Chapter 1

The code

The main part of this thesis has been the implementation of a self-consistent ladder-DΓA calculation in a very accessible[1] way. In this chapter, we present the implementation of the computational framework developed to achieve a fully converged self-energy though the self-consistency algorithm of the ladder-DΓA equations. The code is written in `Python` and relies exclusively on `NumPy` for numerical calculations, ensuring efficient execution through vectorized operations and optimized linear algebra routines. Additionally, symmetries inherent to the problem have been carefully exploited to further reduce computational costs and improve performance.

The code is designed to efficiently compute vertex functions, such as one-particle Green's function, irreducible and full vertices, three-leg vertices, the electron self-energy and many more, while maintaining flexibility for extensions and modifications. Building upon the underlying theoretical foundations which were established already in the previous chapters, this numerical implementation aims to translate the formal expressions into a working algorithm. Special attention has been given to structuring the code in a modular and efficient manner, ensuring that it can be used not only for the ladder-DΓA equations, but also for other algorithms that require multi-point vertex functions within the Matsubara frequency framework in condensed matter physics.

The package will be provided under the MIT license, making it freely available for use, modification, and distribution. This permissive license promotes collaboration and allows academics and developers to make changes on the code without placing restrictions. By publishing the implementation open-source, we hope to encourage accessibility, reproducibility, and community development.

We start with an overview of the overall structure and design choices of the code, followed by a detailed discussion of its core components and algorithms. We then describe the typical workflow for using the code, including input handling, computational steps and output generation. Finally, we discuss validation methods and potential improvements for future work.

1.1 Structure and design choices

Please note that there already exist two toolboxes that allow the calculation of vertex functions through ladder-DΓA: (i) `AbinitioDΓA` [**abinitio dga**, **anna galler thesis**, **abinitio dga project**], which is a one-shot multi-orbital calculation of the DΓA self-energy and (ii) `DGApy` [**dgapy**], which is the one-band version of it, where additional quantities are calculated, such as the superconducting eigenvalues and real-valued Green's functions. The former is written in `Fortran`, whereas the latter has been developed in `Python`. The purpose of the code developed in the course of this thesis is to provide the functionality of `AbinitioDΓA`, while being easily accessible and quick to set-up, similar to `DGApy`. Therefore, we decided to write the program in `Python`. A main advantage of this programming language is its low entry-barrier and the easy set-up and code-execution. In theory, all that is needed is a

[1] Accessible here means, that we provide a very easy-to-understand API (application programming interface), where only a short training period and quick set-up is needed to get the code up and running.

Python environment with the necessary packages[2], DMFT input files, the configuration file and the executable code.
As already mentioned, there already exists a Python package, called DGApy [**dgapy**], written by Paul Worm, which is capable of performing a single-band λ-corrected one-shot ladder-DΓA calculation. The code in this thesis will, however, be concerned with the multi-orbital implementation of the algorithm, additionally employing a self-consistency loop contrary to the λ-correction and will be capable of handling non-local Coulomb interactions. So far, DGApy has successfully been used in several instances, for example in Refs. [**simone unconventional**, **paul spin fluc**], to only mention a few. Since it is already widely used among researchers, the transition from DGApy to this code should be as simple as possible, hence the configuration file and some internal structures are kept very similar.

After the vertex calculation has been performed with w2dyn [**w2dyn**], we have to extract a couple of key quantities, such as the local self-energy, Green's function, interaction and two-particle Green's functions from the output files of w2dyn. There is a script written, called symmetrize.py, which allows for the straightforward extraction of the two-particle Green's function from the vertex file of w2dyn. A simple execution of this script yields a new file, where the density and magnetic components of $G^{\omega\nu\nu'}_{r;1234}$ are written to. This is handy, since the vertex file from w2dyn is usually very large and it can be deleted after the symmetrization process. Then one is left with the 1p-data.hdf5 and g4iw_sym.hdf5 files, which contain the one-particle and two-particle (symmetrized) quantities. These are, next to a file containing the real-space (e.g., wannier_hr.dat) or Fourier-space (e.g., wannier.hk) Hamiltonian, the only input files needed for the execution of the DΓA algorithm. Notice that all the file names are not fixed and can be specified in the dga_config.yaml file, which is the main configuration file for the program, more on that later.

To lower the computation time, we employ the use of the mpi4py library in Python, which allows us to execute the program inside of a process-pool, where it is very easy to slice and pass/gather objects to/from other processes. This — in theory — should result in an n-times faster execution of parallelizable code, where n is the number of available MPI processes. However, this exact speed-up is not always realistic, since there is always some communication overhead that has to be accounted for. When the explicit non-local execution of the DΓA-equations starts, the MpiDistributor-class is set-up. This class allows for an easy scattering and gathering of objects along the $\mathbf{q}$-dimension to and from other processes. By specifying the number of $\mathbf{q}$-points of the objects, the MpiDistributor automatically knows which $\mathbf{q}$-slice belongs to which sub-process. This slicing along the $\mathbf{q}$-dimension is only possible because of the explicit diagonal structure of the BSE (**??**) and the SDE (**??**).

We tried to keep a very object-oriented approach when developing the toolbox for the program, hence instead of directly working with NumPy arrays, we work with (Local)FourPoint, (Local)Interaction, SelfEnergy and GreensFunction classes. In Python, it is very easy to overload operators, i.e. with so-called "dunder" methods, hence we implemented the most common operators "+/-/@/~", alongside multiplication and division with numbers, for these objects to help make the implementation of equations more easy. This might not be a obvious improvement at first, however the underlying structure allows a very simple high-level handling of these objects, where most of the logic is performed in the background. We will discuss the explicit implementation of these dunder methods later.

[2]For a detailed list of requirements including package versions, see the file requirements.txt in the source code directory.