Computer Physics Communications 243 (2019) 135–150

Contents lists available at ScienceDirect

Computer Physics Communications

journal homepage: www.elsevier.com/locate/cpc

# QEngine: A C++ library for quantum optimal control of ultracold atoms[☆]

J.J. Sørensen, J.H.M. Jensen, T. Heinzel, J.F. Sherson [*]

*Department of Physics and Astronomy, Aarhus University, Ny Munkegade 120, 8000 Aarhus C, Denmark*

A R T I C L E   I N F O

*Article history:*
Received 31 July 2018
Received in revised form 1 April 2019
Accepted 23 April 2019
Available online 17 May 2019

*Keywords:*
Quantum optimal control theory
Bose–Einstein condensate
Gross–Pitaevskii equation
GROUP
Bose–Hubbard
C++

A B S T R A C T

We present the first version of the QEngine, an open-source C++ library for performing optimal control of ultracold quantum systems. The most notable systems presented here are Bose–Einstein condensates, many-body systems described by Bose–Hubbard type models, and two interacting particles. These systems can all be realized experimentally using ultracold atoms in various trapping geometries including optical lattices and tweezers. We provide several optimal control algorithms including the GROUP method. The QEngine library has a strong focus on accessibility and performance. We provide several examples of how to prepare simulations of the physical systems and apply optimal control.

**Program summary**
*Program Title*: QEngine
*Program Files doi:* http://dx.doi.org/10.17632/72rcmn65b2.1
*Licensing provisions*: MPL-2.0
*Programming Language*: C++14
*External routines*: Armadillo, LAPACK and BLAS or Intel Math Kernel Library
*Nature of problem*: Quantum optimal control of ultracold systems.
*Solution method*: Numerical simulation of the equation of motion and gradient based quantum control.
*Additional comments:* For further information and downloads see quatomic.com and gitlab.com/quatomic/qengine.

## 1. Introduction

In the last two decades there have been exceptional advances in the ability to engineer and understand complex quantum systems. Especially, ultracold atoms provide an excellent platform for precision measurements [1,2], matter wave interferometry [3–5], quantum simulation [6,7], and quantum gates [8–10]. These systems offer extensive versatility through their purity and the high level of control of both the underlying potential landscape and the interatomic interactions [11]. In order to fully utilize the potential of these quantum systems the design of efficient experimental protocols for preparing quantum states of interest poses an important challenge [12].

Many experimental control protocols rely on simple empirical or adiabatic inspiration, which are typically slow and therefore limited by decoherence, decay, etc. [12,13]. It is often desirable to find fast protocols that avoid decoherence and are robust with respect to system perturbations resulting in typically highly complex controls. Such control protocols can be found within the framework of quantum optimal control. In quantum optimal control improved protocols are found using optimization algorithms that seek to minimize some cost functional [14].

In the context of ultracold atomic physics, quantum optimal control has been applied to improve splitting and driving of Bose–Einstein condensates trapped on an atom chip, which can be used to realize matter wave interferometry and nonlinear atom optics [12,15,16]. Quantum optimal control has also been applied to stabilize ultracold molecules [17,18] and manipulate ultracold many-body systems in optical lattices [19]. In addition, it has been demonstrated that such optimal control pulses are experimentally feasible [12,20]. There has also been fundamental studies showing that quantum optimal control can find controls saturating the

[☆] This paper and its associated computer program are available via the Computer Physics Communication homepage on ScienceDirect (http://www.sciencedirect.com/science/journal/00104655)..

[*] Corresponding author.
*E-mail address:* sherson@phys.au.dk (J.F. Sherson).

https://doi.org/10.1016/j.cpc.2019.04.020
0010-4655/© 2019 Elsevier B.V. All rights reserved.

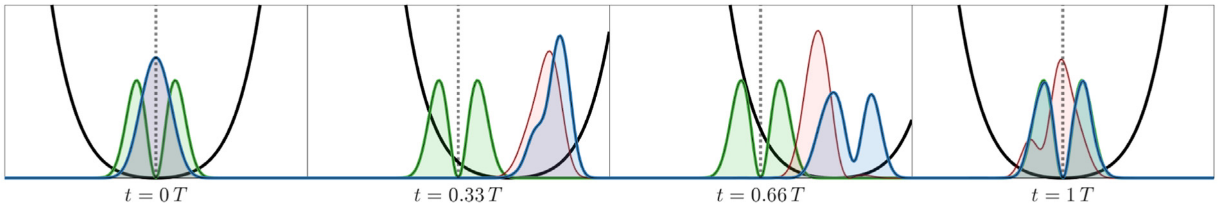

**Fig. 1.** (Color online) Snapshots of the instantaneous condensate density for an optimized (blue) and unoptimized (red) transfer of a ground state BEC into the first excited state (green) in an anharmonic potential (black) — see Section 3.1 for details. The simulations and optimization were performed using the QEngine.

fundamental quantum speed limit [21–23] where similar behavior has also been reported for ultracold atoms in a double well system [24]. Quantum optimal control is a versatile tool that can be applied not only in the context of ultracold atoms but as examples also in nuclear magnetic resonance [25], control of chemical reactions [26] and nitrogen vacancies [27].

Optimal control protocols are typically designed for a certain set of experimental parameter values that may change due to modifications or drifts in the experimental setup, thereby necessitating a recalculation for new optimal controls. For instance, there is a large number of papers that discuss driving a condensate from the ground state into the first excited state with slightly different parameter values [12,15,16,20,28,29]. A barrier for calculating such controls is writing and rigorously testing quantum optimal control programs, which is time consuming and the programs are very slow if not properly implemented. These two requirements, performance and usability, are primary driving forces behind the design of the QEngine.

There exist a number of alternative software packages to the QEngine for performing quantum optimal control. Many of these are implemented in MATLAB like OCTBEC [30], DYNAMO [31] and the recent WavePacket [32,33]. Especially, OCTBEC has a similar focus to the QEngine and it has been a source of inspiration for our work. The Python package QUTIP is also a widespread platform for simulation and optimal control of quantum optics [34]. Collectively these packages offer more functionality than the QEngine but they are implemented in weakly typed programming languages that are inherently less focused on performance.

The QEngine is designed for performance. One way the QEngine achieves this is by a general reliance on templates to provide flexibility instead of virtual functions and pointers. This allows for a high number of compile-time optimizations. Templates are useful for efficiency, but they are typically a programming barrier for physicists who are not C++ professionals. In order to accommodate such users of the library, we have made considerable efforts in providing a simplified API. The `auto`-syntax available in modern C++ together with factory functions and overloaded operators allows a straightforward syntax close to the mathematical equations used by theoretical physicists and weakly typed languages such as MATLAB and Python. The QEngine uses the highly optimized Intel Math Kernel Library (MKL) and the C++ library Armadillo to provide efficient basic linear algebra needed for the quantum simulations and optimizations [35]. In addition, the code has also been profiled and optimized.

A comprehensive documentation for the QEngine is available at quatomic.com/qengine/reference and the source code is available at gitlab.com/quatomic/qengine. The library has a number of example programs that can help users get started. In this paper, we give an introduction to some of the features in QEngine but leave out several details that can be found in the online documentation. The QEngine currently supports simulation and optimal control of the Gross–Pitaevskii description of a BEC, the Bose–Hubbard model, two interacting particles, a single particle, and generic few mode models.

The paper is organized as follows. In Section 2 we give a brief introduction to optimal control theory and the physical models. In Section 3 we discuss how to prepare simulations in two example programs that demonstrate key functionalities in the QEngine. Quantum optimal control theory is explained in Section 4 including the GROUP algorithm we recently introduced in Ref. [36]. Finally in Section 5 we explain how to perform optimal control on the example programs from Section 3, which is the main goal of the library. Section 6 gives a brief summary.

## 2. Overview of the QEngine

### 2.1. Quantum optimal control

The QEngine facilitates simulations of ultracold atoms in trapped geometries defined by external control parameters. The core goal of the QEngine is to enable optimal control on top of these simulations in a straightforward manner. In particular, the QEngine enables the user to solve state transfer problems using quantum optimal control. This type of problem consists in manipulating the system dynamics in order to realize a transfer of an initial state $\psi_0$ into a target state $\psi_t$ for some fixed duration $T$. The manipulatory access to the dynamics is through one or more control fields $u(t)$ parametrizing the Hamiltonian in some way $H(u(t))$. In an experimental setting the control fields usually correspond to physical quantities such as the intensity or position of a laser beam. Optimization algorithms are typically used to iteratively design the control fields. We give a more detailed introduction to quantum optimal control and the algorithms available in the QEngine in Section 4. An example of a state transfer problem is shown in Fig. 1 where a condensate is driven from the ground state into the first excited state using an optimized control field. In this case the control field corresponds to the position of the trap center, which is experimentally realized by adjusting magnetic fields [12,20]. The figure shows snapshots of the transfer process before and after the optimization, illustrating that the optimization algorithm succeeds in finding an optimal control. The QEngine offers a variety of optimization algorithms that can be applied to several physical models.

### 2.2. Models

Quantum optimal control depends intrinsically on the equations of motion in the physical model. In this subsection we discuss the models available in the QEngine. The starting point for ultracold atomic systems is the second quantized Hamiltonian [37]

$$\hat{H} = \int \left( \hat{\Psi}^\dagger(x) \left[ -\frac{\hbar^2}{2m}\frac{\partial^2}{\partial x^2} + \hat{V} \right] \hat{\Psi}(x) + \frac{g_{1D}}{2} \hat{\Psi}^\dagger(x)\hat{\Psi}^\dagger(x)\hat{\Psi}(x)\hat{\Psi}(x) \right) \mathrm{d}x. \tag{1}$$

Here the first term describes the kinetic and potential energy and the second term represents two-particle interactions. Currently, the QEngine supports one-dimensional systems. $\hat{\Psi}(x)$ and $\hat{\Psi}^\dagger(x)$ are the bosonic field operators obeying the usual commutation relations $[\hat{\Psi}(x_1), \hat{\Psi}^\dagger(x_2)] = \delta(x_1 - x_2)$. $\hbar$ is the reduced Planck constant and $m$ is the mass of the atom. In Eq. (1) we have used that two-particle interactions in the ultracold, dilute limit are well-described by an effective contact interaction, which in one-dimension is $V_{\text{int}}(x_1, x_2) = g_{1D}\delta(x_1 - x_2)$ where $g_{1D}$ is the system dependent coupling strength [11,38]. Different physical models described in QEngine emerge from different special cases of Eq. (1). We describe the available models one-by-one,

*Gross–pitaevskii equation.* One important special case for $N$ ultracold bosons is a mean field description where the bosonic field is in a single mode. This gives rise to the Gross–Pitaevskii Equation (GPE) describing the time evolution of a Bose–Einstein condensate (BEC) $\psi = \sqrt{N}\phi$. The corresponding Hamiltonian is

$$\hat{H}_{\text{gp}} = \hat{H}_0 + g_{1D}|\psi(x,t)|^2, \tag{2}$$

where $\hat{H}_0$ is the kinetic and potential energy. The non-linear term represents the condensate self-interaction. The GPE is an important starting point for modeling the dynamics of BECs [3,11,12,15]. The QEngine can calculate the ground state and first excited state of the GPE using an optimal damping algorithm described in Ref. [39] or state mixing. The time evolution is performed using the split-step Fourier method [40,41].

*Two-particle.* Technological advances have enabled the preparation of single atoms in optical lattices or tweezer arrays [11,42,43]. It has been proposed to use these systems as a platform for quantum computation, where the necessary two-qubit gate can for example be realized using controlled ultracold collisions of two atoms [8,10,44,45]. It is convenient to rewrite Eq. (1) in first quantization as

$$\hat{H} = -\frac{\hbar^2}{2m}\frac{\partial^2}{\partial x_1^2} + \hat{V}(x_1) - \frac{\hbar^2}{2m}\frac{\partial^2}{\partial x_2^2} + \hat{V}(x_2) + g_{1D}\delta(x_1 - x_2), \tag{3}$$

where $x_1$ and $x_2$ are the positions of the two atoms. The associated dynamics is also simulated using the split-step Fourier method. In a similar manner it is possible to simulate the dynamics of a single particle in the QEngine.

*Bose–hubbard.* Ultracold atoms can be loaded into a periodic optical lattice [11]. In this system it is convenient to expand $\hat{\Psi}(x)$ in terms of the localized Wannier modes on each lattice site. In a lowest band approximation the expansion reads $\hat{\Psi}(x) = \sum_i \hat{a}_i w_0(x - x_i)$. Also assuming the tight-binding approximation equation (1) for $L$ lattice sites becomes

$$\hat{H} = -J\sum_{i=1}^{L-1}\left(\hat{a}_{i+1}^\dagger \hat{a}_i + \text{h.c.}\right) + \frac{U}{2}\sum_{i=1}^{L}\hat{n}_i(\hat{n}_i - 1) + \sum_{i=1}^{L} V_i \hat{n}_i \tag{4}$$

Here $J$ and $U$ are matrix elements of Eq. (1) with the lowest band Wannier functions, which describe the nearest-neighbor tunneling and on-site interaction. $V_i$ is the local external trapping potential. The ground state of this Hamiltonian exhibits a phase transition from a delocalized superfluid state to a Mott insulating state depending on the ratio $U/J$ [11]. This model is simulated in the QEngine using exact diagonalization with sparse linear algebra. The time evolution is performed using the Krylov–Lanczos method [46].

*Units.* In order to perform any physical simulation it is convenient to transform the Hamiltonians in Eqs. (2)–(4) into dimensionless units. A discussion of the units used in one of the example programs is given in the Appendix.

### 2.3. Program structure

In this subsection we give an overview of the program structure in the QEngine. Instructions for installing the QEngine are included in the `README.md` file or at www.gitlab/quatomic/qengine. We have tested that the QEngine is compliant with recent standard compilers on Windows, Linux, and Mac OSX. Dependencies are included via `git submodules` for easy installation. In addition, a very detailed and complete API documentation can be found at www.quatomic.com/qengine/reference, which describes classes, methods, functions, interfaces etc. The QEngine has nearly 400 unit tests for both simulation and optimal control. The goal of this paper is not to act as a substitute for the online documentation but rather to complement it with a broad overview and enlightening examples.

A schematic overview of the QEngine is presented in Fig. 2. The figure also illustrates a common user workflow consisting in first setting up the appropriate model for simulation and then proceeding to perform optimal control. While formally similar, each specific physical model (Green box in Fig. 2) varies widely in their numerical implementation. This includes the vector representation of quantum states $|\psi\rangle$, the matrix representation of operators $\hat{O}$, diagonalization procedures $\hat{O}|o_i\rangle = o_i|o_i\rangle$, and application of operators to states $\hat{O}|\psi\rangle$. For instance, the nonlinear Gross–Pitaevskii equation requires a special algorithm for diagonalization. However, the most important example is the need for different time evolution algorithms when solving the equations of motion i.e. $|\psi(t+\Delta t)\rangle \approx \exp\bigl(-i\hat{H}\Delta t/\hbar\bigr)|\psi(t)\rangle$. As an example, the matrix exponential may efficiently be directly calculated for small dense Hamiltonians whereas large sparse matrices require more efficient methods such as the Krylov–Lanczos method or the split-step Fourier method.

The models are associated with Hilbert space classes in the QEngine which are broadly categorized as either spatial or quantized. The former describes models with dynamics in a one-dimensional spatial domain defined by a grid of discrete points and a Laplacian approximated using finite differences. For the spatial models, the QEngine implements the fundamental class `FunctionOfX` templated

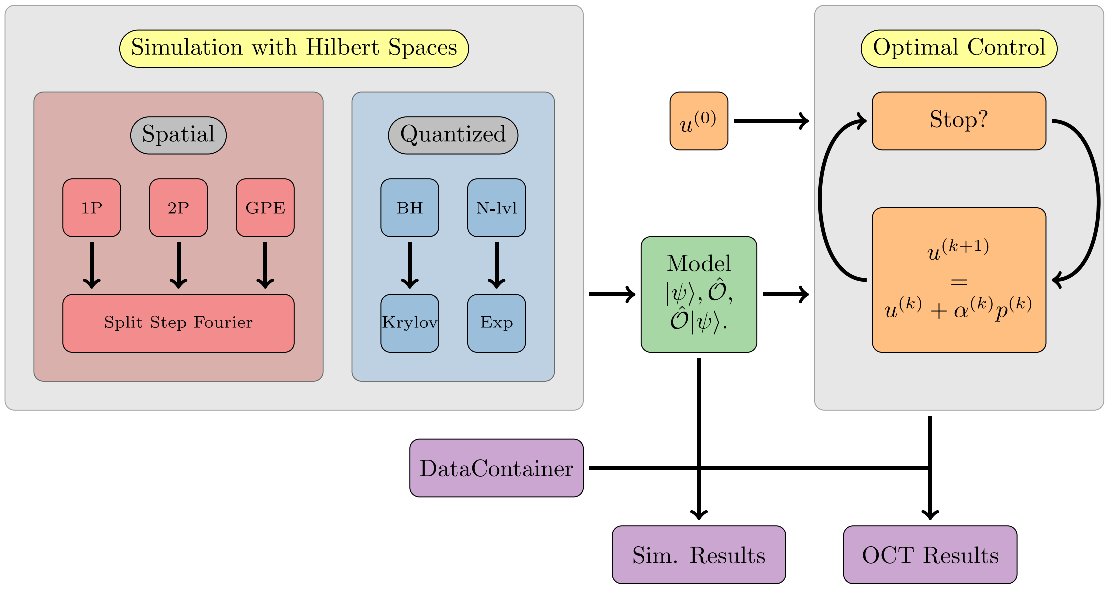
Simulation with Hilbert Spaces
Spatial
1P
2P
GPE
Split Step Fourier
Quantized
BH
N-lvl
Krylov
Exp
Model
$|\psi\rangle, \hat{\mathcal{O}}, \hat{\mathcal{O}}|\psi\rangle$.
$u^{(0)}$
Optimal Control
Stop?
$u^{(k+1)} = u^{(k)} + \alpha^{(k)} p^{(k)}$
DataContainer
Sim. Results
OCT Results

**Fig. 2.** (Color online) Schematic overview of the QEngine. The QEngine consists of a simulation (left) and optimal control (right) part. In the simulation part, the user sets up a physical model. Currently supported models are Gross–Pitaevskii (GPE), single or two-interacting particles (1P and 2P), generic few mode (N-lvl) and Bose–Hubbard (BH). In the source code these models are grouped in spatial and quantized models. The user may then readily obtain simulation results or perform quantum optimal control. In the optimal control part a cost function is iteratively minimized from a given initial control ($u^{(0)}$). In depth information is available in the online documentation at www.quatomic.com/qengine/reference.

on the specific Hilbert space, which describes any function defined on the grid e.g. the potential and wavefunction. The `FunctionOfX` class has several useful functions related to the grid discretization — see www.quatomic.com/qengine/reference for more information. In a one-dimensional single particle or GPE system the `FunctionOfX` describes $|\mathbf{X}\rangle = |x_1\rangle$ whereas for a one-dimensional two particle system it describes the tensor product space $|\mathbf{X}\rangle = |x_1\rangle \otimes |x_2\rangle$.

The other category of Hilbert spaces is quantized describing dynamics in a finite dimensional space. This part of the QEngine implements a set of classes for describing the dynamics of the Bose–Hubbard model. We have also implemented some support for generic N-level systems with a user supplied Hamiltonian matrix. Details are available in the online documentation.

The QEngine has a unified syntax (i.e. overloaded functions) for operations such as diagonalization, calculating the fidelity and performing optimal control. This allows the user to define a simulation on a conceptual level without worrying about the particular numerical intricacies required for optimal performance. To enable this, most of the QEngine is implemented as template classes and functions which also enhances code re-usability. Heavy use of templates usually leads to long unreadable types that obfuscates the conceptual intent. This can be solved with a standard C++ type deduction technique relying on implicit type deduction in factory functions similar to `std::make_tuple` and the `auto` keyword available in modern C++. We follow the standard convention of prefixing factory function names with `make`.

Until now, we have described the simulation part of the QEngine shown in Fig. 2. As stated previously, the main goal of the QEngine is to perform optimal control. We give a brief overview of the theory behind quantum optimal control in Section 4 that introduces the different optimal control algorithms available in the QEngine. The different control algorithms and their numerical implementation is also intrinsically dependent on the different models. This is accommodated in the same way as before with heavy use of templates and associated factory functions. We discuss the details of the optimal control classes and workflow in Section 5.

The results generated by simulation and optimal control may be exported for post processing and analysis with the `DataContainer` class. The `DataContainer` allows for storing multiple numbers, arrays, matrices and strings of different sizes into a single `.mat` (standard MATLAB format) or `.json` file for fast and convenient use in other programs.

In order to help users get started we have included a number of example programs in the folder `example_projects` that are designed to illustrate central features in the QEngine. An overview of these programs is given in Table 1. Rather than simply listing all classes and functions available in the QEngine, we instead discuss how to set up and perform optimal control on a spatial and quantized model through the two example programs `gpe-example.cpp` and `bosehubbard-example.cpp`. This also illustrates the user workflow depicted in Fig. 2 consisting of first setting up a model (part I) and then executing the optimal control algorithms (part II). Although the main goal is to perform optimal control it is convenient to first discuss part I in its own section. For clarity, we omit minor code details in this paper.

## 3. Part I: Simulation example programs

### 3.1. Gross–Pitaevskii example program

As a first example, we describe optimal control of a BEC trapped on an atom-chip. We focus on the control problem described in Refs. [12,16,47] where a BEC is transferred from the ground state into the first excited state as shown in Fig. 1. The physical motivation for this problem is to create a source of twin-atom beams, which is the matter-wave analogue to twin-photon beams [47]. The mechanism behind twin beam emission is binary collisions of two excited atoms. The collision may cause atoms to de-excite into

**Table 1**
The default example programs included in the QEngine. The table shows the file name, execution time and a short description. The results were generated on a 2016 HP Prodesk with an Intel(R) Core$^{TM}$ i7-6700 CPU @ 3.40 GHz processor. Execution times may vary due to randomly generated numbers.

| Example program | Execution time | Description |
|---|---|---|
| gpe-example.cpp | 21s | Optimal control of driving a condensate wave function. |
| bosehubbard-example.cpp | 60s | Optimal control of a superfluid to Mott transfer. |
| twoparticle-example.cpp | 35s | Optimal control of an ultracold atomic gate. |
| oneparticle-example.cpp | 0.87s | Optimal control of a single atom in an optical tweezer. |
| twolevel-example-cpp | 0.47s | Optimal control of the Landau–Zener system. |

the radial ground state mode while simultaneously populating entangled twin momentum states $|\pm k_0\rangle$ along the axial $z$-direction due to conservation of momentum and energy. The characteristic timescale for the collision induced decay is a few milliseconds ($\approx 3$ ms), and as a consequence the duration of the preparation stage into the excited state must be well below this threshold [12]. The atom-chip experiment has two tightly confined transverse directions ($x$ and $y$) and a weakly confined axial direction ($z$). One of the transverse directions (say $y$) has a tighter confinement freezing out excitations. The dynamics along the axial direction is slow compared to the transverse directions, and we may only consider a one-dimensional GPE along the $x$-direction. This requires an appropriate effective coupling constant [12,48]. In this part, we only discuss how to set up the simulation of this system.

The potential along the $x$-direction is parameterized by a single control field $u(t)$, which is well-approximated by the anharmonic potential

$$V(x, u(t)) = p_2(x - u(t))^2 + p_4(x - u(t))^4 + p_6(x - u(t))^6, \tag{5}$$

where the $p_i$'s are constants obtained experimentally [12,47]. The initial state is taken to be the ground state corresponding to $V(x, 0)$, and in the optimal control part we take the target state to be the first excited state corresponding to $V(x, 0)$ as shown in Fig. 1. Measuring length in units of micrometers and time in units of milliseconds, the effective mean field interaction strength is $g_{1D} = 1.8299$ (see Appendix).

To use the QEngine in a program, we need to include the QEngine header file

```
#include <qengine/qengine.h>
#include <iostream>

using namespace qengine;
```

The `qengine.h` header exposes the different simulation models and optimal control algorithms. At the highest level the QEngine library defines the namespace `qengine`. The `qengine` namespace contains most API-functionality across the different types of physics and optimal control. We will also make use of the `DataContainer` class defined in the QEngine, which can be used to save data to a `.json` file format or optionally to a MATLAB `.mat` file format. This makes it easy to export data for visualization and post-processing.

First we set up the control field $u(t)$

```
const auto dt = 0.002;
const auto duration = 1.25; // corresponds to 1.25ms
const auto n_steps = floor(duration/dt) + 1;

const auto ts = makeTimeControl(n_steps,dt);
const auto initialAmplitude = 0.55;
const auto u = initialAmplitude*sin(PI/duration*ts); // control field
```

The `makeTimeControl` factory function returns a single control field with linearly spaced values, which can be used to compose more complicated control fields. Note how the variable types are deduced by the `auto` keyword. In particular, the type deduction of the variable `ts` is performed with the factory function technique as indicated by the prefix `make`. In this case the control field `u` is half a sine period with amplitude 0.55. This will also act as our initial guess in the optimal control algorithms in the later section. It is possible to access the control field values at time index $i$ by calling `u.get(i)`, returning an `RVec` whose entries are the values for each control field at that time index. In the present case, we only have a single control field. The control field values at the first and last time index can be easily accessed with `u.getFront()` and `u.getBack()`, respectively.

The concept of a Hilbert space is mimicked in the QEngine for each type of physical model. For the Gross–Pitaevskii equation,

```
const auto kinFactor = 0.36537;  // T = -kinFactor*d^2/dx^2
const auto s = gpe::makeHilbertSpace(-2,+2,256,kinFactor);
const auto x = s.x(); // FunctionOfX of x-grid values
```

Here we also extract the spatial grid by `s.x()` which has type `FunctionOfX` with appropriate template parameters implicitly determined by the Hilbert space. Having defined both $x$ and $u(t)$ we can create the control-dependent potential $V(x, u(t))$ Eq. (5)

```
const auto p2 =  65.8392;
const auto p4 =  97.6349;
const auto p6 = -15.3850;

const auto V_func = [&x,p2,p4,p6](const real u)
{
    // By saving intermediate calculations we reduce overall computation time
const auto x_u = x - u;
const auto x_uPow2 = x_u * x_u;
const auto x_uPow4 = x_uPow2 * x_uPow2;
const auto x_uPow6 = x_uPow2 * x_uPow4;

return  p2*x_uPow2 + p4*x_uPow4 + p6*x_uPow6;
};

const auto u_initial = u.getFront().front(); // first entry in first time index
const auto V = makePotentialFunction(V_func, u_initial);
```

The lambda function `V_func` takes a `real` number and returns a `FunctionOfX` evaluated with the given control value. To create a potential object the lambda function and an initial control field value are combined in `makePotentialFunction`. The `V` object encapsulates the idea of a potential, and calling `V(newControlValue)` evaluates the `V_func` lambda with `newControlValue` and returns a potential operator. In the present case `newControlValue` is of type `real`. The kinetic energy operator can simply be extracted from the Hilbert space. It is represented by the 5-diagonal approximation to the second derivative with non-periodic boundary conditions. The mean field interaction is equally succinctly handled. Assembling the Hamiltonian operator is then straightforward,

```
const auto T = s.T();
const auto g1D = 1.8299;
const auto meanfield = makeGpeTerm(g1D);

const auto H = T + V + meanfield;
```

The `H` object can be called in the same way as the underlying potential by `H(newControlValue)`. Note that the type of `H` is `auto` deduced to be a GPE Hamiltonian by the compiler. Omitting the `meanfield` term would change the type deduction to a single particle Hamiltonian. This would still be valid code since the GPE Hilbert space is the same as the single particle Hilbert space.

The QEngine defines a convenient syntax for creating general linear combinations of eigenstates for operators. Let `A` be an operator and let $\{\phi_i\}$ be the corresponding eigenstates and suppose we wanted to create the linear combination $\psi = \phi_0 - 2i\phi_1$. This is readily achieved with the lines,

```
const auto comb = A[0] - 2.0*1i*A[1];     // syntax object
const auto psi = makeWavefunction(comb);  // evaluate syntax
```

The states we need for the example are individual eigenstates

```
const auto psi_0 = makeWavefunction(H(u.getFront())[0]);
const auto psi_t = makeWavefunction(H(u.getBack() )[1]);
```

The QEngine currently only supports calculation of the ground state and first excited state for GPE type physics, but will not raise an error if higher excited states are queried. There are no such restrictions for the other types of physics.

Initializing the `DataContainer` and storing time independent data is done by

```
DataContainer dc; // empty data container
dc["dt"] = dt;
dc["duration"] = duration;
dc["x"] =  x.vec();
dc["psi_t"] = psi_t.vec();
```

Once all data has been collected, calling `dc.save("<path/to/dest>.<file-extension>")` creates a file in either the `.json` or `.mat` file format, where e.g. the variable `x` is stored with the field name "x" and corresponding field values. The `FunctionOfX` types must be converted into a regular vector by `.vec` for saving.

To perform time evolution we initialize a stepper with fixed stepping interval length `dt`. We then loop over the entire control and append a few quantities of interest at each instant of time for saving.

```
auto stepper = makeFixedTimeStepper(H,psi_0,dt);
for(auto i = 0; i < n_steps; i++)
{
const auto& psi = stepper.state();
dc["V"].append(V(u.get(i)).vec());
dc["psis"].append(psi.vec());
dc["overlap"].append( overlap(psi, psi_t));
dc["fidelity"].append(fidelity(psi,psi_t));
dc["x_expect"].append(expectationValue(x,psi));
if(i < n_steps-1) stepper.step(u.get(i+1));
}
```

The propagation from $\psi(t)$ to $\psi(t+dt)$ is performed by `stepper.step(u.get(i+1))` using the midpoint rule. Only $u(t+dt)$ is needed since $u(t)$ is stored internally from the previous step. In the program we subsequently take additional steps with the final Hamiltonian held constant by using the `stepper.cstep()` function in an otherwise identical loop to the one above.

Note that the default `make`-functions for spatial time steppers include imaginary boundaries, which attenuates the wave function near the spatial grid extrema. For more information see quatomic.com/qengine/reference.

The result of the simulation is illustrated in Fig. 3 where the density of the condensate is plotted as a function of time. In the atom-chip experiment the objective is to transfer the initial state into the first excited state, which is not accomplished in the unoptimized transfer Fig. 3. Later we will illustrate how to find such a transfer using optimal control.

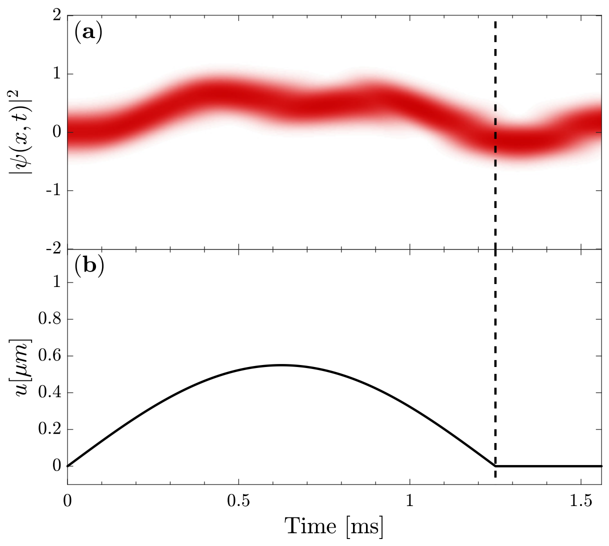

**Fig. 3.** (Color online) (**a**) the density of the condensate $|\psi(x, t)|^2$ when propagated along the unoptimized control (**b**) from the Gross–Pitaevskii example program. The initial control gives $F = 0.23$. After the vertical dashed line the control is held constant.

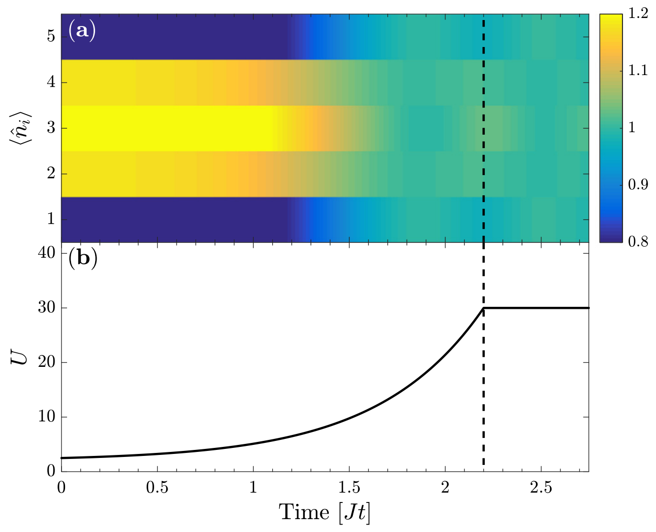

**Fig. 4.** (Color online) (**a**) the on-site density $\langle \hat{n}_i \rangle$ when propagated along the unoptimized control (**b**) from the Bose–Hubbard example program. The initial control gives $F = 0.81$. After the vertical dashed line the control is held constant.

### *3.2. Bose-Hubbard example program*

In this example, we discuss the optimal control of bosons in an optical lattice described by the Bose–Hubbard model equation (4). Creating a Mott state with one particle on each site is important for many experimental applications such as quantum logic gate operations [49–53], quantum simulation [54], and single atom transistors [55]. Experimentally the system is initialized in the superfluid state and must be dynamically transferred into the Mott state [56,57]. However, near the phase transition the gap between the ground state and the first excited state closes in an infinite system. This implies diverging transfer times for adiabatic solutions. There have been both experimental and numerical attempts to find improved transfer protocols [19,56,58,59].

Here we consider a transfer from the superfluid ground state at $U = 2.5$ into a Mott ground state at $U = 30$ with a weak harmonic external potential in units of $J$. It is necessary to impose a minimal $U_{\min} = 2$ since the Bose–Hubbard model assumes a sufficiently deep lattice [11]. In a similar manner, it is not experimentally feasible to have arbitrarily large values of $U$. We take the upper bound to be $U_{\max} = 40$. Later we will apply quantum optimal control to find optimized solutions, which must also satisfy these experimental and modeling constraints on $U$. The constraints can be accommodated by introducing a nonlinear transformation $U(u) = A(\tanh(u) + B)$, where $u$ is a non-physical but unbounded control field. Here $A = U_{\max}/(1 + B)$ and $B = (1 + U_{\min}/U_{\max})/(1 - U_{\min}/U_{\max})$ restricts the physical control $U_{\min} < U < U_{\max}$. Another option to handle these constraints is to use the soft bounds introduced in Eq. (14).

```
const auto Umin = 2.0; const auto Umax = 40.0;

const auto B = (1+Umin/Umax)/(1-Umin/Umax); // transformation params
const auto A = Umax/(1+B);
const auto UFromu = [A,B](auto u){return A*(tanh(u) + B);}; // control to physical U
const auto uFromU = [A,B](auto U){return atanh(U/A-B);};    // physical U to control

const auto dt = 0.002;
const auto duration = 2.0;
const auto n_steps = floor(duration / dt) + 1;

const auto ts = makeTimeControl(n_steps, dt);
const auto u = uFromU(Umin + 0.5*exp(log((30-Umin)/0.5)*ts/duration));
```

Here we use an exponential ramp for $U$, which will later be used as the starting point for the quantum optimal control algorithms [19]. As in the previous example, we begin by creating the underlying Hilbert space and subsequently initialize the terms in the Hamiltonian. We also demonstrate how to add a weak confinement potential.

```
const auto space = bosehubbard::makeHilbertSpace(5,5);

const auto periodicBoundaries = false;
const auto hoppingOperator = space.makeHoppingOperator(periodicBoundaries);
const auto onSiteOperator  = space.makeOnSiteOperator();

const auto sitePositions = linspace(-1.0, +1.0, space.nSites());
const auto potential = 0.1*pow(sitePositions,2);
const auto V = space.transformPotential(potential); // transform to site indices

const auto H_J = -1.0* hoppingOperator; // J = 1.0
const auto H_const = H_J + V; // Constant parts of Hamiltonian is added

const auto H_func = [&H_const, &onSiteOperator, &UFromu](const real u)
{
return H_const + 0.5*UFromu(u)* onSiteOperator;
};

const auto H = makeOperatorFunction(H_func,u.getFront().front());
```

As in the previous example the full Hamiltonian `H` is assembled using a lambda function `H_func` and an initial control value `u.getFront().front()`. After initializing the Hamiltonian, we set up the superfluid state and the Mott state. We then initialize the time stepper. The default stepper is a Lanczos propagator, which uses a user supplied Krylov order [46].

```
const auto psi_0 = makeState(H(u.getFront())[0]);
const auto psi_t = makeState(H(u.getBack())[0]);

const auto krylovOrder = 4;
auto stepper = makeFixedTimeStepper(H,psi_0,krylovOrder,dt);
```

These lines of code complete the necessary steps to set up a Bose–Hubbard simulation. Exactly as in the previous example we propagate over the control and in this case save the single-particle density matrix by

```
dc["rho1"].append(space.singleParticleDensityMatrix(state));
```

`state` is the instantaneous state from the `stepper` when propagated over the control `u`. The result of the Bose–Hubbard simulation is illustrated in Fig. 4 where the on-site density is plotted as a function of time. In the superfluid–Mott transfer the objective is to reach the Mott insulator type state, which is not accomplished in the unoptimized transfer Fig. 4, thus requiring optimal control.

## 4. Theory of quantum optimal control

In the previous section we have described how to set up simulations in the QEngine. As the main goal of the QEngine is to perform quantum optimal control, we review the theory and central algorithms implemented in the QEngine in this section. In the next section we describe how to apply quantum optimal control to the example programs.

Consider the problem of engineering a single control field $u(t)$ realizing the state transfer from $\psi_0$ to $\psi_t$ in duration $T$ constrained by the equation of motion $i\hbar\dot{\psi} = \hat{H}(u)\psi$ for all $t$. We may consider $\hat{H} = \hat{H}_{\mathrm{gp}}$ to be the general case, as taking $g_{1\mathrm{D}} = 0$ produces the usual Schrödinger equation. In quantum optimal control, this problem is posed as a minimization of the cost functional [36]

$$J[\psi,\chi,u] = J_{\mathcal{F}}[\psi] + J_{\gamma}[u] + J_{\mathrm{gp}}[\psi,\chi,u] \tag{6}$$

$$= \frac{1}{2}\left(1 - |\langle\psi_t|\psi(T)\rangle|^2\right) + \frac{\gamma}{2}\int_0^T \dot{u}^2 \mathrm{d}t + \Re\int_0^T \left\langle\chi\middle|\left(i\hbar\partial_t - \hat{H}_0(u) - g_{1\mathrm{D}}|\psi|^2\right)\middle|\psi\right\rangle \mathrm{d}t, \tag{7}$$

where the time dependence of most quantities has been suppressed for readability. The first term is minimal for perfect transfers up to a global phase i.e. when the fidelity $F = |\langle\psi_t|\psi(T)\rangle|^2$ is unity. The second term penalizes rapid temporal fluctuations in the control field, which are typically not experimentally feasible. The relative importance between the first and second term is determined by a regularization hyperparameter $\gamma \geq 0$ where higher values shift preference towards smoother controls. Usually $\gamma \sim 10^{-7} - 10^{-5}$. The last term containing the Lagrange multiplier $\chi(t)$ ensures the equation of motion is obeyed at all times.

### 4.1. GRAPE

Setting the first Gâteaux variations of $J$ wrt the functions $\{\psi(t), \chi(t), u(t)\}$ to zero

$$D_{\delta\psi}J = D_{\delta\chi}J = D_{\delta u}J = 0, \tag{8}$$

and assuming the variations of the control vanish at the boundaries ($t = 0$ and $t = T$) lead to the first order optimality conditions [36]

$$i\hbar\dot{\psi} = (\hat{H}_0(u) + g_{1\mathrm{D}}|\psi|^2)\psi, \qquad \psi(0) = \psi_0, \tag{9a}$$

$$i\hbar\dot{\chi} = \left(\hat{H}_0(u) + 2g_{1\mathrm{D}}|\psi|^2\right)\chi + g_{1\mathrm{D}}\psi^2\chi^*, \qquad \chi(T) = i\langle\psi_t|\psi(T)\rangle\psi_t, \tag{9b}$$

$$\gamma\ddot{u} = -\Re\left\langle \chi|\frac{\mathrm{d}\hat{H}_0(u)}{\mathrm{d}u}|\psi\right\rangle, \qquad u(0) = u_0, \quad u(T) = u_T. \tag{9c}$$

At this point we may think of $J$ as a functional of only $u$, $J = J[u]$, with the corresponding dynamics of $\psi$ and $\chi$ determined by the equations of motion above. An analytical approach to solving this set of equations is not generally feasible. However, we may define the gradient of $J$ wrt $u(t)$ under the $X$ norm as the element $\nabla_X J$ fulfilling the relation

$$D_{\delta u}J = \langle \nabla_X J, \delta u\rangle_X, \tag{10}$$

where $\delta u$ is an arbitrary variation. The common choices of the norm are $X = L^2$ and $X = H^1$ defined as $\langle f, g\rangle_{L^2} = \int_0^T f(t)g(t)dt$ and $\langle f, g\rangle_{H^1} = \int_0^T \dot{f}(t)\dot{g}(t)dt$, respectively [60]. Eq. (10) establishes an indirect way of calculating the gradient and for the norms above we obtain

$$\nabla_{L^2}J(t) = -\Re\left\langle \chi(t)|\frac{\mathrm{d}\hat{H}_0(u(t))}{\mathrm{d}u}|\psi(t)\right\rangle - \gamma\ddot{u}(t), \tag{11}$$

$$\nabla_{H^1}\ddot{J}(t) = -\nabla_{L^2}J(t). \tag{12}$$

The $L^2$ gradient may not vanish at the boundaries so we must artificially enforce $\nabla_{L^2}J(0) = \nabla_{L^2}J(T) = 0$ to respect Eq. (9c). In solving the Poisson equation (12) for the $H^1$ gradient we may conveniently choose Dirichlet boundary conditions $\nabla_{H^1}J(0) = \nabla_{H^1}J(T) = 0$ directly so Eq. (9c) is always fulfilled.

The control is iterated towards a local minimum of the cost functional by the update rule

$$u^{(k+1)} = u^{(k)} + \alpha^{(k)}p^{(k)}, \tag{13}$$

where $\alpha^{(k)}$ is a suitable step size along the search direction $p^{(k)}$ for the k'th iteration — see Fig. 2. Typically the search direction is based on gradient information [61]. The simplest choice is searching in the direction of steepest descent $p_{\mathrm{SD}}^{(k)} = -\nabla_X J[u^{(k)}]$ where we again are free to choose either $X = L^2$ or $X = H^1$. Another common search direction is the Newton direction $p_{\mathrm{N}}^{(k)} = (\nabla_X^2 J[u^{(k)}])^{-1}\nabla_X J[u^{(k)}]$, which takes into account the local curvature of the functional. This requires an expensive calculation of the Hessian $\nabla_X^2 J$, while also having no guarantee of invertibility far from the critical points of $J$. In practice one uses methods like L-BFGS to build an approximation $p_{\text{L-BFGS}}^{(k)} \approx p_{\mathrm{N}}^{(k)}$ to the search direction at iteration $k$ based on the gradients calculated in iterations $1, 2, \ldots, k$ [60]. In passing we note that our numerical experiments suggest that restarting the L-BFGS by erasing the gradient history may improve convergence rates in some situations.

As mentioned above, in a number of experimentally relevant cases there is also a bound on the maximal and minimal values of the control $u_{\min} \leq u(t) \leq u_{\max}$. These bounds can be accommodated by using a non-linear transformation that makes the control unconstrained as in the Bose–Hubbard example program. An alternative is to add a term in the cost function equation (7) that penalizes controls outside the bounds. The latter approach is known as soft bounds and the QEngine supports a parabolic cost penalty

$$J_{\mathrm{b}} = \frac{\sigma}{2}\int_0^T \Theta(u_{\min} - u)(u - u_{\min})^2 + \Theta(u - u_{\max})(u - u_{\max})^2 \mathrm{d}t, \tag{14}$$

where $\Theta$ is the Heaviside step function. The weight factor $\sigma$ is typically of the order $\sigma \sim 10^3 - 10^4$ to heavily penalize controls outside the bounds. It is straightforward to calculate the $L^2$ gradient of this term which is

$$\nabla_{L^2}J_{\mathrm{b}} = \sigma\Big(\Theta(u_{\min} - u)(u - u_{\min}) + \Theta(u - u_{\max})(u - u_{\max})\Big). \tag{15}$$

This gradient is added to Eq. (11) steering the optimization towards a region inside the bounds.

Collectively the updates using gradients described in Eqs. (11)–(12) are known as GRAPE algorithms (GRadient Ascent Pulse Engineering) [25]. Numerically solving Eqs. (9a)–(9b) requires discretizing time in steps of $\Delta t$ with a total number of steps $N = \lfloor T/\Delta t \rfloor + 1$. In the GRAPE algorithm family, the dimensionality $M$ of the optimization problem is equal to the number of simulation time steps, $M = N$, which is usually on the order of thousands.

### 4.2. GROUP

The GROUP algorithm (GRadient Optimization Using Parametrization) [36] consists in expanding the control function $u$ on a reduced basis of functions $f_m(t)$ where $1 \leq m \leq M$

$$u(t; \boldsymbol{c}) = u_{\mathrm{ref}}(t) + S(t)\sum_{m=1}^{M} c_m f_m(t), \tag{16}$$

and performing gradient-based optimization in the $M$-dimensional space of real coefficients $\boldsymbol{c} = [c_1, c_2, \ldots, c_M]^T$, which is usually on the order of tens, $M \ll N$ [36]. This gives GROUP a much smaller optimization dimensionality than GRAPE and is also independent of the duration and size of the time steps. In Eq. (16) $u_{\text{ref}}(t)$ is a reference control and $S(t)$ is a shape function that goes to zero for $t = 0$ and $t = T$, together enforcing appropriate boundary conditions equation (9c). Since we are now optimizing the expansion coefficients $\boldsymbol{c}$, the first Gâteaux variations of $J$ wrt $\{\psi(t), \chi(t), \boldsymbol{c}\}$ are set to zero

$$D_{\delta\psi}J = D_{\delta\chi}J = D_{\delta\boldsymbol{c}}J = 0. \tag{17}$$

This produces the same equations of motion as Eqs. (9a)–(9b). Effectively now $J = J[\boldsymbol{c}]$. Choosing the inner product to be the usual vector dot product for $X = \mathbb{R}^M$, the corresponding gradient of $J$ wrt $\boldsymbol{c}$ is then defined as the element $\nabla_{\mathbb{R}^M}J$ fulfilling the relation

$$D_{\delta\boldsymbol{c}}J = \langle \nabla_{\mathbb{R}^M}J, \boldsymbol{\delta c}\rangle_{\mathbb{R}^M} = \nabla_{\mathbb{R}^M}J \cdot \boldsymbol{\delta c} = \sum_{m=1}^{M} \frac{\partial J}{\partial c_m}\delta c_m, \tag{18}$$

where $\delta\boldsymbol{c}$ is an arbitrary variation. This definition of the gradient coincides with the usual definition of the gradient as a column vector of partial derivatives. The GROUP gradient elements become

$$\frac{\partial J}{\partial c_m} = \int_0^T \left(-\Re\left\langle \chi \left| \frac{d\hat{H}_0(u)}{du} \right| \psi \right\rangle - \gamma\ddot{u}\right) S(t) f_m(t)\mathrm{d}t = \int_0^T \nabla_{L^2}J(t)S(t)f_m(t)\mathrm{d}t, \tag{19}$$

where we identified the term in parenthesis to simply be the $L^2$ GRAPE gradient from Eq. (11). Calculating the GROUP gradient amounts to first calculating the usual $L^2$ GRAPE gradient and subsequently performing $M$ inexpensive one-dimensional integrals [36]. The coefficients are then iterated according to

$$\boldsymbol{c}^{(k+1)} = \boldsymbol{c}^{(k)} + \alpha^{(k)}\boldsymbol{p}^{(k)}, \tag{20}$$

where $\boldsymbol{p}^{(k)}$ is either the steepest descent direction or the L-BFGS direction both utilizing the gradient equation (19).

### 4.3. Dressed GROUP

A caveat of the parametrization equation (16) is that we may induce local trap minima not inherent to the control problem, but rather due to the parametrization itself. These types of traps are known as artificial traps [62]. A method to escape such traps was proposed in Ref. [62]. We may let $f_m = f_m(t; \boldsymbol{\theta}_m)$ where $\boldsymbol{\theta}_m$ is a set of values that is usually drawn at random. For example we may take $f_m(t, \theta_m) = \sin((m + \theta_m)\pi t/T)$ where $-0.5 \leq \theta_m \leq 0.5$ is drawn from a uniform distribution. Then, if the algorithm gets trapped at (possibly) an artificial minimum, we set $u_{\text{ref}}(t) \leftarrow u(t)$, re-initialize the algorithm with coefficients $\boldsymbol{c} = \boldsymbol{0}$, and draw a new set of basis functions $f_m$ defined by a new set of values $\boldsymbol{\theta}_m$,

$$u_{\text{ref}}(t) \leftarrow u(t), \quad \boldsymbol{c} \leftarrow 0, \quad f_m(t; \boldsymbol{\theta}_m) \leftarrow f_m(t; \boldsymbol{\theta}^*_m), \tag{21}$$

where $\boldsymbol{\theta}^*_m$ are new random values. This changes the topography of the optimization landscape and the artificial trap may have been eliminated. Effectively, this corresponds to restarting the GROUP algorithm with a new parametrization basis. These restarts are known as superiterations [62]. This modification is referred to as dressed GROUP, or dGROUP for short [36].

### 4.4. Optimal control program structure

All algorithms described in the previous subsections are readily available in the QEngine. The general structure of the optimization is illustrated in Fig. 5, which elaborates on the optimal control in Fig. 2. The user must first set up a `problem` with the `makeStateTransferProblem` factory function. The `problem` object maintains a control function and is responsible for calculating the associated cost equation (7) and $L^2$-gradient equation (11). The bare `makeStateTransferProblem` does not include the regularization $J_\gamma$ equation (7) nor boundary term $J_b$ equation (14) and must be added as shown in Section 5.2 and the online documentation. The `problem` constitutes the input to the optimal control part in Fig. 5. After instantiating a `problem` object the user decides on an algorithm. The minimal syntax to instantiate a particular algorithm is shown in Table 2. Associated with each algorithm is a particular step-direction $p^{(k)}$ and control parametrization. Each particular algorithm object also internally handles transformation of the $L^2$ gradient (provided by the `problem` object) into the appropriate form, e.g. the $H^1$ GRAPE gradient equation (12) or the GROUP gradient equation (19). Calling `.optimize()` on the algorithm object starts the optimization and the five main steps in each iteration are illustrated in Fig. 5. First the descent direction $p^{(k)}$ is calculated. Next an appropriate step size is found using the `StepSizeFinder` object and thereafter the control is updated according to Eq. (13) or Eq. (20). The algorithm then calls the `Collector` storing information about the optimization in this particular iteration e.g. the fidelity, iteration number, gradient norm, etc. Finally the `Stopper` checks if the optimization should stop or continue to the next iteration. Default values for the `Stopper`, `StepSizeFinder`, or `Collector` objects are used if custom versions are not passed into the algorithm factory function as in Table 2. These optional settings are illustrated in Fig. 5 by dashed boxes. Note that the options are independent of the `problem` and therefore work for any the physical model. The GROUP (dGROUP) algorithm additionally requires a `basis` (`basisMaker`) object.

## 5. Part II: Optimal control example programs

We now extend the example programs from Section 3 by performing optimal control on the systems, which illustrates the core functionality in the QEngine. Initially we will use the minimal API listed in Table 2 to perform quantum optimal control on the Bose–Hubbard example program where we use the default options for the `stopper`, `collector` and `stepSizeFinder`. Afterwards we will show how to customize these options for the GPE example program.

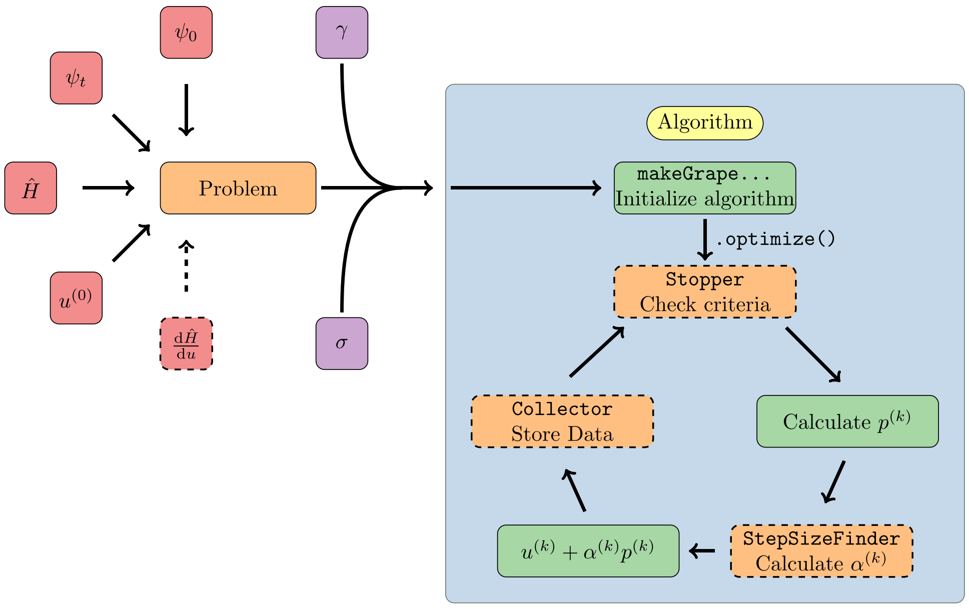

**Fig. 5.** (Color online) Schematic overview of the optimal control part of the QEngine. The user sets up a `problem` with `makeStateTransferProblem(...)` with inputs given by the red boxes. Dashed boxes indicate optional inputs. Regularization ($\gamma$) and bounds ($\sigma$) terms can be added to the optimization `problem`. The resulting `problem` object is input to an algorithm factory function such as `makeGrape...` where the ellipsis indicates an optimization type suffix for example `_bfgs_L2`. The algorithm factory function also optionally accepts a `Stopper`, `Collector` and `StepSizeFinder` as input arguments, otherwise defaults are used. Calling `.optimize()` starts the optimal control loop. The main steps in this loop are depicted in the loop. The same optimization loop is performed for GROUP with $p^{(k)}$ replaced by $\mathbf{c}^{(k)}$ and additional input options such as `basisMaker`.

**Table 2**
Factory functions with minimal arguments to instantiate different control algorithms in the QEngine. The GRAPE algorithms (left) only require a state transfer `problem` object, whereas the GROUP algorithms (right) additionally require a basis specification. The `basisMaker` object creates a new basis in each superiteration according to prescription (21).

| GRAPE | GROUP |
|---|---|
| `makeGrape_steepest_L2(problem)` | `makeGroup_steepest(problem,basis)` |
| `makeGrape_steepest_H1(problem)` | `makeGroup_bfgs(problem,basis)` |
| `makeGrape_bfgs_L2(problem)` | `makeDGroup_steepest(problem,basisMaker)` |
| `makeGrape_bfgs_H1(problem)` | `makeDGroup_bfgs(problem,basisMaker)` |

### 5.1. *Control of Bose–Hubbard Program — Minimal API*

Here we continue the example program from Section 3.2. In this example we prepare a state transfer problem and solve it using GRAPE, GROUP, and dGROUP. A state transfer problem is encapsulated by a `problem` object,

```
auto problem = makeStateTransferProblem(H, psi_0, psi_t, u, krylovOrder);
```

Here `H` is the Hamiltonian, `psi_0` is the initial state, `psi_t` is the target state, and `u` is the initial control field, which in this case is an exponential ramp as shown in Fig. 4**b**. The `krylovOrder` is the order used internally in the time stepper. This `problem` object maintains a control field and is responsible for calculating the corresponding cost functional and gradient. These quantities are used internally by the optimization algorithm to update the control according to Eq. (13) or (20). Having set up the `problem`, we can apply the different algorithms using the minimal API listed in Table 2,

```
///------------------------- GRAPE -------------------------///
auto GRAPE = makeGrape_bfgs_L2(problem);
GRAPE.optimize();
const auto u_grape = GRAPE.problem().control();

///------------------------- GROUP -------------------------///
const auto M = 60; // basis size
const auto shapeFunction = makeSigmoidShapeFunction(ts);
const auto maxRand = 0.5; // -maxRand < theta_m < maxRand
const auto basis =  shapeFunction*makeSineBasis(M,u.metaData(),maxRand);

auto GROUP = makeGroup_bfgs(problem,basis);
GROUP.optimize(); // begin optimization
const auto u_group = GROUP.problem().control(); // extract GROUP optimized control

///------------------------- dGROUP -------------------------///
auto basisMaker = makeBasisMaker([M,maxRand,&u, &shapeFunction]()
{
```

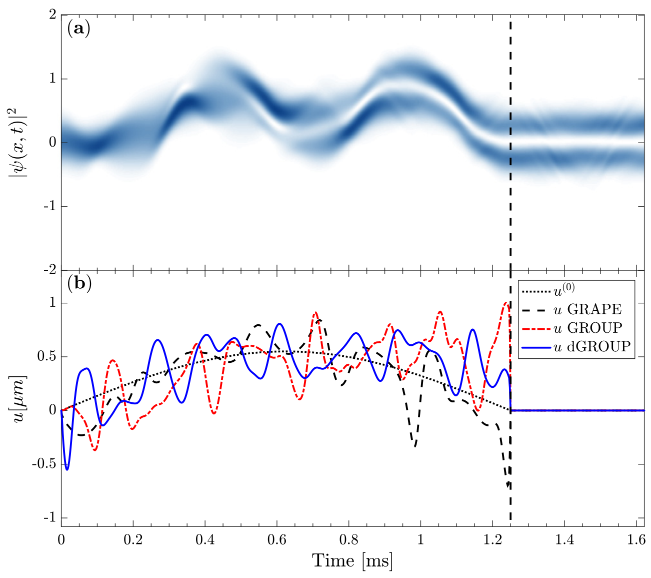

**Fig. 6.** (Color online) (**a**) the density of the condensate $|\psi(x,t)|^2$ when propagated along the optimized dGROUP control from the Gross–Pitaevskii example program. (**b**) the initial control $u^{(0)}$ and the optimized controls from GRAPE $F = 0.99$, GROUP $F = 0.99$ and dGROUP $F = 0.99$ — see legend.

```
return shapeFunction*makeSineBasis(M, u.metaData(), maxRand);
});

auto dGROUP = makeDGroup_bfgs(problem,basisMaker);
dGROUP.optimize(); // begin optimization
const auto u_dgroup = dGROUP.problem().control(); // extract dGROUP optimized control
```

After construction, calling `.optimize()` begins the optimization algorithm. Once the optimization is completed, the optimized control fields are extracted by `.problem().control()`. GRAPE optimizes the control field directly whereas GROUP uses a reduced basis, which must be supplied by the user — see Table 2. In this example we use a sine basis $f_m(t,\theta_m) = \sin((m+\theta_m)\pi t/T)$ where -`maxRand` $\leq \theta_m \leq$ `maxRand` using the `makeSineBasis` factory function which depends on parameters like `dt` and the number of controls, which are conveniently supplied through `u.metaData()`. GROUP also uses a shape function to enforce boundary conditions on the control field equation (16). The default shape function `makeSigmoidShapeFunction` is a symmetric sigmoid function. GROUP uses the same basis for the entire optimization whereas dGROUP uses a new basis in each superiteration through the prescription (21). The new basis is constructed from the `basisMaker` object.

The result of the dGROUP optimization is shown in Fig. 7**a** where the on-site density is plotted as a function of time for the optimized control. This figure should be compared with Fig. 4**a** that also plots the on-site density as a function of time but for the unoptimized control. The optimization increases the fidelity from 0.81 to 0.99 which completes the state transfer. After the vertical dashed line the density and the control is constant since the target state is an eigenstate. In Fig. 7**b** the controls from the other optimization algorithms are plotted.

We have illustrated a central feature in the QEngine, namely how it is possible to generate optimal controls using 3 different algorithms in less than 20 lines of code after the simulation has been set up.

## *5.2. Control of Gross–Pitaevskii program — advanced API*

In this section we continue the example program from Section 3.1. Here we perform GRAPE, GROUP, and dGROUP optimizations using a more advanced API with user specified `Stopper`, `Collector`, and `StepSizeFinder` objects (see Fig. 5).

Calculating the gradient equation (11) requires the derivative of the Hamiltonian wrt the control. The default behavior in the minimal API is to calculate the derivative numerically using finite differences. It is more efficient and accurate to manually supply the analytic derivative using the appropriate factory function.

```
const auto dHdu_func = [&x,p2,p4,p6](const real u)
{
auto x_u = x-u;
auto x_u_Pow2 = x_u*x_u;
auto x_u_Pow3 = x_u*x_u_Pow2;
auto x_u_Pow5 = x_u_Pow2*x_u_Pow3;

return  -(2*p2*x_u + 4*p4*x_u_Pow3 + 6*p6*x_u_Pow5);
};

const auto dHdu = makeAnalyticDiffPotential(makePotentialFunction(dHdu_func,u_initial));

auto problem = makeStateTransferProblem(H,dHdu,psi_0,psi_t,u)
 + 1e-5*Regularization(u)
 + 2e3*Boundaries(u,RVec{-1},RVec{+1});
```

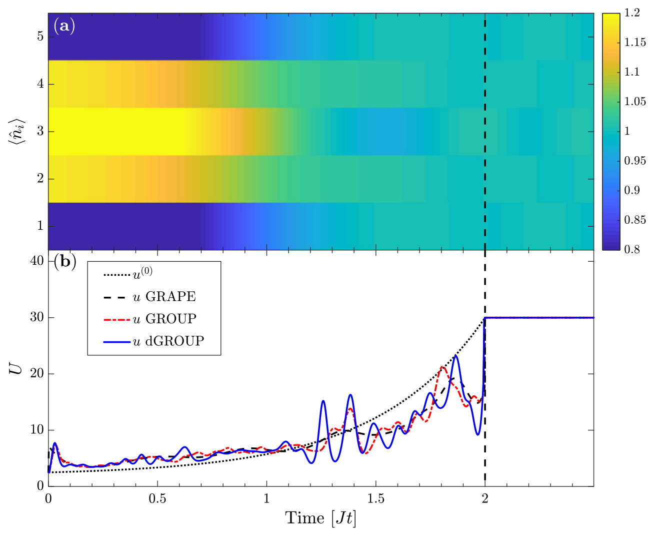

**Fig. 7.** (Color online) (**a**) the on-site density $\langle\hat{n}_i\rangle$ when propagated along the optimized dGROUP control from the Bose–Hubbard example program. (**b**) the initial control $u^{(0)}$ and the optimized controls from GRAPE, GROUP and dGROUP — see legend. All algorithms have fidelity $F = 0.99$ and find almost identical controls.

Note the resemblance to the cost functional equation (7) when initializing the state transfer problem. The last term adds soft bounds to the optimization problem as in Eq. (14) that penalizes control values outside $|u(t)| \geq 1$ μm, which is set by experimental constraints on the control problem [12].

The QEngine allows for arbitrary stopping conditions within its optimization algorithms. After each iteration the optimizer calls a `Stopper` object to check if the optimization should be stopped. This object is instantiated by the `makeStopper` factory function. Any callable taking the optimizer type as its argument and returning a boolean can be passed in to this function. In this example we pass an inline lambda into `makeStopper` using `auto&` to deduce the optimizer type. The interface for the lambda function is fixed since the algorithm expects a callable where it can pass a reference to itself. The same is true for the `Collector` object.

```
// Stopper object
const auto stopper = makeStopper([](auto& optimizer) -> bool
{
bool stop = false;
if (optimizer.problem().fidelity() > 0.99)
{ std::cout << "Fidelity criterion satisfied" << std::endl; stop = true; }
if (optimizer.previousStepSize() < 1e-7)
{ std::cout << "Step size too small" << std::endl; stop = true; };
if (optimizer.iteration() == 200)
{ std::cout << "Max iterations exceeded" << std::endl; stop = true; }
if (stop)
{std::cout << "STOPPING" << std::endl;}
return stop;
});
```

After each iteration the optimizer also calls a `Collector` object and defines what should be displayed, saved, and so on in each iteration. The `Collector` works similarly to the `Stopper` and is instantiated by the `makeCollector` factory function. In this example the fidelity of the current control is saved to the `DataContainer`, and a status message is printed to console.

```
// Collector object
const auto collector = makeCollector([&dc,n_steps](auto& optimizer) {
dc["fidelityHistory"].append(optimizer.problem().fidelity());
std::cout <<
"ITER "       << optimizer.iteration() << " | " <<
"fidelity : " << optimizer.problem().fidelity() << "\t " <<
"stepsize : " << optimizer.stepSize()    << "\t " <<
"fpp : "      << round(optimizer.problem().nPropagationSteps()/n_steps) << "\t " <<
std::endl;
});
```

Next we define a line search algorithm that calculates a suitable step size $\alpha^{(k)}$ along the search direction $p^{(k)}$ in Eqs. (13) and (20). The QEngine supplies an interpolating line search algorithm [63] that can be used out of the box by calling the `makeInterpolating-StepSizeFinder` factory function. It is also possible to create custom line search algorithms. As a simple example, a constant step size strategy is commented out to show the required interface for custom algorithms.

```
// Stepsize finder object
const auto maxStepSize = 5.0;
const auto maxInitGuess = 1.0;
const auto stepSizeFinder = makeInterpolatingStepSizeFinder(maxStepSize,maxInitGuess);

// const auto constStepSize=makeStepSizeFinder([](auto& dir,auto& problem,auto& optimizer)
//{
//return 0.01;
//});
```

Restarting the L-BFGS algorithm is sometimes beneficial as noted in Section 4. In this example, we simply use the default option. The optimizer of choice is then created by calling the corresponding `make` factory function with the additional objects defined above.

```
///------------------------- GRAPE -------------------------///
auto GRAPE = makeGrape_bfgs_L2(problem,stopper,collector,stepSizeFinder);
collector(GRAPE); // collect iteration 0
GRAPE.optimize(); // begin optimization
const auto u_grape = GRAPE.problem().control(); // extract GRAPE optimized control

///------------------------- GROUP -------------------------///
const auto M = 60; // basis size
auto maxRand = 0.5; // -maxRand < theta_m < maxRand
const auto shapeFunction = makeSigmoidShapeFunction(ts);
const auto basis = shapeFunction*makeSineBasis(basisSize,u.metaData(),maxRand);

problem.update(0*u); // update problem such that reference control will be u_ref(t)=0
auto GROUP = makeGroup_bfgs(problem,basis,stopper,collector,stepSizeFinder);

// the initial guess is the first element in the basis, so we may set explicitly:
auto cs = GROUP.problem().coefficients();
cs.at(0).at(0) = initialAmplitude;
GROUP.problem().update(cs); // set initial coefficient vector \vec c = [0.55,0,0,...,0]

collector(GROUP);
GROUP.optimize(); // begin optimization
const auto u_group = GROUP.problem().control(); // extract GROUP optimized control
```

To perform the optimization we invoke `.optimize()` on each algorithm and then retrieve the optimized control. The optimization runs until the `stopper` function returns true. For the GROUP optimizations we use a sine basis equation (16) with randomization. On construction the reference control and coefficients in Eq. (16) are set to u and $\boldsymbol{c} = [0, 0, 0, \dots, 0]^T$. We can manually update them to e.g. $\boldsymbol{c} = [0.55, 0, 0, \dots, 0]^T$ where 0.55 is the initial amplitude as illustrated.

For dGROUP we first prepare a shape function and the `BasisMaker` as in the Bose–Hubbard example program. In dGROUP there is also the possibility to supply a user defined `dressedRestarter` object that in each iteration checks if the algorithm should re-initialize with a new random basis as in Eq. (21), which requires a non-zero bound `maxRand` on the random values $\boldsymbol{\theta}_m$. As a simple example we re-initialize the algorithm every 100 iterations or if there is only a small decrease in the cost

```
///------------------------- dGROUP -------------------------///
const auto basisMaker = makeRandSineBasisMaker(basisSize, shapeFunction, maxRand);

auto restart_func = [tol{ 1e-6 }](const auto& dGROUP) mutable
{
auto stepSize = dGROUP.stepSize();
if (stepSize < tol)
{
std::cout << "New superiteration in dGROUP algorithm." << std::endl;
return true;
}
return false;
};
const auto dRestarter = makeDressedRestarter(restart_func);
problem.update(0*u);
auto dGROUP = makeDGroup_bfgs(problem, basisMaker, stopper,
  collector, stepSizeFinder, dRestarter);

dGROUP.problem().update(cs); // set initial coefficient vector \vec c = [0.55,0,0,...,0]

collector(dGROUP);  // collect iteration 0
dGROUP.optimize(); // begin optimization

const auto u_dgroup = dGROUP.problem().control(); // extract dGROUP optimized control
```

Finally the data from all the optimizations is saved into a `.json` file by calling `dc.save("gpe-example.json")` or alternatively a `.mat` (MATLAB format) file with `dc.save("gpe-example.mat")` if matio has been configured. The result of the dGROUP optimization is shown in Fig. 6**a** where the condensate density is plotted as a function of time. This should be compared with Fig. 3**a** which corresponds to the density evolution when propagating along the unoptimized control. The optimization increases the fidelity from 0.23 to 0.99 which completes the state transfer. The control and the density is constant after the vertical line since the target state (an eigenstate) has been reached. The final controls from all the optimization algorithms are displayed in Fig. 6**b**. The optimized controls are highly complex compared to the initial sine control function. With this example we illustrated how it is easy to configure the control algorithms with custom versions of the `Stopper`, `Collector`, and `StepSizeFinder` objects depicted in Fig. 5. Additionally, we illustrated how to supply analytical derivatives of the Hamiltonian wrt the control and how to configure superiteration restart conditions for dGROUP.

## 6. Summary

We have introduced and described the QEngine as a software library primarily designed for quantum optimal control of several interesting physical models. The interplay between the simulation and control parts were illustrated schematically in a broad sense in Fig. 2 and in more elaborate detail in Fig. 5. Through the Gross–Pitaevskii and Bose–Hubbard example programs in Section 3 we showed how the auto-syntax combined with factory functions allows the user to first set up simulation models in a straightforward manner. In the context of state transfer, we saw in Figs. 3–4 that simple guesses for the control were not enough to obtain a sufficiently high fidelity in either problem which motivated the need for optimal control. In Section 4 we reviewed the relevant theory of quantum optimal control. Finally in Section 5 we showed how to easily apply three different high performance optimal control algorithms to the simulation models. Using quantum optimal control we showed in Figs. 6–7 that the complex optimized solutions solve the state transfer problem. Additionally, we supply users with an extensive API documentation at quatomic.com/qengine/reference.

## Acknowledgments

This work has funded by the European Research Council, Lundbeck Foundation, Carlsberg Foundation, John Templeton Foundation and the Innovation Fund Denmark. This information has also been supplied in the online form. We would also like to thank Birk Skyum for help with testing and building the QEngine.

## Appendix. Units and nondimensionalization

The SI-unit system usually results in very small numerical values for quantum mechanical simulations, which makes simulations impractical or infeasible. For this reason it is beneficial to rescale physical quantities into characteristic scales of the problem such that most values are of order unity. This is achieved using a process known as nondimensionalization where quantities in SI-units are written in product form e.g. $a$ becomes $a = \alpha\tilde{a}$ where $\alpha$ carries both the dimension of $a$ and a magnitude while $\tilde{a}$ is a non-dimensional scaling value. This is done for all quantities and substituted into the equations of motion, which leaves a new set of working equations involving only the dimensionless scaling values. As an example, consider the GPE example program discussed in Section 3.1 where the relevant quantities are

$$x = \chi\tilde{x}, \qquad t = \tau\tilde{t}, \qquad V = \epsilon\tilde{V}, \qquad \psi = \xi\tilde{\psi}, \qquad g_{1\mathrm{D}} = \gamma\tilde{g}_{1\mathrm{D}} \tag{A.1}$$

We may a priori take length to be measured in micrometer and time to be measured in milliseconds.

$$\chi = 1\ \mu\mathrm{m}, \qquad \tau = 1\ \mathrm{ms} \tag{A.2}$$

The three remaining units are chosen conveniently as

$$\epsilon = \frac{\hbar^2}{2\kappa m\chi^2}, \qquad \xi = \sqrt{\frac{N}{\chi}}, \qquad \gamma = \frac{\epsilon}{\xi^2}, \tag{A.3}$$

where $\kappa$ is the kinetic factor. Substituting into the GPE

$$i\frac{\hbar}{\tau}\xi\frac{\partial\tilde{\psi}(\tilde{x},\tilde{t})}{\partial\tilde{t}} = \left(-\kappa\left(\frac{1}{\kappa}\frac{\hbar^2}{2m\chi^2}\right)\frac{\partial^2}{\partial\tilde{x}^2} + \epsilon\tilde{V}(\tilde{x}) + \gamma\xi^2\tilde{g}_{1\mathrm{D}}|\tilde{\psi}(\tilde{x},\tilde{t})|^2\right)\xi\tilde{\psi}(\tilde{x},\tilde{t}) \tag{A.4}$$

Dividing by $\epsilon$ and requiring $\hbar/\tau\epsilon = 1$ or equivalently $\kappa = \tau\hbar/2m\chi^2$ gives

$$i\frac{\partial\tilde{\psi}(\tilde{x},\tilde{t})}{\partial\tilde{t}} = \left(-\kappa\frac{\partial^2}{\partial\tilde{x}^2} + \tilde{V}(\tilde{x}) + \tilde{g}_{1\mathrm{D}}|\tilde{\psi}(\tilde{x},\tilde{t})|^2\right)\tilde{\psi}(\tilde{x},\tilde{t}), \tag{A.5}$$

which is the dimensionless form the GPE solved in the example programs. For the potential we find $\tilde{p}_i$ by comparing

$$\tilde{V} = \frac{V}{\epsilon} = \sum_{i=2,4,6}\frac{p_i}{\epsilon}(x-u)^i = \sum_{i=2,4,6}\left(\frac{p_i\chi^i}{\epsilon}\right)(\tilde{x}-\tilde{u})^i = \sum_{i=2,4,6}\tilde{p}_i(\tilde{x}-\tilde{u})^i \tag{A.6}$$

In these units the nondimensionalized scaling values used in the simulation are [12]

$$\kappa = 0.36537, \qquad \tilde{p}_2 = 65.8392, \qquad \tilde{p}_4 = 97.6349, \qquad \tilde{p}_6 = -15.3850, \qquad \tilde{g}_{1\mathrm{D}} = 1.8299. \tag{A.7}$$

## References

[1] G. Rosi, F. Sorrentino, L. Cacciapuoti, M. Prevedelli, G. Tino, Nature 510 (7506) (2014) 518.
[2] N. Poli, F.-Y. Wang, M. Tarallo, A. Alberti, M. Prevedelli, G. Tino, Phys. Rev. Lett. 106 (3) (2011) 038501.
[3] T. Schumm, S. Hofferberth, L.M. Andersson, S. Wildermuth, S. Groth, I. Bar-Joseph, J. Schmiedmayer, P. Krüger, Nat. Phys. 1 (1) (2005) 57–62.
[4] E. Andersson, T. Calarco, R. Folman, M. Andersson, B. Hessmo, J. Schmiedmayer, Phys. Rev. Lett. 88 (10) (2002) 100401.
[5] Y.-J. Wang, D.Z. Anderson, V.M. Bright, E.A. Cornell, Q. Diot, T. Kishimoto, M. Prentiss, R. Saravanan, S.R. Segal, S. Wu, Phys. Rev. Lett. 94 (9) (2005) 090405.
[6] I. Bloch, J. Dalibard, S. Nascimbene, Nat. Phys. 8 (4) (2012) 267–276.
[7] I. Georgescu, S. Ashhab, F. Nori, Rev. Modern Phys. 86 (1) (2014) 153.
[8] A. Kaufman, B. Lester, M. Foss-Feig, M. Wall, A. Rey, C. Regal, Nature 527 (7577) (2015) 208.
[9] G. De Chiara, T. Calarco, M. Anderlini, S. Montangero, P. Lee, B. Brown, W. Phillips, J. Porto, Phys. Rev. A 77 (5) (2008) 052333.
[10] C. Weitenberg, S. Kuhr, K. Mølmer, J.F. Sherson, Phys. Rev. A 84 (3) (2011) 032322.
[11] I. Bloch, J. Dalibard, W. Zwerger, Rev. Modern Phys. 80 (3) (2008) 885.
[12] S. van Frank, M. Bonneau, J. Schmiedmayer, S. Hild, C. Gross, M. Cheneau, I. Bloch, T. Pichler, A. Negretti, T. Calarco, et al., Optimal control of complex atomic quantum systems, Sci. Rep. 6.
[13] S.J. Glaser, U. Boscain, T. Calarco, C.P. Koch, W. Köckenberger, R. Kosloff, I. Kuprov, B. Luy, S. Schirmer, T. Schulte-Herbrüggen, et al., Eur. Phys. J. D 69 (12) (2015) 279.
[14] J. Werschnik, E. Gross, J. Phys. B: At. Mol. Opt. Phys. 40 (18) (2007) R175.
[15] G. Jäger, U. Hohenester, Phys. Rev. A 88 (3) (2013) 035601.
[16] G. Jäger, D.M. Reich, M.H. Goerz, C.P. Koch, U. Hohenester, Phys. Rev. A 90 (3) (2014) 033628.

[17] M. Ndong, C.P. Koch, Phys. Rev. A 82 (4) (2010) 043437.
[18] C.P. Koch, J.P. Palao, R. Kosloff, F. Masnou-Seeuws, Phys. Rev. A 70 (1) (2004) 013402.
[19] P. Doria, T. Calarco, S. Montangero, Phys. Rev. Lett. 106 (19) (2011) 190501.
[20] R. Bücker, T. Berrada, S. Van Frank, J.-F. Schaff, T. Schumm, J. Schmiedmayer, G. Jäger, J. Grond, U. Hohenester, J. Phys. B: At. Mol. Opt. Phys. 46 (10) (2013) 104012.
[21] T. Caneva, M. Murphy, T. Calarco, R. Fazio, S. Montangero, V. Giovannetti, G.E. Santoro, Phys. Rev. Lett. 103 (24) (2009) 240501.
[22] G.C. Hegerfeldt, Phys. Rev. Lett. 111 (26) (2013) 260501.
[23] S. Deffner, S. Campbell, J. Phys. A 50 (45) (2017) 453001.
[24] I. Brouzos, A.I. Streltsov, A. Negretti, R.S. Said, T. Caneva, S. Montangero, T. Calarco, Phys. Rev. A 92 (6) (2015) 062110.
[25] N. Khaneja, T. Reiss, C. Kehlet, T. Schulte-Herbrüggen, S.J. Glaser, J. Magn. Reson. 172 (2) (2005) 296–305.
[26] A. Assion, T. Baumert, M. Bergt, T. Brixner, B. Kiefer, V. Seyfried, M. Strehle, G. Gerber, Science 282 (5390) (1998) 919–922.
[27] F. Dolde, V. Bergholm, Y. Wang, I. Jakobi, B. Naydenov, S. Pezzagna, J. Meijer, F. Jelezko, P. Neumann, T. Schulte-Herbrüggen, et al., Nat. Commun. 5 (2014) 3371.
[28] J.-F. Mennemann, D. Matthes, R.-M. Weishäupl, T. Langen, New J. Phys. 17 (11) (2015) 113027.
[29] U. Hohenester, P.K. Rekdal, A. Borzì, J. Schmiedmayer, Phys. Rev. A 75 (2) (2007) 023602.
[30] U. Hohenester, Comput. Phys. Comm. 185 (1) (2014) 194–216.
[31] S. Machnes, U. Sander, S. Glaser, P. de Fouquieres, A. Gruslys, S. Schirmer, T. Schulte-Herbrüggen, Phys. Rev. A 84 (2) (2011) 022305.
[32] B. Schmidt, U. Lorenz, Comput. Phys. Comm. 213 (2017) 223–234.
[33] B. Schmidt, C. Hartmann, Comput. Phys. Comm. 228 (2018) 229–244.
[34] J. Johansson, P. Nation, F. Nori, Comput. Phys. Comm. 183 (8) (2012) 1760–1772.
[35] C. Sanderson, R. Curtin, J. Open Source Softw. 1 (2016) 26.
[36] J. Sørensen, M. Aranburu, T. Heinzel, J. Sherson, Phys. Rev. A 98 (2) (2018) 022119.
[37] F. Dalfovo, S. Giorgini, L.P. Pitaevskii, S. Stringari, Rev. Modern Phys. 71 (3) (1999) 463.
[38] M. Olshanii, Phys. Rev. Lett. 81 (5) (1998) 938.
[39] C.M. Dion, E. Cancès, Comput. Phys. Commun. 177 (10) (2007) 787–798.
[40] W. Bao, D. Jaksch, P.A. Markowich, J. Comput. Phys. 187 (1) (2003) 318–342.
[41] T.R. Taha, M.I. Ablowitz, J. Comput. Phys. 55 (2) (1984) 203–230.
[42] D. Barredo, S. De Léséleuc, V. Lienhard, T. Lahaye, A. Browaeys, Science 354 (6315) (2016) 1021–1023.
[43] M. Endres, H. Bernien, A. Keesling, H. Levine, E.R. Anschuetz, A. Krajenbrink, C. Senko, V. Vuletic, M. Greiner, M.D. Lukin, Science 354 (6315) (2016) 1024–1027.
[44] G. De Chiara, T. Calarco, M. Anderlini, S. Montangero, P. Lee, B. Brown, W. Phillips, J. Porto, Phys. Rev. A 77 (5) (2008) 052333.
[45] M. Anderlini, P.J. Lee, B.L. Brown, J. Sebby-Strabley, W.D. Phillips, J. Porto, Nature 448 (7152) (2007) 452–456.
[46] T.J. Park, J. Light, J. Chem. Phys. 85 (10) (1986) 5870–5876.
[47] R. Bücker, J. Grond, S. Manz, T. Berrada, T. Betz, C. Koller, U. Hohenester, T. Schumm, A. Perrin, J. Schmiedmayer, Nat. Phys. 7 (8) (2011) 608–611.
[48] F. Gerbier, Europhys. Lett. 66 (6) (2004) 771.
[49] D. Jaksch, H.-J. Briegel, J. Cirac, C. Gardiner, P. Zoller, Phys. Rev. Lett. 82 (9) (1999) 1975.
[50] O. Mandel, M. Greiner, A. Widera, T. Rom, T.W. Hänsch, I. Bloch, Nature 425 (6961) (2003) 937.
[51] G.K. Brennen, C.M. Caves, P.S. Jessen, I.H. Deutsch, Phys. Rev. Lett. 82 (5) (1999) 1060.
[52] D. Jaksch, J. Cirac, P. Zoller, S. Rolston, R. Côté, M. Lukin, Phys. Rev. Lett. 85 (10) (2000) 2208.
[53] J.K. Pachos, P.L. Knight, Phys. Rev. Lett. 91 (10) (2003) 107902.
[54] E. Jané, G. Vidal, W. Dür, P. Zoller, J.I. Cirac, Quantum Inf. Comput. 3 (1) (2003) 15–37.
[55] A. Micheli, A. Daley, D. Jaksch, P. Zoller, Phys. Rev. Lett. 93 (14) (2004) 140408.
[56] S. Braun, M. Friesdorf, S.S. Hodgman, M. Schreiber, J.P. Ronzheimer, A. Riera, M. del Rey, I. Bloch, J. Eisert, U. Schneider, Proc. Natl. Acad. Sci. 112 (12) (2015) 3641–3646.
[57] M. Greiner, O. Mandel, T. Esslinger, T.W. Hänsch, I. Bloch, Nature 415 (6867) (2002) 39.
[58] J. Zakrzewski, D. Delande, Phys. Rev. A 80 (1) (2009) 013602.
[59] S. Rosi, A. Bernard, N. Fabbri, L. Fallani, C. Fort, M. Inguscio, T. Calarco, S. Montangero, Phys. Rev. A 88 (2) (2013) 021601.
[60] G. Von Winckel, A. Borzì, Inverse Problems 24 (3) (2008) 034007.
[61] J. Nocedal, S.J. Wright, Numerical Optimization, second ed., 2006.
[62] N. Rach, M.M. Müller, T. Calarco, S. Montangero, Phys. Rev. A 92 (6) (2015) 062343.
[63] J.J. Moré, D.J. Thuente, ACM Trans. Math. Software 20 (3) (1994) 286–307.