NESSi: The Non-Equilibrium Systems Simulation package[☆,☆☆]

Michael Schüler [a,b], Denis Golež [a,c], Yuta Murakami [a,d], Nikolaj Bittner [a], Andreas Herrmann [a], Hugo U.R. Strand [c,e], Philipp Werner [a], Martin Eckstein [f,*]

[a] *Department of Physics, University of Fribourg, 1700 Fribourg, Switzerland*
[b] *Stanford Institute for Materials and Energy Sciences, SLAC & Stanford University, Stanford, CA 94025, USA*
[c] *Center for Computational Quantum Physics, Flatiron Institute, 162 Fifth avenue, New York, NY 10010, USA*
[d] *Department of Physics, Tokyo Institute of Technology, Meguro, Tokyo 152-8551, Japan*
[e] *Department of Physics, Chalmers University of Technology, SE-412 96 Gothenburg, Sweden*
[f] *Department of Physics, University of Erlangen-Nürnberg, 91058 Erlangen, Germany*

Keywords:
Numerical simulations
Nonequilibrium dynamics of quantum many-body problems
Keldysh formalism
Kadanoff–Baym equations

A B S T R A C T

The nonequilibrium dynamics of correlated many-particle systems is of interest in connection with pump–probe experiments on molecular systems and solids, as well as theoretical investigations of transport properties and relaxation processes. Nonequilibrium Green's functions are a powerful tool to study interaction effects in quantum many-particle systems out of equilibrium, and to extract physically relevant information for the interpretation of experiments. We present the open-source software package `NESSi` (The **N**on-**E**quilibrium **S**ystems **Si**mulation package) which allows to perform many-body dynamics simulations based on Green's functions on the L-shaped Kadanoff–Baym contour. `NESSi` contains the library `libcntr` which implements tools for basic operations on these nonequilibrium Green's functions, for constructing Feynman diagrams, and for the solution of integral and integro-differential equations involving contour Green's functions. The library employs a discretization of the Kadanoff–Baym contour into time N points and a high-order implementation of integration routines. The total integrated error scales up to $\mathcal{O}(N^{-7})$, which is important since the numerical effort increases at least cubically with the simulation time. A distributed-memory parallelization over reciprocal space allows large-scale simulations of lattice systems. We provide a collection of example programs ranging from dynamics in simple two-level systems to problems relevant in contemporary condensed matter physics, including Hubbard clusters and Hubbard or Holstein lattice models. The `libcntr` library is the basis of a follow-up software package for nonequilibrium dynamical mean-field theory calculations based on strong-coupling perturbative impurity solvers.

Program summary
Program Title: NESSi
CPC Library link to program files: http://dx.doi.org/10.17632/973crf9hgd.1
Licensing provisions: MPL v2.0
Programming language: C++, python
External routines/libraries: cmake, eigen3, hdf5 (optional), mpi (optional), omp (optional)
Nature of problem: Solves equations of motion of time-dependent Green's functions on the Kadanoff–Baym contour.
Solution method: Higher-order solution methods of integral and integro-differential equations on the Kadanoff–Baym contour.

Part I. Core functionalities and usage of the library

In this part, we discuss Core functionalities and usage of the library.

☆ The review of this paper was arranged by Prof. N.S. Scott.
☆☆ 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: martin.eckstein@fau.de (M. Eckstein).

List of abbreviations

Notation	Description
1D	one-dimensional
2B	second-Born
BDF	backward differentiation formula
BZ	Brillouin zone
el-ph	electron-phonon
DMFT	dynamical mean-field theory
GF	Green's function
GFs	Green's functions
HDF5	Hierarchical Data Format version 5
HF	Hartree–Fock
KB	Kadanoff–Baym
KMS	Kubo–Martin–Schwinger
MPI	Message passing interface
NCA	Non-Crossing Approximation
NEGF	nonequilibrium Green's function
NEGFs	nonequilibrium Green's functions
OCA	One-Crossing Approximation
PPSC	Pseudo-Particle Strong Coupling
uMig	unrenormalized Migdal approximation
VIDE	Volterra integro-differential equation
VIDEs	Volterra integro-differential equations
VIE	Volterra integral equation
VIEs	Volterra integral equations

1. Introduction

Calculating the time evolution of an interacting quantum many-body system poses significant computational challenges. For instance, in wave-function based methods such as exact diagonalization or the density matrix renormalization group [1,2], one has to solve the time-dependent Schrödinger equation. The main obstacles here are the exponential scaling of the Hilbert space with system size, or the rapid entanglement growth. Variational methods avoid this problem [3], but their accuracy depends on the ansatz for the wave function. The Green's function formalism [4] provides a versatile framework to derive systematic approximations or to develop numerical techniques (e.g. Quantum Monte Carlo [5]) that circumvent the exponential scaling of the Hilbert space. Moreover, the Green's functions contain useful physical information that can be directly related to measurable quantities such as the photoemission spectrum.

The nonequilibrium Green's function (NEGF) approach, as pioneered by Keldysh, Kadanoff and Baym [6,7], is an extension of the equilibrium (Matsubara) Green's function technique [8,9]. It not only defines the analytical foundation for important concepts in nonequilibrium theory, such as the quantum Boltzmann equation [10], but it also serves as a basis for numerical simulations. The direct numerical solution of the equations of motion for the real-time nonequilibrium Green's functions (NEGFs) [11,12] has been successfully applied to the study of open and closed systems ranging from molecules to condensed matter, with various types of interactions including electron–electron, electron–phonon, or electron–photon couplings [9]. At the heart of these simulations lies the solution of integro-differential equations which constitute non-Markovian equations of motion for the NEGFs, the so-called Kadanoff–Baym equations. Even in combination with simple perturbative approximations, their solution remains a formidable numerical task.

A general NEGF calculation is based on the L-shaped KB contour $\mathcal{C} = \mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{C}_3$ in the complex time plane, which is sketched in Fig. 1. Here, the vertical (imaginary-time) branch $\mathcal{C}_3$ of the contour represents the initial equilibrium state of the system ($\beta = 1/k_B T$ is the inverse temperature), while the horizontal branches $\mathcal{C}_{1,2}$ represent the time evolution of the system starting from this equilibrium state. Correlation functions with time arguments on this contour, and a time ordering defined by the arrows on the contour, are a direct generalization of the corresponding imaginary-time quantities. Hence, diagrammatic techniques and concepts which have been established for equilibrium many-body problems can be directly extended to the nonequilibrium domain.

In this paper, we introduce the **N**on**E**quilibrium **S**ystems **Si**mulation package (`NESSi`) as a state-of-the art tool for solving nonequilibrium many-body problems. `NESSi` provides an efficient framework for representing various types of Green's functions (GFs) on the discretized KB contour, implements the basic operations on these functions and allows to solve the corresponding equations of motion. The library is aimed at the study of transient dynamics from an initial equilibrium state, induced by parameter modulations or electric field excitations. It is not designed for the direct study of nonequilibrium steady states or time-periodic Floquet states, where the memory of the initial state is lost and thus the branch $\mathcal{C}_3$ is not needed. While steady states can be reached in open systems in a relatively short time, depending on parameters and driving conditions, such simulations may be more efficiently implemented with a dedicated steady-state or Floquet code.

The two-fold purpose of this paper is to explain both the numerical details underlying the solution of the integro-differential equations, and the usage and core functionalities of the library. The paper is therefore structured such that a reader who is mainly interested in using the library may consider only the first part of the text (Sections 2–7), while (Sections 8–14) contains an explanation

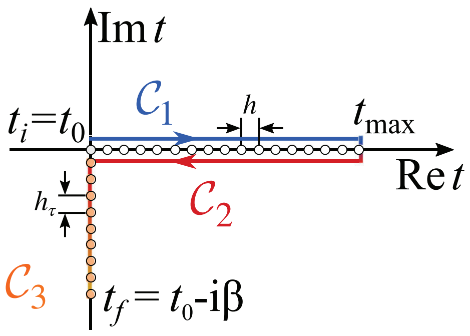


Fig. 1. L-shaped Kadanoff–Baym contour $\mathcal{C}$ in the complex time plane, containing the forward branch $\mathcal{C}_1$, backward branch $\mathcal{C}_2$, and the imaginary (Matsubara) branch $\mathcal{C}_3$. Gray dots indicate the discretization on the real-time branches, while orange dots represent the discretization of the Matsubara branch. The arrows indicate the contour ordering. (color online).

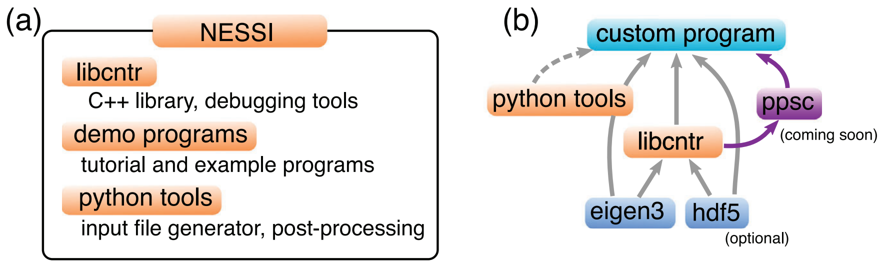


Fig. 2. (a) Basic structure of the `NESSi` software package. The core ingredient is the shared library `libcntr`, which contains basic classes and routines for storing and manipulating nonequilibrium Green functions. Furthermore, we provide a tutorial and demonstration programs which illustrate the usage and functionalities of `libcntr`. (b) Custom programs based on `libcntr` should be linked against the `libcntr` library and the dependencies `eigen3` and `hdf5` (optional). An extension of the `libcntr` library for dynamical mean field theory calculations is in preparation and will be published separately (PPSC library).

of the numerical methods. We remark that the usage of the library is also explained in a detailed and independent online manual on the webpage www.nessi.tuxfamily.org.

This paper is organized as follows. In Section 2 we present an overview of the basic structure of the `NESSi` software package, its core ingredients, and the main functionalities. This overview is kept brief to serve as a reference for readers who are familiar with the formalism. A detailed description of the general formalism is provided in Section 3, while Section 4 introduces the fundamental equations of motion on the KB contour and discusses their solution, as implemented within `NESSi`. Section 5 explains how to compile the `NESSi` software and how to use its functionalities in custom projects. Several illustrative examples are presented and discussed in Section 6, and Section 7 explains how to use the code package to solve a large number of coupled integral equations with a distributed memory parallelization. Finally, the numerical details are presented in Sections 8–10: starting from highly accurate methods for quadrature and integration (Section 8), we explain the numerical procedures underlying the solution of the equations on the KB contour. In Sections 11–14 we discuss the implementation of the main functions.

2. Overview of the program package

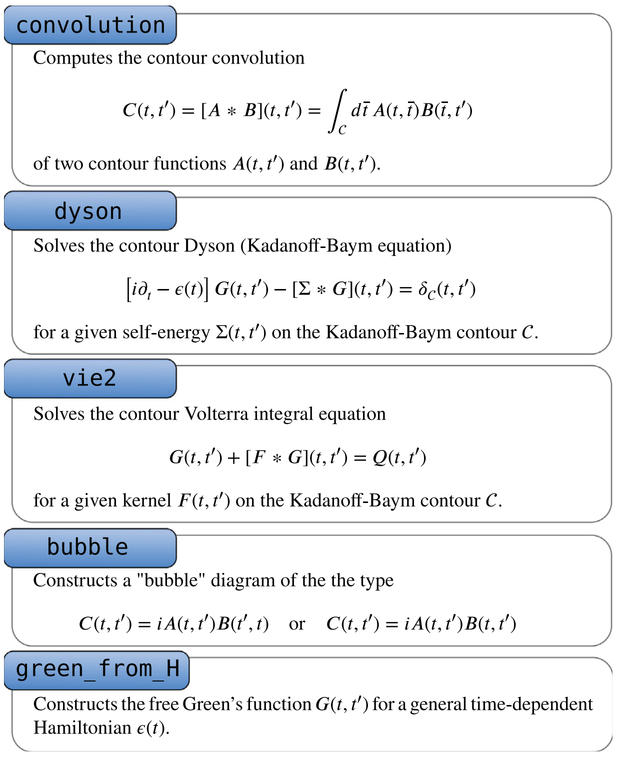
2.1. Structure of the software

Fig. 2 summarizes the content of the `NESSi` package. The core constituent of `NESSi` is the shared library `libcntr`. The `libcntr` library is written in C++ and provides the essential functionalities to treat GFs on the KB contour. To solve a particular problem within the NEGF formalism, the user can write a custom C++ program based on the extensive and easy-to-use `libcntr` library (see Fig. 2(b)). The `NESSi` package also contains a number of simple example programs, which demonstrate the usage of `libcntr`. All important callable routines perform various sanity checks in debugging mode, which enables an efficient debugging of `libcntr`-based programs. Furthermore, we provide a number of python tools for pre- and post-processing to assist the use of programs based on `libcntr`. More details can be found in Section 6, where we present a number of example programs demonstrating the usage of `libcntr` and the python tools. The `libcntr` library and the example programs depend on the `eigen3` library which implements efficient matrix operations. Furthermore, the `hdf5` library and file format can be used for creating binary, machine-independent output for GFs and related objects. We further provide python tools for reading and post-processing GFs from `hdf5` format via the `h5py` python package. The usage of the `hdf5` library in the `NESSi` package is, however, optional.

2.2. Core functionalities

The central task within the NEGF framework is calculating the single-particle Green's function (GF), from which all single-particle observables such as the density or the current can be evaluated. Let us consider the generic many-body Hamiltonian

$$\hat{H}(t) = \sum_{a,b} \epsilon_{a,b}(t)\hat{c}_a^\dagger \hat{c}_b + \hat{V}(t) , \tag{1}$$



`convolution`

Computes the contour convolution

$$C(t,t') = [A * B](t,t') = \int_C d\bar{t}\, A(t,\bar{t})B(\bar{t},t')$$

of two contour functions $A(t,t')$ and $B(t,t')$.

`dyson`

Solves the contour Dyson (Kadanoff-Baym equation)

$$\left[i\partial_t - \epsilon(t)\right] G(t,t') - [\Sigma * G](t,t') = \delta_C(t,t')$$

for a given self-energy $\Sigma(t,t')$ on the Kadanoff-Baym contour C.

`vie2`

Solves the contour Volterra integral equation

$$G(t,t') + [F * G](t,t') = Q(t,t')$$

for a given kernel $F(t,t')$ on the Kadanoff-Baym contour C.

`bubble`

Constructs a "bubble" diagram of the the type

$$C(t,t') = i A(t,t')B(t',t) \quad \text{or} \quad C(t,t') = i A(t,t')B(t,t')$$

`green_from_H`

Constructs the free Green's function $G(t,t')$ for a general time-dependent Hamiltonian $\epsilon(t)$.

Fig. 3. Main routines for constructing and manipulating objects on the KB contour and solving the corresponding equations of motion.

where $\hat{c}_a^\dagger$ ($\hat{c}_a$) denotes the fermionic or bosonic creation (annihilation) operator with respect to some basis labelled by a and $\epsilon_{a,b}(t)$ the corresponding single-particle Hamiltonian, while $\hat{V}(t)$ represents an arbitrary interaction term or the coupling to a bath. In typical problems, the GF is obtained by solving the Dyson equation

$$[i\partial_t - \epsilon(t)]\, G(t,t') - \int_C d\bar{t}\; \Sigma(t,\bar{t})G(\bar{t},t') = \delta_C(t,t') \tag{2}$$

or related integral equations. Here all objects are matrices in orbital indices. The symbols $\int_C$ and $\delta_C(t,t')$ denote an integral and the Dirac delta function defined on the KB contour, respectively, while $\Sigma(t,t')$ is the self-energy, which captures all interaction effects originating from $\hat{V}(t)$. Details are discussed in Section 3.

The `libcntr` library provides accurate methods for solving the Dyson equation (2) and related problems. A brief overview of the core routines is presented in Fig. 3. It includes routines for computing the convolution $[A*B](t,t') = \int_C d\bar{t}\, A(t,\bar{t})B(\bar{t},t')$ (`convolution`), which is an essential part of solving Eq. (2). It furthermore provides a high-order solver for Eq. (2) (`dyson`) along the full KB contour. In particular, the initial thermal equilibrium state and the time evolution are treated on equal footing. Moreover, contour integral equations of the type

$$G(t,t') + [F * G](t,t') = Q(t,t') \tag{3}$$

can be solved efficiently via `vie2`. A typical example is the self-consistent *GW* approximation [13]: the screened interaction obeys the Dyson equation $W = V + V * \Pi * W$, where V denotes the bare Coulomb interaction, while Π stands for the irreducible polarization, see Section 6. Free GF with respect to a given single particle Hamiltonian $\epsilon(t)$ are computed by `green_from_H`. Finally, in many-body theories, the self-energy $\Sigma(t,t')$ can be expressed in terms of Feynman diagrams using the GF themselves. The most common Feynman diagrams consist of products of two propagators, implemented as `bubble` in `libcntr`. All routines work both for fermions and bosons.

2.3. Perspective: dynamical mean-field theory

While the `NESSi` package provides a general framework for the manipulation of real-time GFs and can be used in different types of applications, one particularly fruitful application has been the nonequilibrium extension of DMFT [14]. In order to perform DMFT calculations the library needs to be supplemented with a solver for the DMFT effective impurity problem. Typical approximate approaches are weak coupling expansions [15] (in particular Iterated Perturbation Theory, IPT) and strong coupling methods. The weak coupling expansions can be directly implemented with the help of the routines provided by `libcntr`. The strong coupling based methods [16] are based on pseudo-particle GFs, which are defined for each state in the local Hilbert space and have properties different from the normal GFs introduced below. This formulation solves the atomic problem exactly and treats the hybridization with the environment perturbatively. The first and second order dressed expansion of this method is commonly known as the Non-Crossing Approximation (NCA) and the One-Crossing Approximation (OCA) [17,18]. Currently we are working on a library implementing these methods called the Pseudo-Particle Strong Coupling (PPSC) library – based on `libcntr` – and plan to release it as a future extension of `NESSi`.

3. Basic formalism: NEGFs on the contour

GFs are objects depending on position, orbital and spin arguments (or an equivalent basis representation), as well as on two time arguments. The dependence on multiple time arguments does not only account for the explicit time dependence of observables, it also allows to store information on the characteristic energy scales of the system and its thermal equilibrium state. All these ingredients can be incorporated on equal footing by choosing the time arguments on the KB contour $\mathcal{C}$ illustrated in Fig. 1: $t \in \mathcal{C} = \mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{C}_3$. The directions of the arrows in Fig. 1 define the induced ordering of time arguments $t_1, t_2 \in \mathcal{C}$: we call t_2 later than t_1 (denoted by $t_2 \succ t_1$) if t_2 can be reached by progressing along $\mathcal{C}$ as indicated by the arrows. Thus, contour arguments on the backward branch $t_2 \in \mathcal{C}_2$ are always later than $t_1 \in \mathcal{C}_1$.

Let us furthermore define the many-body Hamiltonian $\hat{H}_{\mathcal{C}}(t)$ on the contour $\mathcal{C}$ by $\hat{H}_{\mathcal{C}}(t) = \hat{H}(t) - \mu\hat{N}$ for $t \in \mathcal{C}_{1,2}$ and $\hat{H}_{\mathcal{C}}(t) = \hat{H}_{\mathrm{eq}} - \mu\hat{N}$ for $t \in \mathcal{C}_3$. Here, $\hat{H}(t)$ denotes the real-time Hamiltonian, while $\hat{H}_{\mathrm{eq}}$ describes the system in thermal equilibrium, given by a grand-canonical ensemble with chemical potential μ and particle number operator $\hat{N}$. The latter is described by the many-body density matrix

$$\hat{\rho} = \frac{1}{Z} e^{-\beta(\hat{H}_{\mathrm{eq}} - \mu\hat{N})} = \frac{1}{Z} e^{-\beta\hat{H}_{\mathcal{C}}(-i\beta)} \tag{4}$$

and the partition function by $Z = \mathrm{Tr}[e^{-\beta(\hat{H}_{\mathrm{eq}} - \mu\hat{N})}] = \mathrm{Tr}[e^{-\beta\hat{H}_{\mathcal{C}}(-i\beta)}]$. The time evolution of any observable is governed by the time-evolution operator

$$\hat{U}(t_1, t_2) = T \exp\left[-i\int_{t_2}^{t_1} dt\, \hat{H}_{\mathcal{C}}(t)\right] \ , \quad t_1 > t_2 \ , \tag{5a}$$

$$\hat{U}(t_1, t_2) = \bar{T} \exp\left[i\int_{t_1}^{t_2} dt\, \hat{H}_{\mathcal{C}}(t)\right] \ , \quad t_2 > t_1 \ , \tag{5b}$$

where T ($\bar{T}$) denotes the chronological (anti-chronological) time ordering symbol. Time-dependent ensemble averages of an operator $\hat{A}(t)$ (here we refer to an explicit time dependence in the Schrödinger picture) are given by time evolving the density matrix according to

$$\langle\hat{A}(t)\rangle = \mathrm{Tr}\left[\hat{U}(t, 0)\hat{\rho}\hat{U}(0, t)\hat{A}(t)\right] \ , \tag{6}$$

which we can formally rewrite as

$$\langle\hat{A}(t)\rangle = \frac{1}{Z}\mathrm{Tr}\left[\hat{U}(-i\beta, 0)\hat{U}(0, t)\hat{A}(t)\hat{U}(t, 0)\right] \ . \tag{7}$$

This is where the contour $\mathcal{C}$ comes into play: the time arguments (from right to left) in Eq. (7) follow the KB contour, passing through $\mathcal{C}_1$, $\mathcal{C}_2$ and finally through $\mathcal{C}_3$. If we now introduce the contour ordering symbol $T_{\mathcal{C}}$ which orders the operators along the KB contour, any expectation value can be written as

$$\langle\hat{A}(t)\rangle = \frac{\mathrm{Tr}\left[T_{\mathcal{C}} \exp\left(-i\int_{\mathcal{C}} d\bar{t}\, \hat{H}_{\mathcal{C}}(\bar{t})\right)\hat{A}(t)\right]}{\mathrm{Tr}\left[T_{\mathcal{C}} \exp\left(-i\int_{\mathcal{C}} d\bar{t}\, \hat{H}_{\mathcal{C}}(\bar{t})\right)\right]} . \tag{8}$$

Note that the integrals over $\mathcal{C}_1$ and $\mathcal{C}_2$ in the denominator cancel, such that it becomes equivalent to the partition function Z, while the matrix exponential in the numerator is equivalent to the time evolution in Eq. (7).

General correlators with respect to operators $\hat{A}(t)$ and $\hat{B}(t')$ are similarly defined on $\mathcal{C}$ as

$$C_{AB}(t, t') = \frac{\mathrm{Tr}\left[T_{\mathcal{C}} \exp\left(-i\int_{\mathcal{C}} d\bar{t}\, \hat{H}_{\mathcal{C}}(\bar{t})\right)\hat{A}(t)\hat{B}(t')\right]}{\mathrm{Tr}\left[T_{\mathcal{C}} \exp\left(-i\int_{\mathcal{C}} d\bar{t}\, \hat{H}_{\mathcal{C}}(\bar{t})\right)\right]} \equiv \langle T_{\mathcal{C}}\hat{A}(t)\hat{B}(t')\rangle \ . \tag{9}$$

Fermionic and bosonic particles are characterized by different commutation relations. Throughout this paper, we associate fermions (bosons) with the negative (positive) sign $\xi = -1$ ($\xi = 1$). Assuming $\hat{A}$, $\hat{B}$ to be pure fermionic or bosonic operators, the contour ordering in the definition (9) is defined by

$$T_{\mathcal{C}}\left\{\hat{A}(t_1)\hat{B}(t_2)\right\} = \begin{cases} \hat{A}(t_1)\hat{B}(t_2) & : t_1 \succ t_2 \\ \xi\hat{B}(t_2)\hat{A}(t_1) & : t_2 \succ t_1 \ . \end{cases} \tag{10}$$

All two-time correlators like the GF (12) fulfill the Kubo–Martin–Schwinger (KMS) boundary conditions

$$G(0, t') = \xi G(-i\beta, t') \ , \quad G(t, 0) = \xi G(t, -i\beta) \ . \tag{11}$$

One of the most important correlators on the KB contour is the single-particle GF, defined by

$$G_{ab}(t, t') = -i\langle T_{\mathcal{C}}\hat{c}_a(t)\hat{c}_b^\dagger(t')\rangle \ , \tag{12}$$

where $\hat{c}_a^\dagger$ ($\hat{c}_a$) denotes the fermionic or bosonic creation (annihilation) operator with respect to the single-particle state a. Higher-order correlators with multiple contour arguments, like two-particle GFs, are defined in an analogous way.

Table 1
Keldysh components of a function $C(t_1, t_2)$ with arguments on $\mathcal{C}$.

$t_1 \in$	$t_2 \in$	Notation	Name
$\mathcal{C}_1$	$\mathcal{C}_1$	$C^{\mathrm{T}}(t_1, t_2)$	Causal (time-ordered)
$\mathcal{C}_1$	$\mathcal{C}_2$	$C^{<}(t_1, t_2)$	Lesser
$\mathcal{C}_1$	$\mathcal{C}_3(t_2 = -i\tau_2)$	$C^{\rceil}(t_1, \tau_2)$	Left-mixing
$\mathcal{C}_2$	$\mathcal{C}_1$	$C^{>}(t_1, t_2)$	Greater
$\mathcal{C}_2$	$\mathcal{C}_2$	$C^{\bar{\mathrm{T}}}(t_1, t_2)$	Anti-causal
$\mathcal{C}_2$	$\mathcal{C}_3(t_2 = -i\tau_2)$	$C^{\rceil}(t_1, \tau_2)$	Left-mixing
$\mathcal{C}_3(t_1 = -i\tau_1)$	$\mathcal{C}_1$	$C^{\lceil}(\tau_1, t_2)$	Right-mixing
$\mathcal{C}_3(t_1 = -i\tau_1)$	$\mathcal{C}_2$	$C^{\lceil}(\tau_1, t_2)$	Right-mixing
$\mathcal{C}_3(t_1 = -i\tau_1)$	$\mathcal{C}_3(t_2 = -i\tau_2)$	$C(-i\tau_1, -i\tau_2)$	Imaginary time-ordered

Generalizing Eq. (9), contour correlators can also be defined with respect to any action $\hat{S}$,

$$C_{AB}(t, t') = \frac{\mathrm{Tr}\left[T_{\mathcal{C}} e^{\hat{S}} \hat{A}(t)\hat{B}(t')\right]}{\mathrm{Tr}\left[T_{\mathcal{C}} e^{\hat{S}}\right]} \equiv \langle T_{\mathcal{C}} \hat{A}(t)\hat{B}(t')\rangle_{\mathcal{S}} \ . \tag{13}$$

For instance, Eq. (9) corresponds to the action $\hat{S} = -i\int_{\mathcal{C}} d\bar{t}\, \hat{H}_{\mathcal{C}}(\bar{t})$, but all the properties and procedures discussed in this work remain valid if the action $\hat{S}$ is extended to a more general form. A typical example is the action encountered in the framework of DMFT [14],

$$\hat{S} = -i\int_{\mathcal{C}} dt\, \hat{H}_{\mathcal{C}}(t) - i\int_{\mathcal{C}} dt \int_{\mathcal{C}} dt'\, \hat{c}^{\dagger}(t)\Delta(t, t')\hat{c}(t') \ , \tag{14}$$

where $\Delta(t, t')$ is the so-called hybridization function.

3.1. Contour decomposition

While the real-time GFs is defined for any pair of arguments (t, t') on the L-shaped KB contour $\mathcal{C}$, it can be decomposed in a number of components where each of the two time arguments is constrained to a specific branch $\mathcal{C}_j$ of the KB contour. These, which will generally be called Keldysh components in the following,[1] are summarized in Table 1. We note that time arguments t, t' are used both to represent contour arguments as well as real times, and whenever a correlator $C(t, t')$ occurs without a superscript specifying the Keldysh components, the time arguments t, t' are to be understood as contour arguments.

In addition to the Keldysh components defined in Table 1, one defines the retarded (advanced) component $C^{\mathrm{R}}(t, t')$ ($C^{\mathrm{A}}(t, t')$) by

$$C^{\mathrm{R}}(t, t') = \theta(t - t')\left[C^{>}(t, t') - C^{<}(t, t')\right] \ , \tag{15}$$

$$C^{\mathrm{A}}(t, t') = \theta(t' - t)\left[C^{<}(t, t') - C^{>}(t, t')\right] \ . \tag{16}$$

Here, $\theta(t)$ denotes the Heaviside step function.

For the component with imaginary time arguments only (last entry in Table 1), we employ the convention to represent it by the Matsubara component

$$C^{\mathrm{M}}(\tau_1 - \tau_2) = -iC(-i\tau_1, -i\tau_2) \ . \tag{17}$$

As the Matsubara function is defined by the thermal equilibrium state, it depends on the difference of the imaginary time arguments only. For the single-particle GF (12), the corresponding Matsubara GF $G^{\mathrm{M}}_{ab}(\tau)$ corresponds to a hermitian matrix, $G^{\mathrm{M}}_{ab}(\tau) = [G^{\mathrm{M}}_{ba}(\tau)]^*$.

Extending the concept of the hermitian conjugate to the real-time and mixed components will prove very useful for the numerical implementation as detailed below. Thus, we formally define the *hermitian* conjugate $[C^{\ddagger}](t, t')$ of a general correlator $C(t, t')$ by

$$C^{\gtrless}(t, t') = -\left([C^{\ddagger}]^{\gtrless}(t', t)\right)^{\dagger} \ , \tag{18a}$$

$$C^{\mathrm{R}}(t, t') = \left([C^{\ddagger}]^{\mathrm{A}}(t', t)\right)^{\dagger} \ , \tag{18b}$$

$$C^{\rceil}(t, \tau) = -\xi\left([C^{\ddagger}]^{\lceil}(\beta - \tau, t)\right)^{\dagger} \ , \tag{18c}$$

$$C^{\lceil}(\tau, t) = -\xi\left([C^{\ddagger}]^{\rceil}(t, \beta - \tau)\right)^{\dagger} \ , \tag{18d}$$

$$C^{\mathrm{M}}(\tau) = \left([C^{\ddagger}]^{\mathrm{M}}(\tau)\right)^{\dagger} \ . \tag{18e}$$

Here the superscript $\dagger$ refers to the usual hermitian conjugate of a complex matrix. The definition is reciprocal, $[C^{\ddagger}]^{\ddagger}(t, t') = C(t, t')$. A contour function C is called hermitian symmetric if $C = C^{\ddagger}$ (which does not mean that $C(t, t')$ is a hermitian matrix, see definition above). In particular, the GF defined by Eq. (12) possesses hermitian symmetry. In contrast, more general objects, such as convolutions (see Section 3.3), do not possess a hermitian symmetry, and hence $C(t, t')$ and $[C^{\ddagger}](t, t')$ are independent.

Note that $C^{\mathrm{R}}(t, t') = 0$ if $t' > t$, which expresses the causality of the retarded component. However, for the implementation of numerical algorithms, it can be convenient to drop the Heaviside function in Eq. (15). Therefore, we define a modified retarded component by

$$\tilde{C}^{\mathrm{R}}(t, t') = C^{>}(t, t') - C^{<}(t, t') \ . \tag{19}$$

[1] In the literature (Ref. [14], for instance), often only the combination $G^{\mathrm{K}}(t, t') = G^{<}(t, t') + G^{>}(t, t')$ is referred to as the Keldysh component or Keldysh GF.

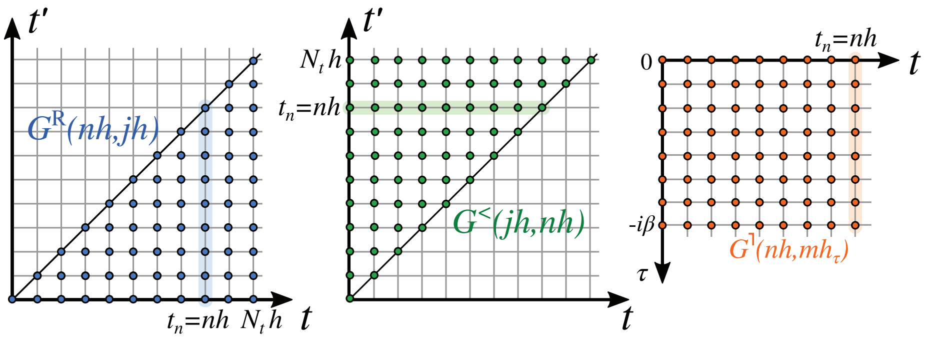


Fig. 4. Storage scheme of the `herm_matrix` class: for $0 \le n \le N_t$ time steps, the class saves $G^{\mathrm{R}}(nh,jh)$ and $G^{<}(jh,nh)$ for $0 \le j \le n$ along with the left-mixing component $G^{\rceil}(nh,mh_\tau)$ for $m = 0,\dots,N_\tau$. The shaded background represents the storage scheme of the time slice $\mathcal{T}[G]_n$, represented by the class `herm_matrix_timestep`.

The modified retarded component of the hermitian conjugate $[C^{\ddagger}](t,t')$ then assumes a similar form as the greater and lesser components:

$$\tilde{C}^{\mathrm{R}}(t,t') = -\left([\tilde{C}^{\ddagger}]^{\mathrm{R}}(t',t)\right)^{\dagger} \ . \tag{20}$$

Assuming the hermitian symmetry $C = C^{\ddagger}$, the number of independent Keldysh components is limited to four. From $C^{>}(t,t') - C^{<}(t,t') = C^{\mathrm{R}}(t,t') - C^{\mathrm{A}}(t,t')$ and Eq. (18b) one finds that the pair $\{C^{>}, C^{<}\}$ or $\{C^{\mathrm{R}}, C^{<}\}$ determines the other real-time components. Furthermore, the hermitian symmetry for the left-mixing component (Eq. (18c)) renders the $C^{\lceil}(\tau,t)$ redundant if $C^{\rceil}(t,\tau)$ is known. Hence, we use in `libcntr` $\{C^{<}, C^{\mathrm{R}}, C^{\rceil}, C^{\mathrm{M}}\}$ as the minimal set of independent Keldysh components.

The KMS boundary conditions (11) establish further relations between the Keldysh components. For the minimal set used here, the corresponding relations are given by

$$C^{\mathrm{M}}(\tau+\beta) = \xi C^{\mathrm{M}}(\tau) \ . \tag{21a}$$

Moreover, if the progagator is continuous on the contour, we have

$$C^{\rceil}(0,\tau) = iC^{\mathrm{M}}(-\tau) \ , \tag{21b}$$

$$C^{<}(t,0) = C^{\rceil}(t,0^{+}) \ . \tag{21c}$$

For the GFs $G(t,t')$, the anti-commutation (commutation) relations for fermions (bosons) determine the retarded component at equal times by

$$G^{\mathrm{R}}_{ab}(t,t) = -i\delta_{a,b} \ . \tag{22}$$

These conditions are used to numerically solve the Dyson equation, see below.

3.2. Numerical representation of NEGFs

In the solvers used for computing the GF numerically, the contour arguments are discretized according to the sketch in Fig. 1. The contour $\mathcal{C}$ is divided into (N_t+1) equidistant points $t_n = nh$, $n = 0,\dots,N_t$ on the real axis (the points correspond to both real time branches $\mathcal{C}_{1,2}$), while $\tau_m = mh_\tau$, $m = 0,\dots,N_\tau$ with $\tau_0 = 0^{+}$, $\tau_{N_\tau} = \beta^{-}$ samples the Matsubara branch. The corresponding discretized contour is denoted by $\mathcal{C}[h, N_t, h_\tau, N_\tau]$.

As discussed in Section 3.1, the contour correlators $C(t,t')$ with hermitian symmetry are represented in `libcntr` by the minimal set of Keldysh components $\{C^{<}, C^{\mathrm{R}}, C^{\rceil}, C^{\mathrm{M}}\}$ on $\mathcal{C}[h, N_t, h_\tau, N_\tau]$. The hermitian symmetry (18a) allows to further reduce the number of points to be stored. We gather this representation of $C(t,t')$ in the class `herm_matrix`, which stores

$$C^{\mathrm{M}}_m = C^{\mathrm{M}}(mh_\tau) \ , \ \ m = 0,\dots,N_\tau \ , \tag{23a}$$

$$C^{<}_{jn} = C^{<}(jh,nh) \ , \ \ n = 0,\dots,N_t, j = 0,\dots,n \ , \tag{23b}$$

$$C^{\mathrm{R}}_{nj} = C^{\mathrm{R}}(nh,jh) \ , \ \ n = 0,\dots,N_t, j = 0,\dots,n \ , \tag{23c}$$

$$C^{\rceil}_{nm} = C^{\rceil}(nh,mh_\tau) \ , \ \ n = 0,\dots,N_t, m = 0,\dots,N_\tau \ . \tag{23d}$$

Hence, the retarded component is only stored on the lower triangle in the two-time plane, while only the upper triangle is required to represent the lesser component (see Fig. 4). For fixed time arguments, the contour function C represents a $d \times d$ square matrix. Note that general two-time functions C (without hermitian symmetry) are also stored in the form of Eq. (23). Hence, to recover the full two-time dependence $C(t,t')$, $C^{\ddagger}(t,t')$ is required.

For some of the algorithms described below, not the full two-time correlator but only a slice with one fixed contour argument is required. To this end, we define a time step $\mathcal{T}[C]_n$ represented by the class `herm_matrix_timestep`, which stores the Keldysh components

$$(\mathcal{T}[C]_n)^{\mathrm{M}}_m = C^{\mathrm{M}}(mh_\tau) \ , \ \ m = 0,\dots,N_\tau \quad (\text{if } n=-1), \tag{24a}$$

$$(\mathcal{T}[C]_n)^{<}_j = C^{<}(jh,nh) \ , j = 0,\dots,n \ , \tag{24b}$$

Table 2
Constructor of the classes `herm_matrix`, `herm_matrix_timestep` and `function`. The arguments in this table correspond to the number of points and the storage scheme discussed above: `nt` is the number of real-time points N_t, `ntau` stands for the number of points N_τ on the Matsubara branch, `tstp` marks the current timestep t_n, whereas `size1` denotes the number of basis functions (orbitals) d. The last argument `sig` for the `herm_matrix` and `herm_matrix_timestep` specifies the fermionic (`sig = -1`) or bosonic (`sig = +1`) statistics.

Class	Constructor
`herm_matrix`	`herm_matrix(int nt, int ntau, int size1, int sig)`
`herm_matrix_timestep`	`herm_matrix_timestep(int tstp, int ntau, int size1, int sig)`
`function`	`function(int nt, int size1)`

$$(\mathcal{T}[C]_n)^{\mathrm{R}}_j = C^{\mathrm{R}}(nh, jh)\ , j = 0, \dots, n\ , \tag{24c}$$
$$(\mathcal{T}[C]_n)^{\rceil}_m = C^{\rceil}(nh, mh_\tau)\ , m = 0, \dots, N_\tau\ . \tag{24d}$$

For later convenience we define $\mathcal{T}[C]_{-1}$, which refers to the Matsubara component only. The stored points in the two-time plane are indicated by the shaded background in Fig. 4. Note that $\mathcal{T}[C]_n$ is a $d \times d$ square matrix for fixed contour argument.

Finally, we introduce contour functions with a single contour argument $f(t)$. While $f(t)$ corresponds to the real times for $t \in \mathcal{C}_1 \cup \mathcal{C}_2$, the function value on the imaginary branch is defined by $f(-i\tau) = f(0^-)$. Single-time contour functions are represented by the `function` class, storing

$$f_n = \begin{cases} f(0^-) & : n = -1 \\ f(nh) & : n = 0, \dots, N_t \end{cases}\ . \tag{25}$$

For fixed n, f_n can be matrix valued ($d \times d$ square matrix).

The initialization of the above contour functions as C++ classes in `libcntr` is summarized in Table 2.

3.3. *Contour multiplication and convolution*

The basic Feynman diagrams can be constructed from products and convolutions of GFs. In this subsection we summarize how such operations can be expressed in terms of Keldysh components.

Product $C_{c_1,c_2}(t,t') = iA_{a_1,a_2}(t,t')B_{b_2,b_1}(t',t)$. — This type of product is often encountered in diagrammatic calculations. Here, $a_1, a_2, b_1, b_2, c_1, c_2$ denote orbital indices, which are kept fixed while the product is computed for all times on a time-slice. For instance, the polarization entering the *GW* approximation is of this form [19]. Its representation in terms of the Keldysh components follows the Langreth rules [9]:

$$C^{\gtrless}_{c_1,c_2}(t,t') = iA^{\gtrless}_{a_1,a_2}(t,t')B^{\lessgtr}_{b_2,b_1}(t',t)\ , \tag{26a}$$
$$C^{\mathrm{R}}_{c_1,c_2}(t,t') = iA^{\mathrm{R}}_{a_1,a_2}(t,t')B^{<}_{b_2,b_1}(t',t) + iA^{<}_{a_1,a_2}(t',t)B^{\mathrm{A}}_{b_2,b_1}(t',t)\ , \tag{26b}$$
$$C^{\rceil}_{c_1,c_2}(t,\tau) = iA^{\rceil}_{a_1,a_2}(t,\tau)B^{\lceil}_{b_2,b_1}(\tau,t)\ , \tag{26c}$$
$$C^{\mathrm{M}}_{c_1,c_2}(\tau) = A^{\mathrm{M}}_{a_1,a_2}(\tau)B^{\mathrm{M}}_{b_2,b_1}(-\tau)\ . \tag{26d}$$

In `libcntr`, we refer to this contour product as `Bubble1`.

Product $C_{c_1,c_2}(t,t') = iA_{a_1,a_2}(t,t')B_{b_1,b_2}(t,t')$. — The direct product of this form also represents a bubble. It is used, for instance, in the calculation of the *GW* self-energy diagram (for additional examples of usage see Section 6). The corresponding representation in terms of the Keldysh components is analogous to the above:

$$C^{\gtrless}_{c_1,c_2}(t,t') = iA^{\gtrless}_{a_1,a_2}(t,t')B^{\gtrless}_{b_1,b_2}(t,t')\ , \tag{27a}$$
$$C^{\mathrm{R}}_{c_1,c_2}(t,t') = iA^{<}_{a_1,a_2}(t,t')B^{\mathrm{R}}_{b_1,b_2}(t,t') + A^{\mathrm{R}}_{a_1,a_2}(t,t')iB^{>}_{b_1,b_2}(t,t')\ , \tag{27b}$$
$$C^{\rceil}_{c_1,c_2}(t,\tau) = iA^{\rceil}_{a_1,a_2}(t,\tau)B^{\rceil}_{b_1,b_2}(t,\tau)\ , \tag{27c}$$
$$C^{\mathrm{M}}_{c_1,c_2}(\tau) = A^{\mathrm{M}}_{a_1,a_2}(\tau)B^{\mathrm{M}}_{b_1,b_2}(\tau)\ . \tag{27d}$$

In `libcntr`, we refer to this contour product as `Bubble2`.

Convolution $C = A * B$. — The convolution of the correlators

$$[A * B](t,t') = \int_{\mathcal{C}} d\bar{t}\, A(t,\bar{t})B(\bar{t},t') \tag{28}$$

is one of the most basic operations on the contour. Using the Langreth rules for the convolution, one obtains

$$\begin{aligned} C^{\gtrless}(t,t') = &\int_0^t d\bar{t}\, A^{\mathrm{R}}(t,\bar{t})B^{\gtrless}(\bar{t},t') + \int_0^{t'} d\bar{t}\, A^{\gtrless}(t,\bar{t})B^{\mathrm{A}}(\bar{t},t') \\ &- i\int_0^{\beta} d\bar{\tau}\, A^{\rceil}(t,\bar{\tau})B^{\lceil}(\bar{\tau},t'), \end{aligned} \tag{29}$$

$$C^{\mathrm{R}}(t,t') = \int_{t'}^{t} d\bar{t}\, A^{\mathrm{R}}(t,\bar{t})B^{\mathrm{R}}(\bar{t},t'), \tag{30}$$

$$C^{\rceil}(t,\tau) = \int_{0}^{t} d\bar{t}\, A^{\mathrm{R}}(t,\bar{t})B^{\rceil}(\bar{t},\tau) + \int_{0}^{\beta} d\tau'\, A^{\rceil}(t,\tau')B^{\mathrm{M}}(\tau'-\tau)\,, \tag{31}$$

$$C^{\mathrm{M}}(\tau) = \int_{0}^{\beta} d\bar{\tau}\, A^{\mathrm{M}}(\tau-\bar{\tau})B^{\mathrm{M}}(\bar{\tau})\,. \tag{32}$$

For the hermitian conjugate one finds $[C^{\ddagger}](t,t') = [B^{\ddagger} * A^{\ddagger}](t,t')$.

3.4. Free Green's functions

Free GFs $G_0(t,t')$ are often required when solving the Dyson equation in integral form. A free Green's function for a time-dependent Hamiltonian $\epsilon(t)$ [Eq. (1)] is obtained from the solution of the equation

$$[i\partial_t - \epsilon(t)]\, G_0(t,t') = \delta_{\mathcal{C}}(t,t') \tag{33}$$

with KMS boundary conditions. This defines a regular differential equation, which can be solved by various standard algorithms. In `libcntr`, free Green's functions can be obtained by the call to a function `green_from_H`. There are several rather obvious interfaces to this function, and we refer to the examples (and the online manual) for more details. The numerical implementation is described in Section 14.

4. Integral equations on $\mathcal{C}$: Overview

4.1. Equations with causal time-dependence

In applications of the Keldysh formalism to problems involving real-time dynamics, one needs to solve various types of differential and integral equations on the contour $\mathcal{C}$. `libcntr` provides algorithms to solve the three most common tasks (convolution of two GFs, solution of a Dyson equation in both integro-differential and integral form) on an equidistant contour mesh $\mathcal{C}[h, N_t, h_\tau, N_\tau]$ with a global error that scales like $\mathcal{O}(h^k, h_\tau^k)$ with k up to $k = 5$.

The precise equations are summarized in Sections 4.2–4.4. A common property of all equations is their *causal* structure, i. e., the solution for the time slice $\mathcal{T}[G]_n$ of the unknown G does not depend on the time slices $m > n$. This causality allows to transform the kth order accurate solution of all integral equations on $\mathcal{C}$ into a time-stepping procedure with the following three steps, which are executed consecutively:

(1) **Matsubara:** Solve the equation for the Matsubara time slice $\mathcal{T}[G]_{-1}$, using the input at time slice $m = -1$.
(2) **Start-up (bootstrapping):** Solve the equation for $\mathcal{T}[G]_j$, $j = 0, \ldots, k$, using $\mathcal{T}[G]_{-1}$ and the input at time slices $j = -1, \ldots, k$. The start-up procedure is essential to keep the $\mathcal{O}(h^k)$ accuracy of the algorithm, as explained in the numerical details (Section 9).
(3) **Time-stepping:** For time slices $n > k$, successively solve the equation for $\mathcal{T}[G]_n$, using $\mathcal{T}[G]_j$ for $j = -1, \ldots, n-1$ and the input at time slices $j = -1, \ldots, n$.

The causality is preserved *exactly* in these algorithms for all time slices $n = -1$ and $n \geq k$. Only for the starting time slices $n = 0, \ldots, k$, the numerical error $\mathcal{O}(h^k, h_\tau^k)$ can also depend on the input at later time slices $j = n+1, \ldots, k$.

We note that for the time-stepping algorithm, a guess for the GF $\mathcal{T}[G]_n$ or the self-energy $\mathcal{T}[\Sigma]_n$ is usually required for starting the self-consistency cycle at time step n. In many cases it is useful to employ a polynomial extrapolation as a predictor $\mathcal{T}[G]_{n-1} \to \mathcal{T}[G]_n$ (see Appendix A.1). The corrector step then involves several iterations of the equations at a given time step until convergence.

In Sections 4.2–4.4 we specify the integral equations implemented in `libcntr`, and present an overview over their input and dependencies on the Matsubara, start-up, and time-stepping parts. The details of the numerical implementation of the kth-order accurate algorithm are explained in Sections 9–13.

4.2. `dyson`*: Dyson equation in integro-differential form*

The Dyson equation for the Green's function $G(t,t')$ can be written as

$$i\partial_t G(t,t') - \epsilon(t)G(t,t') - \int_{\mathcal{C}} d\bar{t}\, \Sigma(t,\bar{t})G(\bar{t},t') = \delta_{\mathcal{C}}(t,t'). \tag{34a}$$

This equation is to be solved for $G(t,t')$ for given input $\epsilon(t)$ and $\Sigma(t,t')$, and the KMS boundary conditions (11). It is assumed that $\Sigma = \Sigma^{\ddagger}$ is hermitian (according to Eq. (18)), and $\epsilon(t) = \epsilon(t)^{\dagger}$, which implies that also the solution G possesses hermitian symmetry. All quantities $\Sigma(t,t')$, $G(t,t')$, and $\epsilon(t)$ can be square matrices of dimension $d \geq 1$. Because of the hermitian symmetry, G can also be determined from the equivalent conjugate equation

$$-i\partial_{t'} G(t,t') - G(t,t')\epsilon(t') - \int_{\mathcal{C}} d\bar{t}\, G(t,\bar{t})\Sigma(\bar{t},t') = \delta_{\mathcal{C}}(t,t'). \tag{34b}$$

In `libcntr`, Eq. (34) is referred to as `dyson` equation. The dependencies between the input and output for the Matsubara, start-up, and time-stepping routines related to the solution of Eqs. (34) are summarized in Table 3.

Table 3
Dependencies between input and output for the Matsubara, start-up, and time-stepping routines associated with the solution of Eqs. (34).

Routine(s)	Input	Output
`dyson_mat`	$\mathcal{T}[\Sigma]_{-1}$, ϵ_{-1}	$\mathcal{T}[G]_{-1}$
`dyson_start`	$\mathcal{T}[\Sigma]_j$ for $j=-1,\dots,k$, ϵ_j for $j=-1,\dots,k$, $\mathcal{T}[G]_{-1}$	$\mathcal{T}[G]_j$, $j=0,\dots,k$
`dyson_timestep(n)` $n>k$	$\mathcal{T}[\Sigma]_j$ for $j=-1,\dots,n$, ϵ_j for $j=-1,\dots,n$, $\mathcal{T}[G]_j$ for $j=-1,\dots,n-1$	$\mathcal{T}[G]_n$

Table 4
Dependencies between input and output for the Matsubara, start-up, and time-stepping routines associated with the solution of Eqs. (37).

Routine(s)	Input	Output
`vie2_mat`	$\mathcal{T}[F]_{-1}$, $\mathcal{T}[F^\ddagger]_{-1}$, $\mathcal{T}[Q]_{-1}$	$\mathcal{T}[G]_{-1}$
`vie2_start`	$\mathcal{T}[F]_j$, $\mathcal{T}[F^\ddagger]_j$ for $j=-1,\dots,k$, $\mathcal{T}[Q]_j$ for $j=-1,\dots,k$, $\mathcal{T}[G]_{-1}$	$\mathcal{T}[G]_j$, $j=0,\dots,k$
`vie2_timestep(n)` $n>k$	$\mathcal{T}[F]_j$, $\mathcal{T}[F^\ddagger]_j$ for $j=-1,\dots,n$, $\mathcal{T}[Q]_n$ $\mathcal{T}[G]_j$ for $j=-1,\dots,n-1$	$\mathcal{T}[G]_n$

A typical application of Eqs. (34a) and (34b) is the solution of the Dyson series in diagrammatic perturbation theory, i. e., a differential formulation of the problem

$$\begin{aligned} G &= G_0 + G_0 * \Sigma * G_0 + G_0 * \Sigma * G_0 * \Sigma * G_0 + \cdots \\ &= G_0 + G_0 * \Sigma * G, \end{aligned} \tag{35}$$

where G_0 satisfies the differential equation

$$i\partial_t G_0(t,t') - \epsilon(t) G_0(t,t') = \delta_{\mathcal{C}}(t,t'). \tag{36}$$

In this case $\epsilon(t)$ is a (possibly time-dependent) single-particle or mean-field Hamiltonian.

4.3. `vie2` : *Dyson equation in integral form*

The second important equation is an integral equation of the form

$$G(t,t') + \int_{\mathcal{C}} d\bar{t}\, F(t,\bar{t}) G(\bar{t},t') = Q(t,t') \quad \Leftrightarrow \quad (1+F)*G = Q, \tag{37a}$$

$$G(t,t') + \int_{\mathcal{C}} d\bar{t}\, G(t,\bar{t}) F^\ddagger(\bar{t},t') = Q(t,t') \quad \Leftrightarrow \quad G*(1+F^\ddagger) = Q. \tag{37b}$$

This linear equation is to be solved for $G(t,t')$ for a given input kernel $F(t,t')$, its hermitian conjugate $F^\ddagger(t,t')$, and a source term $Q(t,t')$, assuming the KMS boundary conditions (21). In the solution of this linear equation, we assume that both Q and G are hermitian. In general, the hermitian symmetry would not hold for an arbitrary input F and Q. However, it does hold when F and Q satisfy the relation

$$F*Q = Q*F^\ddagger, \quad Q = Q^\ddagger, \tag{38}$$

which is the case for the typical applications discussed below. In this case, Eqs. (37a) and (37b) are equivalent.

In `libcntr`, Eq. (37) is referred to as `vie2`. The nomenclature refers to the fact that the equation can be reduced to a Volterra Integral Equation of 2nd kind (see below). The dependencies between the input and output for the Matsubara, start-up, and time-stepping routines associated with the solution of the `vie2` equation are summarized in Table 4.

A typical physical application of Eqs. (37) is given by the summation of a random phase approximation (RPA) series for a susceptibility

$$\begin{aligned} \chi &= \chi_0 + \chi_0 * V * \chi_0 + \chi_0 * V * \chi_0 * V * \chi_0 + \cdots \\ &= \chi_0 + \chi_0 * V * \chi\ . \end{aligned} \tag{39}$$

Here, χ_0 is a bare susceptibility in a given channel (charge, spin, etc, ...), and V is a (possibly retarded) interaction in that channel. Since χ_0 and V are GFs with hermitian symmetry, Eq. (39) can be recast in the form (37a) with

$$F = -\chi_0 * V, \quad F^\ddagger = -V * \chi_0, \quad Q = \chi_0. \tag{40}$$

One can easily verify Eq. (38). Equivalently, one can also recast the Dyson series (35) into the form of a `vie2` equation, with $F = -G_0 * \Sigma$, $F^\ddagger = -\Sigma * G_0$, and $Q = G_0$.

Table 5
Dependencies between input and output for the Matsubara, start-up, and time-stepping routines associated with the solution of Eq. (41).

Routine(s)	Input	Output
`convolution_mat`	$\mathcal{T}[A]_{-1}$, $\mathcal{T}[A^\ddagger]_{-1}$, $\mathcal{T}[B]_{-1}$, $\mathcal{T}[B^\ddagger]_{-1}$, f_{-1}	$\mathcal{T}[C]_{-1}$
`convolution_timestep(n)` for $0 \le n \le k$	for $j = -1, \dots, k$: $\mathcal{T}[A]_j$, $\mathcal{T}[A^\ddagger]_j$, $\mathcal{T}[B]_j$, $\mathcal{T}[B^\ddagger]_j$, f_j	$\mathcal{T}[C]_n$
`convolution_timestep(n)` for $n > k$	for $j = -1, \dots, n$: $\mathcal{T}[A]_j$, $\mathcal{T}[A^\ddagger]_j$, $\mathcal{T}[B]_j$, $\mathcal{T}[B^\ddagger]_j$, f_j	$\mathcal{T}[C]_n$

Table 6
Classes grouped in the name space `cntr`.

Class	Purpose
`function`	Class for representing single-time functions $f(t)$ on the KB contour.
`herm_matrix`	Class for representing two-time functions $C(t,t')$ with hermitian symmetry on the KB contour.
`herm_matrix_timestep`	Class for representing a time slice $\mathcal{T}[G]_n$ of a `herm_matrix` at time step n.
`herm_matrix_timestep_view`	Provides a pointer to a `herm_matrix_timestep` or `herm_matrix` at a particular time step without copying the data.
`distributed_array`	Generic data structure for distributing and communicating a set of data blocks by the Message passing interface (MPI).
`distributed_timestep_array`	Specialization of the `distributed_array` in which data blocks are associated with the `herm_matrix_timestep` objects.

4.4. `convolution`

The most general convolution of two contour Green's functions A and B and a time-dependent function f is given by the integral

$$C(t,t') = \int_{\mathcal{C}} d\bar{t}\, A(t,\bar{t})f(\bar{t})B(\bar{t},t'). \tag{41}$$

In `libcntr` this integral is calculated by the `convolution` routines. The dependencies between the input and output for the Matsubara, start-up, and time-stepping routines related to `convolution` are summarized in Table 5.

In the evaluation of this integral we make in general no assumption on the hermitian properties of A and B. Since the input of the implemented routine is the class of the type `herm_matrix`, both A and B and their hermitian conjugate $A^\ddagger$ and $B^\ddagger$ must be provided, so that $A(t,t')$ and $B(t,t')$ can be restored for arbitrary t, t' on $\mathcal{C}$ (see Section 3). Similarly, the implemented routines calculate the convolution integral only for the components of C corresponding to the domain of the `herm_matrix` type, i. e., the upper/lower triangle representation (23). The full two-time function $C(t,t')$ can be restored by calculating both C and $C^\ddagger$ on the domain of the `herm_matrix` type, where $C^\ddagger$ is obtained from a second call to `convolution`,

$$C^\ddagger(t,t') = \int_{\mathcal{C}} d\bar{t}\, B^\ddagger(t,\bar{t})f^\dagger(\bar{t})A^\ddagger(\bar{t},t'). \tag{42}$$

5. Compiling and using `NESSi`

5.1. Main routines in `libcntr`

The main routines and classes in `libcntr` are grouped under the C++ name space `cntr`. The important classes in `cntr` are summarized in Table 6. The main routines in the `cntr` name space are presented in Table 7 along with a brief description (For information about auxiliary routines see www.nessi.tuxfamily.org). Most of the routines have been introduced above; the remaining functions are explained in the discussion of the example programs in Section 6 and in Appendix A.

Furthermore, the name space `integration` contains the `integrator` class, which contains all the coefficients for numerical differentiation, interpolation and quadrature as explained in Section 8.

Table 7
Main functions available in the name space `cntr`.

Class	Purpose	Reference section
`Bubble1(tstp,C,c1,c2,` `A,A`‡`,a1,a2,B,B`‡`,b1,b2)`	Computes the particle–hole bubble diagram at a time step `tstp` for given two-time objects A, B with indices $a1$, $a2$ and $b1$, $b2$, respectively: $C_{c_1,c_2}(t,t') = iA_{a_1,a_2}(t,t')B_{b_2,b_1}(t',t)$. **output:** object C with indices $c1$, $c2$	3.3
`Bubble2(tstp,C,c1,c2,` `A,A`‡`,a1,a2,B,B`‡`,b1,b2)`	Computes the particle–particle bubble diagram at a time step `tstp` for given two-time objects A, B with indices $a1$, $a2$ and $b1$, $b2$, respectively: $C_{c_1,c_2}(t,t') = iA_{a_1,a_2}(t,t')B_{b_1,b_2}(t,t')$. **output:** object C with indices $c1$, $c2$	3.3
`convolution(C,A,A`‡`,B,B`‡`,` `beta,dt,SolveOrder)`	Computes the convolution $C = A * B$ for given two-time objects A, B in the full two-time plane. **output:** object C	11
`convolution_timestep(n,C,A,` `A`‡`,B,B`‡`,beta,dt,SolveOrder)`	Computes the time step $\mathcal{T}[C]_n$ of the convolution $C = A * B$ with given two-time objects A, B. **output:** object C	11
`convolution_density_matrix(` `tstp,M,A,A`‡`,B,B`‡`,beta,dt, SolveOrder)`	Computes the convolution $-i[A * B]^<(t,t)$ with given two-time objects A, B at a time step `tstp`. **output:** object M	11
`dyson(G,mu,H,Sigma,beta,` `dt,SolveOrder)`	Solves the Dyson equation with given `Sigma`, `H`, and `mu` in the full two-time plane. **output:** object G	12
`dyson_mat(G,Sigma,mu,H,` `beta,SolveOrder,method)`	Solves the Matsubara Dyson equation with given `Sigma`, `mu`, `H`. The argument `method` is optional. **output:** object $\mathcal{T}[G]_{-1}$	12.2
`dyson_start(G,mu,H,Sigma,` `beta,dt,SolveOrder)`	Solves the starting problem of the Dyson equation for $\mathcal{T}[G]_n$, $n = 0, \dots, k$ with given `Sigma`, `mu`, `H`. **output:** object G	12.3
`dyson_timestep(n,G,mu,H,` `Sigma,beta,dt,SolveOrder)`	Solves the Dyson equation for the time step $\mathcal{T}[G]_n$ with given `Sigma`, `mu`, `H`. **output:** object G	12.4
`green_from_H(G,mu,eps, beta,dt)`	Computes the free GF $G_0(t,t')$ for a given Hamiltonian $\epsilon(t)$, and `mu`. **output:** object G.	14
`response_convolution(tstp,` `cc,A,a1,a2,f,b1,b2,` `SolveOrder,beta,dt)`	Computes the convolution of object A with indices $a1$, $a2$ and a function f with indices $b1$, $b2$ at a time step `tstp`: $\int_{\mathcal{C}} d\bar{t} A_{a_1,a_2}(t,\bar{t}) f_{b_1,b_2}(\bar{t})$. **output:** object `cc`.	11
`extrapolate_timestep(n,` `A,ExtrapolationOrder)`	Computes $\mathcal{T}[A]_{n+1}$ by polynomial extrapolation with given `ExtrapolationOrder`. **output:** extrapolated object A	A.1
`correlation_energy(tstp,G,` `Sigma,beta,h,SolveOrder)`	Evaluates the Galitskii–Migdal formula $E_{\mathrm{corr}} = \frac{1}{2}\mathrm{ImTr}[\Sigma * G]^<(t,t)$ at a given time step `tstp`.	11
`distance_norm2(tstp,A,B)`	Returns the distance between given objects A, B with respect to the absolute-value-norm on the KB contour at a time step `tstp`.	A.2
`vie2(G,F,F`‡`,Q,beta,dt, SolveOrder)`	Solves the VIE for given $F(t,t')$ and $Q(t,t')$ in the full two-time plane. **output:** object G	13
`vie2_mat(G, F, F`‡`,Q,beta,` `method,SolveOrder)`	Solves the Matsubara VIE for $\mathcal{T}[G]_{-1}$ with given F, Q. The argument `method` is optional. **output:** object G	13.2
`vie2_start(G,F,F`‡`,Q,beta,` `dt,SolveOrder)`	Solves the starting problem of the VIE for $\mathcal{T}[G]_n$, $n = 0, \dots, k$ with given F, Q. **output:** object G	13.3
`vie2_timestep(tstp,G,F,F`‡`,` `Q,beta,dt,SolveOrder)`	Solves the VIE for the time step $n =$`tstp` $\mathcal{T}[G]_n$ with given F, Q. **output:** object G	13.4

5.2. Compilation of `libcntr`

For compiling and installing the `libcntr` library, we use the `cmake` building environment[2] to generate system specific `make` files. `cmake` can be called directly from the terminal; however, it is more convenient to create a configure script with all compile options. We suggest the following structure:

```
CC=[C compiler] CXX=[C++ compiler] \
cmake \
    -DCMAKE_INSTALL_PREFIX=[install directory] \
    -DCMAKE_BUILD_TYPE=[Debug|Release] \
    -Domp=[ON|OFF] \
    -Dhdf5=[ON|OFF] \
    -Dmpi=[ON|OFF] \
    -DBUILD_DOC=[ON|OFF] \
    -DCMAKE_INCLUDE_PATH=[include directory] \
    -DCMAKE_LIBRARY_PATH=[library directory] \
    -DCMAKE_CXX_FLAGS="[compiling flags]" \
    ..
```

In the first line, the C and C++ compiler are set. The install directory (for instance `/home/opt`) is defined by the `cmake` variable `CMAKE_INSTALL_PREFIX`. Debugging tools are switched on by setting `CMAKE_BUILD_TYPE` to `Debug`; otherwise, all assertions and sanity checks are turned off. The code is significantly faster in `Release` mode, which is recommended for production runs. The `Debug`

2 Version 2.8 or higher is required.

mode, on the other hand, turns on assertions (implemented as C++ standard assertions) of the consistency of the input for all major routines.

The following three lines trigger optional (but recommended) functionalities: Setting `omp` to `ON` turns on the compilation of routines parallelized with `openMP`, while setting `mpi` to `ON` is required for compiling distributed-memory routines based on `MPI`. In this case, `MPI` compilers have to be specified in the first line. Finally, `hdf5=ON` activates the usage of the `hdf5` library.

The path to the libraries that `libcntr` depends upon (`eigen3` and, optionally, `hdf5`) are provided by specifying the include directory `CMAKE_INCLUDE_PATH` and the library path `CMAKE_LIBRARY_PATH`. Finally, the compilation flags are specified by `CMAKE_CXX_FLAGS`. To compile `libcntr`, the flags should include

```
-std=c++11
```

As the next step, create a build directory (for instance `cbuild`). Navigate to this directory and run the configure script:

```
sh ../configure.sh
```

After successful configuration (which generates the `make` files), compile the library by typing

```
make
```

and install it to the install directory by

```
make install
```

After the compilation, the user can check the build by running

```
make test
```

which runs a set of tests based on the `catch` testing environment [20], checking every functionality of `libcntr`. After completing all tests, the message

```
All tests passed
```

indicates that the compiled version of `libcntr` is fully functional.

The C++ code is documented using the automatic documentation tool `doxygen`. For generating the documentation, set the `CMake` variable `BUILD_DOC` to `ON` in the configure script. Running `make` will then also generate an html description of many functions and classes in the `doc/` directory. A detailed and user-friendly manual is provided on the webpage www.nessi.tuxfamily.org.

5.3. *Using* `libcntr` *in custom programs*

In order to include the `libcntr` routines in custom C++ programs, the user needs to:

1. Include the declaration header by

   ```
   #include "cntr/cntr.hpp"
   ```

   This makes available all main routines and classes in the C++ name space `cntr`, as summarized in Table 6. We also offer tools for reading variables from an input file. The respective routines can be used in a program by including

   ```
   #include "cntr/utils/read_inputfile.hpp"
   ```

2. Compile the programs linking the `libcntr` library with the flag `-lcntr`.

The example programs presented below in Section 6 demonstrate how to integrate `libcntr` in custom programs.

5.4. *HDF5 in/output*

In addition to simple input and output from and to text files (which is described in the manual on www.nessi.tuxfamily.org), `libcntr` allows to use the Hierarchical Data Format version 5 (HDF5) to store basic data types for contour functions to disk. HDF5 is an open source library and file format for numerical data which is widely used in the field of scientific computing. The format has two building blocks: (i) *datasets*, that are general multi-dimensional arrays of a single type, and (ii) *groups*, that are containers which can hold datasets and other groups. By nesting groups, it is possible to store arbitrarily complicated structured data, and to create a file-system-like hierarchy where groups can be indexed using standard POSIX format, e.g. `/path/to/data`.

The `libcntr` library comes with helper functions to store the basic contour response function data types in HDF5 with a predefined structure of groups and datasets, defined in the header `cntr/hdf5/hdf5_interface.hpp`. In particular, a `herm_matrix` response function is stored as a group with a dataset for each contour component `mat` ($g^{\mathrm{M}}(\tau)$), `ret` ($g^{\mathrm{R}}(t,t')$), `les` ($g^{<}(t,t')$), and `tv` ($g^{\rceil}(t,\tau)$), respectively, see Section 3.2. The retarded and lesser components are stored in upper and lower triangular contiguous time order respectively. In the `libcntr` HDF5 format each component is stored as a rank 3 array where the first index is time, imaginary time, or triangular contiguous two-time, and the remaining two indices are orbital indices.

To store a contour GF of type `cntr::herm_matrix`, one writes its components into a group of a HDF5 file using the member function `write_to_hdf5`. In C++ this takes the form,

```
#include <cntr/cntr.hpp>
..
// Create a contour Green's function
int nt = 200, ntau = 400, norb = 1;
GREEN A(nt, ntau, norb, FERMION);

// Open HDF5 file and write components of the Green's function A into a group g.
std::string filename = "data.h5";
A.write_to_hdf5(filename.c_str(), "g");
```

For another example of writing contour objects to file see the Holstein example program in Section 6.3. To understand the structure of the resulting HDF5 file one can inspect it with the `h5ls` command line program that can be used to list all groups and datasets in a HDF5 file:

```
$ h5ls -r data.h5
...
/g                       Group
/g/element_size          Dataset {1}
/g/les                   Dataset {20301, 1, 1}
/g/mat                   Dataset {401, 1, 1}
/g/nt                    Dataset {1}
/g/ntau                  Dataset {1}
/g/ret                   Dataset {20301, 1, 1}
/g/sig                   Dataset {1}
/g/size1                 Dataset {1}
/g/size2                 Dataset {1}
/g/tv                    Dataset {80601, 1, 1}
```

One can see that apart from the contour components the Green's function group g contains additional information about the dimensions and the Fermi/Bose statistics (`sig`$= \mp 1$), for details see the API documentation of `herm_matrix` and Section 3.2. To understand the dimensions of the contour components we can look at the number of imaginary time steps `ntau` and number of real time steps `nt` using the `h5dump` command line utility,

```
$ h5dump -d /g/ntau data.h5
HDF5 "data.h5" {
DATASET "/g/ntau" {
   DATATYPE  H5T_STD_I32LE
   DATASPACE  SIMPLE { ( 1 ) / ( 1 ) }
   DATA {
   (0): 400
   }
}
}
$ h5dump -d /g/nt data.h5
HDF5 "data.h5" {
DATASET "/g/nt" {
   DATATYPE  H5T_STD_I32LE
   DATASPACE  SIMPLE { ( 1 ) / ( 1 ) }
   DATA {
   (0): 200
   }
}
}
```

which shows that the dimensions are $n_\tau = 400$ and $n_t = 200$. The size of the `/g/mat` component reveals that this corresponds to $n_\tau + 1 = 401$ imaginary time points. The mixed `/g/tv` component has a slow time index and a fast imaginary time index and is of size $(n_t + 1)(n_\tau + 1) = 80601$ while the two time triangular storage of the `/g/ret` and `/g/les` components contains $(n_t + 1)(n_t + 2)/2 = 20301$ elements.

To simplify postprocessing of contour GFs, `NESSi` also provides the python module `ReadCNTRhdf5.py` for reading the HDF5 format (using the python modules `numpy` and `h5py`) producing python objects with the contour components as members. The python module unrolls the triangular storage of the `ret` and `les` components making it simple to plot time slices. For example, to plot the imaginary part of the retarded Green's function $\mathrm{Im}[G^{\mathrm{R}}(t, t' = 0)]$ as a function of t we may use the commands

```
import h5py
from ReadCNTRhdf5 import read_group

with h5py.File('data.h5', 'r') as fd:
    g = read_group(fd).g

import matplotlib.pyplot as plt
plt.figure(figsize=(6., 1.5))

plt.plot(g.ret[:, 0, 0, 0].imag)

plt.xlabel(r'time step')
plt.ylabel(r'Im$[G^{R}(t, 0)]$')

plt.tight_layout(); plt.savefig('figure_g_ret.pdf')
```

which produces the plot shown in Fig. 5. More advanced usage of the HDF5 interface is exemplified in the example programs, and in the online manual.

6. Example programs

In this section, we present a number of examples of how the described routines can be used to solve typical nonequilibrium problems. It is assumed that the `libcntr` library has been compiled and installed. Furthermore, we assume that the collection of demonstration programs `nessi_demo` has been installed to some directory `nessi_demo/` and compiled in, for instance, `nessi_demo/build/`.

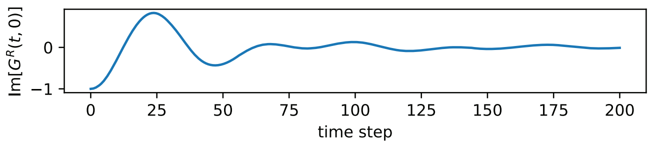


Fig. 5. Result of the python example for reading and plotting $\text{Im}[G^R(t,t'=0)]$ from a HDF5 file. Using the C++ example for generating the HDF5 file would give only zero values. Here, the result from a nontrivial time-dependent calculation is shown instead.

Detailed building instructions can be found in Appendix B. Further examples can be found in the online manual, where one can also find a more detailed description of all member functions and simple helper routines (such as, e.g., adding up Green's functions, scalar multiplication, etc.).

6.1. Test of accuracy and scaling analysis

Overview. — The first example both serves as a minimal application of the vie2 equation (without much physical significance), and at the same time it demonstrates the convergence of the methods described in Section 9 with the time discretization. We consider a 2×2 matrix-valued time-independent Hamiltonian

$$\epsilon = \begin{pmatrix} \epsilon_1 & i\lambda \\ -i\lambda & \epsilon_2 \end{pmatrix} . \tag{43}$$

The corresponding numerically exact GF $G(t,t')$ (assuming fermions) is computed using the routine `green_from_H` mentioned in Section 3.4. Alternatively, one can compute the (1,1) component of the GF by *downfolding*: To this end, we solve

$$(i\partial_t - \epsilon_1)\, g_1(t,t') = \delta_{\mathcal{C}}(t,t') + \int_{\mathcal{C}} d\bar{t}\ \Sigma(t,\bar{t}) g_1(\bar{t},t') \tag{44}$$

with the embedding self-energy $\Sigma(t,t') = |\lambda|^2 g_2(t,t')$. Here, $g_2(t,t')$ is the free GF with respect to ϵ_2,

$$(i\partial_t - \epsilon_2)\, g_2(t,t') = \delta_{\mathcal{C}}(t,t') \ . \tag{45}$$

The solution of the Dyson equation (44) then must be identical to the (1, 1) matrix element of G: $G_{1,1}(t,t') = g_1(t,t')$. The test programs `test_equilibrium.x` and `test_nonequilibrium.x` solve this problem in equilibrium and nonequilibrium, respectively, and compare the error. In the equilibrium case, we define

$$\text{err.} = \frac{1}{\beta}\int_0^\beta d\tau\ |G_{1,1}(\tau) - g_1(\tau)| \ , \tag{46}$$

whereas

$$\begin{aligned} \text{err.} = &\frac{1}{T^2}\int_0^T dt \int_0^t dt' |G^<_{1,1}(t',t) - g^<_1(t',t)| \\ &+ \frac{1}{T^2}\int_0^T dt \int_0^t dt' |G^R_{1,1}(t,t') - g^R_1(t,t')| \\ &+ \frac{1}{T\beta}\int_0^T dt \int_0^\beta d\tau |G^\rceil_{1,1}(t,\tau) - g^\rceil_1(t,\tau)| \end{aligned} \tag{47}$$

for the nonequilibrium case.

Implementation: Equilibrium. — The implementation of the equilibrium solution of the example is found in `programs/test_equilibrium.cpp`. In the following we summarize and explain the main parts:

In `libcntr`, we define the following short-hand types

```
#define GREEN cntr::herm_matrix<double>
#define GREEN_TSTP cntr::herm_matrix<double>
#define CFUNC cntr::function<double>
```

for double-precision objects. They are available to any program including `cntr.hpp`. In the main part of the C++ program, the parameters of the Hamiltonian are defined as constants. In particular, we fix $\epsilon_1 = -1$, $\epsilon_2 = 1$, $\lambda = 0.5$. The chemical potential is set to $\mu = 0$ and the inverse temperature fixed to $\beta = 20$. The input variables read from file are `Ntau` (N_τ) and `SolveOrder` ($k = 1, \ldots, 5$). After reading these variables from file via

```
find_param(argv[1],"__Ntau=",Ntau);
find_param(argv[1],"__SolveOrder=",SolveOrder);
```

we can define all quantities. First we define the Hamiltonian (43) as an `eigen3` complex matrix:

```
cdmatrix eps_2x2(2,2);
eps_2x2(0,0) = eps1;
eps_2x2(1,1) = eps2;
eps_2x2(0,1) = I*lam;
eps_2x2(1,0) = -I*lam;
```

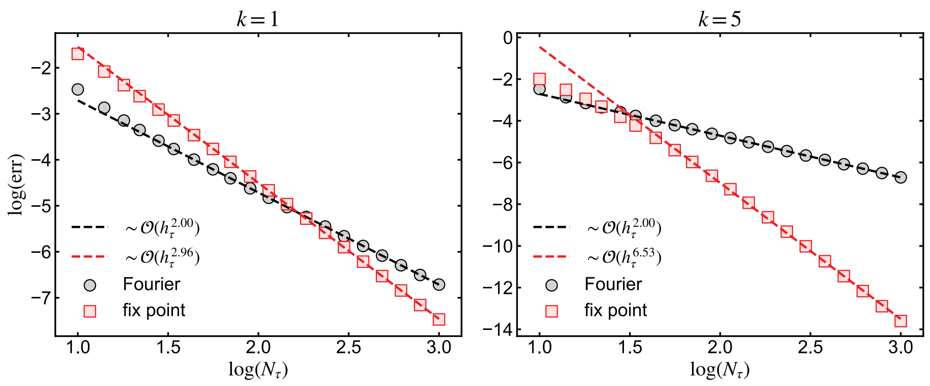


Fig. 6. Average error according to Eq. (46) for $\epsilon_1 = -1$, $\epsilon_2 = 1$, $\lambda = 0.5$, $\mu = 0$, $\beta = 20$ for $k = 1$ and $k = 5$.

The 1×1 Hamiltonian representing ϵ_1 is constructed as

```
CFUNC eps_11_func(-1,1);
eps_11_func.set_constant(eps1*MatrixXcd::Identity(1,1));
```

Here, eps_11_func is a contour function entering the solvers below. Note the first argument in the constructor of CFUNC: the number of real-time points N_t is set to -1. In this case, only the Matsubara part is addressed. Its value is fixed to the constant 1×1 matrix by the last line. With the Hamiltonians defined, we can initialize and construct the free 2×2 exact GF by

```
GREEN G2x2(-1,Ntau,2,FERMION);
cntr::green_from_H(G2x2,mu,eps_2x2,beta,h);
```

Including the libcntr header provides a number of constants for convenience; here, we have used FERMION$\equiv$ -1 (bosons would be described by BOSON$\equiv$ +1). The time step h is a dummy argument here, as the real-time components are not addressed. From the exact GF, we extract the submatrix $G_{1,1}$ by

```
GREEN G_exact(-1,Ntau,1,FERMION);
G_exact.set_matrixelement(-1,0,0,G2x2);
```

Finally, we define the embedding self-energy by

```
GREEN Sigma(-1,Ntau,1,FERMION);
cdmatrix eps_22=eps2*MatrixXcd::Identity(1,1);
cntr::green_from_H(Sigma, mu, eps_22, beta, h);
Sigma.smul(-1,lam*lam);
```

The last line performs the multiplication of $\mathcal{T}[\Sigma]_{-1}$ with the scalar λ^2. After initializing the approximate GF G_approx, we can solve the Matsubara Dyson equation and compute the average error:

```
cntr::dyson_mat(G_approx, Sigma, mu, eps_11_func, beta, SolveOrder, CNTR_MAT_FOURIER);
err_fourier = cntr::distance_norm2(-1,G_exact,G_approx) / Ntau;

cntr::dyson_mat(G_approx, Sigma, mu, eps_11_func, beta,  SolveOrder, CNTR_MAT_FIXPOINT);
err_fixpoint = cntr::distance_norm2(-1,G_exact,G_approx) / Ntau;
```

The error is then written to file. The function distance_norm2 measures the absolute-value-norm of two contour functions, as explained in Appendix A.2.

Running and output: Equilibrium. — For convenience, we provide a driver python3 script for creating the input file, running the program and plotting the results. For running the equilibrium test, go to nessi_demo/ and run

```
python3 utils/test_equilibrium.py k
```

where k=1,. . . ,5 is the integration order. The test solves the Matsubara Dyson equation for $N_\tau = 10^x$ for 20 values of $x \in [1, 3]$. The results are plotted using matplotlib. Fig. 6 shows the corresponding plots for $k = 1$ and $k = 5$. As Fig. 6 demonstrates, the Fourier method described in Section 12.2 scales as $\mathcal{O}(h_\tau^2)$, while solving the Dyson equation in integral form (fixed point integration) results approximately in a $\mathcal{O}(h_\tau^{k+2})$ scaling of the average error for small enough h_τ.

Implementation: Nonequilibrium. — Testing the accuracy of the dyson and vie2 solvers can be done analogous to the equilibrium case above. The source code, which is described below, can be found in programs/test_nonequilibrium.cpp.

We adopt the same parameters as for the equilibrium case. To obtain the NEGFs, the Dyson equation (44) is propagated in time. Equivalently, one can also solve the Dyson equation in integral form

$$g_1(t,t') + [F * g_1](t,t') = g_1^{(0)}(t,t') \,, \tag{48a}$$

$$g_1(t,t') + [g_1 * F^\ddagger](t,t') = g_1^{(0)}(t,t') \,, \tag{48b}$$

where $F = -\Sigma * g_1^{(0)}$ and $F^\ddagger = -g_1^{(0)} * \Sigma$, as explained in Section 4.3. The free GF $g_1^{(0)}(t,t')$ is known analytically and computed by calling the routine green_from_H.

The structure of the test program is analogous to the equilibrium case. First, the input variables N_t, N_τ, $T_{\max}$ and k are read from the input file:

```
find_param(flin,"__Nt=",Nt);
find_param(flin,"__Ntau=",Ntau);
find_param(flin,"__Tmax=",Tmax);
find_param(flin,"__SolveOrder=",SolveOrder);
```

The time step is fixed by $h = T_{\max}/N_t$. After initializing the Hamiltonian and the GFs, the embedding self-energy is constructed via

```
cntr::green_from_H(Sigma, mu, eps_22, beta, h);
for(tstp=-1; tstp<=Nt; tstp++) {
  Sigma.smul(tstp,lam*lam);
}
```

The generic procedure to solve a Dyson equation in the time domain in `libcntr` is

1. Solve the equilibrium problem by solving the corresponding Matsubara Dyson equation,
2. Compute the NEGFs for time steps $n = 0, \dots, k$ by using the start-up algorithm (bootstrapping), and
3. Perform the time stepping for $n = k+1, \dots, N_t$.

For Eq. (44), this task is accomplished by

```
GREEN G_approx(Nt, Ntau, 1, FERMION);

// equilibrium
cntr::dyson_mat(G_approx, mu, eps_11_func, Sigma, beta, SolveOrder);

// start
cntr::dyson_start(G_approx, mu, eps_11_func, Sigma, beta, h, SolveOrder);

// time stepping
for (tstp=SolverOrder+1; tstp<=Nt; tstp++) {
   cntr::dyson_timestep(tstp, G_approx, mu, eps_11_func, Sigma, beta, h, SolveOrder);
}
```

The deviation of the nonequilibrium Keldysh components from the exact solution is then calculated by

```
err_dyson=0.0;
for(tstp=0; tstp<=Nt; tstp++){
  err_dyson += cntr::distance_norm2_les(tstp, G_exact, G_approx) / (Nt*Nt);
  err_dyson += cntr::distance_norm2_ret(tstp, G_exact, G_approx) / (Nt*Nt);
  err_dyson += cntr::distance_norm2_tv(tstp, G_exact, G_approx) / (Nt*Ntau);
}
```

The solution of the corresponding integral formulation (61) is performed by the following lines of source code:

```
// noninteracting 1x1 Greens function (Sigma=0)
GREEN G0(Nt,Ntau,1,FERMION);
cdmatrix eps_11=eps1*MatrixXcd::Identity(1,1);
cntr::green_from_H(G0, mu, eps_11, beta, h);

GREEN G_approx(Nt,Ntau,1,FERMION);
GREEN F(Nt,Ntau,1,FERMION);
GREEN Fcc(Nt,Ntau,1,FERMION);

// equilibrium
GenKernel(-1, G0, Sigma, F, Fcc, beta, h, SolverOrder);
cntr::vie2_mat(G_approx, F, Fcc, G0, beta, SolverOrder);

// start
for(tstp=0; tstp <= SolveOrder; tstp++){
  GenKernel(tstp, G0, Sigma, F, Fcc, beta, h, SolverOrder);
}
cntr::vie2_start(G_approx, F, Fcc, G0, beta, h, SolveOrder);

// time stepping
for (tstp=SolveOrder+1; tstp<=Nt; tstp++) {
  GenKernel(tstp, G0, Sigma, F, Fcc, beta, h, SolverOrder);
  cntr::vie2_timestep(tstp, G_approx, F, Fcc, G0, beta, h, SolveOrder);
}
```

For convenience, we have defined the routine `GenKernel`, which calculates the convolution kernels F and $F^\ddagger$:

```
void GenKernel(int tstp, GREEN &G0, GREEN &Sigma, GREEN &F, GREEN &Fcc, const double beta, const double h,
     const int SolveOrder){
  cntr::convolution_timestep(tstp, F, G0, Sigma, beta, h, SolveOrder);
  cntr::convolution_timestep(tstp, Fcc, Sigma, G0, beta, h, SolveOrder);
  F.smul(tstp,-1);
  Fcc.smul(tstp,-1);
}
```

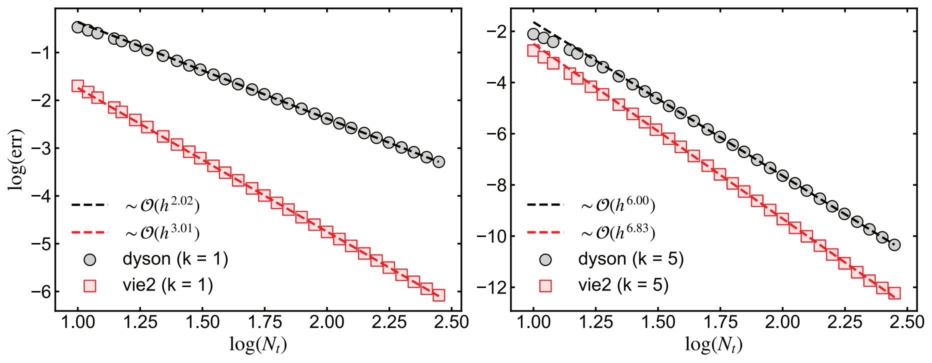


Fig. 7. Average error according to Eq. (47) for $\epsilon_1 = -1$, $\epsilon_2 = 1$, $\lambda = 0.5$, $\mu = 0$, $\beta = 20$ for $k = 1$ and $k = 5$. We have fixed $N_\tau = 800$ and $T_{\max} = 5$.

Running and output: Nonequilibrium. — The python3 driver script `test_nonequilibrium.py` provides an easy-to-use interface for running the accuracy test. In the `nessi_demo/` directory, run

```
python3 utils/test_nonequilbrium.py k
```

where `k` is the solution order. The average error of the numerical solution of Eq. (61) is computed analogously to the Dyson equation in integro-differential form. The output of `test_nonequilibrium.py` is shown in Fig. 7. As this figure confirms, the average error of solving the Dyson equation in the integro-differential form scales as $\mathcal{O}(h^{k+1})$, while the corresponding integral form yields a $\mathcal{O}(h^{k+2})$ scaling.

6.2. Hubbard chain

Overview. — The Hubbard model is one of the most basic models describing correlation effects. It allows to demonstrate the performance, strengths and also shortcomings of the NEGF treatment [21–23]. Here, we consider a one-dimensional (1D) configuration with the Hamiltonian

$$\hat{H}_0 = -J \sum_{\langle i,j\rangle,\sigma} \hat{c}^\dagger_{i\sigma}\hat{c}_{j\sigma} + U\sum_i (\hat{n}_{i\uparrow} - \bar{n})(\hat{n}_{i\downarrow} - \bar{n}) \ , \tag{49}$$

where $\langle i, j\rangle$ constrains the lattice sites i, j to nearest neighbors, while $\bar{\sigma} = \uparrow, \downarrow$ for $\sigma = \downarrow, \uparrow$. We consider M lattice sites with open boundary conditions. Furthermore, we restrict ourselves to the paramagnetic case with an equal number of spin-up ($N_\uparrow$) and spin-down ($N_\downarrow$) particles. The number of particles determines the filling factor $\bar{n} = N_\uparrow/M$. Note that the Hamiltonian (49) contains a chemical potential shift, such that $\mu = 0$ corresponds to filling $\bar{n}$. In analogy to Ref. [22], the system is excited with an instantaneous quench of the on-site potential of the first lattice site to w_0:

$$\hat{H}(t) = \hat{H}_0 + \theta(t) w_0 \sum_\sigma \hat{c}^\dagger_{1\sigma}\hat{c}_{1\sigma} \ . \tag{50}$$

In this example, we treat the dynamics with respect to the Hamiltonian (50) within the second-Born (2B), *GW*, and *T*-matrix (particle–particle ladder) approximations. A detailed description of these approximations can be found, for instance, in Ref. [22]. The numerical representation of the respective self-energy expressions is implemented in the C++ module `hubbard_chain_selfen_impl.cpp`. Below we explain the key routines.

Self-energy approximation: second-Born. — The 2B approximation corresponds to the second-order expansion in terms of the Hubbard repulsion $U(t)$, which we treat here as time dependent for generality. Defining the GF with respect to the lattice basis, $G_{ij,\sigma}(t,t') = -i\langle T_\mathcal{C}\hat{c}_{i\sigma}(t)\hat{c}^\dagger_{j\sigma}(t')\rangle$, the 2B is defined by

$$\Sigma_{ij,\sigma}(t,t') = U(t)U(t')G_{ij,\sigma}(t,t')G_{ij,\bar{\sigma}}(t,t')G_{ji,\bar{\sigma}}(t',t) \ . \tag{51}$$

The 2B self-energy (51) is implemented in two steps. (i) The (spin-dependent) polarization $P_{ij,\sigma}(t,t') = -iG_{ij,\sigma}(t,t')G_{ji,\sigma}(t',t)$ is computed using the routine `Bubble1` and subsequently multiplied by -1. (ii) The self-energy is then given by $\Sigma_{ij,\sigma}(t,t') = iU(t)U(t')G_{ij,\sigma}(t,t')P_{ij,\bar{\sigma}}(t,t')$, which corresponds to a bubble diagram computed by the routine `Bubble2`. Inspecting the Keldysh components of the GFs, one notices that the polarization $P_{ij,\sigma}(t,t')$ is needed on one time slice only. As $G_{ij,\uparrow}(t,t') = G_{ij,\downarrow}(t,t') \equiv G_{ij}(t,t')$ (an analogous statement holds for other contour functions), the spin index can be dropped. The 2B self-energy is computed by the routine `Sigma_2B` as follows:

```
void Sigma_2B(int tstp, GREEN &G, CFUNC &U, GREEN &Sigma){
   int nsites=G.size1();
   int ntau=G.ntau();
   GREEN_TSTP Pol(tstp,ntau,nsites,BOSON);

   Polarization(tstp, G, Pol);

   Pol.right_multiply(tstp, U);
   Pol.left_multiply(tstp, U);
```

```

    for(int i=0; i<nsites; i++){
      for(int j=0; j<nsites; j++){
        cntr::Bubble2(tstp,Sigma,i,j,G,i,j,Pol,i,j);
      }
    }

}
```

First, the polarization `Pol`, which represents $P_{ij}(t,t')$, is defined for the given time step. After computing $P_{ij}(t,t')$ by the function

```
  void Polarization(int tstp, GREEN &G, GREEN_TSTP &Pol){
    int nsites=G.size1();

    for(int i=0; i<nsites; i++){
      for(int j=0; j<nsites; j++){
        cntr::Bubble1(tstp,Pol,i,j,G,i,j,G,i,j);
      }
    }
    Pol.smul(-1.0);
  }
```

the lines

```
Pol.right_multiply(tstp, U);
Pol.left_multiply(tstp, U);
```

perform the operation $P_{ij}(t,t') \to P_{ij}(t,t')U(t')$ and $P_{ij}(t,t') \to U(t)P_{ij}(t,t')$, respectively. Finally, `Bubble2` computes $\Sigma_{ij}(t,t')$.

Self-energy approximation: GW. — As the next approximation to the self-energy, we consider the *GW* approximation. We remark that we formally treat the Hubbard interaction as spin-independent (as in Ref. [22]), while the spin-summation in the polarization P (which is forbidden by the Pauli principle) is excluded by the corresponding prefactor. In terms of the Feynman diagrams, this corresponds to the self-energy including the diagrams with any number of polarization insertins (bubbles). The analogous approximation for the explicitly spin-dependent interaction (spin-*GW*) is also discussed in Ref. [22], whose self-energy includes the diagrams with odd number of the bubbles only. We focus on the former type of GW in the following.

Within the same setup as above, the *GW* approximation is defined by

$$\Sigma_{ij}(t,t') = iG_{ij}(t,t')\delta W_{ij}(t,t') \ , \tag{52}$$

where $\delta W_{ij}(t,t')$ denotes the dynamical part of the screened interaction $W_{ij}(t,t') = U(t)\delta_{ij}\delta_{\mathcal{C}}(t,t') + \delta W_{ij}(t,t')$. We compute $\delta W_{ij}(t,t')$ from the charge susceptibility $\chi_{ij}(t,t')$ by $\delta W_{ij}(t,t') = U(t)\chi_{ij}(t,t')U(t')$. This susceptibility obeys the Dyson equation

$$\chi = P + P * U * \chi \ , \tag{53}$$

where P stands for the irreducible polarization $P_{ij}(t,t') = -iG_{ij}(t,t')G_{ji}(t',t)$. The strategy to compute the *GW* self-energy with `libcntr` thus consists of three steps:

1. Computing the polarization $P_{ij}(t,t')$ by `Bubble1`.
2. Solving the Dyson equation (53) as VIE. By defining the kernel $K_{ij}(t,t') = -P_{ij}(t,t')U(t')$ and its hermitian conjugate, Eq. (53) amounts to $[1+K]*\chi = P$, which is solved for χ using `vie2`.
3. Computing the self-energy (52) by `Bubble2`.

The implementation of step 1 has been discussed above. For step 2, we distinguish between the equilibrium (timestep `tstp=-1`) and time stepping on the one hand, and the starting phase on the other hand. For the former, we have defined the routine

```
  void GenChi(int tstp, double h, double beta, GREEN &Pol,
  CFUNC &U, GREEN &PxU, GREEN &UxP, GREEN &Chi, int SolveOrder){

    PxU.set_timestep(tstp, Pol);
    UxP.set_timestep(tstp, Pol);
    PxU.right_multiply(tstp, U);
    UxP.left_multiply(tstp, U);
    PxU.smul(tstp,-1.0);
    UxP.smul(tstp,-1.0);

    if(tstp==-1){
      cntr::vie2_mat(Chi,PxU,UxP,Pol,beta,SolveOrder);
    } else{
      cntr::vie2_timestep(tstp,Chi,PxU,UxP,Pol,beta,h,SolveOrder);
    }
  }
```

Here, `PxU` and `UxP` correspond to the kernel K_{ij} and its hermitian conjugate, respectively. Analogously, the starting routine is implemented as

```
  void GenChi(double h, double beta, GREEN &Pol, CFUNC &U,
      GREEN &PxU, GREEN &UxP, GREEN &Chi, int SolveOrder){

    for(int n = 0; n <= SolveOrder; n++){
      PxU.set_timestep(n, Pol);
      UxP.set_timestep(n, Pol);
      PxU.right_multiply(n, U);
      UxP.left_multiply(n, U);
      PxU.smul(n,-1.0);
      UxP.smul(n,-1.0);
    }

    cntr::vie2_start(Chi,PxU,UxP,Pol,beta,h,SolveOrder);

  }
```

Finally, the self-energy is computed by

```
  void Sigma_GW(int tstp, GREEN &G, CFUNC &U, GREEN &Chi, GREEN &Sigma){
    int nsites=G.size1();
    int ntau=G.ntau();
    GREEN_TSTP deltaW(tstp,ntau,nsites,BOSON);

    Chi.get_timestep(tstp,deltaW);
    deltaW.left_multiply(tstp,U);
    deltaW.right_multiply(tstp,U);

    for(int i=0; i<nsites; i++){
      for(int j=0; j<nsites; j++){
         cntr::Bubble2(tstp,Sigma,i,j,G,i,j,deltaW,i,j);
      }
    }
  }
```

Self-energy approximation: T-matrix. — The particle–particle ladder $T_{ij}(t,t')$ represents an effective particle–particle interaction, which defines the corresponding self-energy

$$\Sigma_{ij}(t,t') = iU(t)T_{ij}(t,t')U(t')G_{ji}(t',t) \ . \tag{54}$$

The T-matrix, in turn, is obtained by solving the Dyson equation $T = \Phi - \Phi * U * T$, where Φ corresponds to the particle–particle bubble $\Phi_{ij}(t,t') = -iG_{ij}(t,t')G_{ij}(t,t')$. Hence, the procedure of numerically computing the $\Sigma_{ij}(t,t')$ is analogous to the GW approximation:

1. Compute $\Phi_{ij}(t,t')$ by `Bubble2` and multiply by -1.
2. Calculate the kernel $K_{ij}(t,t') = \Phi_{ij}(t,t')U(t')$ and its hermitian conjugate and solve the VIE $[1+K]*T = \Phi$ for T using `vie2`.
3. Perform the operation $T_{ij}(t,t') \to U(t)T_{ij}(t,t')U(t')$ and compute the self-energy by `Bubble1`.

Mean-field Hamiltonian and onsite quench. — So far, we have described how to compute the dynamical contribution to the self-energy. The total self-energy furthermore includes the Hartree–Fock (HF) contribution, which we incorporate into the mean-field Hamiltonian $\epsilon^{\mathrm{MF}}_{ij}(t) = \epsilon^{(0)}_{ij}(t) + U(n_i - \bar{n})$ with the occupation (per spin) $n_i = \langle \hat{c}^\dagger_i \hat{c}_i \rangle$. The shift of chemical potential $-U\bar{n}$ is a convention to fix the chemical potential at half filling at $\mu = 0$. Note that when we write the interaction term as in Eq. (49), the corresponding Fock term is zero because of the spin symmetry in the paramagnetic phase. In the example program, the mean-field Hamiltonian is represented by the contour function `eps_mf`. Updating `eps_mf` is accomplished by computing the density matrix using the `herm_matrix` class routine `density_matrix`.

The general procedure to implement a quench of some parameter λ at $t = 0$ is to represent λ by a contour function λ_n: λ_{-1} corresponds to the pre-quench value which determines the thermal equilibrium, while λ_n with $n \geq 0$ governs the time evolution. In the example program, we simplify this procedure by redefining $\epsilon^{(0)}_{ij} \to \epsilon^{(0)}_{ij} + w_0\delta_{i,1}\delta_{j,1}$ after the Matsubara Dyson equation has been solved.

Generic structure of the example program. —

The source code for the 2B, GW and T-matrix approximation, is found in `hubbard_chain_2b.cpp`, `hubbard_chain_gw.cpp`, `hubbard_chain_tpp.cpp` in folder `programs/`, respectively. The programs are structured similarly as the previous examples. After reading variables from file and initializing the variables and classes, the Matsubara Dyson equation is solved in a self-consistent fashion. The example below illustrates this procedure for the 2B approximation.

```
  tstp=-1;
  gtemp = GREEN(SolveOrder,Ntau,Nsites,FERMION);
  gtemp.set_timestep(tstp,G);

  for(int iter=0;iter<=MatsMaxIter;iter++){
    // update mean field
    hubb::Ham_MF(tstp, G, Ut, eps0, eps_mf);

    // update self-energy
    hubb::Sigma_2B(tstp, G, Ut, Sigma);

    // solve Dyson equation
    cntr::dyson_mat(G, MuChem, eps_mf, Sigma, beta, SolveOrder);

    // self-consistency check
    err = cntr::distance_norm2(tstp,G,gtemp);

  if(err<MatsMaxErr){
     break;
  }
  gtemp.set_timestep(tstp,G);
}
```

Updating the mean-field Hamiltonian (`hubb::Ham_MF`), the self-energy (`hubb::Sigma_2B`) and solving the corresponding Dyson equation (`cntr::dyson_mat`) is repeated until self-consistency has been reached, which in practice means that the deviation between the previous and updated GF is smaller than the given number `MatsMaxErr`. For other self-energy approximations, the steps described above (updating auxiliary quantities) have to be performed before the self-energy can be updated.

Once the Matsubara Dyson equation has been solved up to the required convergence threshold, the start-up algorithm for time steps $n = 0, \dots, k$ can be applied. To reach self-consistency for the first few time steps, we employ the bootstrapping loop:

```
for (int iter = 0; iter <= BootstrapMaxIter; iter++) {
  // update mean field
  for(tstp=0; tstp<=SolveOrder; tstp++){
     hubb::Ham_MF(tstp, G, Ut, eps_0, eps_mf);
  }

  // update self-energy
  for(tstp=0; tstp<=SolveOrder; tstp++){
      hubb::Sigma_2B(tstp, G, Ut, Sigma);
  }

  // solve Dyson equation
  cntr::dyson_start(G, MuChem, eps_mf, Sigma, beta, h, SolveOrder);

  // self-consistency check
  err=0.0;
  for(tstp=0; tstp<=SolveOrder; tstp++) {
    err += cntr::distance_norm2(tstp,G,gtemp);
  }

  if(err<BootstrapMaxErr && iter>2){
    break;
  }

  for(tstp=0; tstp<=SolveOrder; tstp++) {
    gtemp.set_timestep(tstp,G);
  }
}
```

Finally, after the bootstrapping iteration has converged, the time propagation for time steps $n > k$ is launched. The self-consistency at each time step is accomplished by iterating the update of the mean-field Hamiltonian, GF and self-energy over a fixed number of `CorrectorSteps`. As an initial guess, we employ a polynomial extrapolation of the GF from time step $n-1$ to n, as implemented in the routine `extrapolate_timestep` (see Appendix A.1). Thus, the time propagation loop takes the form

```
for(tstp = SolveOrder+1; tstp <= Nt; tstp++){
  // Predictor: extrapolation
  cntr::extrapolate_timestep(tstp-1, G ,SolveOrder);
  // Corrector
  for (int iter=0; iter < CorrectorSteps; iter++){
     // update mean field
     hubb::Ham_MF(tstp, G, Ut, eps0, eps_mf);

     // update self-energy
     hubb::Sigma_2B(tstp, G, Ut, Sigma);

     // solve Dyson equation
     cntr::dyson_timestep(tstp, G, MuChem, eps_mf, Sigma, beta, h, SolveOrder);
   }
}
```

After the GF has been computed for all required time steps, we compute the observables. In particular, the conservation of the total energy provides a good criterion to assess the accuracy of the calculation. The total energy per spin for the Hubbard model (49) is given in terms of the Galitskii–Migdal formula [9].

$$E = \frac{1}{2}\mathrm{Tr}\left[\rho(t)\left(\epsilon^{(0)} + \epsilon^{\mathrm{MF}}(t)\right)\right] + \frac{1}{2}\mathrm{ImTr}\left[\Sigma * G\right]^{<}(t,t)\,. \tag{55}$$

The last term, known as the correlation energy, is most conveniently computed by the routine

```
Ecorr = cntr::correlation_energy(tstp, G, Sigma, beta, h, SolveOrder);
```

Running the example programs. — There are three programs for the 2B, *GW* and *T*-matrix approximation, respectively: `hubbard_chain_2b.x`, `hubbard_chain_gw.x`, `hubbard_chain_tpp.x`. The driver script `demo_hubbard_chain.py` located in the `utils/` directory provides a simple interface to these programs. After defining the parameters and convergence parameters, the script creates the corresponding input file and launches all three programs in a loop. The occupation of the first lattice site $n_1(t)$ and the kinetic and

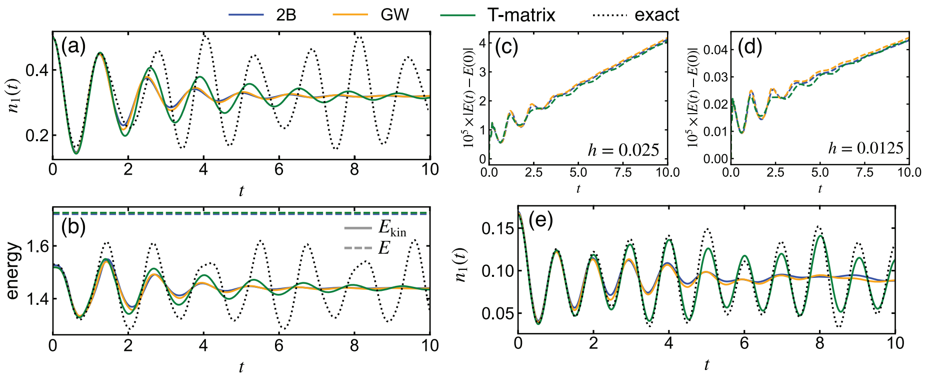


Fig. 8. Dynamics in the Hubbard chain. (a) Occupation on the first site $n_1(t)$ for $M=2$, $U=1$, $n=1/2$ and $w_0=5$. (b) Corresponding kinetic (solid) and total (dashed lines) energy. (c) and (d): deviation of the total energy from the initial value, corresponding to (b), for time steps $h=0.025$ and $h=0.0125$, respectively. (e) Occupation on the first site for $M=4$, $U=1.5$, $n=1/4$ and $w_0=5$.

total energy are then read from the output files and plotted. The script `demo_hubbard_chain.py` also allows to pass reference data as an optional argument, which can be used to compare, for instance, to exact results.

Discussion. — Following Ref. [22], we have selected two prominent examples illustrating the shortcomings of weak-coupling diagrammatic treatments for finite systems and strong excitations. The regimes where the discussed approximations work well are systematically explored in Refs. [22,24].

For the Hubbard dimer ($M=2$) at half filling ($\mu=0$, $\bar{n}=1/2$), a strong excitation (here $w_0=5$) leads to the phenomenon of artificial damping: although the density $n_1(t)$ exhibits an oscillatory behavior for all times in an exact treatment, the approximate NEGF treatment – with either self-energy approximation considered here – shows damping to an unphysical steady state (see Fig. 8(a)–(b)). It is instructive to look at the total energy, shown as dashed lines in Fig. 8(b). The conservation of total energy is illustrated in Fig. 8(c)–(d). For the relatively large time step $h=0.025$, the energy is conserved up to 4×10^{-5} in the considered time interval, while using a half as small step $h=0.0125$ improves the accuracy of the energy conservation by two orders of magnitude.

Fig. 8(e) shows the corresponding dynamics of the occupation for $M=4$ and quarter filling. In the regime of small filling, the T-matrix approximation is known to provide a good description for any strength of the interaction. This is confirmed by Fig. 8(e), where the 2B and GW approximation lead to artificial damping, while the $n_1(t)$ calculated by the T-matrix approximation agrees well with the exact result.

6.3. DMFT for the Holstein model

Overview. — In this section, we study the dynamics of the Holstein model, which is a fundamental model for electron–phonon (el-ph) coupled systems. This example demonstrates a minimal application of `libcntr` within the nonequilibrium dynamical mean-field theory (DMFT) [14], as well as the usage of the phonon (bosonic) GFs.

The Hamiltonian of the single-band Holstein model is

$$H(t)=-J(t)\sum_{\langle i,j\rangle,\sigma}\hat{c}^\dagger_{i,\sigma}\hat{c}_{j,\sigma}-\mu\sum_i\hat{n}_i+\omega_0\sum_i\hat{a}^\dagger_i\hat{a}_i+g(t)\sum_i(\hat{a}^\dagger_i+\hat{a}_i)\hat{n}_i. \tag{56}$$

Here $J(t)$ is the hopping parameter of the electrons, μ is the chemical potential, ω_0 is the phonon frequency, and $g(t)$ is the el–ph coupling. As excitation protocols, we consider modulations of the hopping parameter or the el–ph coupling as excitation protocols. For simplicity, in the following we consider the Bethe lattice (with infinite coordination number). For this lattice, the free electrons have a semi-circular density of states, $\rho_0(\epsilon)=\frac{1}{2\pi J^{*2}}\sqrt{4J^{*2}-\epsilon^2}$, with J^* a properly renormalized hopping amplitude [25]. Here we take $J^*=1$. Assuming spin symmetry, the lattice GFs are introduced as

$$G_{ij}(t,t')=-i\langle T_{\mathcal{C}}\hat{c}_{i,\sigma}(t)\hat{c}^\dagger_{j,\sigma}(t')\rangle, \tag{57a}$$

$$D_{ij}(t,t')=-i\langle T_{\mathcal{C}}\Delta\hat{X}_i(t)\Delta\hat{X}_j(t')\rangle. \tag{57b}$$

Here $\hat{X}_i=\hat{a}^\dagger_i+\hat{a}_i$ and $\Delta\hat{X}_i(t)=\hat{X}_i(t)-\langle\hat{X}_i(t)\rangle$.

We treat the dynamics of the Holstein model using the DMFT formalism [26]. In DMFT, the lattice model is mapped to an effective impurity model with a properly adjusted free electron bath, which is characterized by the so-called hybridization function $\Delta(t,t)$, see Eq. (60a). The hybridization function is self-consistently determined, so that the impurity GF ($G_{\rm imp}(t,t')$) and the impurity self-energy ($\Sigma_{\rm imp}$) are identical to the local Green's function ($G_{\rm loc}=G_{ii}$) and the local self-energy of the lattice problem ($\Sigma_{\rm loc}$), respectively. In practice, the DMFT implementation consists of (i), solving the impurity model for a given $\Delta(t,t')$ to obtain $G_{\rm imp}(t,t')$ and $\Sigma_{\rm imp}$, and (ii), the DMFT lattice self-consistency part, where we update $G_{\rm loc}$ and $\Delta(t,t')$ assuming $\Sigma_{\mathbf{k}}=\Sigma_{\rm imp}$. In the case of a Bethe lattice, the DMFT lattice self-consistency part is simplified and the hybridization function can be determined directly from the GF,

$$\Delta(t,t')=J^*(t)G_{\rm imp}(t,t')J^*(t'). \tag{58}$$

The action of the corresponding effective impurity model in the path integral formalism is[3]

$$\mathcal{S}_{\rm imp} = i\sum_\sigma \int_{\mathcal{C}} dtdt' c^\dagger_\sigma(t)\mathcal{G}_0^{-1}(t,t')c_\sigma(t') + i\int_{\mathcal{C}} dtdt' X(t)\frac{D_0^{-1}(t,t')}{2}X(t') - ig\sum_\sigma \int_{\mathcal{C}} dt X(t)c^\dagger_\sigma(t)c_\sigma(t), \tag{59}$$

where

$$\mathcal{G}_0^{-1}(t,t') = [i\partial_t + \mu]\delta_{\mathcal{C}}(t,t') - \Delta(t,t'), \tag{60a}$$

$$D_0^{-1}(t,t') = \frac{-\partial_t^2 - \omega_0^2}{2\omega_0}\delta_c(t,t'). \tag{60b}$$

The electron and phonon GFs of the impurity problem are determined by the Dyson equations

$$[i\partial_t - \mu - \Sigma^{\rm MF}_{\rm imp}(t)]G_{\rm imp}(t,t') - [(\Delta + \Sigma^{\rm corr}_{\rm imp}) * G_{\rm imp}](t,t') = \delta_{\mathcal{C}}(t,t'), \tag{61a}$$

$$D_{\rm imp}(t,t') = D_0(t,t') + [D_0 * \Pi_{\rm imp} * D_{\rm imp}](t,t'), \tag{61b}$$

and the phonon displacement, $X_{\rm imp}(t) = \langle \hat{X}_{\rm imp}(t)\rangle$, which is described by

$$X_{\rm imp}(t) = -\frac{2g(0)}{\omega_0}n_{\rm imp}(0) + \int_0^t d\bar{t} D_0^{\rm R}(t,\bar{t})[g(\bar{t})n_{\rm imp}(\bar{t}) - g(0)n_{\rm imp}(0)]. \tag{62}$$

Here the mean-field contribution ($\Sigma^{\rm MF}_{\rm imp}(t)$) corresponds to

$$\Sigma^{\rm MF}_{\rm imp}(t) = g(t)X_{\rm imp}(t), \tag{63}$$

$\Sigma^{\rm corr}_{\rm imp}(t,\bar{t})$ is the beyond-mean-field contribution to the self-energy, $D_0(t,t') \equiv -i\langle \Delta\hat{X}(t)\Delta\hat{X}(t')\rangle_0$ is for the free phonon system, $\Pi_{\rm imp}$ is the phonon self-energy and $n_{\rm imp}(t) = \langle \hat{n}_{\rm imp}(t)\rangle$ is the particle number at the impurity site.

After the DMFT loop is converged, one can calculate some observables such as different energy contributions. The expressions for the energies (per site) are given in the following expressions. The kinetic energy is

$$E_{\rm kin}(t) = \frac{1}{N}\sum_{\langle i,j\rangle,\sigma} -J(t)\langle \hat{c}^\dagger_{i,\sigma}(t)\hat{c}_{j,\sigma}(t)\rangle = -2i[\Delta * G_{\rm loc}]^<(t,t). \tag{64}$$

The interaction energy can be expressed as

$$E_{\rm nX}(t) = \frac{g(t)}{N}\sum_i \langle \hat{X}_i \hat{n}_i\rangle = \Sigma^{\rm MF}_{\rm loc}(t)n(t) - 2i[\Sigma^{\rm corr}_{\rm loc} * G_{\rm loc}]^<(t,t). \tag{65}$$

The phonon energy is

$$E_{\rm ph}(t) = \frac{\omega_0}{N}\sum_i \langle \hat{a}^\dagger_i \hat{a}_i\rangle = \frac{\omega_0}{4}[iD^<(t,t) + X(t)^2] + \frac{\omega_0}{4}[iD^<_{\rm PP}(t,t) + P(t)^2]. \tag{66}$$

Here $D_{\rm PP}(t,t') = -i\langle T_{\mathcal{C}}\Delta\hat{P}_i(t)\Delta\hat{P}_i(t')\rangle$ with $\hat{P}_i = \frac{1}{i}(\hat{a}_i - \hat{a}_i^\dagger)$ and $\Delta\hat{P}_i(t) \equiv \hat{P}_i(t) - \langle\hat{P}_i(t)\rangle$. We note that translational invariance is assumed and $X(t) = \langle\hat{X}_i(t)\rangle = X_{\rm imp}(t)$, $P(t) = \langle\hat{P}_i(t)\rangle = P_{\rm imp}(t)$, $D = D_{ii} = D_{\rm imp}$, $\Sigma_{\rm loc} = \Sigma_{\rm imp}$.

In this example, we solve the impurity problem using the simplest weak-coupling expansion as an impurity solver, i.e. the unrenormalized Migdal approximation (uMig) [27,28], where the phonons act as a glue for the electrons as well as a heat bath. On the web page www.nessi.tuxfamily.org we discuss an alternative impurity solver based on the self-consistent Migdal approximation (sMig) [29–31]. Both solvers are implemented in the C++ module `Holstein_impurity_impl.cpp`.

Unrenormalized Migdal approximation as an impurity solver: uMig. — The impurity self-energy for the electron is approximated as

$$\hat{\Sigma}^{\rm uMig,corr}_{\rm imp}(t,t') = ig(t)g(t')D_0(t,t')G_{\rm imp}(t,t'), \tag{67}$$

while we do not consider the self-energy of the phonons. In `NESSi`, $\frac{1}{2}D_0(t,t')$ is obtained by a cntr routine as

```
cntr::green_single_pole_XX(D0,Phfreq_w0,beta,h);
```

In the sample program, the unrenormalized Migdal approximation (uMig) self-energy is computed by the routine `Sigma_uMig`. We provide two interfaces for $0 \leq$ `tstp` $\leq$ `SolveOrder` (bootstrapping within the start-up algorithm) and `tstp` $= -1$, `tstp` $>$ `SolveOrder` (Matsubara part and the time-stepping part), respectively. Here, we show the latter as an example:

```
void Sigma_uMig(int tstp, GREEN &G, GREEN &D0, CFUNC &g_el_ph, GREEN &Sigma){

        int Norb=G.size1();
        int Ntau=G.ntau();

        GREEN_TSTP gGg(tstp,Ntau,Norb,FERMION);
        G.get_timestep(tstp,gGg);//copy time step from G
        gGg.right_multiply(tstp,g_el_ph);
        gGg.left_multiply(tstp,g_el_ph);

```

[3] Here we denote the Grassmann fields by $c^\dagger$ and c and the scalar field as X.

```
        //Get Sig(t,t')=ig^2 D_0(t,t') G(t,t')

        Bubble2(tstp,Sigma,0,0,D0,0,0,gGg,0,0);

 }
```

In this routine, the electron self-energy Eq. (67) is evaluated using `Bubble2`, see Section 3.3.

Generic structure of the example program. — The program for DMFT + uMig is implemented in `Holstein_bethe_uMig.cpp` for normal states. As in the case of the Hubbard chain, the program consists of three main steps: (i) solving the Matsubara Dyson equation, (ii) bootstrapping within the start-up algorithm (`tstp`$\leq$ `SolveOrder`) and (iii) time propagation for `tstp` $>$ `SolveOrder`. Since the generic structure of each step is similar to the respective step for the Hubbard chain, we only show here the time propagation part to illustrate the differences.

```
for(tstp = SolverOrder+1; tstp <= Nt; tstp++){
  // Predictor: extrapolation
  cntr::extrapolate_timestep(tstp-1,G,SolveOrder);
  cntr::extrapolate_timestep(tstp-1,Hyb,SolveOrder);

  // Corrector
  for (int iter=0; iter < CorrectorSteps; iter++){
    //========================
    // Solve Impurity problem
    // ========================
    cdmatrix rho_M(1,1), Xph_tmp(1,1);
    cdmatrix g_elph_tmp(1,1),h0_imp_MF_tmp(1,1);

    //update self-energy
    Hols::Sigma_uMig(tstp, G, D0, g_elph_t, Sigma);

    //update phonon displacement
    G.density_matrix(tstp,rho_M);
    rho_M *= 2.0;//spin number=2
    n_tot_t.set_value(tstp,rho_M);
    Hols::get_phonon_displace(tstp, Xph_t, n_tot_t, g_elph_t, D0, Phfreq_w0, SolveOrder, h);

    //update mean-field
    Xph_t.get_value(tstp,Xph_tmp);
    g_elph_t.get_value(tstp,g_elph_tmp);
    h0_imp_MF_tmp = h0_imp + Xph_tmp*g_elph_tmp;
    h0_imp_MF_t.set_value(tstp,h0_imp_MF_tmp);

    //solve Dyson for impurity
    Hyb_Sig.set_timestep(tstp,Hyb);
    Hyb_Sig.incr_timestep(tstp,Sigma,1.0);
    cntr::dyson_timestep(tstp, G, 0.0, h0_imp_MF_t, Hyb_Sig, beta, h , SolveOrder);

    //==================================
    // DMFT lattice self-consistency (Bethe lattice)
    // ==================================
    //Update hybridization
    Hyb.set_timestep(tstp,G);
    Hyb.right_multiply(tstp,J_hop_t);
    Hyb.left_multiply(tstp,J_hop_t);
  }
}
```

At the beginning of each time step, we extrapolate the local GF and the hybridization function, which serves as a predictor. Next, we iterate the DMFT self-consistency loop (corrector) until convergence is reached. In this loop, we first solve the impurity problem to update the local self-energy and GF. Then we update the hybridization function by the lattice self-consistency condition, which in the case of the Bethe lattice simplifies to Eq. (58).

Running the example programs. — The corresponding executable file is named `Holstein_bethe_uMig.x`. In these programs, we use $\mu_{\rm MF} \equiv \mu - gX(0)$ as an input parameter instead of μ. (μ is determined in a post-processing step.) Excitations via modulations of the hopping and el-ph coupling are implemented, where we need to provide dg ($\equiv g(t) - g(0)$) and $dJ^*(t)$ ($\equiv J(t) - J(0)$) as inputs. The driver script `demo_Holstein.py` located in the `utils/` directory provides a simple interface to the program. After defining the system parameters, numerical parameters (time step, convergence criterion, etc.) and excitation parameters, the script creates the corresponding input file and starts the simulation. After the simulation, the number of particles for each site ($n(t) = \sum_\sigma n_\sigma(t)$), phonon displacement ($X(t)$), phonon momentum ($P(t)$) and the energies are plotted. In addition, the spectral functions of electrons and phonons, respectively,

$$A^{\rm R}(\omega; t_{\rm av}) = -\frac{1}{\pi}\int dt_{\rm rel} e^{i\omega t_{\rm rel}} F_{\rm gauss}(t_{\rm rel}) G^{\rm R}(t_{\rm rel}; t_{\rm av}), \tag{68a}$$

$$B^{\rm R}(\omega; t_{\rm av}) = -\frac{1}{\pi}\int dt_{\rm rel} e^{i\omega t_{\rm rel}} F_{\rm gauss}(t_{\rm rel}) D^{\rm R}(t_{\rm rel}; t_{\rm av}), \tag{68b}$$

are plotted using a `python3` script in `NESSi` for $t_{\rm av} = \frac{Nt\cdot h}{2}$. Here, $F_{\rm gauss}(t_{\rm rel})$ is a Gaussian window function, which can also be specified in `demo_Holstein_impurity.py`.

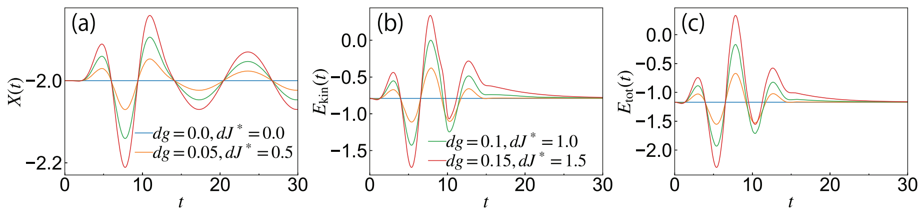


Fig. 9. Time evolution of (a) the phonon displacement, (b) the kinetic energy and (c) the total energy after excitation via the simultaneous modulation of the el–ph coupling and the hopping parameter within DMFT + uMig. We use $g = 0.5$, $\omega_0 = 0.5$ and $\beta = 10.0$ and consider the half-filled case. Here, we use a $\sin^2$ envelope for both modulations with excitation frequency $\Omega = 1.2$ and pulse duration $T_{\rm end} = 15.7$. The strength of the excitation is indicated by dg and dJ^*.

Discussion. — In Fig. 9(a)–(c), we show the time evolution of the phonon displacement $X(t)$, the kinetic energy $E_{\rm kin}(t)$ and the total energy $E_{\rm tot}(t)$ after excitation by the simultaneous modulation of the el-ph coupling and the hopping parameter. Since the energy is gradually absorbed by the phonons after the excitation, both $E_{\rm kin}(t)$ and $E_{\rm tot}(t)$ gradually relax toward the initial value, i.e. the equilibrium value at the phonon temperature. We note that we also provide an example program for the self-consistent Migdal approximation as an impurity solver as well as programs for the Nambu formalism for the superconducting states. Explanations and demonstrations of these sample programs are given on the webpage `http://www.nessi.tuxfamily.org/`.

7. MPI Parallelization

7.1. Parallelization

In `libcntr`, we provide tools for distributed-memory parallelization via the Message Passing Interface (MPI). In particular, the parallel layout is tailored to treat vectors of GFs, which is relevant for the simulation of *extended* systems. In this case, all quantities are additionally labelled by the reciprocal lattice vector $\vec{k}$ chosen from the first Brillouin zone (BZ). For instance, the Dyson equation for the GF $G_{\vec{k}}(t,t')$ takes the form

$$\left(i\partial_t - \epsilon_{\vec{k}}(t)\right) G_{\vec{k}}(t,t') = \delta_{\mathcal{C}}(t,t') + \left[\Sigma_{\vec{k}} * G_{\vec{k}}\right](t,t')\ . \tag{69}$$

This equation can be solved independently for each $\vec{k}$, which can be performed in parallel without communication. Constructing the self-energy $\Sigma_{\vec{k}}(t,t')$ then typically requires information from different points $\vec{k}' \neq \vec{k}$ in the BZ. However, the computational effort to solve the Dyson equation at a time step N scales like $\mathcal{O}(N^2)$, while the amount of data to be communicated scales only like $\mathcal{O}(N)$, so that the problem can be parallelized using a distributed memory parallelization with moderate communication overhead.

To handle the all-to-all communication of Green's functions among MPI ranks, we have implemented the auxiliary class `distributed_timestep_array`. (Simple point-to-point communication of time-steps can be done using member functions of the `herm_matrix_class`, which is described in the online manual.) Below we provide an overview over the `distributed_timestep_array` and its parent class `distributed_array`, and discuss a real-time *GW* simulation as an advanced example for a parallel application (see Section 7.2).

`Distributed_array.` — The class `distributed_array` provides a generic structure for distributing and communicating sets of data blocks. Let us assume the total number of points sampling the BZ is given by N_k and label the points by $k = 1, \ldots, N_k$. The `distributed_array` class is comprised of a vector of length N_k of any base class (provided by a template argument) on *every* MPI rank, as illustrated in Fig. 10(a). This makes the communication particularly straightforward. For instance, after updating an element of the `distributed_array` on rank 2 (see lower panel in Fig. 10(a)), this information is broadcasted to all other ranks using the MPI collective communication function `mpi_bcast`. Further functionalities include sending and receiving blocks among different ranks, as well as gathering all data on one rank (typically the master). The `distributed_array` thus provides a general framework for MPI parallelization, which can be adjusted to a particular situation. The most common usage is distributing instances of the `herm_matrix_timestep` class.

`Distributed_timestep_array.` — The class `distributed_timestep_array` is a specialization of the class `distributed_array`, for which the distributed base class is `herm_matrix_timestep`. Let us sketch the typical usage. Due to the high memory demands for storing two-time GFs as `herm_matrix`, we divide the total number of points N_k into a smaller local (with respect to the MPI rank) number of points $N_{k,\rm loc}$. Two-time functions such as the GF are stored as a vector of $N_{k,\rm loc}$: G_j with $j = 1, \ldots, N_{k,\rm loc}$ (see Fig. 10(b)). For each rank to have access to the full momentum dependence G_k, $k = 1, \ldots, N_k$, the `distributed_timestep_array` class is used to communicate the time slice $\mathcal{T}[G_k]_n$ for $k = 1, \ldots, N_k$. Fig. 10(b) illustrates this layout for the example of $N_{\rm tasks} = 4$ MPI ranks, $N_k = 8$ and, thus, $N_{k,\rm loc} = 2$. The precise calls to perform these communications will be best apparent from the example below, and are also described in detail in the online manual.

7.2. Example: GW for the translationally invariant Hubbard model

Overview. — As in Section 6.2, we will consider the Hubbard Hamiltonian, but we assume a translationally invariant system with periodic boundary conditions. The translational invariance implies that all observables and propagators are diagonal in momentum space. Hence, all quantities can be labelled by the reciprocal lattice vector $\vec{k}$ within the first BZ. This reduces the computational and storage complexity from $\mathcal{O}(N_k^2)$ for the real space formalism introduced in Section 6.2 to $\mathcal{O}(N_k)$. Moreover, the Dyson equation is diagonal in momentum space and can be efficiently parallelized using the distributed-memory parallelization based on MPI.

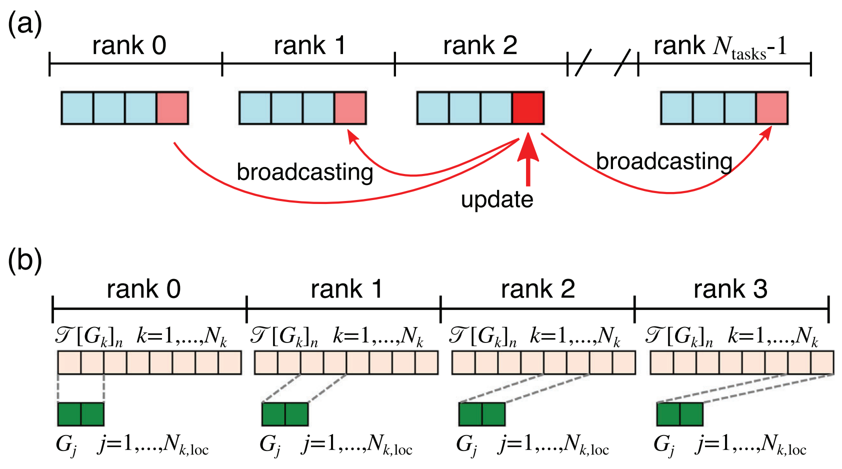

(a)
rank 0
rank 1
rank 2
rank N_{tasks}-1
broadcasting
broadcasting
update
(b)
rank 0
rank 1
rank 2
rank 3
$\mathcal{T}[G_k]_n$ $k=1,\ldots,N_k$
G_j $j=1,\ldots,N_{k,\text{loc}}$

Fig. 10. (a) Sketch of the parallelization in momentum space using the `distributed_array` class. A local update on one MPI rank is broadcasted to all other corresponding elements across all ranks. (b) Example of the parallelization in momentum space using the `distributed_timestep_array` class for $N_k = 8$, distributed over $N_{\text{tasks}} = 4$ MPI ranks. The full `herm_matrix` (represented by the dark green squares) is stored for local indices only, while the time slices for a fixed time step are stored on each rank (light pink squares). The gray dashed lines indicate the connection between local and global indices. (color online).

We will consider a 1D chain described by the Hubbard model, see Eq. (49). The single particle part of the Hamiltonian can be diagonalized as

$$\hat{H}_0 = \sum_{\vec{k},\sigma} \epsilon(\vec{k}) c^\dagger_{\vec{k}\sigma} c_{\vec{k}\sigma}, \tag{70}$$

where we have introduced the free-electron dispersion $\epsilon(\vec{k}) = -2J\cos([\vec{k}]_x)$. We will use a vector notation since the generalization to higher dimensional systems is straightforward. For the 1D chain used in the demonstration program, the momentum has only one component $[\vec{k}]_x = k_x$.

The system is excited via an electromagnetic field, which for a translationally invariant system is conveniently introduced using the Peierls substitution. The latter involves the vector potential $\vec{A}(t)$ (treated within the dipole approximation) as a phase factor in the hopping [32,33], or, equivalently a time-dependent shift in the single-particle dispersion

$$\epsilon(\vec{k}, t) = \epsilon(\vec{k} - q\vec{A}(t)/\hbar). \tag{71}$$

The vector potential is obtained from the electric field as $\vec{A}(t) = -\int_0^t \vec{E}(\bar{t})d\bar{t}$.

In this example, we treat the dynamics within the *GW* approximation, following an implementation similar to Ref. [34] (which considers a four-band model). The numerical evaluation of the respective self-energy expressions is implemented in the C++ module `gw_selfen_impl.cpp`, and the main code is found in `programs/gw.cpp`. Below we explain the key routines.

Self-energy approximation: GW. — In momentum space, the correlation part of the *GW* self-energy can be written as

$$\Sigma_{\vec{k}}(t, t') = \frac{i}{N_k} \sum_{\vec{q}} G_{\vec{k}-\vec{q}}(t, t') \delta W_{\vec{q}}(t, t'), \tag{72}$$

where we have introduced the Fourier transform of the propagator $X_{\vec{q}}(t, t') = (1/N_k) \sum_i \exp(i(\vec{r}_i - \vec{r}_j) \cdot \vec{q}) X_{ij}(t, t')$, see also Section 6.2 for the definition of the propagators. In line with Section 6.2, we have introduced the dynamical part of the effective interaction $\delta W_{\vec{q}}$ via $W_{\vec{q}}(t, t') = U\delta_{\mathcal{C}}(t, t') + \delta W_{\vec{q}}(t, t')$. Due to the translational invariance, the propagators and corresponding Dyson equations are diagonal in momentum space. This leads to a significant speed-up of calculations since the most complex operation, the solutions of the Volterra integral equation (VIE), can be performed in parallel. The retarded interaction is obtained as a solution of the Dyson-like equation

$$W_{\vec{k}}(t, t') = U\delta_{\mathcal{C}}(t, t') + U[\Pi_{\vec{k}} * W_{\vec{k}}](t, t'). \tag{73}$$

and the Fourier transform of the polarization is given by

$$\Pi_{\vec{k}}(t, t') = \frac{-i}{N_k} \sum_{\vec{q}} G_{\vec{k}+\vec{q}}(t, t') G_{\vec{q}}(t', t). \tag{74}$$

In the case of a non-local interaction, the polarization is multiplied by a spin factor $s = 2$.

This structure allows for an easy adaptation of the code to arbitrary lattice geometries. In particular, we provide an implementation of a 1D chain geometry in the class `lattice_1d_1b` within the C++ module `gw_lattice_impl.cpp`. The routine `add_kpoints` evaluates the sum or difference of two vectors $\vec{k} \pm \vec{q}$, where slight care has to be taken to map the vector back to the first BZ. For the modification to other lattices and interaction vertices, the user has to define the first BZ, the single-particle dispersion $\epsilon(\vec{k})$, and the interaction vertex U.

The generalization to the long-range interaction

$$\hat{H}_{\rm int} = U \sum_i \hat{n}_{i\uparrow}\hat{n}_{i\downarrow} + \frac{1}{2}\sum_{i,j} V(\vec{r}_i - \vec{r}_j)\hat{n}_i\hat{n}_j \tag{75}$$

is straightforward. For the purpose of demonstration, we have included the nearest-neighbor interaction $V(\vec{r}_i - \vec{r}_j) = \delta(|\vec{r}_i - \vec{r}_j| = 1)V$ into the example program (input parameter `V`). We should comment that the Fock term is zero for a purely local interaction in the paramagnetic phase as far as the interaction is written in the form of Eq. (75), while it takes a finite value in cases with a finite non-local interaction.

Distribution of momenta over MPI ranks. — As each momentum point is independent, we have introduced a class `kpoint` in the module `gw_kpoints_impl.cpp`. This class includes all propagators at the given momentum point $\vec{k}$, as well as corresponding methods, such as the solution of the Dyson equations for the single-particle propagator $G_{\vec{k}}(t,t')$, see Eq. (69), and the retarded interaction $W_{\vec{k}}(t,t')$. An arbitrary lattice can be represented as a set of `kpoint` objects. In the code, each physical momentum $\vec{k}$ is indexed by an index `k`$\in$ {0, . . .,`Nk-1`}, which we will refer to as the "global index" in the following. `lattice` is the variable which stores information on the lattice (in particular the relation between the index `k` and the physical momentum $\vec{k}$, and `lattice.nk_` returns `Nk`.

Each `kpoint` objects needs to be accessed at only one mpi rank, because the Dyson and vie2 integral equations can be solved independently for each rank. However, the evaluation of the self-energy and polarization diagrams at a given timeslice requires that the timeslice $\mathcal{T}[G_{\vec{k}}]$ and $\mathcal{T}[W_{\vec{k}}]$ at *all* $\vec{k}$ is made available at *all* mpi ranks, see Eq. (72). This is facilitated by introducing a setting with the following variables at a rank which holds `Nkloc` `kpoint` objects:

- `std::vector <kpoint > corrK_rank` : A vector of length `Nkloc`, containing the `kpoint` objects stored locally at the rank.
- `std::vector <int> kindex_rank`: A vector of length `Nkloc`.
  `kindex_rank[j]` returns the global index `k`$\in$ {0, . . .,`Nk-1`} of the `kpoint` j.
- `distributed_timestep_array gk_all_timesteps`, as described in Section 7.1. Can store $\mathcal{T}_n[G_{\vec{k}}]$ at a given timestep n for all `k`$\in$ {0, . . .,`Nk-1`}. `gk_all_timesteps.G()[k]` returns a reference to the data at $\mathcal{T}[G_{\vec{k}}]$. The class has a copy of `kindex_rank` and of the inverse map, so that one can easily launch a communication in which the rank which owns a given `kpoint` would send the corresponding timeslices to all other ranks.
- `distributed_timestep_array wk_all_timesteps`: Can store $\mathcal{T}_n[W_{\vec{k}}]$ at a given timestep n for all `k`$\in$ {0, . . .,`Nk-1`}. Analogous to `gk_all_timesteps`.

The strategy to compute the GW self-energy $\mathcal{T}[\Sigma_{\vec{k}}]_n$ at time step n thus consists of two steps:

1. At time t_n, communicate the latest time slice of the GFs $\mathcal{T}[G_{\vec{k}}]_n$ and retarded interactions $\mathcal{T}[W_{\vec{k}}]_n$ for all momentum points among all MPI ranks.
2. Evaluate the self-energy diagram $\mathcal{T}[\Sigma_{\vec{k}_{\rm rank}}]_n$ in Eq. (72) for a subset of momentum points $\vec{k}_{\rm rank}$ present on a given rank using the routine `Bubble2`.

Step 1 is implemented as

```
void gather_gk_timestep(int tstp,int Nk_rank,DIST_TIMESTEP &gk_all_timesteps,std::vector<kpoint> &corrK_rank
    ,std::vector<int> &kindex_rank){
    gk_all_timesteps.reset_tstp(tstp);
    for(int k=0;k<Nk_rank;k++){
      gk_all_timesteps.G()[kindex_rank[k]].get_data(corrK_rank[k].G_);
    }
    gk_all_timesteps.mpi_bcast_all();
  }
```

where the abbreviation `DIST_TIMESTEP` for `distributed_timestep_array` is used. An analogous routine is used for the bosonic counterpart. We gather the information from all ranks into an object of type `distributed_timestep_array` named `gk_all_timesteps`. `mpi_bcast_all()` is a wrapper around the MPI routine `Allgather` adjusted to the type `distributed_timestep_array`.
Step 2 is implemented as

```
void sigma_GW(int tstp,int kk,GREEN &S,DIST_TIMESTEP &gk_all_timesteps,DIST_TIMESTEP &wk_all_timesteps,
    lattice_1d_1b &lattice,int Ntau,int Norb){
      assert(tstp==gk_all_timesteps.tstp());
      assert(tstp==wk_all_timesteps.tstp());
      GREEN_TSTP stmp(tstp,Ntau,Norb,FERMION);
      S.set_timestep_zero(tstp);
      for(int q=0;q<lattice.nk_;q++){
        double wk=lattice.kweight_[q];
        int kq=lattice.add_kpoints(kk,1,q,-1);
        stmp.clear();
        for(int i1=0;i1<Norb;i1++){
          for(int i2=0;i2<Norb;i2++){
            cntr::Bubble2(tstp,stmp,i1,i2,gk_all_timesteps.G()[kq],gk_all_timesteps.G()[kq],i1,i2,
                wk_all_timesteps.G()[q],wk_all_timesteps.G()[q],i1,i2);
          }
        }
        S.incr_timestep(tstp,stmp,wk);
      }
  }
```

As each rank includes only a subset of momentum points $\vec{k}_{\rm rank}$ we only evaluate the self-energy diagrams $\Sigma_{\vec{k}_{\rm rank}}$ for this subset of momentum points. After the call to gather_gk_timestep, all ranks carry information about the latest timestep for all momentum points and the internal sum over momentum $\vec{q}$ in Eq. (72) can be performed on each rank. The evaluation of the self-energy is done using the bubble2 routines introduced in Section 3.3.

Generic structure of the example program. — As the generic structure is similar to the two previous examples we will focus on the peculiarities connected to the usage of MPI. First, we need to initialize the MPI session

```
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD, &ntasks);
    MPI_Comm_rank(MPI_COMM_WORLD, &tid);
    tid_root=0;
```

and the distributed_timestep_array for the electronic and bosonic propagators

```
    DIST_TIMESTEP gk_all_timesteps(Nk,Nt,Ntau,Norb,FERMION,true);
    DIST_TIMESTEP wk_all_timesteps(Nk,Nt,Ntau,Norb,BOSON,true);
```

The construction of the DIST_TIMESTEP variables generates the map kindex_rank between the subset of points stored on a given rank and the full BZ.

The program consists of three main parts, namely Matsubara, bootstrapping within the start-up algorithm (tstp $\leq$ SolverOrder) and time propagation for tstp $>$ SolverOrder. The self-consistency iterations include the communication of all fermionic and bosonic propagators between different ranks using the routine gather_gk_timestep and the determination of the local propagators. For instance, the Matsubara part (tstp $= -1$) looks as follows

```
      for(int iter=0;iter<=MatsMaxIter;iter++){
        // update propagators via MPI
        diag::gather_gk_timestep(tstp,Nk_rank,gk_all_timesteps,corrK_rank,kindex_rank);
        diag::gather_wk_timestep(tstp,Nk_rank,wk_all_timesteps,corrK_rank,kindex_rank);

        diag::set_density_k(tstp,Norb,Nk,gk_all_timesteps,lattice,density_k,kindex_rank,rho_loc);
        diag::get_loc(tstp,Ntau,Norb,Nk,lattice,Gloc,gk_all_timesteps);
        diag::get_loc(tstp,Ntau,Norb,Nk,lattice,Wloc,wk_all_timesteps);
```

As on each MPI rank, the momentum-dependent single-particle density matrix $\rho(\vec{k})$ is known for the whole BZ, the evaluation of the HF contribution is done as in Section 6.2. The self-energies $\Sigma_{k_{\rm rank}}$ for the momentum points $k_{\rm rank} = 0, \ldots, N_{\rm rank} - 1$ on each rank are obtained by the routine sigma_GW.

```
        // update mean field and self-energy
        for(int k=0;k<Nk_rank;k++){
          diag::sigma_Hartree(tstp,Norb,corrK_rank[k].SHartree_,lattice,density_k,vertex,Ut);
          diag::sigma_Fock(tstp,Norb,kindex_rank[k],corrK_rank[k].SFock_,lattice,density_k,vertex,Ut);
          diag::sigma_GW(tstp,kindex_rank[k],corrK_rank[k].Sigma_,gk_all_timesteps,wk_all_timesteps,lattice,
              Ntau,Norb);
        }
```

and the variable vertex (type: cntr::function) includes the (possibly time-dependent) values of the interaction U.

Similarly, the solution of the Dyson equation for the fermionic (bosonic) propagators for each momentum point is obtained by step_dyson_with_error (step_W_with_error) which is just a wrapper around the Dyson solver. It returns the error corresponding to the difference between the propagators at the previous and current iterations. The momentum-dependent error for the fermionic propagators is stored in err_ele and at the end we use MPI_Allreduce to communicate among the ranks.

```
        // solve Dyson equation
        double err_ele=0.0,err_bos=0.0;
        for(int k=0;k<Nk_rank;k++){
          err_ele += corrK_rank[k].step_dyson_with_error(tstp,iter,SolverOrder,lattice);
          diag::get_Polarization_Bubble(tstp,Norb,Ntau,kindex_rank[k],corrK_rank[k].P_,gk_all_timesteps,
              lattice);
          err_bos += corrK_rank[k].step_W_with_error(tstp,iter,tstp,SolverOrder,lattice);
        }
        MPI_Allreduce(MPI_IN_PLACE,&err_ele,1,MPI_DOUBLE_PRECISION,MPI_SUM,MPI_COMM_WORLD);
        MPI_Allreduce(MPI_IN_PLACE,&err_bos,1,MPI_DOUBLE_PRECISION,MPI_SUM,MPI_COMM_WORLD);
```

The structure of the start-up procedure and the real-time propagation are similar to the Matsubara solver. The main difference lies in the predictor–corrector scheme as explained in Section 4.1. At the beginning of each time step, we extrapolate the momentum-dependent GF $G_{\vec{k}}$ and the retarded interactions $W_{\vec{k}}$, which works as a predictor:

```
        // Predictor: extrapolation
        diag::extrapolate_timestep_G(tstp-1,Nk_rank,SolverOrder,Nt,corrK_rank);
        diag::extrapolate_timestep_W(tstp-1,Nk_rank,SolverOrder,Nt,corrK_rank);
```

Then we perform several iterations at a given time step until convergence, which acts as a corrector.

After the NEGFs are obtained, we evaluate the kinetic energy (per spin) $E_{\rm kin}(t) = \frac{1}{N_k}\sum_{\vec{k}} {\rm Tr}[\rho_{\vec{k}}(t)\epsilon_{\vec{k}}(t)]$. The interaction energy (per spin) is obtained from the Galitskii–Migdal formula

$$E_{\rm int}(t) = \frac{1}{2N_k}\sum_{\vec{k}}\left({\rm Tr}\left[\rho_{\vec{k}}(t)\left(h^{\rm MF}_{\vec{k}} - \epsilon_{\vec{k}}\right)\right] + {\rm ImTr}\left[\Sigma_{\vec{k}} * G_{\vec{k}}\right]^{<}(t,t)\right), \tag{76}$$

using the routine diag::CorrelationEnergy. The two operations include an MPI reduction as the momentum sum is performed over the whole BZ.

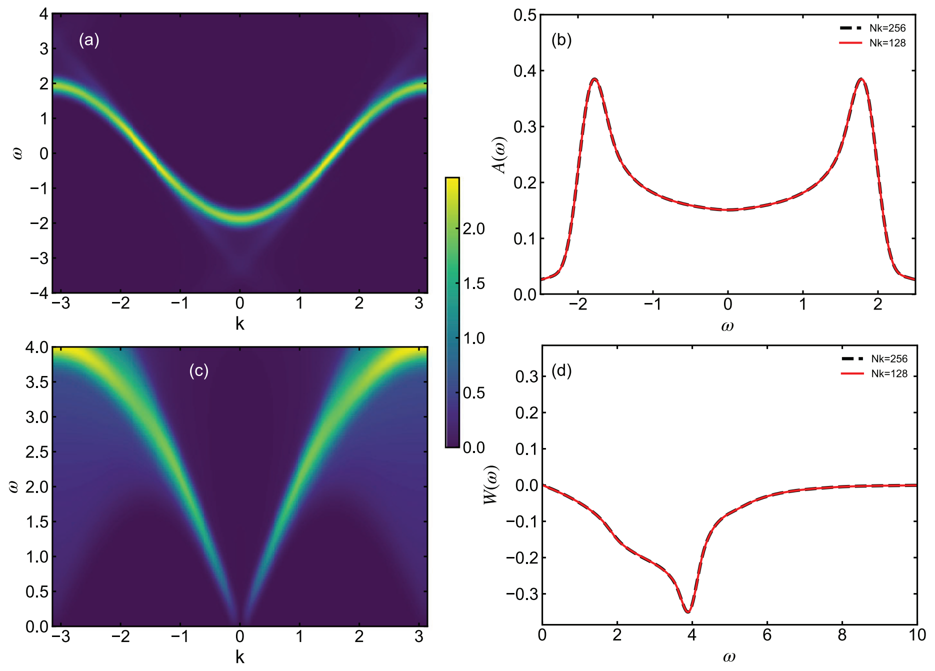


Fig. 11. (a) Momentum-dependent spectral function $A_{\vec{k}}(\omega)$ of the 1D Hubbard model, obtained within the *GW* approximation. (b) Local spectral function $A_{\rm loc}(\omega)$ for two system sizes $N_k = 128$ and $N_k = 256$, respectively. The second row shows the equivalent pair of panels for the effective interaction: (c) the momentum-dependent effective interaction $\mathrm{Im}[W_{\vec{k}}(\omega)]$ and, (d) its local part $\mathrm{Im}[W_{\rm loc}(\omega)]$. The parameters for all the plots are $U = 2$, the inverse temperature is $\beta = 20.0$ and we consider the half-filled case $n = 1$. The momentum-dependent quantities have been obtained with $N_k = 256$ momentum points.

Running the example program. — There is one program for the *GW* calculation, called `gw.x`. The driver script `demo_gw.py` located in the `utils/` directory provides a simple interface to this program. Similar to the examples in Section 6, the script creates an input file and launches the program. The user can specify the shape of the electric pulse, but by default, we use a single-frequency pulse with a Gaussian envelope

$$E(t) = E_0 \exp(-4.6(t - t_0)^2/t_0^2)\sin(\omega(t - t_0)), \tag{77}$$

where $t_0 = 2\pi/\omega N_p$ is determined by the number of cycles N_p. After the simulation, the time evolution of the kinetic energy and potential energy are plotted. The output is determined by two optional parameters. If `SaveGreen` is `true` the local fermionic (G) and bosonic (W) propagators are stored to disk. If `SaveMomentum` is `true` also the momentum-dependent propagators are stored to disk. As the full momentum and time-dependent propagators would require a large amount of memory, we only save selected time slices and their frequency is determined by the parameter `output`. For example, if `output=100`, every 100th timeslice will be stored to disk.

Running the driver script `demo_gw.py` produces the following output files:

1. By default it produces a file `data_gw.h5`, which includes information about the time-evolution of observables like the kinetic energy, density, etc.
2. Setting the parameter `savegf` to 1 will create two additional groups within the file `data_gw.h5`, namely `Gloc` and `Wloc`. These groups include the total two-time information about the local single-particle propagator $G_{\rm loc}(t, t')$ (`Gloc`) and the local two-particle propagator $W_{\rm loc}(t, t')$ (`Wloc`).
3. Setting the parameter `savegk` to 1 will create a set of files for each momentum point. These files include information about the momentum-dependent single-particle propagators $G_{\rm k}(t, t')$ (group `G`) and the corresponding two-particle propagators $W_{\rm k}(t, t')$ (group `W`).

Discussion. — The equilibrium momentum-dependent spectral function $A_k(\omega) = -\frac{1}{\pi}\mathrm{Im}\left[G_{\vec{k}}(\omega)\right]$ and its local part $A_{\rm loc}(\omega) = \frac{1}{N_k}\sum_{\vec{k}} A_k(\omega)$ are presented in Fig. 11. The local spectral function $A_{\rm loc}(\omega)$ shows the typical van Hove singularities present in 1D systems at $\omega \approx \pm 2$. The comparison between two system sizes, namely $N_k = 128$ and $N_k = 256$, shows that the spectrum is converged. The momentum-dependent spectral function $A_{\vec{k}}(\omega)$ closely follows the single-particle dispersion $\epsilon_{\vec{k}}$. The broadening due to many-body effects is small close to the Fermi surface points ($\pm\pi/2$), because of the restricted scattering, but it is increasing with increasing energies. Note that the *GW* approximation cannot capture peculiarities of 1D systems, like the absence of the Fermi surface as described by the Tomanaga–Luttinger liquid [35]. However, this is specific to low-dimensional systems and we consider the 1D case here mainly to avoid heavy calculations in the example program. Another interesting observation is the presence of a shadow band, which is clearly visible for energies away from the chemical potential. The origin of this shadow band is the feedback of the two-particle excitations on the single-particle spectrum.

The information about the two-particle excitation spectrum is contained in the bosonic correlator W. As the latter is antisymmetric in frequency, $\mathrm{Im}[W(\omega)] = -\mathrm{Im}[W(-\omega)]$, we only present results for positive frequencies, see Fig. 11. The local bosonic correlator $\mathrm{Im}[W_{\rm loc}(\omega)]$ is presented in Fig. 11(d) for two system sizes $N_k = 128$ and $N_k = 256$, respectively. The local component $\mathrm{Im}[W_{\rm loc}(\omega)]$

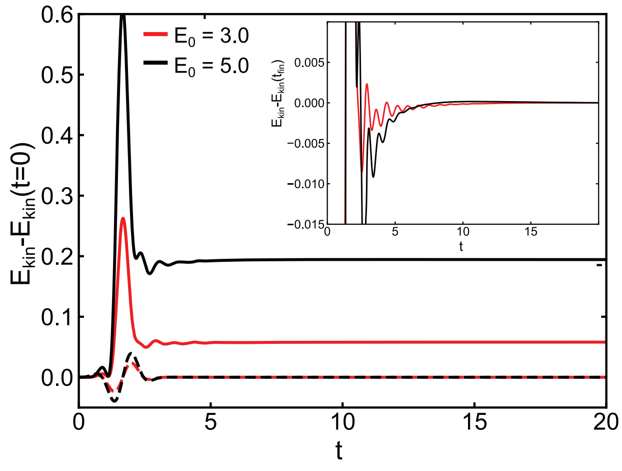


Fig. 12. Time evolution of the kinetic energy for the two excitation strengths $E_0 = 3.0, 5.0$, respectively. The dashed lines show the shape of the electric field pulse scaled down by 100 to fit on the scale. The inset presents a zoom into the relaxation dynamics by subtracting the long-time limit $E_{\text{kin}}(t) - E_{\text{kin}}(t_{\text{fin}})$. Both simulations have been performed with $N_k = 256$, time step $h = 0.01$ and for inverse temperature $\beta = 20$.

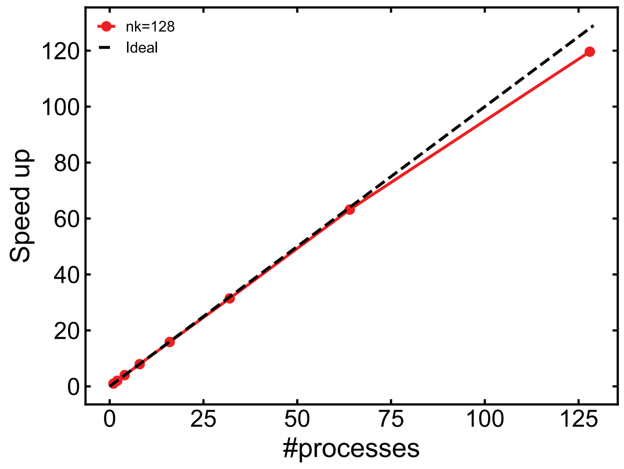


Fig. 13. Speed-up of the total calculation time as a function of the MPI processes for systems with $N_k = 128$ momentum points, where we fixed one task per node. The maximum number of time steps used is $N_t = 2500$. These calculations have been performed on the Popeye cluster at the Flatiron Institute.

shows a strong peak around $\omega \approx 4$, which corresponds to particle–hole excitations between the two van-Hove singularities in the single-particle spectrum. The effective interaction is predominately governed by the particleâĂŞ-hole continuum, which for small momenta scales linearly with momentum. The latter is confirmed by the momentum-dependent bosonic correlator Im$[W_{\tilde{k}}(\omega)]$, see Fig. 11(c). At larger momenta, a deviation from the linear dependence is evident, and close to the edge of the BZ the intensity of the bosonic propagator is maximal as it corresponds to the transition between the two van-Hove singularities in the single particle spectrum.

Now, we turn to the dynamics after the photo-excitation. The system is excited with a short oscillating electric pulse, see Eq. (77), with a single cycle $N_p = 1$. The amplitude of the excitation E_0 determines the absorbed energy. In Fig. 12, we present the time evolution of the kinetic energy for the two excitation strengths $E_0 = 3$ and $E_0 = 5$. As the energy is increased (during the pulse) and the system heats up, the kinetic energy increases. The observed behavior is consistent with thermalization at a higher temperature, but the transient evolution is complicated by the energy exchange between the electronic and bosonic subsystems (plasmon emission). For the strongest excitations, there is a clear relaxation dynamics to the final state, see inset of Fig. 12, accompanied with strongly damped oscillations.

In practice, the main bottleneck to reach longer propagation times is the memory restriction imposed by the hardware. The usage of the MPI parallelization scheme over momentum points reduces this issue due to the distribution of memory among different nodes. This is beneficial as long as the number of momentum points is an integer multiple of the number of cores. The usage of the `distributed_timestep_array` enables a minimal overlap of the stored information between different nodes, which in all practical cases leads to a linear reduction of the memory requirements per MPI rank.

Moreover, the MPI parallelization also speeds up the execution of the program. We have performed a scaling analysis for a system with fixed number of momentum points $N_k = 128$, and parallelization up to 128 processors, see Fig. 13. Moreover, for all tests we have fixed the number of tasks per node to one, since in the real-world scenario we want to maximally distribute the memory. We can see that the scaling is almost perfect up to 128 processors, where a slight deviation from optimal scaling is observed. The main reason for this behavior is the communication overhead, since a substantial amount of data, namely timesteps of propagators for all momentum points, has to be communicated among all nodes. We have tested different communication schemes and the current versions of the library includes the scheme with the best scaling. Of course, we cannot exclude that the scaling for a large number of processors can be improved and this will be an important task for a future update of the library. While the current version can be directly applied to higher dimensional systems (2D, 3D), future applications to the realistic modelling of solids will rely on an efficient parallelization scheme.

Part II. Numerical implementation

In this part, we present the details of the numerical Implementation.

8. Basic integration and differentiation rules

In Sections 9–13 we describe the numerics underlying the at least kth-order accurate solution of the `dyson`, `vie2`, and `convolution` equations in detail. In this section we define, as the first step, the basic notation for polynomial interpolation as well as approximate relations for evaluating differentials (backward differentiation) and integrals (Gregory quadrature rules).

8.1. Polynomial interpolation

Consider a function $y(t)$ which takes the values y_j at the points $t_j = jh$ of an equidistant mesh $j = 0, 1 with time step h, \dots, k$. We denote the kth-order polynomial $\tilde{y}(t)$ passing through the points $y(jh) = y_j$,

$$\tilde{y}(jh) = y_j, \quad j = 0, \dots, k, \tag{78}$$

by $\mathcal{P}^{(k)}[y_0, \dots, y_k](t)$. The interpolation can be cast into the matrix form,

$$\mathcal{P}^{(k)}[y_0, \dots, y_k](t) = \sum_{a,l=0}^{k} h^{-a} t^a P^{(k)}_{al} y_l, \tag{79}$$

$$P^{(k)}_{al} = (M^{-1})_{al} \text{ with } M_{ja} = j^a. \tag{80}$$

With Eqs. (79) and (80), Eq. (78) can be verified directly,

$$\tilde{y}(jh) = \sum_{a,l=0}^{k} j^a P^{(k)}_{al} y_l = \sum_{a,l=0}^{k} M_{ja} (M^{-1})_{al} y_l = y_j. \tag{81}$$

The precomputed weights $P^{(k)}_{al}$ can be obtained from the `integrator` class (see Section 5).

8.2. Polynomial differentiation

An approximation for the derivative dy/dt of a function can be obtained by taking the exact derivative of the polynomial approximant (79),

$$\left.\frac{dy}{dt}\right|_{t=mh} \approx \frac{d}{dt} \mathcal{P}^{(k)}[y_0, \dots, y_k](mh) = \sum_{a=1}^{k} \sum_{l=0}^{k} P^{(k)}_{al} h^{-a} a (mh)^{a-1} y_l \tag{82}$$

$$= h^{-1} \sum_{a=1}^{k} \sum_{l=0}^{k} P^{(k)}_{al} a m^{a-1} y_l. \tag{83}$$

We thus arrive at an approximative relation for *polynomial differentiation*

$$\left.\frac{dy}{dt}\right|_{t=mh} \approx h^{-1} \sum_{l=0}^{k} D^{(k)}_{ml} y_l, \quad \text{with} \tag{84}$$

$$D^{(k)}_{m,l} = \sum_{a=1}^{k} P^{(k)}_{al} a m^{a-1}. \tag{85}$$

The precomputed weights $D^{(k)}_{m,l}$ are stored by the `integrator` class (see Section 5).

8.3. Polynomial integration

In some cases below, the polynomial interpolation formula is also used to get the approximation to an integral. For $0 \le m \le n \le k$,

$$\int_{mh}^{nh} dt\, y(t) \approx \int_{mh}^{nh} dt\, \mathcal{P}^{(k)}[y_0, \dots, y_k](t) = \sum_{a=0}^{k} \sum_{l=0}^{k} \int_{mh}^{nh} dt P^{(k)}_{al} h^{-a} t^a y_l \tag{86}$$

$$= h \sum_{l=0}^{k} \left[\sum_{a=0}^{k} P^{(k)}_{al} \int_{m}^{n} dt\, t^a \right] y_l. \tag{87}$$

Table 8
Weights of the backward differentiation formula (90) up to $k = 6$.

k	a_0	a_1	a_2	a_3	a_4	a_5	a_6
1	1	-1					
2	$\frac{3}{2}$	-2	$\frac{1}{2}$				
3	$\frac{11}{6}$	-3	$\frac{3}{2}$	$-\frac{1}{3}$			
4	$\frac{25}{12}$	-4	$\frac{6}{2}$	$-\frac{4}{3}$	$\frac{1}{4}$		
5	$\frac{137}{60}$	-5	$\frac{10}{2}$	$-\frac{10}{3}$	$\frac{5}{4}$	$-\frac{1}{5}$	
6	$\frac{49}{20}$	-6	$\frac{15}{2}$	$-\frac{20}{3}$	$\frac{15}{4}$	$-\frac{6}{5}$	$\frac{1}{6}$

We thus use the following approximate relation for *polynomial integration*

$$\int_{mh}^{nh} dt\, y(t) \approx h \sum_{l=0}^{k} I^{(k)}_{m,n,l} y_l, \quad \text{with} \tag{88}$$

$$I^{(k)}_{m,n,l} = \sum_{a=0}^{k} P^{(k)}_{al} \frac{n^{a+1} - m^{a+1}}{a+1}. \tag{89}$$

The precomputed weights $I^{(k)}_{m,n,l}$ are implemented in the `integrator` class (see Section 5).

8.4. Backward differentiation

Consider a function $y(t)$ which takes the values y_j at the points $t_j = jh$ of an equidistant mesh $j = 0, 1, \dots, n$, with $n \geq k$. The backward differentiation formula (BDF) of order k approximates the derivative dy/dt at $t = nh$ using the function values $y_n, y_{n-1}, \dots, y_{n-k}$. It is defined via the linear relation

$$\left.\frac{dy}{dt}\right|_{nh} \approx h^{-1} \sum_{j=0}^{k} a^{(k)}_j y_{n-j}. \tag{90}$$

Here the coefficients for the kth order formula are obtained by the derivative $\tilde{y}'(t = 0)$ of the kth order polynomial interpolation $\tilde{y}(t)$ defined by the values $\tilde{y}(jh) = y_{n-j}$ (backward differentiation is thus a special case of the polynomial differentiation),

$$\left.\frac{dy}{dt}\right|_{nh} \approx -\frac{d}{dt} \mathcal{P}^{(k)}[y_n, y_{n-1}, \dots, y_{n-k}](t = 0). \tag{91}$$

Note that the minus sign is due to the reversed order of the interpolated points. Therefore, the coefficients of the BDF are directly related to coefficients for polynomial differentiation: $a^{(k)}_j = -D^{(k)}_{0,j}$. The coefficients for the first k are tabulated in Table 8. The precomputed weights $a^{(k)}_j$ can be obtained from the `integrator` class (see Section 5).

8.5. Gregory integration

The solution of Volterra integral equations (VIEs) discussed below is based on a combination of backward-differentiation formulae with so-called Gregory quadrature rules for the integration. The kth Gregory quadrature rule on a linear mesh is defined by the equation

$$\mathcal{I}_n \equiv \int_0^{nh} dt\, y(t) \approx h \sum_{j=0}^{m(n,k)} w^{(k)}_{n,j} y_j, \quad m(n,k) = \begin{cases} n & n > k \\ k & n \leq k. \end{cases} \tag{92}$$

The weights $w^{(k)}_{n,j}$ are explained below. In general, the approximation for the integral is obtained from function values $\{y_j : 0 \leq j \leq n\}$ *within* the integration interval $[0, nh]$, i.e. $m(n, k) = n$. However, this is not possible for $n < k$, because a kth order accurate quadrature rule cannot be constructed from less than $k + 1$ function values. In the Gregory quadrature for $n < k$, we assume that the function $y(t)$ exists outside the interval $[0, nh]$, and construct an approximation for the integral from values $\{y_j : 0 \leq j \leq k\}$, i.e., $m(n, k) = k$.

The simplest example of a Gregory quadrature rule is the trapezoidal approximation,

$$\int_0^{nh} dt\, y(t) \approx h\Big(\tfrac{1}{2} y_0 + y_1 + \dots + y_{n-1} + \tfrac{1}{2} y_n\Big), \tag{93}$$

which corresponds to $k = 0$, $m(n, k) = n$, and the weights $w^{(k)}_{n,j} = \frac{1}{2}$ for $j \in \{0, n\}$ and $w^{(k)}_{n,j} = 1$ for $0 < j < n$. The weights $w^{(k)}_{n,j}$ for a general kth order accurate rule are implicitly defined by the following procedure:

- $n \leq k$: $\mathcal{I}_n$ is approximated by the exact integral over the polynomial interpolation $\mathcal{P}^{(k)}[y_0, \dots, y_k](t)$,

$$\mathcal{I}_n \approx \int_0^{nh} dt\, \mathcal{P}^{(k)}[y_0, \dots, y_k](t) \equiv h \sum_{j=0}^{k} s^{(k)}_{n,j} y_j. \tag{94}$$

Table 9
Weights of the first few Gregory integration rules, Eq. (92). In the table for each k, the numbers in the left (right) $(k+1)\times(k+1)$ block define the weights $s_{l,j}^{(k)}$ ($\Sigma_{l,j}^{(k)}$), respectively, as shown in the last table. The ω weights can be read off from the last row of the Σ weights, $\omega_j^{(k)} = \Sigma_{k,j}^{(k)}$, $j = 0, \ldots, k$.

$k=0$:						
			0			$\frac{1}{2}$
$k=1$:		0	0		$\frac{5}{12}$	$\frac{7}{6}$
		$\frac{1}{2}$	$\frac{1}{2}$		$\frac{5}{12}$	$\frac{13}{12}$
$k=2$:	0	0	0	$\frac{3}{8}$	$\frac{9}{8}$	$\frac{9}{8}$
	$\frac{5}{12}$	$\frac{2}{3}$	$-\frac{1}{12}$	$\frac{3}{8}$	$\frac{7}{6}$	$\frac{11}{12}$
	$\frac{1}{3}$	$\frac{4}{3}$	$\frac{1}{3}$	$\frac{3}{8}$	$\frac{7}{6}$	$\frac{23}{24}$
$s_{0,0}^{(k)}$	$\cdots$	$s_{0,k}^{(k)}$	$\Sigma_{0,0}^{(k)}$	$\cdots$	$\Sigma_{0,k}^{(k)}$	
$\vdots$		$\vdots$	$\vdots$		$\vdots$	
$s_{k,0}^{(k)}$	$\cdots$	$s_{k,k}^{(k)}$	$\Sigma_{k,0}^{(k)} = \omega_0^{(k)}$	$\cdots$	$\Sigma_{k,k}^{(k)} = \omega_k^{(k)}$	

Hence the weights for $n \le k$, which are denoted as *starting weights* $w_{n,j}^{(k)} = s_{n,j}^{(k)}$, are equivalent to the polynomial integration weights (89),

$$w_{n,j}^{(k)} = I_{0,n,j}^{(k)} \equiv s_{n,j}^{(k)}\,, \quad 0 \le n \le k. \tag{95}$$

- $n > k$: To generate an approximation for the integral $\mathcal{I}(t) = \int_0^t d\bar{t}\, y(\bar{t})$ at $t = nh$ and $n > k$, we consider the differential equation

$$\frac{d}{dt}\mathcal{I}(t) = y(t), \quad \mathcal{I}(0) = 0. \tag{96}$$

This equation is solved by taking the values $\mathcal{I}_j$ for $0 \le j \le k$ from the approximation (94), and solving for $\mathcal{I}_n$ at $n > k$ by applying the BDF . The resulting set of linear equations for $\mathcal{I}_n$ with $n > k$,

$$h^{-1}\sum_{l=0}^{k} a_l^{(k)} \mathcal{I}_{m-l} = y_m, \quad m = k+1, \ldots, n, \tag{97}$$

implicitly determines the values of the integral.

This procedure defines a weight matrix with the following structure:

$$w_{n,j}^{(k)} = \left(\begin{array}{cccccccccc}
s_{00}^{(k)} & \cdots & s_{0k}^{(k)} & 0 & 0 & & & & & \\
\vdots & & \vdots & & & & & & & \\
s_{k0}^{(k)} & \cdots & s_{kk}^{(k)} & 0 & 0 & & & & & \\
\Sigma_{00}^{(k)} & \cdots & \Sigma_{0k}^{(k)} & \omega_0^{(k)} & 0 & & & & & \\
\vdots & & \vdots & & \ddots & \ddots & & & & \\
\Sigma_{k0}^{(k)} & \cdots & \Sigma_{kk}^{(k)} & \omega_k^{(k)} & \cdots & \omega_0^{(k)} & 0 & & & \\
\omega_0^{(k)} & \cdots & \omega_k^{(k)} & 1 & \omega_k^{(k)} & \cdots & \omega_0^{(k)} & 0 & & \\
\omega_0^{(k)} & \cdots & \omega_k^{(k)} & 1 & 1 & \omega_k^{(k)} & \cdots & \omega_0^{(k)} & 0 & \\
\omega_0^{(k)} & \cdots & \omega_k^{(k)} & 1 & \cdots & 1 & \omega_k^{(k)} & \cdots & \omega_0^{(k)} & 0
\end{array}\right). \tag{98}$$

Here the upper block contains the weights obtained from the polynomial approximation. For $n > k$ we have $m(n, k) = n$. For $n \ge 2k+1$ the weights are symmetric $w_{n,j}^k = w_{n,n-j}^k \equiv \omega_j^{(k)}$ for $j \le k$. For $n \ge 2k+1$ the weights satisfy $w_{n,n-j-1}^{(k)} \equiv \omega_j^{(k)}$ for $j \le k$, and furthermore $w_{n,j}^{(k)} = 1$ for $k < j < n-k-1$. The latter property makes this quadrature rule different from, e. g., the Simpson rule, where the weights alternate between $\frac{2}{3}$ and $\frac{4}{3}$ but never become one. Gregory quadrature rules for $n > 2k+1$ can thus be understood as a simple Riemann sum $\mathcal{I}_n \approx h\sum_{j=0}^{n} y_j$ with a *boundary correction* obtained from the function values $\{y_j, y_{n-j} : 0 \le j \le k\}$, thus generalizing the structure of the trapezoidal rule (93). For completeness, the integration rules for some of the lowest k are presented in Table 9. The weights for $k = 1, \ldots, 5$ can be obtained from the `integrator` class (see Section 5).

The advantage of the Gregory integration is a uniform approximation of $\mathcal{I}_n$: for any n, the error scales as $\mathcal{O}(h^{k+2})$ [36]. This is different from Newton–Cotes rules of the same order k, which are only kth order accurate for a certain number of grid points. For instance, the Simpson rule requires an odd number of grid points. The accuracy of the Gregory integration is illustrated in Fig. 14 for the integral $y(x) = \exp(ix)$ with exact integral $\int_0^x dx' y(x') = -i(\exp(ix) - 1)$. In particular, the panel on the right-hand side of Fig. 14 confirms that the average absolute error $(1/N)\sum_{n=0}^{N} |\mathcal{I}_n - \mathcal{I}_n^{\mathrm{ex}}|$ as a function of the number of points N scales as $\mathcal{O}(N^{-p})$ with $p \approx k+2$.[4] We also compare to the Simpson's rule, employing the trapezoidal rule for integrating over $[nh, (n-1)h]$ if n is odd (the total number of points $n+1$ is even). As Fig. 14 shows, this primitive extension of Simpson's rule induces oscillatory behavior of the error, as the accuracy is hampered by the trapezoidal rule. Thus, the scaling of the averaged error is effectively reduced to first order ($k = 1$).

[4] In the solution of the Volterra integral equations, discussed in Section 9, the overall error depends not only on the accuracy of this quadrature rule, but also on the start-up procedure and the differential operator.

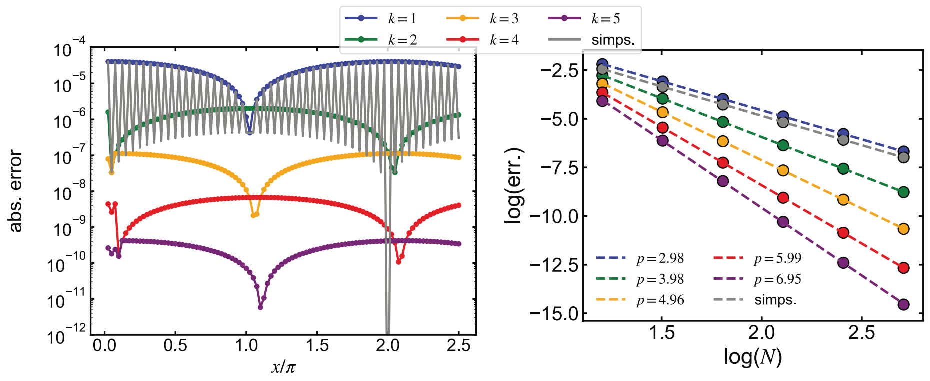


Fig. 14. Left panel: absolute error $|\mathcal{I}_n - \mathcal{I}_n^{\text{ex}}|$ of the Gregory integration for different orders k. Right panel: mean absolute error for integrating up to $x_{\max} = 5\pi/2$, discretizing the interval into N points with $h = 0.025\pi$. The dashed lines are linear fits confirming the order of the Gregory quadrature.

Furthermore, the construction of the Gregory weights makes computing $\mathcal{I}(t)$ by Gregory quadrature and solving the corresponding differential equation by the BDF method numerically equivalent, ensuring consistency of the integral or differential formulation.

8.6. Boundary convolution

In this paragraph, we introduce a kth-order accurate approximation for a special kind of convolution integral, which appears in the context of imaginary time convolutions. Consider the convolution integral

$$c(t) = \int_0^t dt' F(t-t')G(t') \tag{99}$$

between two functions F and G which are only defined for $t > 0$, and cannot be continued into a differentiable function on the domain $t < 0$. On an equidistant mesh with $t = mh$ and $m < k$, the integration range includes less than k points, and the functions must be continued outside the integration range in order to obtain an kth-order accurate approximation. Because of the structure of the convolution integral, $F(t-t')$ can only be continued to the domain $t' < 0$, while $G(t')$ should be continued to the domain $t' > t$.

We use the approximation

$$c(mh) = \int_0^{mh} dt' \mathcal{P}^{(k)}[F_0, \ldots, F_k](mh - t') \mathcal{P}^{(k)}[G_0, \ldots, G_k](t'). \tag{100}$$

Using Eq. (80), this can be transformed into

$$c(mh) = \sum_{r,s,a,b=0}^{k} \int_0^{mh} dt' P^{(k)}_{a,r} h^{-a} (mh-t')^a F_r P^{(k)}_{b,s} h^{-b} (t')^b G_s \tag{101}$$

$$= \sum_{r,s=0}^{k} F_r G_s \Big[\sum_{a,b=0}^{k} \int_0^{mh} dt' P^{(k)}_{a,r} h^{-a} (mh-t')^a P^{(k)}_{b,s} h^{-b} (t')^b \Big]. \tag{102}$$

The terms in brackets are coefficients which can be precomputed, so that finally

$$\int_0^{mh} dt' F(t-t')G(t') = h \sum_{r,s=0}^{k} F_r G_s R^{(k)}_{m;r,s}, \tag{103}$$

$$R^{(k)}_{m;r,s} = \sum_{a,b=0}^{k} P^{(k)}_{a,r} P^{(k)}_{b,s} \int_0^m dx (m-x)^a x^b. \tag{104}$$

The precomputed weights $R^{(k)}_{m;r,s}$ can be obtained from the `integrator` class.

9. Numerical details: Volterra integral equations

In this section, we present kth-order discrete approximations for various contour convolution integrals which appear in the solution of the `dyson`, `vie2`, and `convolution` problems. The integrals constitute different contributions to the convolution (41), which we separate into the Matsubara, retarded, mixed or lesser components of a contour function C. All equations are obtained in a straightforward way from the Gregory integration (92) if the integration interval includes more than $k+1$ function values, and from the polynomial integration (89) or the boundary convolution (103) otherwise.

Below, we indicate by the label $\stackrel{c}{=}$ those equations which are *exactly* causal, i. e., the result C at real time arguments $\leq nh$ does not depend on the input functions A, f, B with (one or both) real time arguments larger than n. For the other equations, causality is satisfied only up to the numerical accuracy. Furthermore, by adding a tilde $\tilde{A}$, $\tilde{B}$ over the input functions in an equation we indicate that some of the input values of A and B lie outside the domain of the `herm_matrix` type and must be obtained from the hermitian conjugates $A^{\ddagger}$ and $B^{\ddagger}$, respectively (see Section 3).

Matsubara.

$$C_1^{\mathrm{M}}[A,f,B](m) = \int_0^{mh_\tau} d\tau' A^{\mathrm{M}}(mh_\tau - \tau')f(0^-)B^{\mathrm{M}}(\tau') = \tag{105}$$

$$\begin{cases} \overset{c}{=} h_\tau \sum_{j,l=0}^{k} R^{(k)}_{m;j,l} A^{\mathrm{M}}_j f_{-1} B^{\mathrm{M}}_l & m \le k \\ \overset{c}{=} h_\tau \sum_{l=0}^{m} w^{(k)}_{m,l} A^{\mathrm{M}}_{m-l} f_{-1} B^{\mathrm{M}}_l & m > k. \end{cases} \tag{106}$$

$$C_2^{\mathrm{M}}[A,f,B](m) = \int_{mh_\tau}^{\beta} d\tau' A^{\mathrm{M}}(mh_\tau - \tau')f(0^-)B^{\mathrm{M}}(\tau') = \tag{107}$$

$$\begin{cases} \overset{c}{=} h_\tau \sum_{j,l=0}^{k} R^{(k)}_{N_\tau-m;j,l} \xi A^{\mathrm{M}}_{N_\tau-j} f_{-1} B^{\mathrm{M}}_{N_\tau-l} & m \ge N_\tau - k \\ \overset{c}{=} h_\tau \sum_{l=0}^{N_\tau-m} w^{(k)}_{N_\tau-m,l} \xi A^{\mathrm{M}}_{N_\tau-l} f_{-1} B^{\mathrm{M}}_{m+l} & m < N_\tau - k. \end{cases} \tag{108}$$

In the second equation $A^{\mathrm{M}}(\tau)$ at the values $\tau \in [-\beta, 0]$ is obtained by using the periodicity property $A^{\mathrm{M}}(\tau+\beta) = \xi A^{\mathrm{M}}(\tau)$.

Retarded.

$$C^{\mathrm{R}}[A,f,B](n,m) = \int_{mh}^{nh} d\bar{t} A^{\mathrm{R}}(nh,\bar{t})f(\bar{t})B^{\mathrm{R}}(\bar{t},mh) = \tag{109}$$

$$\begin{cases} \overset{c}{=} h \sum_{j=m}^{n} w^{(k)}_{n-m,j-m} A^{\mathrm{R}}_{n,j} f_j B^{\mathrm{R}}_{j,m} & n > k, n-m > k \\ \overset{c}{=} h \sum_{j=0}^{k} w^{(k)}_{n-m,j} A^{\mathrm{R}}_{n,n-j} f_{n-j} \tilde{B}^{\mathrm{R}}_{n-j,m} & n > k, n-m \le k \\ = h \sum_{j=0}^{k} I^{(k)}_{m,n;j} \tilde{A}^{\mathrm{R}}_{n,j} f_j \tilde{B}^{\mathrm{R}}_{j,m} & n \le k. \end{cases} \tag{110}$$

As mentioned above, the tilde $\tilde{B}^{\mathrm{R}}_{j,m}$ in the third equation in Eq. (110) indicates that $B^{\mathrm{R}}_{j,m}$ is also evaluated outside the domain $j \ge m$ of the `herm_matrix` type, and thus needs to be reconstructed from $B^\ddagger$, i.e., $\tilde{B}^{\mathrm{R}}_{j,m} = B^{\mathrm{R}}_{j,m} = -(B^\ddagger)^{\mathrm{R}}_{m,j}$. Analogous definitions hold for $\tilde{A}^{\mathrm{R}}_{n,j}$ in the third equation, and $\tilde{B}^{\mathrm{R}}_{n-j,m}$ in the second equation.

Left-mixing components.

$$C_1^{\rceil}[A,f,B](n,m) = \int_0^{nh} d\bar{t} A^{\mathrm{R}}(nh,\bar{t})f(\bar{t})B^{\rceil}(\bar{t},mh_\tau) \tag{111}$$

$$\begin{cases} \overset{c}{=} h \sum_{j=0}^{n} w^{(k)}_{n,j} A^{\mathrm{R}}_{n,j} f_j B^{\rceil}_{j,m} & n > k, \\ = h \sum_{j=0}^{k} w^{(k)}_{n,j} \tilde{A}^{\mathrm{R}}_{n,j} f_j B^{\rceil}_{j,m} & n \le k, \end{cases} \tag{112}$$

$$C_2^{\rceil}[A,f,B](n,m) = \int_0^{mh_\tau} d\tau A^{\rceil}(nh,\tau')f(0^-)B^{\mathrm{M}}(\tau' - mh_\tau) \tag{113}$$

$$\begin{cases} \overset{c}{=} h_\tau \sum_{j,l=0}^{k} R^{(k)}_{m;j,l} A^{\rceil}_l f_{-1} \xi B^{\mathrm{M}}_{N_\tau-j} & m \le k \\ \overset{c}{=} h_\tau \sum_{l=0}^{m} w^{(k)}_{m,l} A^{\rceil}_{m-l} f_{-1} \xi B^{\mathrm{M}}_{N_\tau-l} & m > k, \end{cases} \tag{114}$$

$$C_3^{\rceil}[A,f,B](n,m) = \int_{mh_\tau}^{\beta} d\tau A^{\rceil}(nh,\tau')f(0^-)B^{\mathrm{M}}(\tau' - mh_\tau) \tag{115}$$

$$\begin{cases} \overset{c}{=} h_\tau \sum_{j,l=0}^{k} R^{(k)}_{N_\tau-m;j,l} A^{\rceil}_{N_\tau-l} f_{-1} B^{\mathrm{M}}_j & m \ge N_\tau - k \\ \overset{c}{=} h_\tau \sum_{l=0}^{N_\tau-m} w^{(k)}_{N_\tau-m,l} A^{\rceil}_{m+l} f_{-1} B^{\mathrm{M}}_l & m < N_\tau - k. \end{cases} \tag{116}$$

Lesser components $n \le m$.

$$C_1^{<}[A,f,B](n,m) = \int_0^{nh} d\bar{t} A^{\mathrm{R}}(nh,\bar{t})f(\bar{t})B^{<}(\bar{t},mh) \tag{117}$$

$$\begin{cases} \overset{c}{=} h \sum_{j=0}^{n} w^{(k)}_{n,j} A^{\mathrm{R}}_{n,j} f_j B^{<}_{j,m} & n > k, \\ = h \sum_{j=0}^{k} w^{(k)}_{n,j} \tilde{A}^{\mathrm{R}}_{n,j} f_j B^{<}_{j,m} & n \le k. \end{cases} \tag{118}$$

$$C_2^{<}[A,f,B](n,m) = \int_0^{mh} d\bar{t} A^{<}(nh,\bar{t})f(\bar{t})B^{\mathrm{A}}(\bar{t},mh) = \tag{119}$$

$$\begin{cases} \overset{c}{=} h \sum_{j=0}^{m} w^{(k)}_{m,j} A^{<}_{n,j} f_j B^{\mathrm{A}}_{j,m} & m > k, \\ = h \sum_{j=0}^{k} w^{(k)}_{m,j} A^{<}_{n,j} f_j \tilde{B}^{\mathrm{A}}_{j,m} & m \le k. \end{cases} \tag{120}$$

$$C_3^{<}[A,f,B](n,m) = -i \int_0^{\beta} d\tau A^{\rceil}(nh,\tau')f(0^-)B^{\lceil}(\tau,mh) = \tag{121}$$

$$= -ih_\tau \sum_{j=0}^{N_\tau} w^{(k)}_{N_\tau,j} A^{\rceil}_{n,j} f_{-1} B^{\lceil}_{j,m}. \tag{122}$$

Because the advanced and right-mixing components are not stored by the `herm_matrix` type, these quantities must be reconstructed from the hermitian conjugate. For example, $B^{\lceil}_{j,m} = -\xi [B^\ddagger]^{\rceil}_{m,j}$ in the third equation, and $B^{\mathrm{A}}_{j,m} = [B^\ddagger]^{\mathrm{R}}_{j,m}$ in the second equation.

10. Numerical details: convolution integrals on $\mathcal{C}$

The numerical solution of the `vie2` and `dyson` integral equations is based on a mapping of these equations onto a set of coupled VIEs or Volterra integro-differential equations (VIDEs). In this section, we first explain kth-order accurate algorithms for the solution of Volterra equations. These algorithms are discussed in detail in the book by Brunner and van Houven [37].

10.1. *Volterra Integro-differential equation*

We consider a Volterra integro-differential equation (VIDE) of the form

$$\frac{dy}{dt} + p(t)y(t) + \int_0^t ds\, k(t,s)y(s) = q(t). \tag{123}$$

For given $k(t,s)$, $p(t)$, and $q(t)$ and an *initial condition* specifying $y(0)$, this equation must be solved to determine $y(t)$ in the domain $t > 0$. In the numerical solution all functions are known or determined on an equidistant mesh $t_j = jh$, $j = 0, 1, 2, \ldots$, and we use the notation $k_{j,l} = k(jh, lh)$, $p_l = p(lh)$, etc. Here and in the following, the values of the functions k, p, q, y can be complex matrices of size $d > 1$.

The at least kth-order accurate solution of this equation is obtained by combining the kth-order Gregory quadrature in two steps:

(1) *Start-up (bootstrapping):* A procedure which is used to obtain a solution y_j for $j = 1, \ldots, k$.
(2) *Time-stepping:* A procedure to obtain y_n for $n > k$ from $\{y_j : j < n\}$.

In the algorithm explained below, the time-stepping is causal, i.e., the solution y_n does not depend on the input $k_{l,j}$, p_j, q_j at $l > n$ or $j > n$. For the start-up, the numerical error at y_n for $n < k$ can depend on the input $k_{l,j}$, p_j, q_j at $0 \le l, j \le k$. Furthermore, in the numerical implementation we assume that the kernel $k(t,t')$ can be defined as a differentiable function on the whole domain $0 \le t, t'$, although only the values at $0 \le t' \le t$ enter the exact integral.

10.1.1. *Start-up procedure*

For the start-up (bootstrapping) procedure we express the derivative in (123) in terms of the polynomial differentiation equations (84) and (85), and use the Gregory integration (92) for the convolution

$$h^{-1}\sum_{l=0}^{k} D^{(k)}_{n,l} y_l + p_n y_n + h\sum_{l=0}^{k} w^{(k)}_{n,l}\, k_{n,l} y_l = q_n \text{ for } n = 1, \ldots, k. \tag{124}$$

This defines a linear equation

$$\begin{pmatrix} M_{1,1} & \cdots & M_{1,k} \\ \vdots & & \vdots \\ M_{k,1} & \cdots & M_{k,k} \end{pmatrix}\begin{pmatrix} y_1 \\ \vdots \\ y_k \end{pmatrix} = \begin{pmatrix} q_1 - M_{1,0}y_0 \\ \vdots \\ q_n - M_{k,0}y_0 \end{pmatrix}, \tag{125}$$

where the Matrix M is given by

$$M_{n,l} = h^{-1}D^{(k)}_{n,l} + \delta_{n,l}p_n + hw^{(k)}_{n,l}\, k_{n,l}. \tag{126}$$

This $k \times k$ dimensional linear equation is solved directly. Note that when y, k, p, and q are d-dimensional matrices, the solution of the linear system amounts to inverting a matrix of size $(kd) \times (kd)$.

10.1.2. *Time-stepping*

The time-stepping is done using a combination of backward differentiation (90) and Gregory integration (92)

$$h^{-1}\sum_{l=0}^{k} a^{(k)}_l y_{n-l} + p_n y_n + h\sum_{l=0}^{n} w^{(k)}_{n,l} k_{n,l} y_l = q_n. \tag{127}$$

If the y_j are known for $j < n$ we obtain a linear equation for y_n,

$$\Big[h^{-1}a^{(k)}_0 + p_n + hw^{(k)}_{n,n}k_{n,n}\Big]y_n = \Big[q_n - h^{-1}\sum_{l=1}^{k} a^{(k)}_l y_{n-l} - h\sum_{l=0}^{n-1} w^{(k)}_{n,l}k_{n,l}y_l\Big], \tag{128}$$

which is solved for y_n.

10.1.3. *Conjugate equation*

For later convenience we also define the start-up and time-stepping relations to solve an equivalent conjugate equation

$$\frac{dy}{dt} + y(t)p(t) + \int_0^t ds\, y(s)k(s,t) = q(t). \tag{129}$$

The start-up determines the values $y_1, \ldots, y_k$ by solving the $k \times k$ linear equation

$$\sum_{l=1}^{k} y_l M_{l,n} = q_n - y_0 M_{0,n}, \quad 1 \le n \le k, \tag{130}$$

where the Matrix M is given by

$$M_{l,n} = h^{-1}D^{(k)}_{n,l} + \delta_{n,l}p_n + hw^{(k)}_{n,l}\,k_{l,n}. \tag{131}$$

In the time-stepping, y_n for $n > k$ is determined by solving the linear equation

$$y_n\Big[h^{-1}a^{(k)}_0 + p_n + hw^{(k)}_{n,n}k_{n,n}\Big] = \Big[q_n - h^{-1}\sum_{l=1}^{k} a^{(k)}_l y_{n-l} - h\sum_{l=0}^{n-1} w^{(k)}_{n,l}y_l k_{l,n}\Big]. \tag{132}$$

10.2. Volterra Integral equation of the second kind

Small modifications of Eqs. (123) and (129) lead to the VIEs of the second kind

$$y(t) + \int_0^t ds\, k(t,s)y(s) = q(t), \tag{133}$$

$$y(t) + \int_0^t ds\, y(s)k(s,t) = q(t), \tag{134}$$

with the same assumptions on the domain and the kernel. These equations are solved with the initial condition $y_0 = q_0$.

Start-up and time-stepping procedures are obtained from the integro-differential equation by setting $p(t) = 1$ and omitting the differential: The start-up procedure for Eq. (133) determines the values $y_1, \dots, y_k$ by solving the $k \times k$ linear equation

$$\sum_{l=1}^{k} M_{n,l}y_l = q_n - M_{n,0}y_0, \quad 1 \le n \le k, \tag{135}$$

where the Matrix M is given by

$$M_{n,l} = \delta_{n,l} + hw^{(k)}_{n,l}\,k_{n,l}. \tag{136}$$

In the time-stepping, y_n for $n > k$ is determined by solving the linear equation

$$\Big[1 + hw^{(k)}_{n,n}k_{n,n}\Big]y_n = \Big[q_n - h\sum_{l=0}^{n-1} w^{(k)}_{n,l}k_{n,l}y_l\Big]. \tag{137}$$

The start-up procedure for the conjugate equation (134) determines the values $y_1, \dots, y_k$ by solving the $k \times k$ linear equation

$$\sum_{l=1}^{k} y_l M_{l,n} = q_n - y_0 M_{0,n}, \quad 1 \le n \le k, \tag{138}$$

where the Matrix M is given by

$$M_{l,n} = \delta_{n,l} + hw^{(k)}_{n,l}\,k_{l,n}. \tag{139}$$

In the time-stepping, y_n for $n > k$ is determined by solving the linear equation

$$y_n\Big[1 + hw^{(k)}_{n,n}k_{n,n}\Big] = \Big[q_n - h\sum_{l=0}^{n-1} w^{(k)}_{n,l}y_l k_{l,n}\Big]. \tag{140}$$

11. Implementation: `convolution`

11.1. Langreth rules

In this section, we present the implementation of the `convolution` routine which solves Eq. (41). Using the Langreth rules, the convolution integral (41) is split into contributions from the Matsubara, retarded, left-mixing, and lesser components

$$C^{\mathrm{M}}(\tau) = \int_0^\beta d\tau' A^{\mathrm{M}}(\tau - \tau')f(0^-)B^{\mathrm{M}}(\tau'), \tag{141}$$

$$C^{\mathrm{R}}(t,t') = \int_{t'}^{t} d\bar{t}A^{\mathrm{R}}(t,\bar{t})f(\bar{t})B^{\mathrm{R}}(\bar{t},t'), \tag{142}$$

$$\begin{aligned} C^{\rceil}(t,\tau) = &\int_0^t d\bar{t}A^{\mathrm{R}}(t,\bar{t})f(\bar{t})B^{\rceil}(\bar{t},\tau) \\ &+ \int_0^\beta d\tau A^{\rceil}(t,\tau')f(0^-)B^{\mathrm{M}}(\tau' - \tau), \end{aligned} \tag{143}$$

$$\begin{aligned} C^{<}(t,t') = &\int_0^t d\bar{t}A^{\mathrm{R}}(t,\bar{t})f(\bar{t})B^{<}(\bar{t},t') + \int_0^{t'} d\bar{t}A^{<}(t,\bar{t})f(\bar{t})B^{\mathrm{A}}(\bar{t},t') \\ &- i\int_0^\beta d\tau A^{\rceil}(t,\tau')f(0^-)B^{\lceil}(\tau,t'). \end{aligned} \tag{144}$$

kth-order approximations to these individual components have been presented in Section 10.

It is also convenient to introduce the convolution of a two-time contour object with a function as

$$c(t) = \int_{\mathcal{C}} d\bar{t} A(t,\bar{t}) f(\bar{t}) \,. \tag{145}$$

Eqs. (141)–(144) can be adapted to this case by replacing $B(t,t')$ by the identity function.

11.2. Matsubara

The evaluation of $C^{\mathrm{M}}(\tau)$, i.e., C at timeslice $\mathcal{T}[C]_{-1}$ is implemented as (cf. Eqs. (105) and (107))

$$C^{\mathrm{M}}(mh_\tau) = C_1^{\mathrm{M}}[A,f,B](m) + C_2^{\mathrm{M}}[A,f,B](m) \text{ for } m = 0,\dots,N_\tau\,. \tag{146}$$

11.3. Time steps

The evaluation of C at timeslice $\mathcal{T}[C]_n$ for $n \geq 0$ is implemented as follows:

- For $m = 0,\dots,n$ [cf. Eq. (109)]:

$$C^{\mathrm{R}}(nh, mh) = C_1^{\mathrm{R}}[A,f,B](n,m). \tag{147}$$

- For $m = 0,\dots,N_\tau$ [cf. Eqs. (111), (113), (115)]:

$$\begin{aligned} C^{\rceil}(nh, mh_\tau) = {} & C_1^{\rceil}[A,f,B](n,m) + C_2^{\rceil}[A,f,B](n,m) \\ & + C_3^{\rceil}[A,f,B](n,m). \end{aligned} \tag{148}$$

- For $m = 0,\dots,n$ [cf. Eqs. (117), (119), (121)]:

$$\begin{aligned} C^{<}(mh, nh) = {} & C_1^{<}[A,f,B](m,n) + C_2^{<}[A,f,B](m,n) \\ & + C_3^{<}[A,f,B](m,n). \end{aligned} \tag{149}$$

Comparison with the causal properties of Eqs. (105) to (121) shows that the causal time-dependence indicated in Table 5 is satisfied.

Response convolutions of the type of Eq. (145) are obtained by replacing $B \to 1$ and simplifying the integration formulae in Section 10 accordingly.

12. Implementation: `dyson`

12.1. Langreth rules

In this section, we present the implementation of the `dyson` routine which solves Eq. (34a). To solve Eq. (34a), we again invoke the Langreth rules to split the equation of motion on the KB contour into the respective equations for the Matsubara, lesser, and left-mixing components,

$$-\partial_\tau G^{\mathrm{M}}(\tau) - \epsilon(0^-) G^{\mathrm{M}}(\tau) - \int_0^\beta d\tau'\, \Sigma^{\mathrm{M}}(\tau-\tau') G(\tau') = \delta(\tau), \tag{150}$$

$$i\partial_t G^{\mathrm{R}}(t,t') - \epsilon(t) G^{\mathrm{R}}(t,t') - \int_{t'}^{t} d\bar{t}\, \Sigma^{\mathrm{R}}(t,\bar{t}) G^{\mathrm{R}}(\bar{t},t') = 0 \tag{151}$$

$$\begin{aligned} i\partial_t G^{\rceil}(t,\tau) - \epsilon(t) G^{\rceil}(t,\tau) - \int_0^t d\bar{t}\, \Sigma^{\mathrm{R}}(t,\bar{t}) G^{\rceil}(\bar{t},\tau) \\ = \int_0^\beta d\tau\, \Sigma^{\rceil}(t,\tau') G^{\mathrm{M}}(\tau'-\tau), \end{aligned} \tag{152}$$

$$\begin{aligned} i\partial_t G^{<}(t,t') - \epsilon(t) G^{<}(t,t') - \int_0^t d\bar{t}\, \Sigma^{\mathrm{R}}(t,\bar{t}) G^{<}(\bar{t},t') \\ = \int_0^{t'} d\bar{t}\, \Sigma^{<}(t,\bar{t}) G^{A}(\bar{t},t') - i\int_0^\beta d\tau\, \Sigma^{\rceil}(t,\tau') G^{\lceil}(\tau,t'). \end{aligned} \tag{153}$$

Here Eq. (150) must be solved with the boundary condition

$$G^{\mathrm{M}}(-\tau) = \xi G^{\mathrm{M}}(\beta-\tau), \tag{154}$$

and the remaining equations are solved with the initial conditions

$$G^{\mathrm{R}}(t,t) = -i, \tag{155}$$

$$G^{\rceil}(0,\tau) = iG^{\mathrm{M}}(-\tau) = i\xi G^{\mathrm{M}}(\beta-\tau), \tag{156}$$

$$G^{<}(0,t') = -[G^{\rceil}(t',0)]^\dagger. \tag{157}$$

In the solution of the `dyson` problem, we will use in part the conjugate equation (34b) for the retarded and lesser components. These equations translate into

$$-i\partial_{t'}G^{\mathrm{R}}(t,t')-G^{\mathrm{R}}(t,t')\epsilon(t')-\int_{t'}^{t}d\bar{t}\,G^{\mathrm{R}}(t,\bar{t})\Sigma^{\mathrm{R}}(\bar{t},t')=0, \tag{158}$$

$$\begin{aligned}&-i\partial_{t'}G^{<}(t,t')-G^{<}(t,t')\epsilon(t')-\int_{0}^{t}d\bar{t}\,G^{\mathrm{R}}(t,\bar{t})\Sigma^{<}(\bar{t},t')\\&\quad=\int_{0}^{t'}d\bar{t}\,G^{<}(t,\bar{t})\Sigma^{\mathrm{A}}(\bar{t},t')-i\int_{0}^{\beta}d\tau\,G^{\rceil}(t,\tau')\Sigma^{\lceil}(\tau,t').\end{aligned} \tag{159}$$

which are solved with the initial conditions (155) and

$$G^{<}(t,0)=G^{\rceil}(t,0). \tag{160}$$

12.2. Matsubara

The Matsubara GF is obtained by solving Eq. (150). Unlike the Dyson equations for the real-time and mixed components, Eq. (150) constitutes a boundary-value integro-differential equation.

Fourier series representation. — The (anti-) periodicity $G^{\mathrm{M}}(\tau+\beta)=\xi G^{\mathrm{M}}(\tau)$ allows to express the Matsubara GF by the Fourier series

$$G^{\mathrm{M}}(\tau)=\frac{1}{\beta}\sum_{m=-N_\omega}^{N_\omega}e^{-i\omega_m\tau}G^{\mathrm{M}}(i\omega_m) \tag{161}$$

with $N_\omega\to\infty$, where

$$\omega_m=\begin{cases}\frac{2m\pi}{\beta} & \text{: bosons}\\ \frac{2(m+1)\pi}{\beta} & \text{: fermions}\end{cases} \tag{162}$$

denote the Matsubara frequencies. The Fourier coefficients $G^{\mathrm{M}}(i\omega_m)$ are, in turn, determined by

$$G^{\mathrm{M}}(i\omega_m)=\int_0^\beta d\tau\,G^{\mathrm{M}}(\tau)e^{i\omega_m\tau}\ . \tag{163}$$

Defining the imaginary frequency representation of the self-energy $\Sigma^{\mathrm{M}}(i\omega_m)$ in an analogous fashion, the Dyson equation (150) is transformed into the algebraic equation

$$\left(i\omega_m-\epsilon(0^-)\right)G^{\mathrm{M}}(i\omega_m)=G^{\mathrm{M}}(i\omega_m)\Sigma^{\mathrm{M}}(i\omega_m)\ , \tag{164}$$

which is readily solved for $G^{\mathrm{M}}(i\omega_m)$. Evaluating the Fourier sum (161) then yields $G^{\mathrm{M}}(\tau)$.

Due to the discontinuity of $G^{\mathrm{M}}(\tau)$ at $\tau=0$ and $\tau=\beta$, GFs show the asymptotic behavior $G(i\omega_n)\sim(i\omega_n)^{-1}$. These tails must be treated exactly in order to assure convergence of the Fourier sum (161). Modifying the Matsubara GF in $0\le\tau\le\beta$ according to

$$\widetilde{G}^{\mathrm{M}}(\tau)=\begin{cases}G^{\mathrm{M}}(\tau)+\frac{1}{2} & \text{: fermions}\\ G^{\mathrm{M}}(\tau)+\frac{\tau}{\beta}-\frac{1}{2} & \text{: bosons,}\end{cases} \tag{165}$$

and in an (anti-) periodic fashion outside this interval, removes the discontinuity at $\tau=0$ and $\tau=\beta$, so that $\widetilde{G}^{\mathrm{M}}(\tau)$ becomes a continuous function. The Fourier coefficients are thus obtained by

$$G^{\mathrm{M}}(i\omega_m)=-\frac{\xi}{i\omega_m}+\widetilde{G}^{\mathrm{M}}(i\omega_m)\ , \tag{166}$$

where $\widetilde{G}^{\mathrm{M}}(i\omega_m)$ is analogous to Eq. (163). We numerically perform the back-transformation (161) on $\widetilde{G}^{\mathrm{M}}(i\omega_m)$, and then obtain $G^{\mathrm{M}}(\tau)$ from (165).

For the Fourier transform, we use a piecewise cubic interpolation, yielding a cubically corrected discrete Fourier transformation as described in chapter 13.9 of Ref. [38]. The convergence of this method is determined by the number of frequency points N_ω. We chose $N_\omega=pN_\tau$ in the Fourier sum (161), where p is an *oversampling* factor (typically $p=10$).

In practice, the convergence of this method is limited by the tail correction and thus the average error scales as $\mathcal{O}(h_\tau^2)$ (see Section 6.1 for an illustrative example). The accuracy can be improved to $\mathcal{O}(h_\tau^{k+2})$[5] by solving the integral equation (150). For convenience, we reformulate the Dyson equation in terms of the integral equation

$$G^{\mathrm{M}}(\tau)=g^{\mathrm{M}}(\tau)+[K*G]^{\mathrm{M}}(\tau)\ ,\ K^{\mathrm{M}}(\tau)=[g*\Sigma]^{\mathrm{M}}(\tau)\ , \tag{167}$$

where $g^{\mathrm{M}}(\tau)$ solves Eq. (150) with $\Sigma^{\mathrm{M}}=0$. The exact solution reads

$$g^{\mathrm{M}}(\tau)=-\bar{f}_\xi(\epsilon(0^-)-\mu)\exp(-\epsilon(0^-)\tau)\ , \tag{168}$$

where $\bar{f}_\xi(\omega)=1+\xi f_\xi(\omega)$ and $f_\xi(\omega)$ denote the Fermi ($\xi=-1$) or Bose ($\xi=1$) distribution, respectively. Eq. (167) constitutes a linear equation for $G^{\mathrm{M}}(mh_\tau)$.

We have implemented a variation of Newton's method for solving this equation iteratively:

[5] The accuracy of solution of an integral equation $G+F*G=Q$ is identical to the accuracy of the quadrature rule if the convolution integral is bounded such that $\|F*\delta G\|<const.\|\delta G\|$.

Newton Iteration. — After solving for $G^{\mathrm{M}}(mh_\tau)$ via the Fourier method, the residual

$$R(mh_\tau) = G^{\mathrm{M}}(mh_\tau) - [K * G]^{\mathrm{M}}(mh_\tau) - g^{\mathrm{M}}(mh_\tau) \tag{169}$$

is generally not zero, as the accuracy of the Fourier method is different from the kth-order accurate convolution. We can regard R defined in Eq. (169) as a functional $R[G]$. Finding the root $R[G] = 0$ of the functional is equivalent to solving the Dyson equation in integral form. To find the root, we set up an iteration in the form

$$G^{\mathrm{M},(i+1)}(mh_\tau) = G^{\mathrm{M},(i)}(mh_\tau) - \Delta G^{\mathrm{M},(i)}(mh_\tau)\ , \tag{170}$$

where the update to the ith iteration, $\Delta G^{\mathrm{M},(i)}(mh_\tau)$, obeys the equation

$$\Delta G^{\mathrm{M},(i)}(mh_\tau) - [K * \Delta G^{(i)}]^{\mathrm{M}}(mh_\tau) = R^{(i)}(mh_\tau)\ .$$

To estimate the update, the above equation is solved using the Fourier method. This procedure provides a rapidly converging[6] iteration to minimize the magnitude of the resolvent (169). As an initial guess $G^{\mathrm{M},(0)}(mh_\tau)$, we again employ the Fourier method. This procedure can be considered as the Newton iteration for finding the root of the functional $R[G]$ with an approximation for the derivative $\delta R/\delta G$.

The routine `dyson_mat` provides a general interface for both methods. The optional argument `method` can be set to `CNTR_MAT_FOURIER` if the Fourier method is to be used, or to `CNTR_MAT_FIXPOINT` for the Newton iteration.

12.3. Start

The `dyson_start` routine evaluates G on the time-slices $\mathcal{T}[G]_n$ for $0 \le n \le k$ (cf. Table 3).

- To determine $G^{\mathrm{R}}(nh, mh)$ for $0 \le n \le k$ and $n \le m \le k$ we consider Eq. (151) with initial condition (155). The solution is similar to the start-up procedure for a Volterra equation (123): At each fixed m, we use a polynomial approximation for $y(t) = G^{\mathrm{R}}(t, mh)$ with $G^{\mathrm{R}}_{n,m} = y_n$,

$$y_n = \begin{cases} G^{\mathrm{R}}_{n,m} & m < n \le k \\ -i & m = n \\ -[G^{\mathrm{R}}_{m,n}]^\dagger & 0 \le n < m. \end{cases} \tag{171}$$

  Here the values y_n for $n < m$ amount to a continuous extrapolation of $G^{\mathrm{R}}(t, t')$ to the domain $t < t'$. When Eq. (151) is solved successively for $m = 0, 1, \dots, k$, the values y_n are already known for $n \le m$. Inserting the polynomial ansatz for $y(t)$ into (151) yields

$$ih^{-1}\sum_{l=0}^{k} D^{(k)}_{n,l} y_l + \epsilon_n y_n - h\sum_{l=0}^{k} I^{(k)}_{m,n;l}\tilde{\Sigma}^{\mathrm{R}}_{n,l} y_l = 0. \tag{172}$$

  This is transformed into a $(k-m)\times(k-m)$ linear problem,

$$\sum_{l=m+1}^{k} M_{n,l} y_l = -\sum_{l=0}^{m} M_{n,l} y_l \equiv Q_n, \quad n = m+1, \dots, k, \tag{173}$$

$$M_{n,l} = ih^{-1} D^{(k)}_{n,l} + \delta_{n,l}\epsilon_n - hI^{(k)}_{m,n;l}\tilde{\Sigma}^{\mathrm{R}}_{n,l}. \tag{174}$$

  Because the input $y_{l\le m}$ for Q_n has been computed previously, this equation can be solved for $y_{l>m}$.
- To determine $G^{\rceil}(nh, mh_\tau)$ for $0 \le n \le k$ and $0 \le m \le N_\tau$ we consider Eq. (152) with initial condition (156). For each given m, this equation provides a Volterra equation of standard type (123), with the replacement

$$y(t) = G^{\rceil}(t, \tau)\ ,\ \ p(t) = i\epsilon(t)\ ,\ \ k(t,s) = i\Sigma^{\mathrm{R}}(t,s), \tag{175}$$

$$q(t) = -i\int_0^\beta d\bar\tau\, \Sigma^{\rceil}(t, \bar\tau) G^{\mathrm{M}}(\bar\tau - \tau). \tag{176}$$

  For $0 \le n \le k$, the Volterra equation is solved using the start-up algorithm (125), where the convolution routines Eqs. (113) and (115) are used to evaluate $q(nh)$,

$$q(t) = -iC^{\rceil}_2[\Sigma, 1, G](n, m) - iC^{\rceil}_3[\Sigma, 1, G](n, m). \tag{177}$$

- To determine $G^<(mh, nh)$ for $0 \le n \le k$ and $0 \le m \le n$ we consider Eq. (153) with the initial condition (157). For each given n, this equation corresponds to a Volterra equation of standard type (123), with the replacement

$$y(t) = G^<(t, nh)\ ,\ \ p(t) = i\epsilon(t)\ ,\ \ k(t,s) = i\Sigma^{\mathrm{R}}(t,s)\ , \tag{178}$$

  and a source term $q(t)$ which is obtained from the convolution routines Eqs. (119) and (121),

$$q(t) = -iC^<_2[\Sigma, 1, G](m, n) - iC^<_3[\Sigma, 1, G](m, n). \tag{179}$$

  Note that $G^<$ must be calculated *after* $G^{\rceil}$ and G^{R} have been evaluated at time-slices $\mathcal{T}[G]_{0\le n\le k}$, so that the input for the latter convolution is already known at this stage of the algorithm. For $0 \le m \le k$, the Volterra equation is solved using the start-up algorithm (125).

[6] If the error of solving the auxiliary equation for ΔG^{M} can be neglected, exactly one iteration is required to reach convergence.

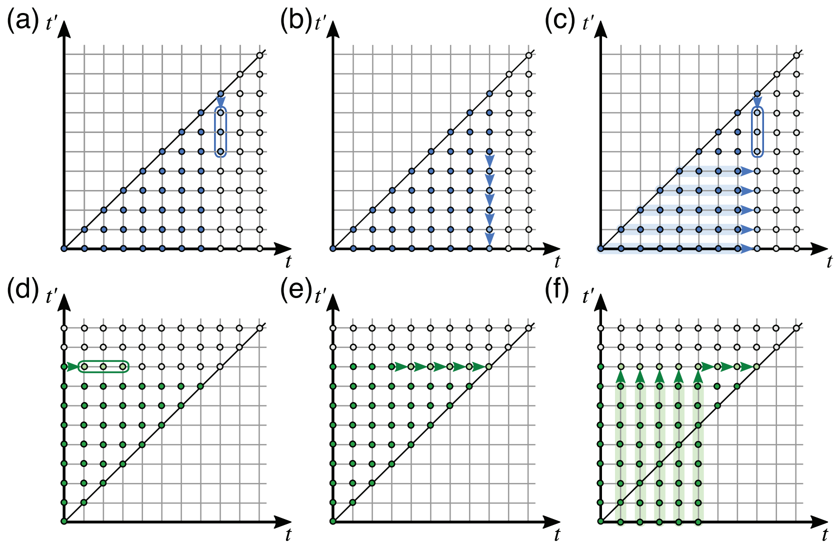


Fig. 15. Propagation scheme of dyson with $k = 3$. (a) Starting at the diagonal $G^{\mathrm{R}}_{n,n}$ with the initial condition (155), the start-up algorithm determines $G^{\mathrm{R}}_{n,m}$ for $m = n-1, \dots, n-k$. (b) After the start-up procedure, the remaining values of $G^{\mathrm{R}}_{n,m}$, $m = n-k = 1, \dots, 0$ can be computed. (c) Parallel version of the dyson solver for the retarded component: the values $G^{\mathrm{R}}_{m,n}$ can be computed in parallel for $m = 0, \dots, n-k$, while the boundary values are obtained as in (a). (d) Start-up procedure for $G^{<}_{m,n}$ for $m = 0, \dots, k$ and subsequent time stepping (e). (f) Parallel algorithm for calculating $G^{<}_{m,n}$ for $m = 1, \dots, n-k$.

12.4. Time stepping

The `dyson_timestep` routines evaluate G from Eq. (34a) on time-slice $\mathcal{T}[G]_n$ for $n > k$, provided that G is already known at time-slices $\mathcal{T}[G]_j$ for $j < n$ (cf. Table 3). $\mathcal{T}[G]_n$ is calculated successively for the retarded, left-mixing, and lesser components:

- To determine $G^{\mathrm{R}}(nh, mh)$ for fixed n and $0 \le m \le n$ there are two alternatives:

  (A) We can consider Eq. (158) with initial condition (155). The equation reduces to a standard Volterra equation (129), with the replacement

$$y(\bar{t}) = G^{\mathrm{R}}(nh, nh - \bar{t})\ ,\ \ p(\bar{t}) = i\epsilon(nh - \bar{t}),$$
$$k(\bar{t}, s) = i\Sigma^{\mathrm{R}}(nh - s, nh - \bar{t})\ ,\ \ y(0) = -i. \tag{180}$$

  The equation is solved using the start-up algorithm (130) for $\bar{t} = lh$, $0 \le l \le k$ (i.e., to compute $G^{\mathrm{R}}_{n,m}$ for $n-k \le m \le n$), while the time stepping algorithm (132) in $\bar{t}$ is applied for $\bar{t} = lh$, $l > k$ (i.e., to compute $G^{\mathrm{R}}_{n,m}$ for $0 \le m < n-k$). The time-stepping scheme is sketched in Fig. 15(a) and (b).

  (B) We can consider Eq. (151) with the initial condition (155). The equation reduces to a standard Volterra equation (123) with the replacement

$$y(\bar{t}) = G^{\mathrm{R}}(mh + \bar{t}, mh)\ ,\ \ p(\bar{t}) = i\epsilon(mh + \bar{t}),$$
$$k(\bar{t}, s) = i\Sigma^{\mathrm{R}}(mh + \bar{t}, mh + s)\ ,\ \ y(0) = -i. \tag{181}$$

  For each $0 \le m < n-k$ this equation is solved for the single time $\bar{t} = (n-m)h$ (i.e. $t = nh$), using the time-stepping method (128). Implementation (B) seems to have, in some cases, a slightly larger numerical error than the alternative (A). However, the Volterra time-steps for $0 \le m < n-k$ can be carried out in parallel, while the implementation (A) is inherently serial. Hence we use alternative (B) for the `openMP` parallel implementations `dyson_timestep_omp`, while (A) is used for the serial implementation `dyson_timestep`. For simplicity and better stability, the values $G^{\mathrm{R}}_{n,m}$ for $n-k \le m \le n$ are always determined from the implementation (A). Fig. 15(c) illustrates the parallel propagation scheme.

- To determine $G^{\rceil}(nh, mh_\tau)$ for fixed $n > k$, we consider Eq. (152) with initial condition (156). For each given m, this equation provides a Volterra equation of standard type (129), with the replacement

$$y(t) = G^{\rceil}(t, \tau)\ ,\ \ p(t) = i\epsilon(t)\ ,\ \ k(t, s) = i\Sigma^{\mathrm{R}}(t, s) \tag{182}$$

  with a source term $q(t)$ that is evaluated using the convolution routines Eqs. (113) and (115),

$$q(nh) = -iC_2^{\rceil}[\Sigma, 1, G](n, m) - iC_3^{\rceil}[\Sigma, 1, G](n, m). \tag{183}$$

  The Volterra equation is solved using the time stepping (128) at the single step n.
- To determine $G^{<}(mh, nh)$ for given n and $0 \le m \le n$ we again have two alternatives:

  (A) We consider Eq. (153) with the initial condition (157). For each given n, this equation becomes a Volterra equation of standard type (123), with the replacement

$$y(t) = G^{<}(t, nh)\ ,\ \ p(t) = i\epsilon(t)\ ,\ \ k(t, s) = i\Sigma^{\mathrm{R}}(t, s), \tag{184}$$

and a source term $q(t)$ which is obtained from the convolution routines Eqs. (119) and (121)

$$q(t) = -iC_2^<[\Sigma, 1, G](m,n) - iC_3^<[\Sigma, 1, G](m,n). \tag{185}$$

The equation is solved using the start-up algorithm (125) for $0 \le m \le k$ (see Fig. 15(d)) and the successive time stepping according to Eq. (128) for $k < m \le n$ (Fig. 15(e)).

(B) Alternatively, we consider Eq. (159) with the initial condition (160). For each given m, this equation provides a Volterra equation of standard type (129), with the replacement

$$y(\bar t) = G^<(mh, \bar t)\ ,\ \ p(\bar t) = -i\epsilon(\bar t)\ ,\ \ k(s,\bar t) = -i\Sigma^{\mathrm{R}}(s,\bar t), \tag{186}$$

and a source term $q(t)$ which is obtained from the convolution routines Eqs. (119) and (121),

$$q(nh) = iC_2^<[G, 1, \Sigma](m,n) + iC_3^<[G, 1, \Sigma](m,n). \tag{187}$$

The equation is solved using a single time step (132) $\bar t = nh$ for each $0 \le m < n-k$. Since all these steps are independent, they can be performed in parallel. Hence, we have implemented a parallelized version `dyson_timestep_omp` based on `openMP` threads. The boundary values $m = n-k, \ldots, n$ are obtained using the serial implementation (A), *after* $G^<_{m,n}$ has been obtained from implementation (B) at $0 \le m \le n-k$. The scheme is sketched in Fig. 15(f).

13. Implementation: `vie2`

13.1. Langreth rules

In this section we present the implementation of the `vie2` routine which solves Eq. (37a). The solution is largely equivalent to `dyson`, but it reduces to a VIE instead of a VIDE. To solve Eq. (37a), we again employ the Langreth rules to obtain the individual equations for the Matsubara, lesser, and left-mixing components,

$$G^{\mathrm{M}}(\tau) + \int_0^\beta d\tau'\, F^{\mathrm{M}}(\tau-\tau')G(\tau') = Q^{\mathrm{M}}(\tau), \tag{188}$$

$$G^{\mathrm{R}}(t,t') + \int_{t'}^t d\bar t\, F^{\mathrm{R}}(t,\bar t)G^{\mathrm{R}}(\bar t,t') = Q^{\mathrm{R}}(t,t') \tag{189}$$

$$\begin{aligned} G^\rceil(t,\tau) &+ \int_0^t d\bar t\, F^{\mathrm{R}}(t,\bar t)G^\rceil(\bar t,\tau) \\ &= Q^\rceil(t,\tau) - \int_0^\beta d\tau F^\rceil(t,\tau')G^{\mathrm{M}}(\tau'-\tau), \end{aligned} \tag{190}$$

$$\begin{aligned} G^<(t,t') &+ \int_0^t d\bar t F^{\mathrm{R}}(t,\bar t)G^<(\bar t,t') \\ &= Q^<(t,t') - \int_0^{t'} d\bar t\, F^<(t,\bar t)G^A(\bar t,t') + i\int_0^\beta d\tau\, F^\rceil(t,\tau)G^\lceil(\tau,t'). \end{aligned} \tag{191}$$

Here Eq. (188) must be solved with the boundary condition

$$G^{\mathrm{M}}(-\tau) = \xi G^{\mathrm{M}}(\beta-\tau)\ , \tag{192}$$

while the remaining equations are solved with initial conditions

$$G^{\mathrm{R}}(t,t) = Q^{\mathrm{R}}(t,t) \tag{193}$$

$$G^\rceil(0,\tau) = iG^{\mathrm{M}}(-\tau) = i\xi G^{\mathrm{M}}(\beta-\tau), \tag{194}$$

$$G^<(0,t') = -[G^\rceil(t',0)]^\dagger \tag{195}$$

13.2. Matsubara

The solution of the VIE for the Matsubara component (Eq. (188)) is analogous to `dyson_mat`. After transforming to the imaginary frequency representation (cf. Eq. (166)), Eq. (188) is transformed to the algebraic equation

$$G^{\mathrm{M}}(i\omega_m) + F^{\mathrm{M}}(i\omega_m)G^{\mathrm{M}}(i\omega_m) = Q^{\mathrm{M}}(i\omega_m)\ . \tag{196}$$

Solving this linear system and calculating the Fourier sum (161) then yields $G^{\mathrm{M}}(\tau)$.

The accuracy of solving Eq. (188) can again be elevated to $\mathcal{O}(h_\tau^{k+2})$ order by the Newton iteration. The algorithm is analogous to the one discussed in Section 12.2, upon replacing $g^{\mathrm{M}} \to Q^{\mathrm{M}}$, $K^{\mathrm{M}} \to -F^{\mathrm{M}}$.

The interface `vie2_mat` allows to choose either method by specifying the argument `method = CNTR_MAT_FOURIER` for the Fourier method, and `method = CNTR_MAT_FIXPOINT` for the Newton iteration, respectively. By default, Newton's method is employed.

13.3. Start

The `vie2_start` routine evaluates G on the time-slices $\mathcal{T}[G]_n$, $0 \le n \le k$ (cf. Table 4).

- To determine $G^{\mathrm{R}}(nh, mh)$ for $0 \le n \le k$ and $n \le m \le k$ we consider Eq. (189) with initial condition (193). The solution is similar to the start-up procedure for a Volterra equation (123): At each fixed m, we use a polynomial approximation for $y(t) = G^{\mathrm{R}}(t, mh)$ with $G^{\mathrm{R}}_{n,m} = y_m$

$$y_n = \begin{cases} G^{\mathrm{R}}_{n,m} & m < n \le k \\ Q^{\mathrm{R}}_{n,n} & m = n \\ -[G^{\mathrm{R}}_{m,n}]^{\dagger} & 0 \le n < m. \end{cases} \tag{197}$$

Here the values y_n for $n < m$ amount to a continuous extrapolation of $G^{\mathrm{R}}(t, t')$ to the domain $t < t'$. When Eq. (189) is solved successively for $m = 0, 1, \dots, k$, the values y_n are already known for $n \le m$. Inserting the polynomial ansatz for $y(t)$ into (189) yields

$$y_n + h \sum_{l=0}^{k} I^{(k)}_{m,n;l} \tilde{F}^{\mathrm{R}}_{n,l} y_l = Q^{\mathrm{R}}_{n,m}. \tag{198}$$

This is transformed into an $(k - m) \times (k - m)$ linear problem,

$$\sum_{l=m+1}^{k} M_{n,l} y_l = - \sum_{l=0}^{m} M_{n,l} y_l, \quad n = m + 1, \dots, k, \tag{199}$$

$$M_{n,l} = \epsilon_n + h I^{(k)}_{m,n;l} \tilde{F}^{\mathrm{R}}_{n,l}. \tag{200}$$

Because the input $y_{l \le m}$ for the right-hand side has been computed previously, this equation can be solved for $y_{l>m}$.

- To determine $G^{\rceil}(nh, mh_\tau)$ for $0 \le n \le k$ and $0 \le m \le N_\tau$ we consider Eq. (190) with the initial condition (194). For each given m, this equation provides a Volterra equation of standard type (133), with the replacement

$$y(t) = G^{\rceil}(t, \tau), \;\; k(t, s) = F^{\mathrm{R}}(t, s), \tag{201}$$

where the source $q(t)$ is evaluated using the convolution routines Eqs. (113) and (115),

$$q(nh) = -C_2^{\rceil}[F, 1, G](n, m) - C_3^{\rceil}[F, 1, G](n, m) + Q^{\rceil}_{n,m}. \tag{202}$$

For $0 \le n \le k$, the Volterra equation is solved using the start-up algorithm (135).

- To determine $G^{<}(mh, nh)$ for $0 \le n \le k$ and $0 \le m \le n$ we consider Eq. (191) with the initial condition (195). For each given n, this equation provides a Volterra equation of standard type (133), with the replacement

$$y(t) = G^{<}(t, nh), \;\; k(t, s) = F^{\mathrm{R}}(t, s), \tag{203}$$

and a source term $q(t)$ which is obtained from the convolution routines Eqs. (119) and (121),

$$q(t) = -C_2^{<}[F, 1, G](m, n) - C_3^{<}[F, 1, G](m, n) + Q^{<}_{m,n}. \tag{204}$$

Note that $G^{<}$ must be calculated *after* $G^{\rceil}$ and G^{R} have been evaluated at the time-slices $\mathcal{T}[G]_{0 \le n \le k}$, so that the input for the latter convolution is already known at this stage of the algorithm. For $0 \le m \le k$, the Volterra equation is solved using the start-up algorithm (135).

13.4. Time stepping

Once the start-up problem has been solved and $\mathcal{T}[G]_n$ is known for $n = 0, \dots, k$, time-stepping can be employed (see Table 4). Mapping the VIEs (189)–(191) to the standard VIE (133) allows to directly adopt the algorithm from Section 10.2. Suppose $G^{\mathrm{R}}(jh, mh)$, $G^{\rceil}(jh, lh_\tau)$ and $G^{<}(mh, jh)$ are known for $j = 0, \dots, n - 1$, $m = 0, \dots, j$ and $l = 0, \dots, N_\tau$. Then the next time step $\mathcal{T}[G]_n$ is obtained as follows:

- In order to compute $G^{\mathrm{R}}(nh, mh)$, for $m = 0, \dots, n-1$ (since $G^{\mathrm{R}}(nh, nh) = Q^{\mathrm{R}}(nh, nh)$), we approximate the convolution by Eq. (109). Hence, setting $y(t) = G^{\mathrm{R}}(t, mh)$ for fixed m maps Eq. (189) to the standard VIE (123) for $n - m > k$. One obtains

$$y_n = q_n + h \sum_{j=m}^{n} w^{(k)}_{n-m,j-m} F^{\mathrm{R}}_{n,j} y_j \ ,$$

where the continuous extension (197) is implied. The above equation is then solved for y_n. For $n - m \le k$, the procedure is similar: via the approximation (109), the VIE translates to

$$y_n = q_n + h \sum_{j=0}^{k} w^{(k)}_{n-m,j} F^{\mathrm{R}}_{n,j} y_{n-j} \ ,$$

which is readily solved for y_n. Note that only $G^{\mathrm{R}}(nh, mh)$ needs to be extrapolated to the upper triangle, while the kernel $F^{\mathrm{R}}_{n,m} = F^{\mathrm{R}}(nh, mh)$ is strictly causal. Except for the case $n - m \le k$, the time step $n - 1 \to n$ can be carried out independently for every $m = 0$. Therefore, these time steps can be performed in parallel, as implemented in the `openMP`-based function `vie2_timestep_omp`.

- The VIE (190) maps to the standard VIE (133) upon identifying $y(t) = G^{\rceil}(t, mh_\tau)$, $k(t,s) = F^{\mathrm{R}}(t,s)$, while the source term $q(t)$ is obtained by the identification (202). The time-stepping algorithm (137) can be used directly. All steps depend only parametrically on m, so parallel propagation is straightforward.
- Once $G^{\mathrm{R}}(nh, mh)$ and $G^{\rceil}(nh, lh_\tau)$ $(m = 0, \dots, n,\ l = 0, \dots, N_\tau)$ have been obtained, the lesser component $G^<(mh, nh)$ can be computed. As for `dyson_timestep`, there are two options for proceeding:
  - (A) The substitutions (203) and (204) map the lesser VIE (191) to the standard VIE (133). For $m = 0, \dots, k$, the resulting equation is solved by the start-up method (135), using the initial condition (195). For $m = k+1, \dots n$, the time propagation proceeds by solving Eq. (137). This scheme of time stepping is sequential by construction.
  - (B) Instead of starting from Eq. (191), the conjugate equation

    $$G^<(t,t') + [G * F^{\ddagger}]^<(t,t') = Q^<(t,t')$$

    can serve as a starting point. The substitution

    $$y(t) = G^<(mh, t), \ \ k(s,t) = [F^{\ddagger}]^{\mathrm{A}}(s,t)$$
    $$q(mh) = -C_1^<[G, 1, F^{\ddagger}](m,n) - C_3^<[G, 1, F^{\ddagger}](m,n) + Q^<(mh, nh)$$

    leads the to the conjugate VIE (134), which can then be propagated by invoking Eq. (140). This time-stepping scheme can be performed for all $m = 0, \dots, n-1$ in parallel, as implemented in `vie2_timestep_omp`. The last point $G^<(nh, nh)$ can be computed once $G^<(nh, (n-1)h) = -[G^<((n-1)h, nh)]^\dagger$ is known.

14. Implementation: Free Green's functions

Free GFs $G_0(t,t')$ are determined from the equation of motion [cf. Eq. (33)]

$$[i\partial_t - \epsilon(t)]\, G_0(t,t') = \delta_{\mathcal{C}}(t,t') \tag{205}$$

as follows. Let us denote the eigenvalues of the Hamiltonian matrix $\epsilon(0^-)$ by ε_α and the corresponding basis transformation matrix by R, such that $\epsilon(0^-) = R\,\mathrm{diag}\{\epsilon_\alpha\}R^\dagger$ ($\mathrm{diag}\{\varepsilon_\alpha\}$ stands for the diagonal matrix containing the energies ε_α). The Matsubara component is then given by

$$G_0^{\mathrm{M}}(\tau) = R\,\mathrm{diag}\{f_\xi(\mu - \varepsilon_\alpha)e^{-(\epsilon_\alpha - \mu)\tau}\}R^\dagger \tag{206}$$

for $\tau \in (0, \beta)$.

All other Keldysh components of $G_0(t,t')$ are governed by the unitary time evolution (defined in Eq. (5)) with respect to the single-particle Hamiltonian $\epsilon(t)$.

14.1. Commutator-free matrix exponentials

On the equidistant grid $t_n = nh$, we approximate the propagator $U_{n,j} \equiv U(nh, jh)$ by the commutator-free matrix exponential approximation described in Ref. [39]. In particular, we have implemented the fourth-order approximation

$$\begin{aligned} U_{n+1,n} &= \exp\left[-i(a_1\epsilon((n+c_1)h) + a_2\epsilon((n+c_2)h))\right] \\ &\quad \times \exp\left[-i(a_2\epsilon((n+c_1)h) + a_1\epsilon((n+c_2)h))\right] + \mathcal{O}(h^5)\ , \end{aligned} \tag{207}$$

where $a_1 = (3 - 2\sqrt{3})/12$, $a_2 = (3 + 2\sqrt{3})/12$, $c_1 = (1 - 1/\sqrt{3})/2$ and $c_2 = (1 + 1/\sqrt{3})/2$. Using the semi-group property $U_{n,j} = U_{n,n-1}U_{n-1,n-2}\dots U_{j+1,j}$, we can thus express the propagator up to $\mathcal{O}(h^4)$. The Hamiltonian at the intermediate points $(n + c_{1,2})$ entering Eq. (207) is approximated by polynomial interpolation, using the points $n-k+1, \dots, n, n+1$ (see Section 8). If $\epsilon(t)$ represents a mean-field Hamiltonian, which is self-consistently determined in the course of the time step $n \to n+1$, ϵ_{n+1} is typically not known before the GF at time step $n+1$ has been computed. Hence, we employ polynomial extrapolation to provide a guess for ϵ_{n+1} before interpolating.

14.2. Real-time and mixed components

Based on the commutator-free matrix exponential approximation, the remaining Keldysh components are determined by

$$G_0^{\rceil}(nh, \tau) = -i\xi U_{n,0}(nh, 0) R\,\mathrm{diag}\{f_\xi(\varepsilon_\alpha - \mu)e^{(\epsilon_\alpha - \mu)\tau}\}R^\dagger\ , \tag{208a}$$

$$G_0^{\mathrm{R}}(nh, jh) = -iU_{n,j} = U_{n,0}[U_{j,0}]^\dagger\ , \tag{208b}$$

$$G_0^<(jh, nh) = iU_{j,0}R\,\mathrm{diag}\{f_\xi(\varepsilon_\alpha - \mu)\}R^\dagger[U_{n,0}]^\dagger\ . \tag{208c}$$

Note that for a time-independent Hamiltonian, Eq. (208) is numerically exact up to round-off errors. Furthermore, the structure of Eq. (208) allows to compute the time slice $\mathcal{T}[G_0]_n$ directly.

15. Conclusions

We have presented the NESSi library, a **N**on-**E**quilibrium **S**ystems **Si**mulation package. This open-source computational physics library provides a simple and efficient framework for simulations of quantum many-body systems out of equilibrium, based on the Greens function formalism. The numerical routines employed in the solution of the Kadanoff–Baym equations and the evaluation of Feynman diagrams have been described in detail. We have exemplified the usage of the library by several applications ranging from simple two-level problems to the state-of-the-art simulations of interacting lattice systems. This information should enable users of the library to implement and run custom applications.

NESSi is an open source library and we encourage contributions and feedback from the user community. We will continue to work on extensions of the library. Planned near-term improvements include the publication of a software package for nonequilibrium impurity and dynamical mean-field theory calculations based on strong-coupling perturbative solvers, non-equilibrium steady state solvers, and truncation schemes for the memory integrals in the integral equations. The latest updates will posted on the web page www.nessi.tuxfamily.org, which also contains a link to the repository, installation instructions, a detailed manual of all relevant classes and routines, and additional example programs. Contributions to the future extensions are welcome, although we recommend to coordinate with the main NESSi developers before embarking on any major coding effort. Any issues encountered in the use of the library should be exclusively reported via the contact address specified on the web site www.nessi.tuxfamily.org.

CRediT authorship contribution statement

Michael Schüler: Software, Validation, Writing - original draft, Writing - review & editing, Visualization, Supervision. **Denis Golež:** Software, Validation, Writing - original draft, Writing - review & editing, Visualization, Supervision. **Yuta Murakami:** Software, Writing - original draft, Writing - review & editing, Visualization. **Nikolaj Bittner:** Validation, Writing - review & editing, Visualization. **Andreas Herrmann:** Software, Validation, Writing - review & editing. **Hugo U.R. Strand:** Validation, Writing - original draft, Writing - review & editing, Visualization. **Philipp Werner:** Conceptualization, Resources, Writing - review & editing, Supervision, Project administration, Funding acquisition. **Martin Eckstein:** Conceptualization, Methodology, Software, Writing - original draft, Writing - review & editing, Supervision, Project administration, Funding acquisition.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

We thank Marcus Kollar, Naoto Tsuji, Jiajun Li, and Nagamalleswararao Dasari, for important feedback while using the library, and for collaborations on early stages of the library. The development of this library has been supported by the Swiss National Science Foundation through SNF Professorship PP0022-118866 (ME,PW), Grants 200021-140648 and 200021-165539 (DG), and NCCR MARVEL (MS,YM), as well as the European Research Council through ERC Starting Grant Nos. 278023 (AH,HS,PW) and 716648 (ME), and ERC Consolidator Grant No. 724103 (MS,NB,PW,YM). MS thanks the Alexander von Humboldt Foundation, Germany for its support with a Feodor Lynen scholarship. The Flatiro Institute as a division of the Simons Foundation.

Appendix A. Contour function utilities

In this appendix, we describe how contour functions can be extrapolated by polynomial extrapolation. Furthermore, we define an absolute-value-norm for contour functions.

A.1. Extrapolation of contour functions

Based on polynomial interpolation (see Section 8.1), we define the polynomial extrapolation by

$$y_{n+1} = \sum_{l=0}^{k} C_l^{(k)} y_{n-l} \ , \tag{A.1}$$

where $y_l = y(lh)$. The coefficients $C_l^{(k)}$ are obtained by inserting $t = (n+1)h$ into Eq. (79). For extrapolations in the two-time plane, we have implemented the following algorithm:

- To approximate $G^{\rceil}((n+1)h, \tau)$ we set $y(t) = G^{\rceil}(t, \tau)$ for fixed τ and apply Eq. (A.1).
- For extrapolating the retarded component, we set $y(t) = \widetilde{G}^{\mathrm{R}}(t, jh)$ for $j = 0, \dots, k$ and apply Eq. (A.1). For the remaining points, we extrapolate along lines parallel to the time diagonal by identifying $y(t) = G^{\mathrm{R}}(t, t - jh)$ for $j = 0, \dots, n-k$. Using Eq. (A.1) then yields the extrapolation to $G^{\mathrm{R}}((n+1)h, (n+1-j)h) \approx y_{n+1}$.
- Similarly, the lesser component can be extrapolated by identifying

$$y(t) = \begin{cases} G^<(jh, t) & : t \geq jh \\ -[G^<(t, jh)]^\dagger & : t < jh \end{cases} .$$

Polynomial extrapolation (A.1) then yields $G^<(jh, (n+1)h)$ for $j = 0, \dots, k$. Analogous to the retarded component, the remaining points in the two-time plain are obtained by applying Eq. (A.1) to $y(t) = G^<((j-n-1)h + t, t)$ for $j = k+1, \dots, n+1$. Note that this includes the diagonal $G^<((n+1)h, (n+1)h)$.

Eq. (A.1) can also be applied to single-time contour functions $f(t)$. The above algorithm is implemented in the function `extrapolate_timestep`.

A.2. Absolute-value-norm

For assessing the convergence of self-consistent algorithms, we introduce a absolute-value-norm for contour functions. Consider two time slices $\mathcal{T}[A]_n$, $\mathcal{T}[B]_n$ at time step n. We define the distance for the individual components as

$$\|A-B\|^{\mathrm{M}} = \sum_{m=0}^{N_\tau}\sum_{a,b}\left|A^{\mathrm{M}}_{a,b}(mh_\tau) - B^{\mathrm{M}}_{a,b}(mh_\tau)\right| \ , \tag{A.2a}$$

$$\|A-B\|^{\rceil}_n = \sum_{m=0}^{N_\tau}\sum_{a,b}\left|A^{\rceil}_{a,b}(nh, mh_\tau) - B^{\rceil}_{a,b}(nh, mh_\tau)\right| \ , \tag{A.2b}$$

$$\|A-B\|^{\mathrm{R}}_n = \sum_{j=0}^{n}\sum_{a,b}\left|A^{\mathrm{R}}_{a,b}(nh, jh) - B^{\mathrm{R}}_{a,b}(nh, jh)\right| \ , \tag{A.2c}$$

$$\|A-B\|^{<}_n = \sum_{j=0}^{n}\sum_{a,b}\left|A^{<}_{a,b}(jh, nh) - B^{<}_{a,b}(jh, nh)\right| \ . \tag{A.2d}$$

The total distance at time step n is then defined by

$$\|A-B\|_n = \begin{cases} \|A-B\|^{\mathrm{M}} & : n=-1 \\ \|A-B\|^{\rceil}_n + \|A-B\|^{\mathrm{R}}_n + \|A-B\|^{<}_n & : n\geq 0 \end{cases} \ . \tag{A.3}$$

The absolute-value-norm Eq. (A.3) is implemented in the function `distance_norm2`.

Appendix B. Instructions for installation and running : nessi_demo example programs

We assume that the `libcntr` library has been compiled successfully and installed under the prefix `/home/opt`. Hence, `/home/opt/lib` contains the shared library `libcntr.so` (or `libcntr.dylib` under MacOSX), while `/home/opt/include` contains the directory `cntr` with all required headers. After downloading or cloning the repository `nessi_demo`, navigate into `examples` in `nessi` and create a build directory (for instance, `cbuild`). The installation procedure is similar to the compilation of `libcntr` (see Section 5.2). We recommend creating a configuration script similar to

```
CC=[C compiler] CXX=[C++ compiler] \
cmake \
    -DCMAKE_BUILD_TYPE=[Debug|Release] \
    -Domp=[ON|OFF] \
    -Dhdf5=[ON|OFF] \
    -Dmpi=[ON|OFF] \
    -DCMAKE_INCLUDE_PATH=[include directory] \
    -DCMAKE_LIBRARY_PATH=[library directory] \
    -DCMAKE_CXX_FLAGS="[compiling flags]" \
    ..
```

For compiling all examples including the translationally invariant Hubbard model (Section 7.2), MPI compilers need to be provided for the C and the C++ compiler. Furthermore, set `mpi=ON`.

`CMAKE_INCLUDE_PATH` needs to include the path used to compile `libcntr` (containing the `eigen3` and `hdf5` headers) and, additionally, `/home/opt/include`. The paths provided to `CMAKE_LIBRARY_PATH` should include all the library paths used to compile `libcntr`, extended by `/home/opt/lib`. We recommend using the same compiler flags as for the compilation of `libcntr`, including

```
-std=c++11
```

After creating the above configure script (for instance, `configure.sh`), navigate to the build directory and run

```
sh ../configure.sh
make
```

to compile the example programs. The executables are placed under `nessi_demo/exe`.

The `utils/` directory contains useful python driver scripts which simplify the execution of the example programs. In order to run the python script, we need to make sure to set the python path to `nessi/libcntr/python` and/or `nessi/libcntr/python3`. The scripts should be ran from `nessi/examples` as follows:

```
python3 utils/*******.py k
```

Here, “k” is input only necessary for `test_equilibrium.py` and `test_nonequilibrium.py`. In the Table B.10, we summarize the python scripts and the corresponding exe files and provide brief explanations of what is done in the python scripts.

Table B.10
Summary of the python scripts to run examples.

Python scripts in `utils`	Explanation	Sec.
`test_equilibrium.py`	Runs the execute file `test_equilibrium.x` to show the scaling of accuracy of the Matsubara Dyson solvers with the specified order as an input. A figure for the scaling against N_τ is created.	6.1
`test_nonequilibrium.py`	Runs `test_nonequilibrium.x` to show the scaling of accuracy of the integro-differential (Dyson) and integral (VIE2) formulation with the specified order as an input. A figure for the scaling against N_t is created.	6.1
`demo_hubbard_chain.py`	Runs `hubbard_chain_**.x` to simulate quench dynamics of the Hubbard chain. Here, `**` (= `2b, gw, tpp`) indicates different many body approximations, which can be specified in the python script. Figure for the time evolution of density and energies are created.	6.2
`demo_Holstein_impurity.py`	Runs `Holstein_impurity_singlebath_**.x` to simulate dynamics against modulation of system parameters in the Holstein-type impurity with a single bath site. Here, `**` (= `Migdal, uMig`) indicates different approximate impurity solvers, which can be specified in the python script. The spectra of electrons and phonons and the time evolution of phonon displacement and energies are plotted.	6.3
`demo_Holstein.py`	Runs `Holstein_bethe_**.x` to simulate dynamics of the Holstein model against modulation of system parameters within DMFT. The rest is the same as `demo_Holstein_impurity.py`.	6.3
`demo_Holstein_sc.py`	Runs `Holstein_bethe_Nambu_**.x`, which is a generalized version of `Holstein_bethe_**.x` to treat *s*-wave superconductor (SC). In addition to figures for the spectra and evolution of densities and energies, evolution of the SC order parameter is plotted.	6.3
`demo_gw.py`	Runs `gw.x` to simulate the 1dim chain of the extended Hubbard model within the GW approximation using MPI parallelization. The script create figures for the electric field and the change in the kinetic energy.	7.2
`demo_integration.py`	Runs `integration.x` to demonstrate the accuracy of the Gregory integration implemented in `nessi`. The same figure as Fig. 14(a) is created.	8.5

References

[1] A.J. Daley, C. Kollath, U. Schollwöck, G. Vidal, J. Stat. Mech. Theor. Exp. 2004 (04) (2004) P04005, http://dx.doi.org/10.1088/1742-5468/2004/04/p04005.
[2] S.R. White, A.E. Feiguin, Phys. Rev. Lett. 93 (2004) 076401, http://dx.doi.org/10.1103/PhysRevLett.93.076401.
[3] K. Ido, T. Ohgoe, M. Imada, Phys. Rev. B 92 (2015) 245106, http://dx.doi.org/10.1103/PhysRevB.92.245106.
[4] G.D. Mahan, Many-Particle Physics, Plenum Press, New York, 1990.
[5] J.E. Gubernatis, N. Kawashima, P. Werner, Quantum Monte Carlo methods, Cambridge University Press, Cambridge, 2016.
[6] L.P. Kadanoff, G. Baym, Quantum Statistical Mechanics, W. A. Benjamin, New York, 1962.
[7] L. Keldysh, JETP 20 (4) (1965) 1018.
[8] A.A. Abrikosov, L.P. Gorkov, I.E. Dzyaloshinski, Methods of Quantum Field Theory in Statistical Physics, Dover, New York, 1975.
[9] G. Stefanucci, R.v. Leeuwen, Nonequilibrium Many-Body Theory of Quantum Systems: A Modern Introduction, Cambridge University Press, 2013.
[10] A. Kamenev, Field Theory of Non-Equilibrium Systems, Cambridge University Press, 2011.
[11] K. Balzer, M. Bonitz, Nonequilibrium Green'S Functions Approach To Inhomogeneous Systems, Springer, 2012.
[12] A. Stan, N.E. Dahlen, R. van Leeuwen, J. Chem. Phys. 130 (22) (2009) 224101, http://dx.doi.org/10.1063/1.3127247.
[13] L. Hedin, J. Phys.: Condens. Matter 11 (42) (1999) R489–R528, http://dx.doi.org/10.1088/0953-8984/11/42/201.
[14] H. Aoki, N. Tsuji, M. Eckstein, M. Kollar, T. Oka, P. Werner, Rev. Modern Phys. 86 (2014) 779–837, http://dx.doi.org/10.1103/RevModPhys.86.779.
[15] N. Tsuji, P. Werner, Phys. Rev. B 88 (2013) 165115, http://dx.doi.org/10.1103/PhysRevB.88.165115.
[16] M. Eckstein, P. Werner, Phys. Rev. B 82 (2010) 115115, http://dx.doi.org/10.1103/PhysRevB.82.115115.
[17] H. Keiter, J.C. Kimball, J. Appl. Phys. 42 (4) (1971) 1460–1461, http://dx.doi.org/10.1063/1.1660293.
[18] T. Pruschke, N. Grewe, Z. Phys. B 74 (4) (1989) 439–449, http://dx.doi.org/10.1007/BF01311391.
[19] F. Aryasetiawan, O. Gunnarsson, Rep. Progr. Phys. 61 (3) (1998) 237–312, http://dx.doi.org/10.1088/0034-4885/61/3/002.
[20] [A modern, C]++-native, header-only, test framework for unit-tests, TDD and BDD: using C++11, C++14, C++17 and later (or C++03 on the Catch1.x branch) - catchorg/Catch2, 2010, original-date: 2010-11-08T18:22:56Z (Feb. 2019). URL https://github.com/catchorg/Catch2.
[21] M. Puig von Friesen, C. Verdozzi, C.-O. Almbladh, Phys. Rev. Lett. 103 (2009) 176404, http://dx.doi.org/10.1103/PhysRevLett.103.176404.
[22] M. Puig von Friesen, C. Verdozzi, C.-O. Almbladh, Phys. Rev. B 82 (2010) 155108, http://dx.doi.org/10.1103/PhysRevB.82.155108.
[23] N. Schlünzen, M. Bonitz, Contrib. Plasma Phys. 56 (1) (2016) 5–91, http://dx.doi.org/10.1002/ctpp.201610003.
[24] N. Schlünzen, J.-P. Joost, F. Heidrich-Meisner, M. Bonitz, Phys. Rev. B 95 (2017) 165139, http://dx.doi.org/10.1103/PhysRevB.95.165139.
[25] W. Metzner, D. Vollhardt, Phys. Rev. Lett. 62 (1989) 324–327, http://dx.doi.org/10.1103/PhysRevLett.62.324.
[26] A. Georges, G. Kotliar, W. Krauth, M.J. Rozenberg, Rev. Modern Phys. 68 (1996) 13–125, http://dx.doi.org/10.1103/RevModPhys.68.13.
[27] A.F. Kemper, M.A. Sentef, B. Moritz, J.K. Freericks, T.P. Devereaux, Phys. Rev. B 90 (2014) 075126, http://dx.doi.org/10.1103/PhysRevB.90.075126.
[28] M.A. Sentef, A.F. Kemper, A. Georges, C. Kollath, Phys. Rev. B 93 (2016) 144506, http://dx.doi.org/10.1103/PhysRevB.93.144506.
[29] Y. Murakami, P. Werner, N. Tsuji, H. Aoki, Phys. Rev. B 91 (2015) 045128, http://dx.doi.org/10.1103/PhysRevB.91.045128.
[30] Y. Murakami, P. Werner, N. Tsuji, H. Aoki, Phys. Rev. B 93 (2016) 094509, http://dx.doi.org/10.1103/PhysRevB.93.094509.
[31] M. Schüler, J. Berakdar, Y. Pavlyukh, Phys. Rev. B 93 (2016) 054303, http://dx.doi.org/10.1103/PhysRevB.93.054303.
[32] R. Peierls, Z. Phys. 80 (11) (1933) 763–791, http://dx.doi.org/10.1007/BF01342591.

[33] J.M. Luttinger, Phys. Rev. 84 (1951) 814–817, http://dx.doi.org/10.1103/PhysRev.84.814.
[34] D. Golež, P. Werner, M. Eckstein, Phys. Rev. B 94 (2016) 035121, http://dx.doi.org/10.1103/PhysRevB.94.035121.
[35] T. Giamarchi, Quantum Physics in One Dimension, vol. 121, Clarendon press, 2003.
[36] J. Steinberg, Numer. Math. 19 (3) (1972) 212–217, http://dx.doi.org/10.1007/BF01404691.
[37] H. Brunner, P.J.v.d. Houwen, The Numerical Solution of Volterra Equations, North-Holland, 1986, Sole distributors for the U.S.A. and Canada, Elsevier Science Pub. Co., Amsterdam; New York; New York, N.Y., U.S.A., 1986, oCLC: 13760699.
[38] W.H. Press, S.A. Teukolosky, W.T. Vetterling, B.P. Flannery, Numerical Recipes 3rd Edition: the Art of Scientific Computing, Cambridge University Press, 2007.
[39] A. Alvermann, H. Fehske, J. Comput. Phys. 230 (2011) 5930, http://dx.doi.org/10.1016/j.jcp.2011.04.006.