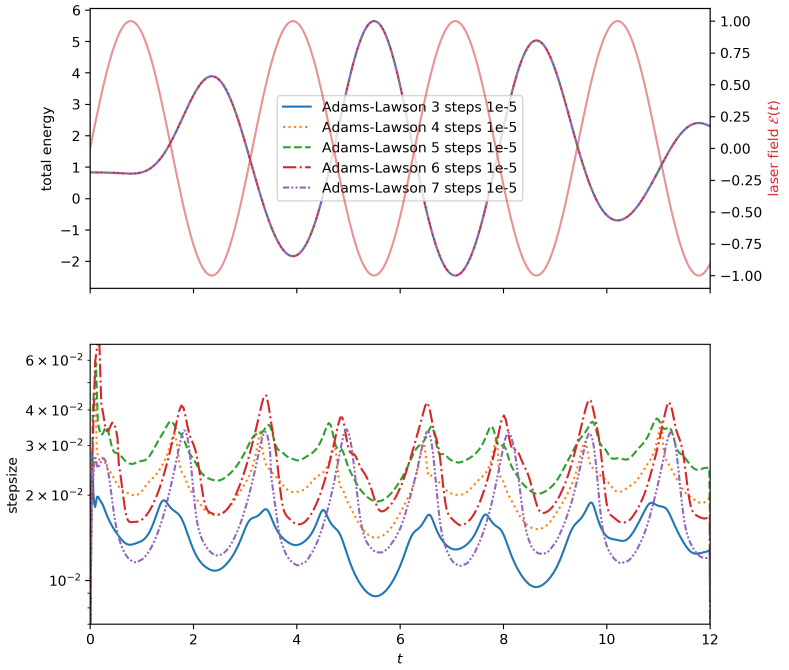

Figure A.3.19.: Visualisation of the step size change for the adaptive Adams-Lawson method using between 3 and 7 steps at tolerance 10^{-5} for the quantum dot experiment with $N = 6$. The local minima of the step sizes correspond to local extrema of the energy curve.

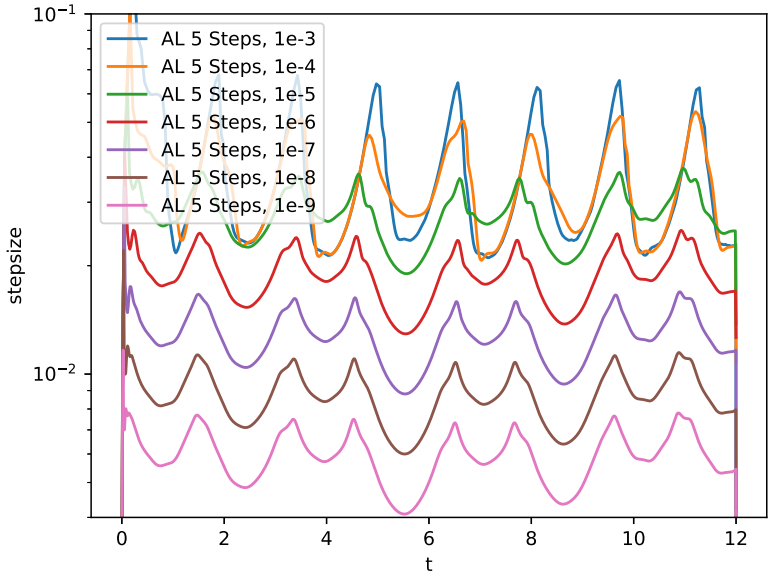

Figure A.3.20.: Visualisation of the step size change for different tolerances for the quantum dot experiment with $N = 6$. There is not much change in step size between high tolerances. From 10^{-6} and below we can see clear increments in the number of steps.

B. Code listings

B.1. Cubic Schrödinger equation

Listing B.1: gep_singlestep_4096.jl

```
using TSSM
TSSM_dir = joinpath(homedir(), ".julia/dev/TSSM")
include(joinpath(TSSM_dir, "examples/time_propagators.jl"))
include("global_error_prop.jl")
using DataFrames, CSV

# exact solution
function soliton(x, t)
    a = 2.0
    b = 1.0
    c = 0.0
    h = (a^2 - b^2)/2*t - b*x
    (a./cosh(a*(b*t+x-c))).*exp(1im*h)
end

nx = 2^12
xmin = -64
xmax = +64
V(x,t) = -0.5*abs(soliton(x,t)).^2
m = Schroedinger1D(nx, xmin, xmax, potential_t=V, cubic_coupling=-0.5)
psi = wave_function(m)

t0 = 0.0
tend = 10.0
results = DataFrame(Method = String[], Time = Float64[], Error = Float64
    [], dt = Float64[], B_Calls = Int[])

suzuki_comp_coeffs = [1/(4-4^(1/3)),
                      1/(4-4^(1/3)),
                      -4^(1/3)/(4-4^(1/3)),
                      1/(4-4^(1/3)),
                      1/(4-4^(1/3))]
yoshida_comp_coeffs = [1.351207191959657634,
                       -1.702414383919315268,
                       1.351207191959657634]
methods = [("strang", SplittingMethod([0.5, 0.5], [1.0, 0.0]))]#,
           ("strang_rk2", SplittingRK4BMethod([0.5, 0.5], [1.0, 0.0],
               secondorder=true)),
           ("suzuki", SplittingMethod(get_coeffs_composition(
               suzuki_comp_coeffs)...)),
           ("suzuki_rk2", SplittingRK4BMethod(get_coeffs_composition(
               suzuki_comp_coeffs)..., secondorder=true)),
```

```
            ("yoshida", SplittingMethod(get_coeffs_composition(
                yoshida_comp_coeffs)...)),
            ("yoshida_rk2", SplittingRK4BMethod(get_coeffs_composition(
                yoshida_comp_coeffs)..., secondorder=true))]
            ("suzuki_rk4", SplittingRK4BMethod(get_coeffs_composition(
                suzuki_comp_coeffs)...)),
            ("yoshida_rk4", SplittingRK4BMethod(get_coeffs_composition(
                yoshida_comp_coeffs)...)),
            ("suzuki_mp2it", CompositionMethod(suzuki_comp_coeffs,2)),
            ("yoshida_mp2it", CompositionMethod(yoshida_comp_coeffs,2)),
            ("suzuki_mp3it", CompositionMethod(suzuki_comp_coeffs,3)),
            ("yoshida_mp3it", CompositionMethod(yoshida_comp_coeffs,3)),
            ("suzuki_mp4it", CompositionMethod(suzuki_comp_coeffs,4)),
            ("yoshida_mp4it", CompositionMethod(yoshida_comp_coeffs,4)),
            ("krogstad", ExponentialRungeKutta(:krogstad)),
            ("rklawson", ExponentialRungeKutta(:lawson))]
println("Time interval [", t0, ",", tend, "]",
", space interval [", xmin, ",", xmax, "], nx=", nx)
for (method_name, method) in methods
    println("Method = ", method_name)
    for dt in 1 ./ Int.(round.(64 * 1.3 .^ (0:10)))
        println("dt = ", dt)
        global results
        set!(psi, soliton, t0)
        times, errs, bcalls = global_error_prop(method, psi, soliton, t0,
            tend, dt, steps=10, print_flag=true)
        results = vcat(results, DataFrame(Method=method_name, Time=times,
            Error=errs, dt=dt, B_Calls=bcalls))
        end
end

# different step sizes for RK4
for (method_name, method) in [("rk4", ExponentialRungeKutta(:rk4))]
    println("Method = ", method_name)
    for dt in 1 ./ Int.(round.(1778 * 1.0005 .^ (0:10)))
        println("dt = ", dt)
        global results
        set!(psi, soliton, t0)
        times, errs, bcalls = global_error_prop(method, psi, soliton, t0,
            2.0, dt, steps=2, print_flag=true)
        results = vcat(results, DataFrame(Method=method_name, Time=times,
            Error=errs, dt=dt, B_Calls=bcalls))
        end
end

CSV.write("results_singlestep_"*string(nx)*".csv", results)
```

Listing B.2: gep_multistep_4096.jl

```
using TSSM
TSSM_dir = joinpath(homedir(), ".julia/dev/TSSM")
include(joinpath(TSSM_dir, "examples/time_propagators.jl"))
include("global_error_prop.jl")
using DataFrames, CSV
# exact solution
function soliton(x, t)
```

```
    a = 2.0
    b = 1.0
    c = 0.0
    h = (a^2 - b^2)/2*t - b*x
    (a./cosh(a*(b*t+x-c))).*exp(1im*h)
end

nx = 2^12
xmin = -64
xmax = +64
V(x,t) = -0.5*abs(soliton(x,t)).^2
m = Schroedinger1D(nx, xmin, xmax, potential_t=V, cubic_coupling=-0.5)
psi = wave_function(m)

mutable struct ExactSolution <: TimePropagationMethod
end
function step!(m::ExactSolution, psi::WaveFunction,
                t0::Real, dt::Real, steps::Int, step::Int)
               set!(psi, soliton, t0+(step+1)*dt)
end

t0 = 0.0
tend = 10.0
results = DataFrame(Method = String[], Time = Float64[], Error = Float64
    [], dt = Float64[], B_Calls = Int[])

suzuki_comp_coeffs = [1/(4-4^(1/3)),
                      1/(4-4^(1/3)),
                      -4^(1/3)/(4-4^(1/3)),
                      1/(4-4^(1/3)),
                      1/(4-4^(1/3))]
yoshida_comp_coeffs = [1.351207191959657634,
                       -1.702414383919315268,
                       1.351207191959657634]
methods = [("expmulti_4_0", ExponentialMultistep(4, iters=0,
    starting_method=ExactSolution())),
           ("expmulti_3_1", ExponentialMultistep(3, iters=1,
               starting_method=ExactSolution())),
           ("expmulti_5_0", ExponentialMultistep(5, iters=0,
               starting_method=ExactSolution())),
           ("expmulti_4_1", ExponentialMultistep(4, iters=1,
               starting_method=ExactSolution())),
           ("expmulti_6_0", ExponentialMultistep(6, iters=0,
               starting_method=ExactSolution())),
           ("expmulti_5_1", ExponentialMultistep(5, iters=1,
               starting_method=ExactSolution())),
           ("adamslawson_4_0", ExponentialMultistep(4, iters=0, version
               =2, starting_method=ExactSolution())),
           ("adamslawson_3_1", ExponentialMultistep(3, iters=1, version
               =2, starting_method=ExactSolution())),
           ("adamslawson_5_0", ExponentialMultistep(5, iters=0, version
               =2, starting_method=ExactSolution())),
           ("adamslawson_4_1", ExponentialMultistep(4, iters=1, version
               =2, starting_method=ExactSolution())),
           ("adamslawson_6_0", ExponentialMultistep(6, iters=0, version
               =2, starting_method=ExactSolution())),
```

```
            ("adamslawson_5_1", ExponentialMultistep(5, iters=1, version
                =2, starting_method=ExactSolution()))
            ]
println("Time interval [", t0, ",", tend, "]",
", space interval [", xmin, ",", xmax, "], nx=", nx)
for (method_name, method) in methods
    println("Method = ", method_name)
    for dt in 1 ./ Int.(round.(64 * 1.3 .^ (0:10)))
        println("dt = ", dt)
        global results
        set!(psi, soliton, t0)
        times, errs, bcalls = global_error_prop(method, psi, soliton, t0,
             tend, dt, steps=10, print_flag = true)
        results = vcat(results,DataFrame(Method=method_name, Time=times,
            Error=errs, dt=dt, B_Calls=bcalls))
        end
end

CSV.write("results_multistep_"*string(nx)*"_rk4.csv", results)
```

Listing B.3: global_error_prop.jl

```
using Printf
using TSSM
function global_error_prop(method::TimePropagationMethod,
        psi::WaveFunction, reference_solution::Function,
       t0::Real, tend::Real, dt::Real; steps, print_flag=false)
    psi_ref = wave_function(psi.m)
    set!(psi_ref, reference_solution, t0)
    errs = Vector{Float64}()
    times = Vector{Int}()
    Bcount = Vector{Int}()
    wf_save_initial_value = clone(psi)
    copy!(wf_save_initial_value, psi)

    total_steps = Int(floor((tend-t0)/dt))
        step_times = Vector(1:steps)*(tend-t0)/steps
        i = 1
        for (step, tsi) in EquidistantTimeStepper(method, psi, get_time(
            psi), dt, total_steps)
                t = get_time(tsi.psi)
                if t-dt/2 <= step_times[i] < t+dt/2
                        i+=1
                        set!(psi_ref, reference_solution, t)
                        err = distance(psi, psi_ref)
                        append!(errs, err)
                        append!(times, Int(round(t)))
                        append!(Bcount, COUNT_B)
                        if print_flag
                                @printf("%12.6e   %2.2f\n", err, get_time
                                    (psi))
                        end
                end
        end
        return times, errs, Bcount
end
```

B.2. Helium experiment

Listing B.4: helium_stability.jl

```
using TSSM
using JLD
TSSM_dir = joinpath(homedir(), ".julia/dev/TSSM")
include(joinpath(TSSM_dir, "MCTDHF/mctdhf1d.jl"))
include(joinpath(TSSM_dir, "MCTDHF/check.jl"))
#include(joinpath(TSSM_dir, "MCTDHF/propagators.jl"))
include(joinpath(TSSM_dir, "examples/time_propagators.jl"))

V1(x) = -2/sqrt(x^2+0.7408^2)
V2(x,y) = 1/sqrt((x-y)^2+0.7408^2)
V(x,y) = V1(x) + V1(y) + V2(x,y)

function create_groundstate(N)
    m = MCTDHF1D(2, N, 512, -16, 16, potential1=V1, potential2=V2,
        spin_restricted=true)
    psi = wave_function(m)
    groundstate!(psi, dt=0.1, max_iter=10000, output_step=10, tol=1e-5)
    TSSM.save(psi, "wf_f2_n"*string(N)*"_nx512_t0.h5")

    return psi
end

function time_propagation(variant::String, steps::Int)
    N = 6
    T = 2*pi/0.1837

    t0=0
    tend=2*T
    dt=tend/steps
    steps=steps

    times = Vector{Float64}()
    energies = Vector{Float64}()
    norms = Vector{Float64}()

    V1_t(x, t) = 0
    m = MCTDHF1D(2, N, 512, -16, 16, potential1=V1, potential2=V2,
        spin_restricted=true);
    psi = wave_function(m);

    #create_groundstate(N)
    TSSM.load!(psi, "wf_f2_n"*string(N)*"_nx512_t0.h5")
    println(steps, "␣steps␣to␣calculate")
    set_time!(psi, t0)
    time0 = time()
    k = 1
    told=t0
    if variant == "yoshidaRK4"
        g = [1.351207191959657634, -1.702414383919315268,
            1.351207191959657634] # Yoshida
        a, b = get_coeffs_composition(g)
        method = SplittingRK4BMethod(a,b)
```

```
    println("yoshida splitting")
end
if variant == "yoshidaMP"
    g = [1.351207191959657634, -1.702414383919315268,
        1.351207191959657634] # Yoshida
    method = CompositionMethod(g, 3)
    println("yoshida splitting using MP")
end
if variant == "suzukiRK4"
    g = [1/(4-4^(1/3)),1/(4-4^(1/3)),-4^(1/3)/(4-4^(1/3)),
        1/(4-4^(1/3)), 1/(4-4^(1/3))] # Suzuki
    a, b = get_coeffs_composition(g)
    method = SplittingRK4BMethod(a,b)
    println("suzuki splitting")
end
if variant == "suzukiMP"
    g = [1/(4-4^(1/3)),1/(4-4^(1/3)),-4^(1/3)/(4-4^(1/3)),
        1/(4-4^(1/3)), 1/(4-4^(1/3))] # Suzuki
    method = CompositionMethod(g, 3)
    println("suzuki splitting using MP")
end
if variant == "strang"
    a, b = [0.5, 0.5], [1.0, 0.0]
    method = SplittingRK4BMethod(a, b, secondorder=true)
    println("strang splitting")
end
if variant == "krogstad"
    method = ExponentialRungeKutta(:krogstad)
    println("krogstad")
end
if variant == "rk4"
    method = ExponentialRungeKutta(:rk4)
    println("rk4")
end
if variant == "rklawson"
    method = ExponentialRungeKutta(:lawson)
    println("runge kutta lawson")
end
if variant == "adamslawson31"
    method = ExponentialMultistep(3, version=2, iters=1,
        starting_method=ExponentialRungeKutta(:lawson),
        starting_subdivision=50)
    println("adamslawson 3 + 1it")
end
if variant == "adamslawson40"
    method = ExponentialMultistep(4, version=2, iters=0,
        starting_method=ExponentialRungeKutta(:lawson),
        starting_subdivision=50)
    println("adamslawson 4 + 0it")
end
if variant == "expmultistep31"
    method = ExponentialMultistep(3, version=1, iters=1,
        starting_method=ExponentialRungeKutta(:lawson),
        starting_subdivision=50)
    println("exponential multistep 3 + 1it")
end
```

```
    if variant == "expmultistep40"
        method = ExponentialMultistep(4, version=1, iters=0,
            starting_method=ExponentialRungeKutta(:lawson),
            starting_subdivision=50)
        println("exponential multistep 4 + 0it")
    end

    if variant == "adamslawson51"
        method = ExponentialMultistep(5, version=2, iters=1,
            starting_method=ExponentialRungeKutta(:lawson),
            starting_subdivision=50)
        println("adamslawson 5 + 1it")
    end
    if variant == "adamslawson60"
        method = ExponentialMultistep(6, version=2, iters=0,
            starting_method=ExponentialRungeKutta(:lawson),
            starting_subdivision=50)
        println("adamslawson without 6 + 0it")
    end
    if variant == "expmultistep51"
        method = ExponentialMultistep(5, version=1, iters=1,
            starting_method=ExponentialRungeKutta(:lawson),
            starting_subdivision=50)
        println("exponential multistep 5 + 1it")
    end
    if variant == "expmultistep60"
        method = ExponentialMultistep(6, version=1, iters=0,
            starting_method=ExponentialRungeKutta(:lawson),
            starting_subdivision=50)
        println("exponential multistep without 6 + 0it")
    end

    for (step, tsi) in EquidistantTimeStepper(method, psi, t0, dt, steps)
        n = norm(psi)
        E_pot = potential_energy(psi)
        E_kin = kinetic_energy(psi)
        E_tot = E_pot+E_kin
        norm_psi = norm(psi)
        @printf("%5i  %14.10f  %14.10f  %14.10f  %14.10f  %14.10f  %14.10
            f  %14.10f  %10.2f\n",
            k, get_time(psi), n, E_pot, E_kin, E_tot, norm_psi, dt, time
                ()-time0)
        append!(norms, norm_psi)
        append!(times, get_time(psi))
        append!(energies, E_tot)
        k += 1
    end
    JLD.save("results/wf_f2_n"*string(N)*"_nx"*string(nx)*"_"*variant*"_"
        *string(steps)*"steps_stability.jld", "times", times, "energies",
         energies, "dt", dt, "norms", norms)
    return
end
time_propagation(ARGS[1], parse(Int, ARGS[2]))
```

Listing B.5: helium_conv.jl

```
using TSSM
using JLD
TSSM_dir = joinpath(homedir(), ".julia/dev/TSSM")
include(joinpath(TSSM_dir, "MCTDHF/mctdhf1d.jl"))
include(joinpath(TSSM_dir, "MCTDHF/check.jl"))
#include(joinpath(TSSM_dir, "MCTDHF/propagators.jl"))
include(joinpath(TSSM_dir, "examples/time_propagators.jl"))

V1(x) = -2/sqrt(x^2+0.7408^2)
V2(x,y) = 1/sqrt((x-y)^2+0.7408^2)
V(x,y) = V1(x) + V1(y) + V2(x,y)

function create_groundstate(N, tol)
    m = MCTDHF1D(2, N, 512, -16, 16, potential1=V1, potential2=V2,
        spin_restricted=true)
    psi = wave_function(m)
    groundstate!(psi, dt=0.1, max_iter=10000, output_step=10, tol=tol)
    TSSM.save(psi, "wf_f2_n"*string(N)*"_nx512_t0"*string(tol)*".h5")

    return psi
end

# gaussian envelope
function f(t)
    cycle = 2*pi/0.1837
    return 1.2*exp(-5e-4*(t-3*cycle)^2)
end

yoshida_coefficients = [1.351207191959657634, -1.702414383919315268,
    1.351207191959657634]
suzuki_coefficients = [1/(4-4^(1/3)),1/(4-4^(1/3)),-4^(1/3)/(4-4^(1/3)),
    1/(4-4^(1/3)), 1/(4-4^(1/3))]

function time_propagation(variant::String, half_exp::Int)
    N = 4
    boundary = 256
    nx = boundary * 32
    println("[-", boundary, ",", boundary,"], nx = ", nx)
    times = Vector{Float64}()
    energies = Vector{Float64}()
    #laser field
    E(t) = 0.1894*f(t)*sin(0.1837*t)

    V1_t(x, t) = E(t)*x
    V_t(x,y,t) = E(t)*(x+y)

    m = MCTDHF1D(2, N, nx, -boundary, boundary, potential1=V1,
        potential1_t=V1_t, potential2=V2,
        spin_restricted=true);
    psi = wave_function(m);
    psi_ref = wave_function(m);
    T = 2*pi/0.1837

    t0=0
    tend=80
    dt=0.5^half_exp
```

```
#create_groundstate(N, tol)
TSSM.load!(psi, "heliumwf_f2_n"*string(N)*"_nx512_t0_1.0e-9.h5")
set_time!(psi, t0)
time0 = time()
k = 1
told=t0
if variant == "yoshidaRK4"
    g = [1.351207191959657634, -1.702414383919315268,
        1.351207191959657634] # Yoshida
    a, b = get_coeffs_composition(g)
    method = SplittingRK4BMethod(a,b)
    println("yoshida splitting")
end
if variant == "yoshidaMP"
    g = [1.351207191959657634, -1.702414383919315268,
        1.351207191959657634] # Yoshida
    method = CompositionMethod(g, 3)
    println("yoshida splitting using MP")
end
if variant == "suzukiRK4"
    g = [1/(4-4^(1/3)),1/(4-4^(1/3)),-4^(1/3)/(4-4^(1/3)),
        1/(4-4^(1/3)), 1/(4-4^(1/3))] # Suzuki
    a, b = get_coeffs_composition(g)
    method = SplittingRK4BMethod(a,b)
    println("suzuki splitting")
end
if variant == "suzukiMP"
    g = [1/(4-4^(1/3)),1/(4-4^(1/3)),-4^(1/3)/(4-4^(1/3)),
        1/(4-4^(1/3)), 1/(4-4^(1/3))] # Suzuki
    method = CompositionMethod(g, 3)
    println("suzuki splitting using MP")
end
if variant == "strang"
    a, b = [0.5, 0.5], [1.0, 0.0]
    method = SplittingRK4BMethod(a, b, secondorder=true)
    println("strang splitting")
end
if variant == "krogstad"
    method = ExponentialRungeKutta(:krogstad)
    println("krogstad")
end
if variant == "rk4"
    method = ExponentialRungeKutta(:rk4)
    println("rk4")
end
if variant == "rklawson"
    method = ExponentialRungeKutta(:lawson)
    println("runge kutta lawson")
end

if variant == "adamslawson31"
    method = ExponentialMultistep(3, version=2, iters=1,
        starting_method=ExponentialRungeKutta(:lawson),
        starting_subdivision=50)
    println("adamslawson 3 + 1it")
end
```

```
if variant == "adamslawson40"
    method = ExponentialMultistep(4, version=2, iters=0,
        starting_method=ExponentialRungeKutta(:lawson),
        starting_subdivision=50)
    println("adamslawson 4 + 0it")
end
if variant == "expmultistep31"
    method = ExponentialMultistep(3, version=1, iters=1,
        starting_method=ExponentialRungeKutta(:lawson),
        starting_subdivision=50)
    println("exponential multistep 3 + 1it")
end
if variant == "expmultistep40"
    method = ExponentialMultistep(4, version=1, iters=0,
        starting_method=ExponentialRungeKutta(:lawson),
        starting_subdivision=50)
    println("exponential multistep 4 + 0it")
end

if variant == "adamslawson51"
    method = ExponentialMultistep(5, version=2, iters=1,
        starting_method=ExponentialRungeKutta(:lawson),
        starting_subdivision=50)
    println("adamslawson 5 + 1it")
end
if variant == "adamslawson60"
    method = ExponentialMultistep(6, version=2, iters=0,
        starting_method=ExponentialRungeKutta(:lawson),
        starting_subdivision=50)
    println("adamslawson without 6 + 0it")
end
if variant == "expmultistep51"
    method = ExponentialMultistep(5, version=1, iters=1,
        starting_method=ExponentialRungeKutta(:lawson),
        starting_subdivision=50)
    println("exponential multistep 5 + 1it")
end
if variant == "expmultistep60"
    method = ExponentialMultistep(6, version=1, iters=0,
        starting_method=ExponentialRungeKutta(:lawson),
        starting_subdivision=50)
    println("exponential multistep without 6 + 0it")
end

for (step, tsi) in EquidistantTimeStepper(method, psi, t0, dt, Int((
    tend-t0)/dt))
    n = norm(psi)
    E_pot = potential_energy(psi)
    E_kin = kinetic_energy(psi)
    E_tot = E_pot+E_kin
    if k % 10 == 0
        @printf("%5i %14.10f %14.10f %14.10f %14.10f %6i %10.2f\n",
            k, get_time(psi), E_tot, dt, n, COUNT_B, time()-time0)
    end
    append!(times, get_time(psi))
    append!(energies, E_tot)
```

```
        k += 1
        if n > 1e10
            break
        end
    end
    time1 = time()-time0
    println("Finished after ", time1, " seconds at t = ", get_time(psi))
    if isapprox(get_time(psi), tend)
        TSSM.save(psi, "results_conv_"*string(nx)*
            "/heliumwf_f2_n"*string(N)*"_nx"*string(nx)*"_t"*string(tend)
                *
            "_"*variant*"_0.5^"*string(half_exp)*".h5")
    end
    JLD.save("results_conv_"*string(nx)*
        "/heliumwf_f2_n"*string(N)*"_nx"*string(nx)*"_t"*string(tend)*
        "_"*variant*"_0.5^"*string(half_exp)*".jld",
        "times", times, "energies", energies, "dt", dt,
        "B_calls", COUNT_B, "runtime", time1)
    return
end
#                variant  halfexp (dt = 0.5^halfexp)
time_propagation(ARGS[1], parse(Int, ARGS[2]))
```

Listing B.6: helium_adaptive.jl

```
using TSSM
using JLD
TSSM_dir = joinpath(homedir(), ".julia/dev/TSSM")
include(joinpath(TSSM_dir, "MCTDHF/mctdhf1d.jl"))
include(joinpath(TSSM_dir, "MCTDHF/check.jl"))
#include(joinpath(TSSM_dir, "MCTDHF/propagators.jl"))
include(joinpath(TSSM_dir, "examples/time_propagators.jl"))

V1(x) = -2/sqrt(x^2+0.7408^2)
V2(x,y) = 1/sqrt((x-y)^2+0.7408^2)
V(x,y) = V1(x) + V1(y) + V2(x,y)

function create_groundstate(N, tol)
    m = MCTDHF1D(2, N, 512, -16, 16, potential1=V1, potential2=V2,
        spin_restricted=true)
    psi = wave_function(m)
    groundstate!(psi, dt=0.1, max_iter=10000, output_step=10, tol=tol)
    TSSM.save(psi, "wf_f2_n"*string(N)*"_nx512_t0"*string(tol)*".h5")

    return psi
end

# gaussian envelope
function f(t)
    cycle = 2*pi/0.1837
    return 1.2*exp(-5e-4*(t-3*cycle)^2)
end

function time_propagation(version::Int, steps::Int, tolexp::Int, N::Int,
    int_bound::Int)
    times = Vector{Float64}()
```

```
    energies = Vector{Float64}()
    stepsizes = Vector{Float64}()
    #laser field
    E(t) = 0.1894*f(t)*sin(0.1837*t)

    V1_t(x, t) = E(t)*x
    V_t(x,y,t) = E(t)*(x+y)

    nx = int_bound*32
    m = MCTDHF1D(2, N, nx, -int_bound, int_bound, potential1=V1,
        potential1_t=V1_t, potential2=V2, spin_restricted=true);
    psi = wave_function(m);

    T = 2*pi/0.1837

    # Part 1
    t0=0
    tend=250
    #tend=0.2
    tol=10.0^(-tolexp)
    #create_groundstate(N, tol)
    TSSM.load!(psi, "heliumwf_f2_n"*string(N)*"_nx512_t0_1.0e-9.h5")
    set_time!(psi, t0)
    time0 = time()
    k = 1
    told=t0
    method = AdaptiveAdamsLawson(steps, version=version)
    save_step = 10
    for (t, tsi) in AdaptiveTimeStepper(method, psi, t0, tend, tol,
        0.00001)
        n = norm(psi)
        E_pot = potential_energy(psi)
        E_kin = kinetic_energy(psi)
        E_tot = E_pot+E_kin
        stepsize = t - told
        @printf("%5i␣␣%14.10f␣␣%14.10f␣␣%14.10f␣␣%14.10f␣␣%10.2f␣%6i\n",
            k, get_time(psi), n, E_tot, stepsize, time()-time0, COUNT_B)
        append!(times, get_time(psi))
        append!(energies, E_tot)
        append!(stepsizes, stepsize)
        told = t
        k += 1
    end
    TSSM.save(psi, "results_adaptive_t"*string(tend)*"/heliumwf_f2_n"*
        string(N)*"_nx"*string(nx)*"_v"*string(version)*"_steps"*string(
        steps)*"_results_t"*string(tend)*"_tol"*string(tol)*".h5")
    JLD.save("results_adaptive_t"*string(tend)*"/heliumwf_f2_n"*string(N)
        *"_nx"*string(nx)*"_v"*string(version)*"_steps"*string(steps)*"
        _results_t"*string(tend)*"_tol"*string(tol)*".jld", "times",
        times, "energies",
        energies, "stepsizes", stepsizes, "B_calls", COUNT_B, "steps",
            steps)
    return
end

using ArgParse
```

```
s = ArgParseSettings()
@add_arg_table s begin
    "--steps"
        help = "Number of steps for the method"
        arg_type = Int
        default = 5
    "--variant"
        help = "Set 1 for Exp. Multistep and 2 for Adams-Lawson method."
        arg_type = Int
        default = 2
    "--tol"
        help = "Prescribe tolerance for adaptive methods. Give positive
            exponent as argument (1e-[tol] will be used)"
        arg_type = Int
        default = 5
    "--N"
        arg_type = Int
        default = 4
    "--int_size"
        help = "Size of the spatial interval (symmetric, give upper bound
            as integer"
        arg_type = Int
        default = 256
end

parsed_args = parse_args(s)
v = parsed_args["variant"]
steps = parsed_args["steps"]
tolexp = parsed_args["tol"]
N = parsed_args["N"]
interval = parsed_args["int_size"]
println("version=", v, ", steps = ", steps,  ", tol = 10^-", tolexp, ", N
    ", N, ", interval = [-", interval, ",", interval, "]")
time_propagation(v, steps, tolexp, N, interval)
```

B.3. Quantum dot experiment

Listing B.7: quantum_dot.jl

```
# Parse parameters
using ArgParse
s = ArgParseSettings()
@add_arg_table s begin
    "--exp"
        help = "Specify the experiment to run (groundstate, stab, conv, adapt)."
        required = true
    "--method"
        help = "Specify the method to use."
        arg_type = String
        required = true
    "--N"
        arg_type = Int
        default = 4
    "--steps"
        help = "Number of steps used in the time interval [0,12]."
        arg_type = Int
        default = 1000
    "--no-ref"
        help = "Don't use reference solution in conv experiment, instead just save resulting wf."
        action = :store_true
    "--tol"
        help = "Tolerance for the adaptivity experiment"
        arg_type = Float64
        default = 1e-5
end

parsed_args = parse_args(s)
const method_string = parsed_args["method"]
const steps = parsed_args["steps"]
const N = parsed_args["N"]
const experiment = parsed_args["exp"]
const adaptive_tol = parsed_args["tol"]

# Run experiment
using TSSM
using JLD
TSSM_dir = joinpath(homedir(), ".julia/dev/TSSM")
include(joinpath(TSSM_dir, "MCTDHF/mctdhf1d.jl"))
include(joinpath(TSSM_dir, "MCTDHF/check.jl"))
include(joinpath(TSSM_dir, "examples/time_propagators.jl"))

V1(x) = 1. / 32 * (x^2)
V2(x,y) = 1/sqrt((x-y)^2+1/16)
V(x,y) = V1(x) + V1(y) + V2(x,y)

const interval_bound = 20
const nx = Int(512*interval_bound/10)
const t0 = 0
const groundstate_tol=1e-9
```

```
SHIFT_B = 0

println("N = ", N, ", nx = ", nx, " on [-", interval_bound, ",",
    interval_bound, "]")

function method_from_string(variant::String)
    if variant == "yoshidaRK4"
        g = [1.351207191959657634, -1.702414383919315268,
            1.351207191959657634] # Yoshida
        a, b = get_coeffs_composition(g)
        method = SplittingRK4BMethod(a,b)
        println("yoshida splitting")
    elseif variant == "yoshidaMP"
        g = [1.351207191959657634, -1.702414383919315268,
            1.351207191959657634] # Yoshida
        method = CompositionMethod(g, 3)
        println("yoshida splitting using MP")
    elseif variant == "suzukiRK4"
        g = [1/(4-4^(1/3)),1/(4-4^(1/3)),-4^(1/3)/(4-4^(1/3)),
            1/(4-4^(1/3)), 1/(4-4^(1/3))] # Suzuki
        a, b = get_coeffs_composition(g)
        method = SplittingRK4BMethod(a,b)
        println("suzuki splitting")
    elseif variant == "suzukiMP"
        g = [1/(4-4^(1/3)),1/(4-4^(1/3)),-4^(1/3)/(4-4^(1/3)),
            1/(4-4^(1/3)), 1/(4-4^(1/3))] # Suzuki
        method = CompositionMethod(g, 3)
        println("suzuki splitting using MP")
    elseif variant == "strang"
        a, b = [0.5, 0.5], [1.0, 0.0]
        method = SplittingRK4BMethod(a, b, secondorder=true)
        println("strang splitting")
    elseif variant == "krogstad"
        method = ExponentialRungeKutta(:krogstad)
        println("krogstad")
    elseif variant == "rk4"
        method = ExponentialRungeKutta(:rk4)
        println("rk4")
    elseif variant == "rklawson"
        method = ExponentialRungeKutta(:lawson)
        println("runge kutta lawson")

    elseif variant == "adamslawson31"
        method = ExponentialMultistep(3, version=2, iters=1,
            starting_method=ExponentialRungeKutta(:lawson),
            starting_subdivision=50)
        println("adamslawson 3 + 1it")
    elseif variant == "adamslawson40"
        method = ExponentialMultistep(4, version=2, iters=0,
            starting_method=ExponentialRungeKutta(:lawson),
            starting_subdivision=50)
        println("adamslawson 4 + 0it")
    elseif variant == "expmultistep31"
        method = ExponentialMultistep(3, version=1, iters=1,
            starting_method=ExponentialRungeKutta(:lawson),
```

```
            starting_subdivision=50)
        println("exponential multistep 3 + 1it")
    elseif variant == "expmultistep40"
        method = ExponentialMultistep(4, version=1, iters=0,
            starting_method=ExponentialRungeKutta(:lawson),
            starting_subdivision=50)
        println("exponential multistep 4 + 0it")

    elseif variant == "adamslawson51"
        method = ExponentialMultistep(5, version=2, iters=1,
            starting_method=ExponentialRungeKutta(:lawson),
            starting_subdivision=50)
        println("adamslawson 5 + 1it")
    elseif variant == "adamslawson60"
        method = ExponentialMultistep(6, version=2, iters=0,
            starting_method=ExponentialRungeKutta(:lawson),
            starting_subdivision=50)
        println("adamslawson without 6 + 0it")
    elseif variant == "expmultistep51"
        method = ExponentialMultistep(5, version=1, iters=1,
            starting_method=ExponentialRungeKutta(:lawson),
            starting_subdivision=50)
        println("exponential multistep 5 + 1it")
    elseif variant == "expmultistep60"
        method = ExponentialMultistep(6, version=1, iters=0,
            starting_method=ExponentialRungeKutta(:lawson),
            starting_subdivision=50)
        println("exponential multistep without 6 + 0it")
    else
        throw(ArgumentError("\""*variant*"\" is not a supported method"))
    end

    try
        s = parse(Int, variant[end-1])
        println(s, " step method needs ", s-1, " additional starting
            values")
        global SHIFT_B = -(s-1)*4*50
        println("shift COUNT_B by ", SHIFT_B)
    catch ArgumentError
    end
    return method
end

function conv(variant::String, steps::Int)
    #laser field
    E(t) = sin(2*t)
    tend = 12
    dt=tend/steps

    V1_t(x, t) = E(t)*x
    V_t(x,y,t) = E(t)*(x+y)

    m = MCTDHF1D(2, N, nx, -interval_bound, interval_bound, potential1=V1
        , potential1_t=V1_t, potential2=V2, spin_restricted=true)
    psi = wave_function(m)
```

```
    TSSM.load!(psi, "wfquantumdot_f2_n"*string(N)*"_nx"*string(nx)*"
        _t0_1e-9.h5")
    set_time!(psi, t0)

    method = method_from_string(variant)
    times, energies, norms = propagate_equidistant!(method, psi, t0, dt,
        steps)

    if !isapprox(times[end], tend)
        println("Method did not converge!")
        return
    end

    TSSM.save(psi, "results_conv/wfquantumdot_conv_f2_n"*string(N)*"_nx"*
        string(nx)*"_t"*string(tend)*"_"*variant*"_"*string(steps)*".h5")

    if !parsed_args["no-ref"]
        psi_ref = wave_function(m);
        TSSM.load!(psi_ref, "results_conv/wfquantumdot_conv_f2_n"*string(
            N)*"_nx"*string(nx)*"_t"*string(tend)*"_adamslawson51_12000.
            h5")
        err = distance(psi, psi_ref)
        println("err:", err)
        JLD.save("results_conv/wfquantumdot_conv_f2_n"*string(N)*"_nx"*
            string(nx)*"_t"*string(tend)*"_"*variant*"_"*string(steps)*".
            jld", "times", times, "energies", energies, "dt", dt, "err",
            err, "B_calls", COUNT_B+SHIFT_B)
    end
end

function stab(variant::String, steps::Int)
    #no laser field
    E(t) = 0
    tend = 70
    dt=tend/steps

    V1_t(x, t) = E(t)*x
    V_t(x,y,t) = E(t)*(x+y)

    m = MCTDHF1D(2, N, nx, -interval_bound, interval_bound, potential1=V1
        , potential1_t=V1_t, potential2=V2, spin_restricted=true);
    psi = wave_function(m);

    TSSM.load!(psi, "wfquantumdot_f2_n"*string(N)*"_nx"*string(nx)*"
        _t0_1e-9.h5")
    set_time!(psi, t0)

    method = method_from_string(variant)

    times, energies, norms = propagate_equidistant!(method, psi, t0, dt,
        steps)

    JLD.save("results_stab/wfquantumdot_stab_f2_n"*string(N)*"_nx"*string
        (nx)*"_t"*string(tend)*"_"*variant*"_"*string(steps)*".jld", "
        times", times, "energies", energies, "dt", dt, "norms", norms)
end
```

```
function adapt(variant::String, tol::Float64)
    E(t) = sin(2*t)
    tend = 12

    V1_t(x, t) = E(t)*x
    V_t(x,y,t) = E(t)*(x+y)

    m = MCTDHF1D(2, N, nx, -interval_bound, interval_bound, potential1=V1
        , potential1_t=V1_t, potential2=V2, spin_restricted=true);
    psi = wave_function(m);
    TSSM.load!(psi, "wfquantumdot_f2_n"*string(N)*"_nx"*string(nx)*"
        _t0_1e-9.h5")
    set_time!(psi, t0)

    times, energies, norms, stepsizes = propagate_adaptive!(psi, t0, tend
        , variant, tol)

    TSSM.save(psi, "results_adapt/wfquantumdot_adapt_f2_n"*string(N)*"_nx
        "*string(nx)*"_t"*string(tend)*"_"*variant*"_"*string(tol)*".h5")
    if !parsed_args["no-ref"]
        psi_ref = wave_function(m);
        TSSM.load!(psi_ref, "results_conv/wfquantumdot_conv_f2_n"*string(
            N)*"_nx"*string(nx)*"_t"*string(tend)*"_adamslawson51_12000.
            h5")
        err = distance(psi, psi_ref)
        println("err:", err)
        JLD.save("results_adapt/wfquantumdot_adapt_f2_n"*string(N)*"_nx"*
            string(nx)*"_"*variant*"_"*string(tol)*".jld", "times", times
            , "energies", energies, "norms", norms, "stepsizes",
            stepsizes, "B_calls", COUNT_B, "err", err)
    end
end

function propagate_equidistant!(method, psi, t0, dt, steps)
    times = Vector{Float64}()
    energies = Vector{Float64}()
    norms = Vector{Float64}()
    step_counter = 1
    time0 = time()
    for (step, tsi) in EquidistantTimeStepper(method, psi, t0, dt, steps)
        n = norm(psi)
        E_pot = potential_energy(psi)
        E_kin = kinetic_energy(psi)
        E_tot = E_pot+E_kin
        if step_counter % 50 == 0 || step == steps
            @printf("%5i␣␣%14.10f␣␣%14.10f␣␣%14.10f␣␣%14.10f␣␣%14.10f␣␣
                %14.10f␣%6i␣%10.2f\n",
                step_counter, get_time(psi), E_pot, E_kin, E_tot, dt, n,
                    COUNT_B+SHIFT_B, time()-time0)
        end
        append!(times, get_time(psi))
        append!(energies, E_tot)
        append!(norms, n)
        step_counter += 1
        if n > 1e10
```

```
            break
        end
    end
    return times, energies, norms
end

function propagate_adaptive!(psi, t0, tend, variant, tol)
    version = 0
    if variant[1:end-1] == "adamslawson"
        version = 2
    elseif variant[1:end-1] == "expmultistep"
        version = 1
    else
        throw(ArgumentError("\""*variant*"\" is not a valid method"))
    end
    method = try
        AdaptiveAdamsLawson(parse(Int, variant[end]), version=version)
    catch ArgumentError
        throw(ArgumentError("The number of steps could not be parsed from
            \""*variant*"\". Only single digits are supported"))
    end

    times = Vector{Float64}()
    energies = Vector{Float64}()
    stepsizes = Vector{Float64}()
    norms = Vector{Float64}()
    step_counter = 1
    time0 = time()
    told = t0
    for (t, tsi) in AdaptiveTimeStepper(method, psi, t0, tend, tol,
        0.00001)
        n = norm(psi)
        E_pot = potential_energy(psi)
        E_kin = kinetic_energy(psi)
        E_tot = E_pot+E_kin
        stepsize = t - told
        if step_counter % 50 == 0
            @printf("%5i  %14.10f  %14.10f  %14.10f  %14.10f  %14.10f
                %14.10f  %10.2f\n",
                step_counter, get_time(psi), n, E_pot, E_kin, E_tot,
                    stepsize, time()-time0)
        end
        append!(times, get_time(psi))
        append!(energies, E_tot)
        append!(stepsizes, stepsize)
        append!(norms, n)
        told = t
        step_counter += 1
    end
    return times, energies, norms, stepsizes
end

if experiment == "groundstate"
    println("calculate groundstate with tolerance ", groundstate_tol)
    include(joinpath(TSSM_dir, "MCTDHF/propagators.jl"))
    m = MCTDHF1D(2, N, nx, -interval_bound, interval_bound, potential1=V1
```

```
        , potential2=V2, spin_restricted=true)
    psi = wave_function(m)
    groundstate!(psi, dt=0.1, max_iter=500000, output_step=100, tol=
        groundstate_tol)
    TSSM.save(psi, "wfquantumdot_f2_n"*string(N)*"_nx"*string(nx)*"_t0_"*
        string(groundstate_tol)*".h5")
elseif experiment == "conv"
    println("calculate␣solution␣using␣method␣", method_string, "␣with␣",
        steps, "␣steps")
    conv(method_string, steps)
elseif experiment == "stab"
    stab(method_string, steps)
elseif experiment == "adapt"
    println("calculate␣solution␣using␣adaptive␣method␣", method_string, "
        ␣with␣tolerance␣", adaptive_tol)
    adapt(method_string, adaptive_tol)
else
    throw(ArgumentError("\"" * experiment * "\"␣is␣not␣an␣implemented␣
        experiment␣type"))
end
```

Bibliography

[1] Schrodinger equation with a cubic nonlinearity. `http://eqworld.ipmnet.ru/en/solutions/npde/npde1401.pdf`. Accessed: 2019-03-23.

[2] W. Bao and Y. Cai. Mathematical theory and numerical methods for Bose–Einstein condensation. *Kinet. Relat. Mod.*, 6:1–135, 2013.

[3] J. Caillat, J. Zanghellini, M. Kitzler, W. Kreuzer, O. Koch, and A. Scrinzi. Correlated multielectron systems in strong laser pulses — an MCTDHF approach. *Phys. Rev. A*, 71:012712, 2005.

[4] Peter Deuflhard and Folkmar Bornemann. *Numerische Mathematik 2.* de Gruyter Studium. de Gruyter, Berlin [u.a.], 4. edition.

[5] E. Hairer, Ch. Lubich, and G. Wanner. *Geometric Numerical Integration.* Springer-Verlag, Berlin–Heidelberg–New York, 2nd edition, 2006.

[6] M. Hochbruck and A. Ostermann. Exponential integrators. *Acta Numer.*, 19:209–286, 2010.

[7] Harald Hofstätter. Time splitting spectral methods (tssm.jl) package. `https://github.com/HaraldHofstaetter/TSSM.jl`.

[8] O. Koch, W. Kreuzer, and A. Scrinzi. Approximation of the time-dependent electronic Schrödinger equation by MCTDHF. *Appl. Math. Comput.*, 173:960–976, 2006.

[9] J.D. Lawson. Generalized Runge–Kutta processes for stable systems with large Lipschitz constants. *SIAM J. Numer. Anal.*, 4:372—380, 1967.

[10] Robert I. McLachlan and G. Reinout W. Quispel. Splitting methods. *Acta Numerica*, 11:341–434, 2002.

[11] L. Trefethen. *Spectral Methods in MATLAB.* SIAM, Philadelphia, 2000.

[12] J. Zanghellini, M. Kitzler, T. Brabec, and A. Scrinzi. Testing the multi-configuration time-dependent Hartree-Fock method. *J. Phys. B: At. Mol. Phys.*, 37:763–773, 2004.