Chapter 1 The code

The main part of this thesis is the implementation of a self-consistent ladder-DΓA calculation with an emphasis on usability and minimal setup effort. The code is designed to be both accessible and performant, targeting fast execution through efficient parallelization and extensive memory optimizations. It is written in Python and relies mainly on NumPy for numerical computations, making use of vectorized operations and optimized linear algebra routines. In addition, the code carefully exploits symmetries inherent to the physical problem to further reduce computational cost and memory usage. In this chapter, we present the implementation of the computational framework developed to achieve a fully converged self-energy through the self-consistency algorithm of the ladder-DΓA equations and a subsequent calculation of the superconducting eigenvalues and gap functions for both singlet and triplet electron pairing.

The code is designed to efficiently compute vertex functions, such as the 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.

In this chapter, 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 programs that allow the calculation of vertex functions and the self-energy through ladder-DΓA: (i) AbinitioDΓA [**Galler2019**, **Galler2018**, **Galler2017**], which is

a one-shot multi-orbital calculation of the DΓA self-energy and (ii) DGApy [**Worm2023**], 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 Python environment with the necessary packages[1], DMFT input files and the configuration file.
As already mentioned, there already exists a Python package, called DGApy [**Worm2023**], 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. [**DiCataldo2024**, **Worm2024**], 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 modules are designed very similar.

1.1.1 Parallelization via MPI

To lower the computation time, we employ the use of the mpi4py library in Python, which enables distributed computing across multiple processes in a straightforward and Pythonic manner. This allows the program to run inside a process pool, where objects can easily be sliced and passed to or gathered from other processes. While an ideal speed-up factor of n for n processes is theoretically possible, in practice, communication overhead and memory transfer costs typically limit the effective gain. Nevertheless, for large-scale computations involving high-dimensional vertex functions or large frequency boxes, this parallelization leads to significant reductions in runtime.
The central class handling parallelization is the MpiDistributor, which is initialized at the point where the explicitly non-local parts of the ladder-DΓA algorithm begin. This class abstracts the communication logic and provides a consistent interface for scattering and gathering objects across processes. The current implementation slices the data along the transferred momentum $\mathbf{q}$, which is possible due to the block-diagonal structure of most equations up to the Schwinger-Dyson equation (**??**). Each MPI process is responsible for a specific set of $\mathbf{q}$-points, allowing for highly parallel and independent evaluation of vertex contractions.
Internally, the slicing is guided by knowledge of the irreducible Brillouin zone, and the distributor automatically determines the mapping from global $\mathbf{q}$-grid indices to process-local ones. This separation of concerns between parallelization logic and physics routines enhances maintainability and flexibility of the code. Further MPI optimizations, such as asynchronous communications or load balancing across uneven grid sizes, could be explored in future versions.

1.1.2 Memory reduction through symmetry

The memory demands of a full ladder-DΓA calculation can be substantial, especially for multi-orbital models and large frequency ranges. To mitigate this, the code incorporates several symmetry-based optimizations that significantly reduce the memory footprint and enable efficient scaling to larger systems.
First, the calculation of non-local quantities is performed exclusively within the irreducible Brillouin zone. Thanks to the crystal symmetries of common lattice structures (e.g., square or quasi-1D lattices),

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

only a small subset of the full momentum grid needs to be explicitly computed. The code determines this set via symmetry operations specified in the brillouin_zone module, which ensures compatibility with arbitrary lattice symmetries defined in the configuration. For instance, on a $16 \times 16 \times 1$ grid, only 45 out of 256 **q**-points are needed for a square lattice, which directly reduces both computational and memory load by more than a factor of five and even higher for larger grid sizes.
To further save memory, we keep most of the objects in half of their bosonic Matsubara frequency range: Due to the symmetry $F^{\omega\nu\nu'}_{1234} = (F^{(-\omega)(-\nu)(-\nu')}_{4321})^* = (F^{(-\omega)\nu\nu'}_{4321})^*$[2], we can restrict the objects to their positive bosonic Matsubara frequency region only. This saves half of the memory and thus a very large amount for the biggest objects, such as the auxiliary susceptibility or the full vertex.
These symmetry-based reductions are implemented in a transparent way for the user: the internal representation is compressed, but the high-level API provides access to the full data. This allows for both memory efficiency and usability, enabling the user to work with large systems that would otherwise be intractable on a single machine.

1.1.3 Object-oriented API

We tried to keep a very object-oriented approach when developing the toolbox for the program, as opposed to both DGApy and AbinitioDGA. Hence instead of directly working with NumPy arrays, we work with (Local)FourPoint, (Local)Interaction, SelfEnergy, GreensFunction and GapFunction classes. The reason being that in Python, it is very easy to overload operators, i.e. with so-called "dunder" methods, hence we implemented the most common operators "+/-/@/~"[3], alongside multiplication and division with numbers, for these objects to help make the implementation of equations easier. This might not be an 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 and only has to be coded once. We will discuss the explicit implementation of these dunder methods later.

Compared to DGApy, we stripped down the module responsible for the Matsubara frequency handling, since most of the features now can be called directly from the objects themselves. Instead of calling a method from the matsubara_frequencies module to cut an objects' frequency range, it is now a method on the object itself, e.g., object.cut_niv(10). Most of these methods are only implemented once in a base class which all subtypes inherit from. These base classes already cover most of the objects' functionality, where the subclass only implements features unique to this object type.

All modules should be very self-explanatory, their names already provide a good description of what the module does and what its responsibilities are. For any details, we refer the reader to the documentation of the code inside the repository.

One benefit of the object-oriented approach is the now easy implementation of various equations. For example, writing an arbitrary equation with arbitrary (non-local) vertex functions A, B, C and D like

$$D^{q\nu\nu'}_{1234} = \frac{1}{2\beta} \sum_{\nu_1;ab} A^{q\nu\nu_1}_{12ab} \left(B^{\omega\nu_1}_{ba34} + C^{\omega\nu_1\nu'}_{ba34} \right) \tag{1.1}$$

[2]The transformation from $-\nu, -\nu'$ to ν, ν' is possible due to the symmetrized nature of the two-particle Green's function and the inherited vertex functions.

[3]Corresponding to addition, subtraction, multiplication and inversion.

can be done with a call of D = 1 / (2 * beta) * A.matmul(B.add(C)) or the dunder methods for matrix multiplication "@" and addition "+", i.e., D = 1 / (2 * beta) * A @ (B + C), where the index contraction, frequency handling and summation will be done in the background. This eases readability in the code and reduces redundancy. Furthermore, it allows for very straightforward optimization of contractions and multiplications through this architecture. Furthermore, inverting a vertex function is also very easy now, all that needs to be called is the obj.invert() function (or with a "~" prefix, i.e. ~obj) on any object and the code performs the inversion in the background using NumPy's vectorization and SciPy's memory-optimized inversion by converting the object to compound indices. Furthermore, other operations, such as addition/subtraction in the form of A.add(B)/A.sub(B) or just simply $A \pm B$ are possible and automatically handle different amounts of frequency dimensions etc. for you. There are a bunch of other convenience methods available, where we would like to refer the interested reader to the code and its documentation for more details.

1.1.4 Setup and execution

The code is very easy to set-up as it is available as an installable Python package, called scdga. After installing the necessary packages denoted in the requirements.txt file to the Python environment, one simply has to run "pip install -e ." from the repository directory in the terminal to install the package in editable mode. This allows the user to now access any module and class inside the scdga package. The main entry point to the program is the file dga_main.py file, which can be started with either "python dga_main.py" for single-core execution (mostly used for testing purposes) or "mpiexec -np <n_proc> python dga_main.py" for multi-core processing. There are two additional command line parameters available to append to the execution:

- "-p": With "-p <path>" you can specify the path to the configuration file, which contains all run-specific parameters. This is useful if one wants to store multiple configuration files in different directories. If this parameter is not set, the path defaults to the location of the repository directory.

- "-c": With "-c <config name>" you can specify the name of the configuration file you want to load. This defaults to dga_config.yaml.

As an example, the following shell command runs the code using 8 MPI processes and loads the configuration file my_config.yaml from the path /configs/:
"mpiexec -np 8 python dga_main.py -p /configs/ -c my_config.yaml".
Additionally, we feature extensive logging: every important step in the calculation will be logged. If it is started from a terminal, the logging will be done to the standard output. If the code is executed on a slurm-based cluster, you will find the logs in the job output file. The reason we employ a lot of logging is the ease of finding errors that might occur during a calculation.

1.1.5 Configuration

The configuration file is split into small blocks of configuration sections, which will be explained in detail in the following. For those entries where only a single data type is expected, the code will try to parse the input to this specific type. If the parsing was unsuccessful, a warning is logged and a default value for this variable is used.
The first block of the configuration is concerned with the number of Matsubara frequencies one wants to use for the calculation:

```
box_sizes:
  niw_core: 50  # int, default: -1
  niv_core: 30  # int, default: -1
  niv_shell: 20 # int, default:  0
```

The "core" region defines the frequency box used for explicitly solving the Bethe-Salpeter and Schwinger-Dyson equations, while the "shell" region sets the size of the asymptotic tails needed for vertex reconstruction through the method described in Sec. ??. It is possible to set the core frequencies to "-1", which means that the number of Matsubara frequencies from the DMFT calculation will be taken.

The next section is concerned with the Hamiltonian of the system, which currently is a little bit complex, since we wanted to adapt the existing configuration file from DGApy and added new features. It defines the lattice symmetries, the momentum grid sizes and the kinetic and interaction part of the Hamiltonian:

```
lattice:
  symmetries: "two_dimensional_square"
                           # string, default: "two_dimensional_square"
  type: "from_wannier90"    # string, default: "from_wannier90"
  hr_input: "/path/to/file" # string | list[float], default: None
  interaction_type: "local_from_dmft"
                           # string, default: "local_from_dmft"
  interaction_input: "/path/to/file"
                           # string | list[float], default: None
  nk: [16, 16, 1]          # list[int], default: [16, 16, 1]
  nq: [16, 16, 1]          # list[int], default: [16, 16, 1]
```

Here, we have a bunch of settings available that can be combined in different ways. First, we can define the lattice symmetry, where one has to enter a set of predefined symmetry sets. It is possible to extend the symmetries the code supports by adding the corresponding symmetry operations and symmetry sets in the code. The available symmetry sets are two_dimensional_square, quasi_one_dimensional_square, simultaneous_x_y_inversion and quasi_two_dimensional_square_symmetries. Next is the type of input to the kinetic part of the Hamiltonian we require. It can either be from_wannier90, from_wannierHK or t_tp_tpp (only for single-band input). The first input type is used if the subsequent field hr_input references to a file, where the hopping elements are written in real space, whereas from_wannierHK is used if the Hamiltonian in Fourier space is available as a file[4]. The last option is only used for single-band input, where the band distribution is that of a t, t' and t'' hopping model only. If this type is specified, then a list of three float values in hr_input is expected.
Since this code is capable of performing multi-orbital calculations with non-local interactions, we require the definition of a interaction type and an input. The available interaction types are local_from_dmft, kanamori_from_dmft, kanamori and custom. If local_from_dmft is specified, we expect a single-band input, where the interaction strength is taken from DMFT. Similar to that, the kanamori_from_dmft option takes the interaction values from DMFT and constructs the correct interaction matrix with the Kanamori-specific entries. In both cases, the field interaction_input will not be read. If kanamori is specified, then we expect a list of four values in interaction_input, which contains the number of orbitals as the first entry and then U, J and V as the next three entries. One can enter whole numbers for the number of bands and floating point values for the interaction strengths. Lastly, if one specifies custom as the interaction type, we then expect a path to a file, which is structured similarly to a real-space

[4] Note that for the former, it is possible to use arbitrary $\mathbf{k}$-grid sizes, since the Fourier transform will be performed in the code, whereas for the latter one the correct grid size has to be entered.

Hamiltonian file but with four orbital entries instead of just two. It is possible to include non-local interactions there. Furthermore, it is possible to set the sizes of both the **k**- and **q**-grid in the fields nk and nq, respectively. It is possible to not set nq, where nk will be used for the **q**-grid. Note, that the execution of the Eliashberg equation is only possible if both the **k**- and the **q**-grid are of the same size.

The next configuration section is concerned with the self-consistency calculation and defines the convergence behavior and mixing strategy of the self-consistency loop:

```
self_consistency:
  max_iter: 20                    # int, default: 20
  save_iter: True                 # bool, default: True
  epsilon: 1e-6                   # float, default: 1e-4
  mixing: 0.3                     # float, default: 0.2
  mixing_strategy: "pulay"        # string, default: "linear"
  mixing_history_length: 2        # int, default: 3
  previous_sc_path: "path/to/files" # string, default: "./"
```

The field max_iter is to set an upper boundary on the number of iterations during the self-consistency cycle. If the self-energy does not converge within max_iter iterations, the iteration will stop. When setting save_iter to True, the code will save the non-local DΓA self-energy for each iteration. The next parameter, epsilon, is set to determine the absolute value difference allowed for the last two calculated sigmas to be considered converged. The convergence will be tested on a predefined set of **k**-points and frequencies ν. Next, we can specify the mixing parameter that will be used in the mixing scheme and is a floating point number between $(0, 1]$. We can furthermore specify the mixing scheme in the next parameter, called mixing_strategy. Here, we can choose from either linear or pulay mixing, where if pulay is specified, it takes mixing_history_length number of previous self-energy results to construct a prediction. If the number of iterations is less than the history length, we will use linear mixing for these iterations and switch to Pulay mixing afterwards. The last parameter, previous_sc_path allows us to specify the path to a previous self-consistency iteration folder, which might not have been converged or which needs to be converged further or with different parameters. The program then takes the last iterations of the already performed calculation and uses these as a starting point for the next calculation. If the number of iterations from the previous calculation is equal or larger than mixing_history_length and pulay is specified, then Pulay mixing will be applied, otherwise we mix linearly.

Given the improved speed and scalability of this code compared to DGApy, we have also implemented two λ-correction schemes which only work for single-band cases and can be configured with the following short section:

```
lambda_correction:
  perform_lambda_correction: True # bool, default: False
  type: "spch"                    # string, default: "spch"
```

Here, setting perform_lambda_correction to True will perform a λ-correction if the input is a single-band model. Otherwise, an error is raised and the code exits. Notice that if one wants a one-shot lambda-corrected self-energy and calculation of the Eliashberg equation, then max_iter and mixing in the self_consistency section need to be set to "1". The other parameter is the type of λ-correction which will be performed. Setting this to spch will renormalize both the density and magnetic susceptibilities, whereas setting it to sp will only renormalize the magnetic susceptibility.

The next section is a very important one, since it is concerned with the location of the DMFT files and mainly defines their location and file names:

```
dmft_input:
  type: "w2dyn"                 # string, default: "w2dyn"
  input_path: "/path/to/files"  # string, default: "./"
  fname_1p: "filename"          # string, default: "1p-data.hdf5"
  fname_2p: "filename"          # string, default: "g4iw_sym.hdf5"
  do_sym_v_vp: True             # bool, default: True
```

Currently, the type argument only supports w2dyn as input type, but this could be extended in the future to also support input files from other sources with other file structures. Next, input_path needs to be set to the folder that contains the output data from w2dynamics. We require two files, one containing one-particle quantities (such as the DMFT Green's function and self-energy) and one containing two-particle quantities (such as the two-particle Green's function); their filenames can be set in fname_1p and fname_1p, respectively. Notice that w2dynamics outputs a large Vertex.hdf5-file, where we have to extract the relevant worm components beforehand. This can be done with the script symmetrize.py, which is also part of the repository. This will automatically extract all relevant worm components of the two-particle Green's function and writes it to a separate file containing the density and magnetic contributions. Lastly, we have the option to symmetrize the two-particle Green's function in ν and ν' before processing them further. This might be necessary sometimes to ensure numerical symmetry of the vertex functions.

The second to last configuration section we will cover here is the output section which, as the name already suggests, covers the program's output settings, such as the output path and plotting instructions:

```
output:
  output_path: "/path"               # string, default: "./"
  do_plotting: True                  # bool, default: "True"
  plotting_subfolder_name: "Plots"   # string, default: "Plots"
  save_quantities: True              # bool, default: True
```

We are able to choose the path where the output of the program is saved to. If this field is chosen empty, the output will be put into a sub-folder inside the DMFT input folder. If we specify do_plotting, then we plot a selective set of quantities and save them to the sub-folder specified in plotting_subfolder_name. We then have the possibility to additionally save almost all quantities that are calculated during the main program as numpy files, except for the pairing vertex and the full ladder-DΓA vertex. These are separately handled in the Eliashberg section, since they are only calculated if you want to perform the calculation of the superconducting eigenvalue and gap function.

Finally, the last configuration section is concerned with the Eliashberg equation and is configuring the eigenvalue and gap function-finding through a Lanczos method:

```
eliashberg:
  perform_eliashberg: True      # bool, default: False
  save_pairing_vertex: False    # bool, default: False
  save_fq: False                # bool, default: False
  n_eig: 2                      # int, default: 2
  epsilon: 1e-9                 # float, default: 1e-6
  symmetry: "random"            # string, default: "random"
  include_local_part: True      # bool, default: True
  subfolder_name: "Eliashberg"  # string, default: "Eliashberg"
```

Here, the first setting is to let the program know if you want to perform the Eliashberg equation to obtain the superconducting singlet and triplet eigenvalues and gap functions, respectively. If this setting is set to "False", then the program will exit after the self-energy has been calculated. The next two settings allow to save the pairing vertex $\Gamma^{(q=0)kk'}$ and the full ladder vertex $F^{q\nu\nu'}$ to a file. These

are set to “False” per default, as these objects can be quite large, especially the full vertex, which can reach up to hundreds of gigabytes. Both will be saved for the irreducible Brillouin zone only to save memory. The following three settings, n_eig, epsilon and symmetry are concerned with the power iteration that is performed to solve the Eliashberg equation. Here, it is possible to specify the number of eigenvalues and eigenvectors one wants to calculate in n_eig. Notice, that we will always calculate n_eig+1 eigenvalues. This is due to the fact that first we calculate the single largest eigenvalue with a Lanczos method and then calculate the n_eig eigenvalues that are closest to one using shift-invert mode. Most of the time, however, the eigenvalues of both calculations are equal. epsilon is the setting that determines the target precision of the power iteration which is performed and symmetry determines the starting vector. There are a couple of options available for symmetry that may help speeding up the power iteration convergence if one knows the pairing symmetry of the gap function a priori: random, p-wave-x, p-wave-y and d-wave. In almost all cases it should suffice to pick a random starting vector for the power iteration.
In principle one also needs the *purely local* full and reducible vertices for the pairing vertex, however these can be omitted if one expects d-wave symmetry of the gap function, where the local vertex functions do not contribute to the eigenvalue and gap function due to symmetry. Setting include_local_part to “False” does exactly that. Not calculating the local part allows for a slightly larger frequency box and a better estimation of the eigenvalue and gap function for d-wave symmetry cases.
The last setting is the sub-folder name where the Eliashberg output will be saved to.

1.1.6 Input files

After the vertex calculation has been performed with w2dynamics [**Wallerberger2019**], 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 located in the repository, 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 no file name is fixed and they can be specified in the corresponding dga_config.yaml file.

1.2 Result validation

In order to check if the program’s output is correct, we created a few simple test cases to validate our results:

- A single-band test, where the code should reproduce the results of DGApy
- A two-band test with the same parameter as the case above, where each band is indistinguishable and electrons in one band do not interact with electrons in the other band. The outputs should be equal to the single-band case in two orbital components and zero elsewhere.
- A rotation of the previous test-case. When rotating the DMFT input data, i.e. the local one- and two-particle Green’s function and the self-energy, and the Hamiltonian in orbital space, we

expect the output to produce the same results as the second test case after a orbital rotation in the opposite direction.