Michael Draxler, BSc

Development of a Python toolbox for the drift kinetic equation solver NEO-2

Master's Thesis

to achieve the university degree of

Diplom-Ingenieur

Master's degree programme: Technical Physics

submitted to

Graz University of Technology

Supervisor

Ass.Prof. Dipl.-Ing. Dr.techn. Winfried Kernbichler

Institute of Theoretical and Computational Physics

Graz, January 2021

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe. Das in TUGRAZonline hochgeladene Textdokument ist mit der vorliegenden Masterarbeit identisch.

Datum	Unterschrift

Kurzfassung

Kernfusionsreaktoren haben das Potenzial, große Mengen an nachhaltig produzierter Energie zu liefern. Allerdings ist diese vielversprechende Technologie noch nicht weit genug fortgeschritten, um in großem Maßstab elektrische Energie zu produzieren. Für einen stabilen Betrieb sind noch erhebliche Verbesserungen in vielen verschiedenen Bereichen erforderlich. Diese Arbeit soll zur Forschung im Bereich der Kernfusion beitragen, indem Möglichkeiten für einen einfachen Einstieg in dieses Gebiet aufgezeigt werden.

Diese Arbeit stellt verschiedene Werkzeuge zur Verfügung, um die Anwendung eines etablierten Simulationspakets (NEO-2) zu erleichtern. NEO-2 löst die driftkinetische Gleichung und ermöglicht ein besseres Verständnis der Transportmechanismen in Fusionsreaktoren. Im Rahmen dieser Arbeit wurde ein Python Paket implementiert, welches das Open-Source-Tool Jupyter Notebook mit NEO-2 verbindet. Das entwickelte Python Paket wird neo2tools genannt. Neo2tools bietet eine Schnittstelle, die ein einfaches Aufsetzen und Starten von NEO-2 Simulationen für eine Reihe häufiger Anwendungsfälle ermöglicht. Zusätzlich werden Methoden zur Visualisierung von NEO-2 Ergebnissen, bis zur Ebene der Verteilungsfunktion, vorgestellt. Die interaktiven Funktionen von Jupyter Notebook werden genutzt, um die erzeugten Daten auf intuitive Weise zu präsentieren.

Abstract

Nuclear fusion reactors have the potential to provide large amounts of energy in a sustainable fashion. However, this promising technology is not yet advanced enough to serve as an energy source on a large scale. Significant improvement in many different fields is still required for a steady state operation. This thesis aims to contribute to the fusion related research activities by lowering the barriers to get active in this field.

This thesis provides different tools to facilitate the application of an established simulation package (NEO-2). NEO-2 provides a solver for the drift kinetic equation and enables a better understanding of transport mechanisms in fusion reactors. During this thesis a Python package, which connects the popular open source tool Jupyter Notebook with NEO-2, was implemented. The developed Python toolbox is called neo2tools. Neo2tools provides an interface, that enables straight forward deployment of NEO-2 runs for a number of common use cases. Additionally, methods to visualize the results of NEO-2 runs, up to the level of the distribution function, are provided within neo2tools. The interactive features of Jupyter Notebook are utilized to present the produced data in an intuitive way.

Contents

1 Introduction

In the last decades, the emission of greenhouse gases and the correlated anthropogenic climate change has been causing rising temperature all around the world [1]. This development is accelerated by the rapid growth of population and wealth and the corresponding increasing energy demand.

Fossil fuels cannot be the right means to supply this demand, due to the emission of greenhouse gases and their limited abundance on our planet.

For a sustainable world, the primary energy source has to be renewable. Existing renewable energy sources, like solar and wind, can already supply large amounts of energy. However, the power supplied by these sources strongly relies on external influences like weather conditions that cannot be controlled. Peaks of high-energy demand do generally not match with peaks of energy production of these fluctuating sources. Therefore, a significant part of energy production must be covered by on-demand energy sources, like the combustion of fossil fuels and nuclear fission.

Nuclear fission enables the controllable supply of energy without the emission of greenhouse gases, but the problem of the storage of the nuclear waste with long half-life periods remains unsolved.

Nuclear fusion has the potential to supply large amounts of energy in a controllable way, without emitting greenhouse gases and burdening future generations with radioactive waste.

Because of the large potential of the promising nuclear fusion technology, large efforts are made to make it a useable energy source. A promising concept is magnetically confined fusion. Recently two different approaches to magnetically confined fusion were realized on a large scale. Iter, in the south of France, is a fusion reactor based on the tokamak approach and Wendelstein 7x, in northeast Germany, relies on the stellarator approach. Both of these facilities are research objects and cannot be operated steady state for energy production.

To make this technology useable, significant progress in many fields is still necessary. Problems still to be solved include, precise control of the plasma, huge thermal stress on the walls, fuel and energy injection and waste depositing, and the fuel production for a steady-state operation. The research to overcome these problems is not restricted to experiments but is always led by simulation results.

In this thesis, the framework for simulations, to improve the precise control of the plasma, was implemented. It was implemented using open-source software. The fact that open-source software is easily available and free of charge makes it a well-suited tool for international research projects in a globalized world. The implemented framework facilitates the application of an established simulation package and lowers the barriers to get active in the field of nuclear fusion related simulations.

One fundamental equation to describe the plasma is the drift kinetic equation. It follows from the neoclassical transport theory [2]. The code NEO-2 is developed to apply this theory to calculate transport in fusion reactors [3]. It is a cooperation of the Institute of Theoretical and Computational Physics at Graz University of Technology, Austria and the National Science Center Kharkov Institute of Physics and Technology, Ukraine.

The scope of this work was to build a Python toolbox for the NEO-2 code. It enables preprocessing and postprocessing of NEO-2 calculations, generation of meaningful data by consecutive application, and comprehensive graphical representation of data for further analysis.

2 Theoretical background

The derivations of this section follow in large parts the books of P.Helander [2] and D'haeseleer [4]. In Section 2.1 the basics of a coordinate system in general and the Boozer coordinate system, in particular, are explained. In Section 2.2 the main quantities of the neoclassical theory are derived.

2.1 Coordinates

Depending on the geometry of a specific problem choosing one or another coordinate system can make calculations more or less cumbersome. Devices in magnetically confined fusion physics have a toroidal geometry. Therefore, toroidal coordinates offer a good possibility to describe problems in such devices.

Tokamaks are built axisymmetrically and the magnetic field is treated axisymmetrically. Sometimes small non-axisymmetric perturbations are added. Of course, these coordinate systems can also be used in stellarators, which are not axisymmetric but possess periodicity in the toroidal direction.

An additional useful concept is flux coordinates. Flux coordinates are oriented on flux surfaces. These flux surfaces are labeled. This flux surface labeling is comparable to the small radius in a toroidal description. A special set of flux coordinates are straight magnetic field line coordinates and one special set of straight magnetic field line coordinates are Boozer coordinates. Flux coordinates are well suited to describe the non-axisymmetric plasma in a stellarator, but can also be used to describe the plasma in a tokamak.

Any spatial vector can be described in a variety of different coordinate systems. Generally, a vector $\mathbf{r}$ in 3D space can be described by three coordinates:

$$\mathbf{r} = \mathbf{r}(u^1, u^2, u^3). \tag{2.1}$$

These coordinates have to be linearly independent to reach every spatial point with a weighting of these coordinates in exactly one combination.

These three coordinates define the basis vectors. There are two possibilities to define the basis vectors, co-variant and contra-variant basis vectors. Both basis will later be used to describe the magnetic field. The co-variant vector can be described as a tangent vector. The co-variant basis vectors are defined as

$$\mathbf{e}_1 = \frac{\partial \mathbf{r}}{\partial u^1}, \quad \mathbf{e}_2 = \frac{\partial \mathbf{r}}{\partial u^2}, \quad \mathbf{e}_3 = \frac{\partial \mathbf{r}}{\partial u^3}. \tag{2.2}$$

The basis vector $\mathbf{e}_i$ is tangent and is called a co-variant basis vector. This can be seen as $\mathbf{e}_i$ is parallel to the curve described by u^i.

The contra-variant basis vector can be described by the coordinate surface. The other two coordinates define the surface and the contra-variant basis vector is perpendicular to this surface. The contra-variant basis vectors can be written with the gradients

$$\mathbf{e}^1 = \nabla u^1, \quad \mathbf{e}^2 = \nabla u^2, \quad \mathbf{e}^3 = \nabla u^3. \tag{2.3}$$

These two basis vector sets are connected and called reciprocal sets of vectors. The transformation of a co-variant basis to a contra-variant basis is done with

$$\mathbf{e}^i = \frac{\mathbf{e}_j \times \mathbf{e}_k}{\sqrt{g}} \tag{2.4}$$

and vice versa:

$$\mathbf{e}_i = \sqrt{g} \cdot \mathbf{e}^j \times \mathbf{e}^k, \tag{2.5}$$

with (i, j, k) as the three cyclic indices for the three dimensions, and with $\sqrt{g}$ as the metric determinant.

The metric determinant can also be written as

$$\sqrt{g} = \left| \frac{\partial \mathbf{r}}{x^i} \right|. \tag{2.6}$$

A generic vector field can then be described by

$$\mathbf{D}(\mathbf{r}) = D^i(\mathbf{r})\mathbf{e_i}(\mathbf{r}) = D_i(\mathbf{r})\mathbf{e^i}(\mathbf{r}) \tag{2.7}$$

with summation over the index i.

So far, only basis vectors have been used. In Equation 2.7 vector components have been introduced. The quantities

$$D^i = \mathbf{D} \cdot \mathbf{e}^i \tag{2.8}$$

are the contra-variant components and

$$D_i = \mathbf{D} \cdot \mathbf{e}_i \tag{2.9}$$

are the co-variant components.

2.1.1 Boozer coordinates

This subsection aims to formulate the magnetic field in Boozer coordinates. Because the magnetic field is a vector field, the magnetic field **B** can be written in two representations as Equation 2.7 demonstrates.

The vector of the magnetic field line is described by the position vector **r**, such that

$$\mathbf{B} = c \cdot \mathrm{d}\mathbf{r}, \tag{2.10}$$

with c as a constant.

If the magnetic field line is followed long enough, it will either ergodically fill a magnetic surface or the magnetic field line will be closed at some point. In the latter case, the magnetic field line will still on this described magnetic surface.

The next step is to define such a surface properly. The force balance [5] in the magneto-hydrodynamic equilibrium is

$$\mathbf{j} \times \mathbf{B} = \nabla p, \tag{2.11}$$

with **j** as the current density and p as the pressure.

By multiplying Equation 2.11 with **B** and applying a little vector calculus,

$$\mathbf{B}\nabla p = 0. \tag{2.12}$$

This leads to the consequence that flux surfaces of constant pressure exist. These surfaces of constant pressure define the first coordinate of the Boozer coordinates.

Through stellarator optimization and due to the axisymmetric nature of tokamaks, it it is implied, that these flux surfaces are nested. Although in general, magnetic islands may disturb this assumption [6].

Boozer coordinates are written as

$$\mathbf{r}(u^1, u^2, u^3) = \mathbf{r}(r, \vartheta, \varphi). \tag{2.13}$$

In comparison to toroidal coordinates, Boozer coordinates look similar but in a deformed way. Toroidal coordinates are orthogonal, but Boozer coordinates are not. r acts like the small radius. φ and ϑ are angle-like variables.

In Boozer coordinates, different surfaces can be defined. With the definition of the surfaces $S_{tor}, S^r_{pol}, S^d_{pol},$ from ref. [4], the corresponding magnetic fluxes Ψ_{tor}, Ψ^r_{pol}, Ψ^d_{pol}, are calculated:

$$\Psi = \iint_S \mathbf{B} d\mathbf{S}. \tag{2.14}$$

The currents $I_{tor}, I^r_{pol}, I^d_{pol},$ within these surfaces can also be calculated:

$$I = \iint_S \mathbf{j} d\mathbf{S}. \tag{2.15}$$

A magnetic field with the Boozer coordinates in the contra-variant form can be written as

$$\mathbf{B} = B^\vartheta \mathbf{e}_\vartheta + B^\varphi \mathbf{e}_\varphi. \tag{2.16}$$

It should be noted that $B^r = 0$ and B^r is therefore missing because as a consequence of Equation 2.10.

With the flux from Equation 2.14 and their corresponding surfaces, the contra-variant components of the magnetic field are written as:

$$B^\vartheta = \frac{1}{2\pi\sqrt{g}} \frac{\partial \Psi^r_{pol}}{\partial r}, \quad B^\varphi = \frac{1}{2\pi\sqrt{g}} \frac{\partial \Psi_{tor}}{\partial r} \tag{2.17}$$

Boozer coordinates are straight field line coordinates and therefore the magnetic field line appears as a straight line. The safety factor q, which represents the twisting of the magnetic field lines, is defined through the ratio

$$q(r) = \frac{\mathrm{d}\Psi_{tor}}{\mathrm{d}\Psi_{pol}}, \tag{2.18}$$

which is equal to

$$q(r) = \frac{B^{\varphi}}{B^{\vartheta}}. \tag{2.19}$$

The magnetic field in the co-variant form is represented by

$$\mathbf{B} = B_r \nabla r + B_{\vartheta} \nabla \vartheta + B_{\varphi} \nabla \varphi, \tag{2.20}$$

containing the components

$$B_r = -\frac{4\pi}{c}\eta + \frac{\partial \Phi_m}{\partial r}, \quad B_{\vartheta} = \frac{2}{c} I_{tor} + \frac{\partial \Phi_m}{\partial \vartheta}, \quad B_{\varphi} = \frac{2 I^d_{pol}}{c} + \frac{\partial \Phi_m}{\partial \varphi}, \tag{2.21}$$

with the speed of light c, the normalized invariant η and Φ_m as the scalar magnetic potential, which is an integration constant defined from the boundary conditions.

2.2 Drift kinetic equation

In this section, the basics of the neoclassical transport ansatz is explained. Therefore, the drift kinetic equation will be derived. The result will be the neoclassical transport matrix and the derived neoclassical fluxes.

A particle can be exactly described through its position $\mathbf{r}$ and its velocity $\mathbf{v}$ at time t. Combining position and velocity gives the six-dimensional phase-space $\mathbf{z} = (\mathbf{r}, \mathbf{v})$. To describe a large number of particles of the same kind, a distribution function $f_a(\mathbf{z}, t)$ is used, where you get the particle density of species a, in a point of the six-dimensional phase-space $\mathbf{z}$ at time t.

Using Gauss's theorem a continuity equation can be written, when no collisions appear:

$$\frac{\partial f_a(\mathbf{z}, t)}{\partial t} + \frac{\partial}{\partial \mathbf{r}}(\dot{\mathbf{r}} f_a(\mathbf{r}, t)) + \frac{\partial}{\partial \mathbf{v}}(\dot{\mathbf{v}} f_a(\mathbf{v}, t)) = 0. \tag{2.22}$$

When collisions appear, Equation 2.22 is not valid anymore. To compensate for that, a Coloumb collision operator $C(f)$ on the right side has to be introduced:

$$\frac{\partial f_a(\mathbf{z}, t)}{\partial t} + \frac{\partial}{\partial \mathbf{r}}(\dot{\mathbf{r}} f_a(\mathbf{r}, t)) + \frac{\partial}{\partial \mathbf{v}}(\dot{\mathbf{v}} f_a(\mathbf{v}, t)) = C(f). \tag{2.23}$$

Equation 2.23 is called the Fokker-Planck equation. The contributions to the collision operator are now explained. Equation 2.22 is not valid for Coulomb collisions, because little jumps in the velocity occur. The Coulomb collisions do not affect the position. Looking closer at a collision, and comparing the situation before and after a collision, the velocity jumps immediately from one value to another without going through the velocities between. These jumps in the velocity space generate rather small holes, but they still have to be taken into account. These holes in velocity space have now to be filled up. Therefore a series expansion for the flux in velocity space ($\mathbf{j} = \dot{\mathbf{v}} f_a$) is made:

$$j_i f = a_i f + b_{ij} \frac{\partial f}{\partial v_j} + c_{ijk} \frac{\partial^2 f}{\partial v_j \partial v_k} + ..., \tag{2.24}$$

with $a_i, b_{ij}, c_{ijk}...$ as coefficients of the series expansion and j_i as the i-th component of the flux $\mathbf{j}$.

The zeroth and the first order of Equation 2.24 satisfy the required accuracy. Comparing the second term of the continuity condition in

Equation 2.22, with the zeroth-order expansion of Equation 2.24 follows that the zeroth-order expansion represents the continuous flow. The first-order expansion can be interpreted as a diffusion operator D, which is a tensor in general. Rewriting Equation 2.24 with these changes gives

$$\mathbf{j} = \dot{\mathbf{v}} f_a - D \nabla_v f_a. \tag{2.25}$$

The quantity $\dot{\mathbf{v}}$ can be expressed by the external force $\mathbf{F}_a^{(e)}$, which acts on the particles:

$$\dot{\mathbf{v}} = \frac{\mathbf{F}_a^{(e)}}{m_a}, \tag{2.26}$$

with m_a as the mass of the particle species a.

With the first-order correction of the flow, the force, which acts on the particles, has also to be changed. In comparison to Equation 2.26 a friction force $\mathbf{F}_{col}$ due to the collisions was introduced:

$$\dot{\mathbf{v}} = \frac{1}{m_a}(\mathbf{F}_a^{(e)} + \mathbf{F}_{col}). \tag{2.27}$$

Grouping the collision terms of Equation 2.25 and Equation 2.27 to the collision operator $C(f_a)$, results in Equation 2.23.

For further simplifications, the neoclassical ordering is introduced. Due to the Lorentz force, particles gyrate around the magnetic field lines. This gyration leads to a complex description of the particles' trajectories. In the guiding center formalism, only the trajectory of the guiding center is described. This formalism is simpler than describing the whole gyration motion and the wanted derived quantities are not sufficiently affected by this simplification. This simplification is possible because different physical effects play a different role on different length scales. By introducing an ordering parameter ϵ, these different length scales are formalized [7].

It is defined as

$$\epsilon = \frac{\rho}{L}, \tag{2.28}$$

with L as the macroscopic scale length, e.g. the major radius of a tokamak and ρ, as the Larmor radius

$$\rho = \frac{m_a v_{T_a}}{e_a B}, \tag{2.29}$$

with v_{T_a} as thermal velocity and e_a as the charge of particles a.

The Larmor radius is in the range of μm for electrons and L is in the range of meters. Therefore $\epsilon << 1$. The distribution function can be expanded in powers of ϵ:

$$f_a = f_{a_0} + f_{a_1} + \mathcal{O}(\epsilon^2). \tag{2.30}$$

The subscript denotes the order of ϵ. The expansion is truncated at $\mathcal{O}(\epsilon^2)$.

To describe the dynamics of the particles, it is useful to formulate a Lagrangian. For the guiding-center description of the particles and using the ordering parameter ϵ, a Lagrangian was found by Littlejohn [7].

With this Lagrangian, Euler-Lagrange equations can be formulated, resulting in the equations of motions. In a slightly different notation than that of LittleJohn, Kapper derived it very detailed in [8]. The latter notation is used in this thesis.

The result of the six equations of motion is

$$\dot{\mathbf{r}}_g = \mathbf{v}_g, \qquad \dot{J}_\perp = 0, \qquad \dot{\phi} = \frac{-\omega_{c,a}}{\epsilon}, \qquad \dot{\omega} = e_a \dot{\mathbf{r}}_g \mathbf{E}^{(A)}, \tag{2.31}$$

with three equations packed in the vector formalism in the first term. For the velocity space $\mathbf{v}$, the perpendicular adiabatic invariant $J_\perp$, the gyrophase ϕ, and ω as the total energy are introduced. $\mathbf{E}^{(A)}$ is the electric field and $\omega_{c,a}$ as the cyclotron frequency. The subscript g stands for the guiding center description.

The distribution function f_a still depends on six coordinates. There was no constraint on the spatial coordinates. But straight field line flux coordinates $\mathbf{r}_g(r, \vartheta, \varphi,)$ as explained in Subsection 2.1.1 will be used. With these equations of motions, Equation 2.23 can be written as

$$\frac{\partial f_a}{\partial t} + \dot{\mathbf{r}}_g \cdot \nabla_{\mathbf{r}_g} f_a + \dot{\phi} \frac{\partial f_a}{\partial \phi} + \dot{\omega} \frac{\partial f_a}{\partial \omega} = C(f_a). \tag{2.32}$$

Analyzing the terms before each f_a in Equation 2.32 on which order of ϵ they depend on, results in

$$\begin{aligned} \frac{\partial}{\partial t} &\sim \epsilon^2, \\ \dot{\phi} &\sim \epsilon^{-1}, \\ v_g^{\varphi} &\sim \epsilon^0 + \epsilon^1, \\ v_g^{\vartheta} &\sim \epsilon^0 + \epsilon^1, \\ v_g^{r} &\sim \epsilon^1, \\ \dot{\omega} &\sim \epsilon^1. \end{aligned} \tag{2.33}$$

Because the gyration $\dot{\phi}$ is from the order ϵ^{-1}, it will be considered that it does not affect f_a. The term $\dot{\phi}\frac{\partial f_a}{\partial \phi}$ is considered to be 0 and $\dot{\omega}\frac{\partial f_a}{\partial \omega}$ and $C(f_a)$ are now gyro-averaged values. Equation 2.32 can then be written as

$$\frac{\partial f_a}{\partial t} + \dot{\mathbf{r}}_g \cdot \nabla_{\mathbf{r}_g} f_a + \dot{\omega} \frac{\partial f_a}{\partial \omega} = C(f_a). \tag{2.34}$$

The distribution function f_a now depends only on four coordinates anymore.

With the notation of $C(f_a)$ only collisions of particles a with particles a were considered. For a generalization, a second particle species b is introduced. The Stoss operator St represents the collisions between two species a and b. Together with the expansion from Equation 2.30, the two-particle interactions, written in terms of the Stoss operator St:

$$St(f_a, f_b) = St(f_{a_0}, f_{b_0}) + St(f_{a_1}, f_{b_0}) + St(f_{a_0}, f_{b_1}) + St(f_{a_1}, f_{b_1}). \quad (2.35)$$

$St(f_{a_0}, f_{b_0}) = 0$, if f_{a_0} and f_{b_0} have the same flow velocity and the same temperature. $St(f_{a_1}, f_{b_1})$ is from the order $\mathcal{O}(\epsilon^2)$ and therefore not further considered. Depending on which particle is interacting with which other particle, the collision operator can be a very complex problem. Various names and operators cover different physical effects, on handling and calculating the collision operator. [2].

The further considerations are done with the linearized collision operator:

$$\hat{L}_{C_L} f \equiv St(f_a, f_{b_0}) + St(f_{a_0}, f_b), \quad (2.36)$$

with a test particle part and a field particle part. The field particle part can be solved with the help of Rosenbluth potential as shown in [2].

The next steps are defining thermodynamic forces and sources and include them in Equation 2.34. For consistency with NEO-2, the test particle part is written with $\hat{L}_{ab}^{(D)}$ and the field particle part with $\hat{L}_{ab}^{(I)}$. Equation 2.36 is written as

$$\hat{L}_{C_L} f \equiv \hat{L}_{ab}^{(D)}(f_a, f_{b_0}) + \hat{L}_{ab}^{(I)}(f_{a_0}, f_b). \quad (2.37)$$

The zeroth-order drift kinetic equation is defined by only considering the zeroth order of ϵ in Equation 2.32:

$$\dot{\mathbf{r}}_g \cdot \nabla_{\mathbf{r}_g} f_0 = \hat{L}_{C_L} f_0. \quad (2.38)$$

Following the Boltzmann H-theorem the lowest order of the distribution function f_a must be a Maxwellian only depending on r and ω:

$$f_0 = f_M(r,\omega) = \frac{n}{\pi^{\frac{3}{2}} v_T^3} e^{-\frac{(\omega - e\phi)}{T}}, \tag{2.39}$$

with T as the temperature.The first-order drift kinetic equation is then written as

$$v_g^r \frac{\partial f_0}{\partial r} + v_g^\varphi \frac{\partial f_1}{\partial \varphi} + v_g^\vartheta \frac{\partial f_1}{\partial \vartheta} + \dot{\omega} \frac{\partial f_0}{\partial \omega} = \hat{L}_{C_L} f_1, \tag{2.40}$$

keeping in mind that f_1 has also an ϵ^1 dependence. The new operator

$$\hat{L} f_1 \equiv v_g^\varphi \frac{\partial f_1}{\partial \varphi} + v_g^\vartheta \frac{\partial f_1}{\partial \vartheta} - \hat{L}_{C_L} f_1, \tag{2.41}$$

is defined to simplify the notation. Constructing the total time derivative of $f_0(r,\omega)$ yields, together with Equation 2.41 and Equation 2.40

$$\dot{f}_0 = v_g^r \frac{\partial f_0}{\partial r} + \dot{\omega} \frac{\partial f_0}{\partial \omega} \equiv \hat{L} f_1. \tag{2.42}$$

The derivative of the Maxwellian can be written as

$$\begin{aligned} \dot{f}_M(r,\omega) &= v_g^r \left(\frac{\mathrm{mv}^2 - 3T}{2T^2} \cdot f_M \right) + \dot{\omega} \left(-\frac{1}{T} \cdot f_M \right) \\ &= -f_M \sum_{k=1}^{3} q_k A_k + \frac{e f_M}{T} v_\parallel h \cdot \nabla \delta\phi, \end{aligned} \tag{2.43}$$

with the thermodynamic forces A_1, A_2, and A_3:

$$A_1 = \frac{1}{n_a} \frac{\partial n_a}{\partial r} - \frac{e_a E_r}{T_a} - \frac{3}{2T_a} \frac{\partial T_a}{\partial r}, \qquad A_2 = \frac{1}{T_a} \frac{\partial T_a}{\partial r}, \qquad A_3 = \frac{e_a \langle E_\parallel B \rangle}{T_a \langle B^2 \rangle}, \tag{2.44}$$

where $\langle\rangle$ denotes the flux surface average and with the sources q_1,q_2, q_3:

$$q_1 = -v_g^r, \qquad q_2 = -\frac{m_a v^2}{2T_a} v_g^r, \qquad q_3 = v_\parallel B. \tag{2.45}$$

The second term in the thermodynamic force A_1 is here to take into account the precession of deeply trapped particles with $\cos\alpha \simeq 0$ [9] and does not derive directly from the Maxwellian, but from the gyro averaged collision operator.

f_1 can then be written as a superposition of solutions for each thermodynamic force:

$$f_1 = \sum_{k=1}^{3} f_{1,k} A_k - \frac{e\delta\phi f_M}{T}, \tag{2.46}$$

with

$$\hat{L} f_M f_{1,k} = q_k f_M. \tag{2.47}$$

Solving Equation 2.47 is a central task of the program package NEO-2, which will be explained in Section 3.1.

The magnetic differential equation of Equation 2.43,

$$h \cdot \nabla \delta\phi = B \frac{e_a \langle E_\parallel B \rangle}{\langle B^2 \rangle} - E_\parallel, \tag{2.48}$$

is separately solved.

The influence of the different sources q_j, to the different distribution function $f_{1,k}$ can be described by the 9 neoclassical diffusion coefficients:

$$D_{jk} = \frac{1}{n_a} \left\langle \int d^3v \; q_j f_M f_{1,k} \right\rangle. \tag{2.49}$$

The neoclassical diffusion coefficients are one result of the postprocessing part of neo2tools as described in Section 4.3. This derivation of D_{jk} only accounts for particles of the same species. The derivation of D_{jk}^{ab}, with a and b for different particle species can be found in [10].

Together with the thermodynamic forces A_k from Equation 2.44 one can reconstruct the thermodynamic flux

$$I_j = -n_a \sum_{k=1}^{3} D_{jk} A_k, \tag{2.50}$$

in which namely, I_1 is the particle flux density, I_2 the heat flux density and I_3 the parallel flow.

3 Computational principles

This thesis's main goal was to implement a NEO-2 interface in Python, which can be controlled through a Jupyter Notebook. In this chapter, the computational fundamentals for this thesis will be explained. These are, on the one hand the implementation of the NEO-2 code and, on the other hand, Python.

3.1 NEO-2

The code NEO-2 solves the drift kinetic equation and computes the neoclassical transport coefficients [3].

NEO-2 is a large Fortran project and has about 100000 lines of code in two main branches. Around 250 input parameters can be set. These input parameters are not all independent, and some parameters are already obsolete. Additionally, there are input files for different plasma profiles and magnetic fields.

The three-dimensional toroidal magnetic field configuration is normally passed to NEO-2 in magnetic flux coordinates, like Boozer coordinates, but NEO-2 can also handle magnetic field representations in real space coordinates. NEO-2 solves the simplified drift kinetic equation, shown in Equation 2.47. There, $f_{1,k}$ is the variable of interest. A variety of physical properties can be determined by integrating $f_{1,k}$, to obtain velocity space moments. Examples of these physical properties can be particle, energy, and heat fluxes. NEO-2 covers the flux surface with the magnetic field line integration technique [11]. To obtain a physical property, F, the flux surface average is built:

$$\langle F \rangle = \lim_{L\to\infty} \left(\int_0^L \frac{dl}{B} \right)^{-1} \int_0^L dl \frac{F}{B}, \tag{3.1}$$

where L is the length of the magnetic field line.

The safety factor q, from Equation 2.19, is the proportion of toroidal revolutions to poloidal revolutions of the field line. Therefore, the order of the proportion of q specifies the field line length.

In simulations, L is limited, but the length of the field line is so adjusted, that the field line closes smoothly. With a low-order rational q, the closing of the magnetic field line happens fast and the field line is short. With a high-order rational q, the field line will be accordingly longer. Too short field lines may falsify the flux surface average; too long field lines increase the computational costs. For that reason, a good adjustment of the magnetic field line length is essential.

To solve the drift kinetic equation, the magnetic field line is divided into so-called field periods. The drift kinetic equation is solved in these field periods separately [12].

NEO-2 works in a wide range of collisional regimes. Most codes in the low collisionality regime work with Monte Carlo methods. For high collisionality regimes, the neoclassical transport ansatz is often also used in other codes. For even higher collisionalities magnetohydrodynamic theory is applied. NEO-2 is based on the neoclassical transport ansatz.

At the trapped-passing boundary, a collision has the most impact because it decides if a particle after a collision is in a trapped or in a passing orbit. In a passing orbit, the probability of a collision is orders of magnitude lower than in a trapped orbit. Therefore, it is important to highly resolve the collisions near the trapped-passing boundary. On the other hand, collisions far away from the trapped-passing boundary do not need such a high resolution in the velocity space. After a collision, the particle will be in the trapped region anyway, because a collision does only change slightly the direction of the particle. In non-axisymmetric magnetic fields, there are many classes of trapped particles. Between two local maxima of the magnetic field strength, particles can be trapped. In addition, there it is important to highly resolve the transition layer between these various trapped particle classes. NEO-2 has an adaptive grid, that highly resolves such boundary layers. Outside of these boundary layers, memory is saved with a reduced resolution.

3.1.1 NEO-2 branches

Stellarators have 3D magnetic fields. Tokamaks have, due to the toroidal symmetric structure, 2D magnetic fields with small perturbations. To properly treat these two magnetic field geometries, NEO-2 is divided into two branches. Depending on how the drift kinetic equation is solved, either the parallel branch (NEO-2-PAR) [8] or the quasilinear branch (NEO-2-QL) [10] is used.

In 3D magnetic field configurations, NEO-2 is used to compute the neoclassical transport coefficients. This is handled with the parallel branch (NEO-2-PAR). For this computation, it is required to have a negligible poloidal drift of particles due to the radial electric field. Furthermore, it is possible to calculate the efficiency of microwave radiation absorption for the electron cyclotron resonance heating (ECRH). The generalized Spitzer function plays an important role in finite low-collisionality regimes to determine this efficiency. The use of the reconstruction run makes it possible to visualize the generalized Spitzer functions.

In nearly axisymmetric magnetic field configurations, like in tokamaks, NEO-2 uses a quasilinear approach (NEO-2-QL) to handle small non-axisymmetric magnetic field perturbations. The toroidal torque produced from this perturbation can be calculated. This toroidal torque is called neoclassical toroidal viscosity. In contrast to the calculations of stellarators, the computation of the neoclassical transport coefficients is not limited to the slow plasma rotation regime. The perturbation field and the employment of the quasilinear approach enable to solve non-slow plasma rotations.

3.1.2 NEO-2 input

The code NEO [11] is the predecessor of NEO-2. NEO-2 uses the magnetic field handling parts of NEO. To control the NEO parts, there is an input file named `neo.in`. Except for the unperturbed magnetic field settings, the settings in `neo.in` do not influence NEO-2. All other settings are handled in the `neo2.in` input file. The `neo2.in` input file is a Fortran input file and the command and control file for NEO-2. The input file is grouped in several namelists, and can be generally divided into two sections of settings. These two sections are the physical settings and numerical settings and do not necessarily coincide with the namelists' grouping.

In Section 2.2, the drift kinetic equation is split into three thermodynamic forces. Each thermodynamic force can be assigned to a physical effect. With the physical settings, every force can be individually controlled and turned on or off. One example is the parallel electric field acting on the particles. Other settings are the type of collision operator and the physical effects considered by the influence of small perturbed magnetic fields, or different plasma parameter settings.

The numerical settings control different numerical approximations. The type of basis functions used and the level placement for calculating the collision operator can be controlled. These mentioned settings constitute only a small fraction of the large number of settings in NEO-2. The two already mentioned magnetic fields, the unperturbed and the perturbed one, are passed to NEO-2 as separated files. In these magnetic field files, the toroidal magnetic field is usually described in Boozer coordinates. These magnetic field equilibria are calculated with the code `nemec` beforehand [13]. The perturbed magnetic field is only utilized from the NEO-2-QL version of the code. For multispecies calculations with the

NEO-2-QL version, each species' temperature and density distributions have to be passed.

3.1.3 Matrix elements of the collision operator

Parts of the collision operator can be represented in an arbitrary basis with respect to the velocity. In this representation, the collision operator can be expressed by matrix elements. The velocity distribution is expanded with the help of basis functions. This representation of parts of the collision operator makes it possible to precalculate these parts. Therefore, the calculation of the distribution function is facilitated.

As described in Equation 2.37 the linearized collision operator is separated into a test particle part $\hat{L}_{ab}^{(D)}$ and a field particle part $\hat{L}_{ab}^{(I)}$. In the linearized collision operator, the particles, momentum, and energy are conserved and not lost in the linearization process [2]. Using the linearization of the collision operator, the drift kinetic equation can be written as

$$v\lambda h^{\vartheta} f_{a_0} \frac{\partial g_a}{\partial \vartheta} + i\omega f_{a_0} g_a - \sum_b \left(\hat{L}_{ab}^{(D)} f_{a_1} + \hat{L}_{ab}^{(I)} f_{b_1} \right) = q_a f_{a_0}, \tag{3.2}$$

with ω as the rotation frequency of the plasma. The indices a and b denotes different particle species. g_a is a newly introduced distribution function defined by $f_{a_1} = f_{a_0} g_a$.

So far, the dependencies of the distribution function have not been closer explained. A convenient choice for the velocity space is

$$\eta = \frac{v_\perp^2}{v^2 B} \tag{3.3}$$

and the magnitude of the velocity with

$$x = \frac{v}{v_{T_a}} \tag{3.4}$$

as dimensionless variable, with v_{T_a} as the thermal velocity. Because η has not the information of the sign of the velocity, the sign σ was introduced. Equation 3.5 shows the approximation of the velocity distribution by basis functions $\varphi_{m'}(x)$ and coefficients $g_{a,m'}(\vartheta,\eta,\sigma)$. Different approaches to the underlying basis functions have been made. Local and global acting basis functions have been used. Laguerre Polynomials, B-splines of different orders, and Taylor expansion have been used and combined. [14] [8, p.33ff]. With

$$g_a(\vartheta,x,\eta,\sigma) = \sum_{m'=0}^{M} g_{a,m'}(\vartheta,\eta,\sigma)\varphi_{m'}(x), \tag{3.5}$$

Equation 3.2 can be written:

$$\begin{aligned} &v\lambda h^{\vartheta} f_{a_0} \sum_{m'=0}^{M} \varphi_{m'}(x)\frac{\partial g_{a,m'}(\vartheta,\eta,\sigma)}{\partial\vartheta} + i\omega f_{a_0}\left(\sum_{m'=0}^{M} g_{a,m'}(\vartheta,\eta,\sigma)\varphi_{m'}(x)\right) \\ &-\sum_b \left(\hat{L}_{ab}^{(D)} f_{a_1} + \hat{L}_{ab}^{(I)} f_{b_1}\right) = q_a f_{a_0}. \end{aligned} \tag{3.6}$$

After the reordering

$$\begin{aligned} &\sum_{m'=0}^{M}\left(v\lambda h^{\vartheta} f_{a_0}\varphi_{m'}(x)\frac{\partial g_{a,m'}(\vartheta,\eta,\sigma)}{\partial\vartheta} + i\omega f_{a_0} g_{a,m'}(\vartheta,\eta,\sigma)\varphi_{m'}(x)\right) \\ &-\sum_b \left(\hat{L}_{ab}^{(D)} f_{a_1} + \hat{L}_{ab}^{(I)} f_{b_1}\right) = q_a f_{a_0}, \end{aligned} \tag{3.7}$$

with

$$f_{a_1} = f_{a_0} g_a = f_{a_0}\left(\sum_{m'=0}^{M} g_{a,m'}(\vartheta,\eta,\sigma)\varphi_{m'}(x)\right) \tag{3.8}$$

follows

$$\sum_{m'=0}^{M}\left(v\lambda h^{\vartheta} f_{a_0}\varphi_{m'}(x)\frac{\partial g_{a,m'}(\vartheta,\eta,\sigma)}{\partial\vartheta}+i\omega f_{a_0} g_{a,m'}\varphi_{m'}(x)\right.$$
$$\left.-\sum_{b}\left(\hat{L}^{(D)}_{ab} f_{a_0} g_{a,m'}\varphi_{m'}+\hat{L}^{(I)}_{ab} f_{b_0} g_{b,m'}\varphi_{m'}\right)\right)=q_a f_{a_0}. \tag{3.9}$$

To define a compact notation, a scalar product was introduced. The definition of the scalar product shows

$$\langle i|j\rangle \equiv \frac{1}{n_a v_{T_a}^{2+\alpha}}\int_0^\infty dv v^{3+\alpha} e^{-\beta \frac{v}{v_T}^2} i(v) j(v) = \frac{v_{T_a}^2}{n_a}\int_0^\infty dx x^{3+\alpha} e^{-\beta x^2} i(x) j(x), \tag{3.10}$$

with α and β as weighting constants in the velocity space. The matrix elements

$$\rho_{m,m'} = \left\langle \varphi_m\left(\frac{v}{v_{T_a}}\right)\middle| v f_{a_0}(v) \middle| \varphi_{m'}\left(\frac{v}{v_{T_a}}\right)\right\rangle, \tag{3.11}$$

$$\omega_{m,m'} = \left\langle \varphi_m\left(\frac{v}{v_{T_a}}\right)\middle| \omega f_{a_0}(v) \middle| \varphi_{m'}\left(\frac{v}{v_{T_a}}\right)\right\rangle, \tag{3.12}$$

$$\hat{L}^{(D),ab}_{m,m'} = \left\langle \varphi_m\left(\frac{v}{v_{T_a}}\right)\middle| \hat{L}^{(D)}_{ab} f_{a_0}(v) \middle| \varphi_{m'}\left(\frac{v}{v_{T_a}}\right)\right\rangle, \tag{3.13}$$

$$\hat{L}^{(I),ab}_{m,m'} = \left\langle \varphi_m\left(\frac{v}{v_{T_a}}\right)\middle| \hat{L}^{(I)}_{ab} f_{a_0}(v) \middle| \varphi_{m'}\left(\frac{v}{v_{T_a}}\right)\right\rangle, \tag{3.14}$$

$$q_{a,m} = \left\langle \varphi_m\left(\frac{v}{v_{T_a}}\right)\middle| q_a\right\rangle, \tag{3.15}$$

are shown in respect to the represented basis functions φ_m and $\varphi_{m'}$. The nomenclature of these elements are the same as in [8].

With the definitions above, Equation 3.9 evolves to

$$\begin{aligned}\sum_{m'=0}^{M} \Bigg(&\rho_{m,m'} \lambda h^{\vartheta} \frac{\partial g_{a,m'}}{\partial \vartheta} + i\omega_{m,m'} g_{a,m'} \\ &- \sum_{b} \left(\hat{L}^{(D),ab}_{m,m'} g_{a,m'} + \hat{L}^{(I),ab}_{m,m'} g_{b,m'} \right) \Bigg) = q_{a,m}.\end{aligned} \tag{3.16}$$

Part of this thesis was the handling of the matrix elements described above. In Section 4.2, the implementation of utilizing the matrix elements more efficiently is presented.

3.1.4 NEO-2 output

As NEO-2 is based on the neoclassical transport theory, the output is neoclassical transport coefficients. The neoclassical transport coefficients are explained in detail in Section 2.2. With the neoclassical transport coefficients different fluxes can be calculated.

In axisymmetric magnetic fields with a non-axisymmetric perturbation, the neoclassical transport coefficients are split into an axisymmetric and a non-axisymmetric contribution [10]. The distribution functions g_k, with $k = 1, 2, 3$, are also an output by NEO-2, if the reconstruction run is applied. The bootstrap current is an important prediction of the neoclassical theory. For example, the coefficient D_{31}, is connected to this current, and is called bootstrap coefficient. Equation 2.49 shows that D_{31} is connected to the gradient driven distribution function g_1. With the use of the reconstruction run g_1 can be displayed.

With the use of radio-frequency waves a parallel current in fusion reactors can be driven. In high collision plasmas, the classical Spitzer function can describe the electron current drive efficiency with high precision. For low collision plasmas, the generalized Spitzer function has been developed. In the transition between high collision plasmas to collision less plasmas, the generalized Spitzer function reveals new physical properties [15]. The distribution function g_3 is linked to the generalized Spitzer function and can be shown by applying the reconstruction run.

Representations of the distribution functions g_1 and g_3, are shown in Section 4.3.

Reconstruction run The reconstruction run is a special mode of NEO-2 to reconstruct the distribution functions. These distribution functions are normally not saved, as the main interest of NEO-2 runs are normally the neoclassical transport coefficients. Nevertheless, there are use cases where the distribution function is required. One use case is the representation of the generalized Spitzer functions.

The distribution functions have up to four dimensions and to save all of them, a large amount of storage is required. Depending on the magnetic field line length, saving the distribution functions can take up to several 100 GB. The correct usage of the reconstruction run is explained in detail in Section 4.3.

3.2 Programming environment

The Jupyter NEO-2 interface relies not only on NEO-2, as described in the last section. It also relies on the program Jupyter Notebook. Therefore, the fundament of Jupyter Notebooks is outlined in this section. It is the programming language Python and its interactive version IPython. As further development of Jupyter Notebooks, a short forecast to JupyterLabs is given. So the consecutive structure of Python, IPython, Jupyter Notebooks, and JupyterLabs are explained in this section.

3.2.1 Python

Python is a high-level programming language invented by Guido van Rossum in 1991. It has a less formal syntax approach and focuses on high readability. For example, control commands, like loops, definitions, or conditionals, are structured by indentation and not by curly braces, like in the programming language C. It is also not necessary to end statements with a semicolon. Python uses the concept of object-orientation and implements this concept continuously. Therefore, in Python, everything is an object, like a list, numbers, classes, etc. The programming done for this thesis uses the concept of object-orientation.

Python is a popular language, in particular to the wide range of available libraries. In the scientific community, Numpy, Scipy, and matplotlib are heavily used. Numpy and Scipy implement calculations of vectors and matrices. These two libraries also offer many numeric routines. The matplotlib library is an interface to visualize data. In this thesis, a child class of matplotlib will be used as described in Section 4.3.

3.2.2 IPython

IPython[16] (Interactive Python) as the name already suggests, offers many features you normally only get in an Integrated Development Environment (IDE). The name IPython refers to both, the IPython kernel and the IPython shell, which communicates with the kernel. IPython has a cell-based execution workflow. Historically there was also an IPython Notebook, but the notebook utility moved to the Jupyter Project.

IPython works after the principle of REPL (Read, Eval, Print, Loop). Although Python itself supports a REPL mode, IPython has some other important features, supports a wide field of code completion and path completion. Furthermore, it can plot with different graphic backends.There are also the powerful so-called magic commands, where you can run external commands in the shell, use timing functions of the executing cell or define plot settings, and many more.

3.2.3 Jupyter Notebooks

As already mentioned, the Jupyter project is the evolution of the IPython Notebook. This evolution was necessary because the Jupyter project contains also kernels from programming languages other than Python, e.g. R and Julia. In the modules explained in this thesis, only Python with the IPython kernel is used.

Through the growing scientific interest in open source software and open data [17], one aim of this thesis was to write an interface for NEO-2 in Python. The interface should be easy to use, also for people who are not in the development of the NEO-2 code itself.

Each run of the NEO-2 code produces a lot of data, and if you want to have meaningful results, you have to make dozens or even hundreds of runs, which are producing up to 100 GB per configuration [18]. Therefore if you want to use the produced data, you also need an robust framework and tools for analyzing the results.

It has been stated in ref. [19] that notebook style is the new form of scientific writing A large scientific community is already using the Jupyter Notebooks. One is namely the LIGO Scientific Collaboration and Virgo Collaboration. They have set up a wide range of tutorials, beginning from understanding signal processing up to reproducing single plots of papers [20]. They have also published their data [21]. To reproduce the data of an important scientific paper, everyone can use the prepared Juptyer Notebooks and verify the results.

As a consequence of an easy to use and easy to reproduce concept, IPython/Jupyter Notebooks was chosen. It is also possible to use online tools like Binder or Azure to remotely run the Notebooks and reproduce the results, without taking care of the server-client connection.

An important part of this thesis is also about IPyWidgets shown in Subsection 4.3.4. IPyWidgets is a library for Python for HTML interactive controls inside the Jupyter Notebooks. It adds, user interfaces to the Jupyter Notebook, like sliders, buttons, and text fields. Because the output can be displayed nearly immediately, it is easier to 'explore' the data.

A Jupyter Notebook is running on a server-client structure. So first, you have to start the notebook server. This server can run locally on your machine, or you can run it remotely and use it, e.g. with ssh and port forwarding. This server can then be accessed via the browser, which automatically opens by default, when starting the server on a local machine. To access the server also a unique token is required. This token is normally passed automatically as an extension of the URL to the server address.

As the Jupyter project evolves, JupyterLabs were developed. JupyterLabs is the next generation of the Jupyter Notebook. JupyterLabs offers the comfort of a web-based integrated development environment. JupyterLabs still uses the notebook format for its files.

4 Implementation/Results

The name of the program package, which was implemented during this thesis, is neo2tools. Neo2tools is written in Python and designed to be used inside of a Jupyter Notebook. The expression "NEO-2 Jupyter interface" implies the interaction of neo2tools with the Jupyter Notebook.

As in Section 3.1 already mentioned, NEO-2 is a large project with about 250 input parameters and different input files for magnetic fields and profiles. The NEO-2 Jupyter Interface, tries to simplify to control this process. With the NEO-2 Jupyter Interface, it should be possible that persons, who are not familiar with the internal structure of NEO-2, can start runs and get meaningful results.

This chapter explains and shows how the program package neo2tools is implemented and laid out. Furthermore, step by step instructions, how to apply the package are provided. Examples of the graphical interactive output using different physical boundary conditions are shown.

This chapter is divided into three sections. The first section, Section 4.1 shows the preparing procedures performed before a single or multiple NEO-2 runs can be started. Section 4.2 focuses on optimizations of the NEO-2 runtime. The interaction of the program with the matrix elements is explained in detail. The final section, Section 4.3 covers plotting of the reconstruction run and displaying of the distribution function. Illustration and interactive display of the NEO-2 output are shown, and the implementation is explained.

4.1 Preprocessing

The preprocessing part of neo2tools enables running the NEO-2 code using the Jupyter Notebook. During the preprocessing, the required structure of files and directories is initialized. Starting NEO-2 runs and monitoring them, also on remote machines and computing nodes, is handled from the preprocessing part of neo2tools.

Common style for the class diagrams is used. Each box represents a class. The top section of the box is the name of each class. The middle section lists the class's attributes, and the bottom section specifies the class's methods. The arrows display relations between two classes. The unfilled triangle arrowhead indicates class inheritance. The filled rectangle arrowhead implies composition. One instance of the composited class is then an attribute of the other class. The blue name next to the arrow annotates the connected attribute. Hidden methods and attributes are indicated by a leading underscore.

In Figure 4.1, the class diagram for the preprocessing part is shown. The three classes `Neo2QL`, `Neo2Par` and `Neo2_common_objects`, constitute the base structure, inspired by the NEO-2 source code structure. The classes and their functionalities are now described.

4.1.1 Description of classes

Neo2_common_objects The abstract class `Neo2_common_objects` collects variables and methods required for both branches. Therefore these methods and attributes are generic. Base of all NEO-2 runs is the NEO-2 program, and the base of the NEO-2 program is the source code. If a `NEO2PATH` environment variable on the running system is defined, the hidden attribute, `_path2code`, is automatically set and points to the source code. The `_path2code` attribute can also be set manually. To build the executable, the `compile()` method is called. The `compile()` method relies on the NEO-2 build process, which is based on cmake and make.

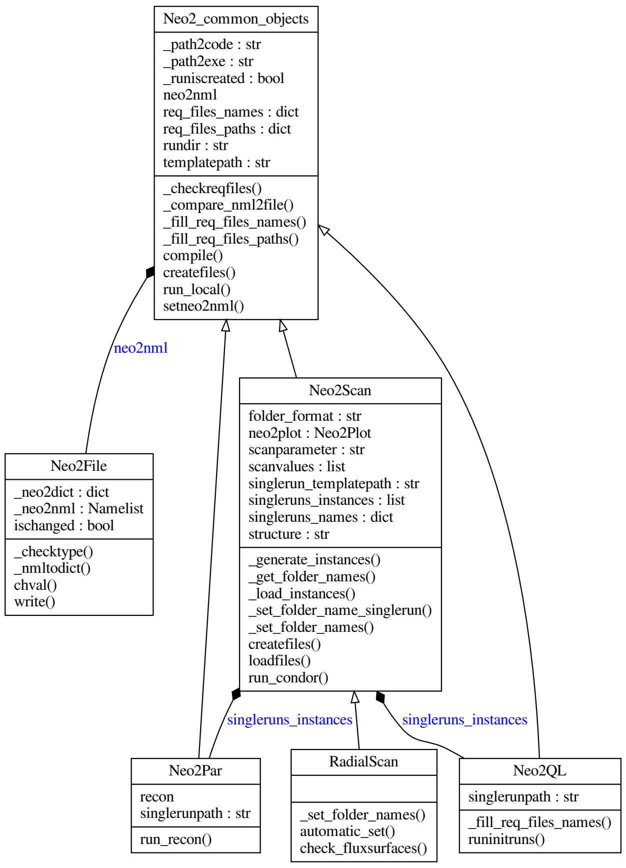


Figure 4.1: Classdiagram of the preprocessing part of neo2tools

During the build process, the `_path2exe` attribute will be set and saves the executable's absolute path.

An instance of `Neo2File` class is saved in the `neo2nml` attribute by using the method `setneo2nml()`. If a new `neo2.in` input file is loaded, it is automatically compared to the already done settings with the hidden method `_compare_nml2file()`. The `Neo2File` class handles the `neo2.in` input file and will be described later on.

Each NEO-2 run needs different additional input and control files. These files will be called required files and they are necessary for successfully starting a NEO-2 run. Depending on the branch of NEO-2, the required files are altering and are not always the same. The basic required files are shown in Figure 4.2 and Figure 4.3. Also the executable is defined as of one of the required files.

Some of the required files are specified as parameters in other required files. For example, in `neo2.in` the name of the file, which contains the perturbed magnetic field, is defined. The interplay of the attributes `req_files_names` and `req_files_paths` and the two methods `_fill_req_files_names()` and `_fill_req_files_paths()`, make sure, that all required files are found and set. With the use of the `templatepath` attribute, an existing run and all the settings can be taken from there. The `rundir` attribute defines the absolute path, where the prepared NEO-2 run will be executed.

With the `createfiles()` method, all files and directories are written to disk. The `_checkreqfiles()` method ensures, that the creation is only done, when all required files are available.

The `run_local()` method starts the NEO-2 run on the local machine and the `_runiscreated` attribute is set to true.

Neo2File The `neo2.in` input file is the command and control file for the NEO-2 runs. Because of the importance of this file, the class `Neo2File` has been created.

The `Neo2File` class handles the correct setting of the `neo2.in` input file. The input file is a fortran namelist file and for a proper handling in Python the `Neo2File` class is derived from the f90nml package [22]. A type checking method explicitly for NEO-2 is handled by the hidden method `_checktype()`. The methods `chval()` and `write()`, change desired parameters in the namelist file and write them to the disk. The `_nmltodict()` method together with the `_neo2dict` and `_neo2nml` attributes handle the internal handling of the attributes. If parameters are changed but the file is not written to disk, this is marked with the boolean attribute `ischanged`.

Neo2Par The `Neo2Par` class is for running single NEO-2 runs from the NEO-2-Par branch. Every single NEO-2 run needs its own folder and the required files. The required files for the NEO-2-PAR branch are the `neo.in` and `neo2.in` input files, the executable `neo_2.x` and the magnetic field file `boozer.bc`. These files are shown in Figure 4.2.

The `Neo2Par` class utilizes mostly the methods from the class `Neo2_common_objects`. One additional method is `run_recon()`. This method starts the reconstruction run. To plot data from the reconstruction runs directly from this class, the `recon` attribute is an instance of the `ReconPlot` class and is explained in Section 4.3.

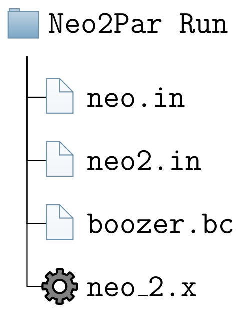


Figure 4.2: File structure for NEO-2-PAR runs

In addition to the `rundir` attribute of the `Neo2_common_objects` class the correct location of the NEO-2 run is saved in the `singlerunpath` attribute.

Neo2QL For starting runs of the NEO-2-QL branch, the `Neo2QL` class is used. The methods and attribute of this class are shown in Figure 4.1. NEO-2 runs from the NEO-2-QL branch need additional files compared to the NEO-2-PAR branch. With the `multi_spec.in` file, multispecies runs are managed. There is a special method for the multispecies runs called `runinitruns()`, which is normally called right before the start of a NEO-2 run. To handle the perturbation of the axisymmetric magnetic field (`axi.bc`), with the quasilinear ansatz, the perturbed magnetic field is provided in the `pert.bc` file. The required files are shown in Figure 4.3.

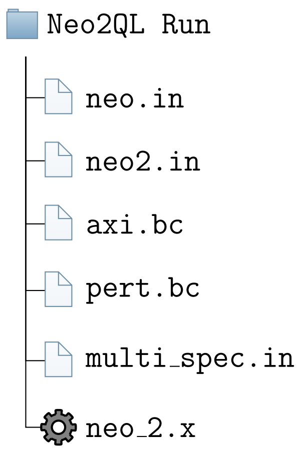


Figure 4.3: File structure for NEO-2-QL runs

Note, that with the use of the precomputation, explained in Section 4.2, the required files change. To account for the additionally required files, the `_fill_req_files_names()` method of the base class is overwritten. The `singlerunpath` attribute is the same as in the `Neo2Par` class.

Neo2Scan The `Neo2Scan` class facilitates parameter scans using NEO-2. Comparing results of single runs to experimental data is often difficult. Therefore, typically several NEO-2 runs scanning over a range of parameter values are performed. The Python package presented in this thesis aims to offer straightforward method to performing these scans.

The `Neo2Scan` class is inherited from the `Neo2_common_objects` class. The file and directory handling get greater importance than in the two single run classes `Neo2Par` and `Neo2QL`. The base of a parameter scan are the single NEO-2 runs, with each in its own folder. Each folder is containing the required files, depending on the NEO-2 branch, as shown in Figure 4.2 and Figure 4.3.

The `Neo2Scan` class can create new NEO-2 scans or load existing NEO-2 scans. The `createfiles()` method will create all files for a new scan from the `single_runs_instances` list. If the list is not filled or the validity check fails the `_set_folder_names()` and `_generate_instances()` methods are called. The `_set_folder_names()` method is responsible for setting all the files and directories. The `_single_run_names` attribute is populated by repeated calls of the `_set_folder_name_singlerun()` method. The `structure` attribute defines how the storage hierarchy is constructed. There are different possibilities to realize this, which are explained in Subsection 4.1.2

During the creation of the structure each layer of directories will be expanded depending on the scanned parameter `scanparameter`. If the name of one directory layer conforms to a parameter of the NEO-2 input file, this will be marked. If this marked directory layer is then also conform with the `scanparameter` attribute, for each value in the `scanvalues` attribute list, a new directory will be created. If more than one scanning parameter is defined, the spanning of the directories applies to the subdirectories.

The `_set_folder_name_singlerun()` method creates the directory name for a single run. The directory name is a joined name of the parameter itself, saved in the attribute `scanparameter`, and the value of this parameter from the `scanvalues` list. The exact typesetting can be controlled with the attribute `folder_format`.
The `_generate_instances()` method generates a list of instances, saved in the `singleruns_instances` attribute, of either the `Neo2Par` class or the `Neo2QL` class. The `singlerun_templatepath` attribute points to the directory of the required files, which are needed to create a NEO-2 run. With this instantiation of all single NEO-2 runs, generic modifications of parameter settings can quickly be passed and written to each single run. Finally all files from the `singleruns_instances` list are written to disk.

`run_condor()` is a particular method to deploy the NEO-2 runs to the high-throughput HTCondor[1] distribute computation system [23]. The HTCondor system is a batch system, which assign jobs to different nodes with enough free resources in an heterogeneous computer infrastructure. Consequently the HTCondor system also monitors the running jobs and reassigns jobs to other machines, if necessary. The documentation of the HTCondor system can be found online [24]. Therefore a NEO-2 scan is solved in an efficient way, utilizing the whole available computer infrastructure.

Pre-existing NEO-2 scans can be loaded using the `loadfiles()` method. This method first calls `_get_folder_names()` to determine the file structure. After this, the individual NEO-2 runs are loaded using the method `_load_instances()`. These can then be graphically analysed using the `Neo2Plot` class saved in the `neo2plot` attribute. A showcase of the `Neo2Plot` class is shown in Section 4.3.
The `_get_singlerun_names()` method is used for analysing the existing runs and fills the `singleruns_names` and the `singleruns_instances` attributes.

RadialScan The `RadialScan` class is inherited from the `Neo2Scan` class. The `RadialScan` class enables straightforward implementation of radial scans by a number of preset parameters and additional checking routines. The `RadialScan` class provides access to the radial dimension of the magnetic field. With the method `automatic_set()`, the resolution in the radial dimension is controlled by the number of the flux surfaces. Each flux surface corresponds to a single NEO-2 run, which name is set with the `_set_folder_names()` method.

The `check_fluxsurface()` method ensures that the field line's number of toroidal field rotations is between a minimum and a maximum. The minimum guarantees a sufficient covering of the flux surface with the magnetic field line. The maximum restricts the calculation time of the program, because the length of the field line is proportional to the calculation time.

[1]The system's name was originally Condor. It was renamed, due to legal reasons.

A low-order rational surface can cause the field line to close before the minimum of toroidal field rotations is reached. The `RadialScan` class, in this case, slightly shifts the flux surface to a surface of higher rational order.

4.1.2 Jupyter Notebook NEO-2 integration

In this subsection the correct usage of the neo2tools, which was created in this thesis, is shown. First, the procedure for creating a single run is explained in detail. Second, the correct usage for the `Neo2Scan` class is demonstrated. Third the automatic setting and checking of a radial scan using the `RadialScan` class is shown. The single run starts with an empty Jupyter Notebook with Python3. The explanation of the other classes continues in the same Jupyter Notebook.

Single run All the single run examples will be shown on the `Neo2QL` class. It would also be possible to use the `Neo2Par` class. Input cell `[1]` shows the first commands, which are required to start a single run. In the first line the neo2tools package is imported.

```
In [1]:  import neo2tools
         runpath = "/temp/Neo2QL/newrun/"
         templatepath = "/temp/Neo2QL/oldrun/"
         Job_ql = neo2tools.Neo2QL(runpath, templatepath)
```

The `runpath` parameter locates the preparing NEO-2 run. Because this is the preparing procedure for a NEO-2 run, the `runpath` parameter does not have to exist. This will be managed from the preprocessing part of neo2tools. The second required parameter is the `templatepath` value. The `templatepath` is the location of a template NEO-2 run directory. This can be an already existing NEO-2 directory or it can be an explicit template directory.

The forth line in input cell `[1]` shows how to create an instance of the `Neo2QL` class. The two path parameters are passed in the instantiation.

```
In [2]:  Job_ql._path2code
         Job_ql.compile()

Out[2]:  "/temp/Codedir/"
```

After the instantiation it is checked, if the path to the NEO-2 source code is correctly set. This step is shown in the first line in input cell [2]. The path to the NEO-2 source code is printed into the output cell. The path has to point, where the NEO-2 Fortan source code is located. Using the `compile()` method the executable is built. To compile the executable, the programs make and cmake are applied. If the path to the NEO-2 code is not correct, the `compile()` method fails and an error message will appear. A manual set of the `_path2code` parameter is then necessary. The command `Job1._path2code = "/temp/correctpath/"` sets the path.

```
In [3]:  Job_ql.setneo2nml("/temp/neo2.in")
```

To manually set a `neo2.in` file, the `setneo2nml()` method has to be invoked. Input cell [3] shows this. If the template directory contains a `neo2.in` input file, this is set automatically as default. The setting is important, because a copy of the `neo2.in` file is saved as the attribute `neo2nml`. The attribute `neo2nml` is an instance of the `Neo2File` class, which is handling the changing of parameters.

```
In [4]:  Job_ql.neo2nml.lag

Out[4]:  lag = 4
```

Every parameter check and change is done with the attribute `neo2nml`. The first line in input cell [4] checks the parameter `lag`. The parameter `lag` is representing the number of basisfunctions. The output shows the current value.

```
In [5]:  Job_ql.neo2nml.lag = 3
```

Input cell [5] demonstrates how to change the parameter `lag` to the value 3.

In [6]:
```
Job_ql.run_local()
```
Out[6]:
```
(Neo2 output)
```

When all parameter settings are done, the NEO-2 run can be started with the method `run_local()`. Input cell `[6]` displays this. The output `[6]` displays the output of the NEO-2 program. Because the output of the NEO-2 program is large, the output is not explicitly shown here. However, in Jupyter Notebook, the output of a cell can be folded or hidden and therefore also large outputs integrate conveniently in the notebook.

NEO-2 Scan A NEO-2 scan consists of many single NEO-2 runs. In comparison to the `Neo2QL` class, there are a few new attributes in the `Neo2Scan` class. As noted before, the Jupyter Notebook continues in the same Jupyter Notebook as the explanation of the `Neo2QL` class.

In [7]:
```
runpath = "/temp/Neo2Scan/"
templatepath = "/temp/Neo2QL/oldrun/"
Job_scan = neo2tools.Neo2Scan(runpath,
                              templatepath)
```

Input cell `[7]` shows the instantiation of the `Neo2Scan` class. In comparison to the input cell `[1]` the `runpath` changed to a new directory. The `templatepath` stayed the same, as the `templatepath` is the template for each single NEO-2 run within the NEO-2 scan. If the template directory contains a compiled NEO-2 program, the compilation process, as described in input cell `[2]` for the single run, can be skipped. The compiled program in the template directory will be utilized. For changing parameter and for setting the NEO-2 input file, the same procedure as for the single run described in input cells `[5]` and `[3]` can be used.

A NEO-2 scan is for example a radial scan. NEO-2 labels the radial component as `boozer_s`. Compared to Equation 2.13 the Boozer coordinates are now written as (s, ϑ, φ) with s as the radial component. Line 1 of input cell `[8]` assigns the string `boozer_s` to the attribute `scanparameter`.

This parameter has to be part of NEO-2 input file, otherwise the scan fails. The desired scan values are passed as a list. For a radial scan with three radial points, line 2 of input cell [8] demonstrates the passing.

```
In [8]:  Job_scan.scanparameter = "boozer_s"
          Job_scan.scanvalues = [1, 0.1, 0.01]
```

For this simple, one-dimensional scan a flat directory hierarchy is appropriate and input cell [9] shows how to set this flat hierarchy using the structure attribute.

```
In [9]:   Job_scan.structure = "boozer_s/"
```

The directory structure of this example is illustrated on the left side of Figure 4.4. The directory is containing the three single run folder, which are containing the required files for each run. Each single run folder's name is a combination of the name and the value of the scanparameter.

Often it is necessary that complex scans with more than one changing parameter are made. With the prerequisite that each NEO-2 run needs a distinct folder, more complex storage hierarchies arise. The structure attribute of the Neo2Scan class defines where each single run of a NEO-2 scan is located. The Neo2Scan class can span a whole tree of directories. The starting point of this tree is the runpath parameter defined in input cell [7]. Each arm of the tree ends in a single NEO-2 run. The user only defines one arm, and the program spans the rest of the tree automatically. The structure attribute is the path, describing the defining arm of the directory tree. The furcations of the tree are the directories and the subdirectories and the subsubdirectories. Using the path limiter '/', each level of directory is defined. If a level's directory name agree with a parameter of the NEO-2 input file, the value will be added to the directory's name. The spanning occurs if the parameter is the scan parameter.

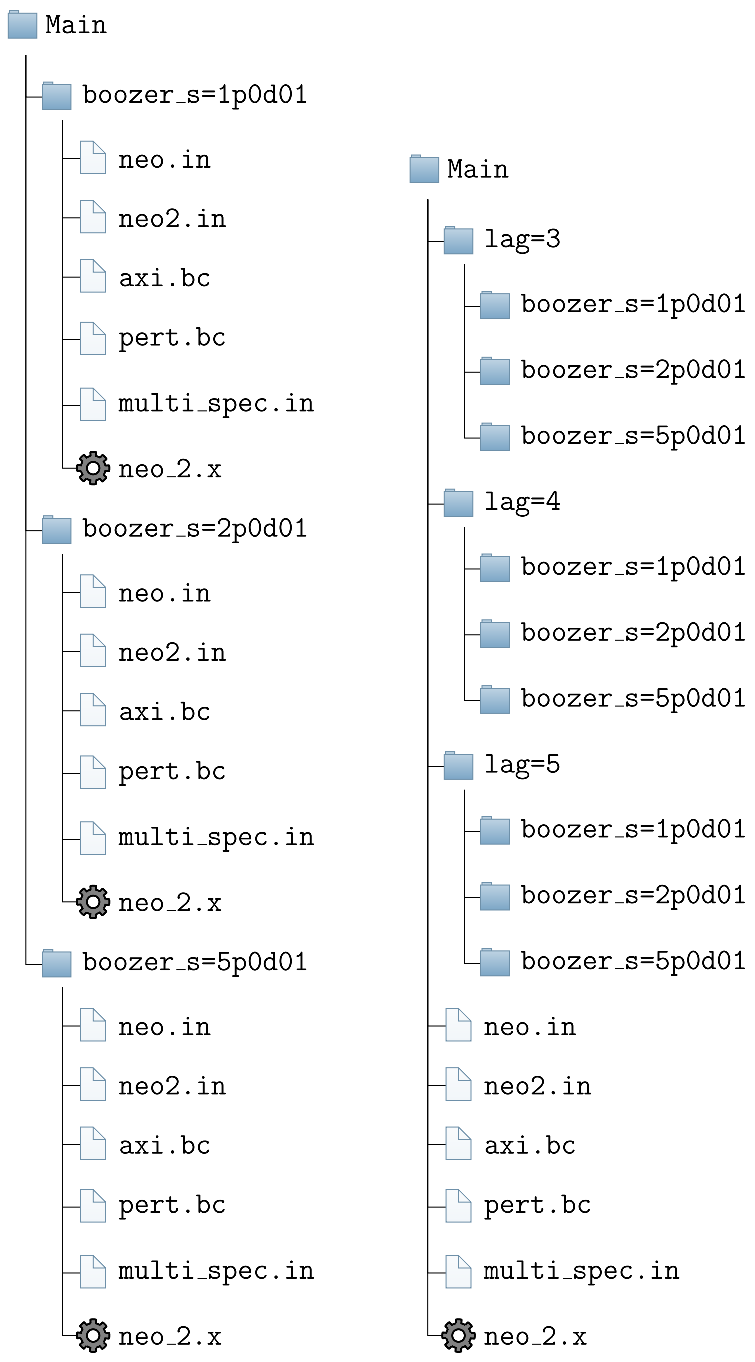


Figure 4.4: Folder hierarchy for a one-dimensional and a two-dimensional NEO-2 scan

Input cell [10] shows the setting for a two and a three-dimensional scan. There are the new instances Job_scan2d and Job_scan3d to emphasize, that these are seperated runs.

```
In [10]:  Job_scan2d.structure = "lag/boozer_s/"
          Job_scan3d.structure = "boozer_s/lag/leg/"
```

The illustration of the directory hierarchy for the two-dimensional scan is shown on the right side of Figure 4.4. In this example the template files for the single runs are in the Main folder. For the sake of clarity the required files for each single run are not displayed.

```
In [11]:  Job_scan.run_condor()
```

Additionally to the run_local() method, explained in input cell [6], another method for the Neo2Scan class has been implemented. Input cell [11] shows the use of the run_condor() method utilizing the network infrastructure. Each single run is running on a different machine and therefore the NEO-2 scan is finished faster. The condor method is described in detail above.

RadialScan The RadialScan enables straightforward implementation of radial scans. Without the RadialScan class these radial scans are more cumbersome as shown above in the example of the Neo2Scan class (Job_scan). In the first line of input cell [12], the new scan is instantiated as in the previous examples. The second line of input cell [12] creates the flux surfaces. With the method automatic_set(), the number of 20 flux surfaces is set with the numbers parameter. The further passed parameters to the automatic_set() method restricts the radial range of the created flux surfaces. In this example the the minimum (min_s) is set to 0 and the maximum (max_s) to 1.

```
In [12]:  Job_radial=RadialScan(runpath, templatepath)
          Job_radial.automatic_set(numbers=20, min_s=0,
                                   max_s=1)
```

In contrast to performing radial scans with the Neo2Scan class, the RadialScan class offers additional checking routines for guaranteeing the minimum field line length. The scan is then started locally or remote with the methods shown in input cell [6] or input cell [11].

4.1.3 Reconstruction run

The reconstruction run is a special mode of NEO-2. The reconstruction run has to be run if the distribution function should be saved. A default saving of the four-dimensional distribution function is not intended, because of the high computational cost as explained in Subsection 3.1.4. For a more precise analysis, the reconstruction run can be applied and the distribution function is saved.

So far, the reconstruction run is only implemented in the parallel version of NEO-2, as the method run_reconstruction(). Therefore, in Figure 4.1 can be seen, that only the Neo2Par class has the run_reconstruction() method.

The setup of an instance for the Neo2Par class is the same as for Neo2QL class. The instance Job_par passed through the same steps from input cell [1] to [6]. In input cell [13] can be seen, that running the reconstruction needs no further input parameter.

```
In [13]: Job_par.run_reconstruction()
```

Internally the prop_reconstruct value of the neo2.in input file is set to one. Subsequently, a NEO-2 run is performed. After the run has finished, prop_reconstruct value is increased by one, and another NEO-2 run is performed. This is repeated until the value of prop_reconstruct reaches three. In Table 4.1 the meaning of each value can be seen. To complete the reconstruction run, all values are run in order.

Table 4.1: Values for the NEO-2 reconstruct setting

prop_reconstruct	Description
0	Normal NEO-2 run
1	Preparing for the reconstruction run
2	Actual reconstruction run
3	Clean up run

The reconstruction run is highly cost- and memory intensive. The output of the reconstruction run is saved in the file final.h5. The size of the file can up to be several 100 GB. In Subsection 4.3.3 the plotting of the distribution function is explained.

4.2 Runtime optimization

4.2.1 Precomputation of the matrix elements

The calculation of the collision operator's matrix elements is an important part of the calculation for the NEO-2 runs. A detailed derviation of the collision operator and its matrix elements is shown in [14].

Table 4.2: Computed matrix elements of the collision operator

Variable	Shape	Mathematical Expression
`anumm_aa`	lag x lag x leg x leg	$\hat{\nu}^{ab}_{m,m'}$
`anumm_inf`	lag x lag	$\hat{\nu}^{a\infty}_{m,m'}$
`ailmm_aa`	5D	$\hat{L}^{(I),ab}_{m,m'}$
`denmm_aa`	4D	$\hat{L}^{(D),ab}_{m,m'}$
`C_m`	lag	C_m
`asource`	lag x 3	q_m
`M_transform`	lag x lag	$\rho_{m,m'}$
`M_transform_inv`	lag x lag	$\rho^{-1}_{m,m'}$
`Amm`	lag x lag	$\rho_{m,m'}$

Equation 3.16 describes the relation of the matrix elements and the collision operator. The matrix elements that are used in this calculation are listed in Table 4.2. This table relates the mathematical expressions, as used in Subsection 3.1.3, to the variable names of the matrix elements used in the code. The shapes of the matrix elements depend on the input parameters `lag` and `leg`. These dependencies are given in the second column of Table 4.2

As seen in Subsection 3.1.3, the basis functions used, are the fundaments of the matrix elements. In addition to the type of the basic functions and the temperatures and masses of each particle species, the weighting in velocity space and the order of Legendre polynomials act as the input parameters for calculating the matrix elements of the collision operator.

Table 4.3 lists the names of the input parameters as used in the code, their type, and descriptions.

Table 4.3: Required input parameter for the matrix elements of the collision operator

Input Parameter	Type	Description
lag	scalar	Number of basisfunctions in direction v
leg	scalar	Number of Legendre polynominals
collop_base_exp	scalar	type of expanded basisfunction
collop_base_prj	scalar	type of projected basisfunction
lsw_multispecies	logical	single species run or multispecies
n_spec	scalar	density of each species
m_spec	array	mass of each species
T_spec	array	Temperature of each species
scalprod_alpha	scalar	weighting in velocity space
scalprod_beta	scalar	weighting in velocity space

All input parameters of the Table 4.3 are located in the `neo2.in` input file in the namelist `collision`. It is noted here that the parameter `lag` was historically used for Laguerre polynomials, but now it is also used for all other basis functions. Depending on the basis function used and the order of the basis functions and the number of particle species used, the matrix elements' calculation can be of significant computational cost compared to the rest of the NEO-2 run. This is especially the case if the number of species is high. However, the memory demand is usually low. In the conventional approach of NEO-2 [12], the computation of the matrix elements and the solving the drift kinetic equation is performed in one single run without intermediate calculation steps. As NEO-2 is usually memory limited and there is only a limited amount of high-memory computing nodes available, it is useful to split the matrix elements' calculation from the rest of the calculations. Splitting the computation of the matrix elements from the rest of the calculations has been implemented in this thesis. This splitting has the additional advantage that various sets of computations can then use the precomputed matrix elements because they are largely independent of the plasma parameters.

The program flow of NEO-2, before (hollow arrow) and after (filled arrow) the implementation of the matrix elements precomputation is schematically illustrated in Figure 4.5.

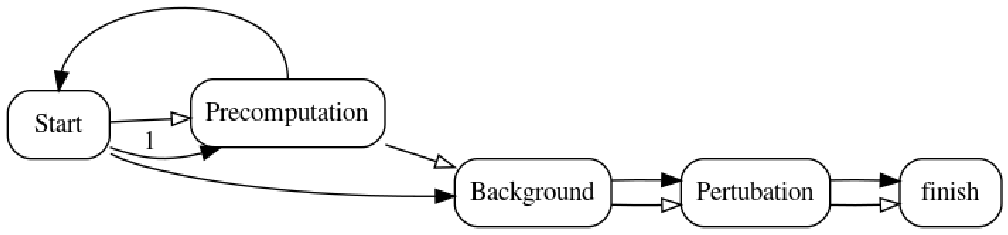


Figure 4.5: Program flow of NEO-2, before (hollow arrow) and after (filled arrow) the implementation of the matrix elements precomputation

As a result of this thesis, the conventional NEO-2 input file is extended by two new boolean variables. These two variables, named `lsw_read_precom` and `lsw_write_precom`, define whether matrix elements are read from or written to the file `precom_collop.h5`. The new introduced variables `lsw_write_precom` and `lsw_read_precom` are added to the namelist `collisions` in the `neo2.in` input file. The following paragraphs describe the behaviour of these two variables.

If the `lsw_write_precom` is set to true, only the precomputation of the matrix elements will be performed, and then the program stops without solving the drift kinetic equation. This behaviour was implemented because the precomputation of the matrix elements often will be performed on another machine than the actual run. The precalculation of the matrix elements can be done on a machine with comparably small memory. The memory demanding part of solving the drift kinetic equation can then be executed on another machine with more memory. For comparison of these values: the matrix elements' memory demand is in the order of a few GB, solving the drift kinetic equation needs several 10 or up to 100's GB of memory. As soon as the matrix elements are computed, they are saved in an hdf5-file named `precom_collop.h5`. After the matrix elements are saved, the NEO-2 runs are started as normal with the `neo2.in` input file and the normal executable of the NEO-2-QL version.

If the `lsw_read_precom` variable is set to true, an algorithm checks the new NEO-2's run compatibility with the precomputed matrix elements saved in the file `precom_collop.h5`. The compatibility is given, if the input file's parameters that are listed in Table 4.3 agree with the corresponding values in the `precom_collop.h5` file. In addition to these parameters also the compatibility of various auxiliary parameters is checked. If the compatibility is not given, the run will be aborted. If this compatibility check is successful, the run will be started utilizing the precomputed matrix elements from the file `precom_collop.h5`.

If `lsw_read_precom` and `lsw_write_precom` are set to false, NEO-2 is first calculating the collision matrix, without saving it, and then solving the drift kinetic equation, as it used to be.

4.3 Postprocessing

The postprocessing is described in four parts. The first part is the description of the involved classes. The second part covers collecting different data from different runs. The third part addresses the graphical data representation and is described in Subsection 4.3.3. This section covers how to easily display the produced data and how neo2tools enables interaction with the produced data. The final section demonstrates the interactive mode of the neo2tools package.

Figure 4.6 shows the class diagram of the relevant classes in postprocessing. The classes shown in the diagram are explicitly explained in the following subsection.

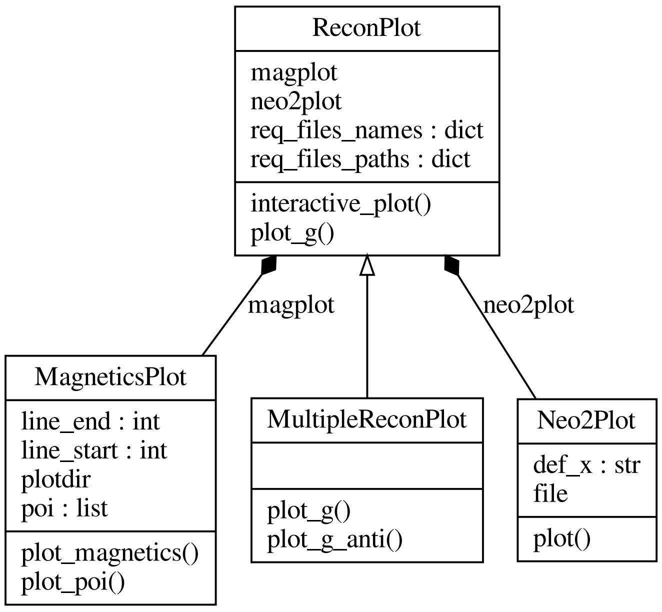


Figure 4.6: Classdiagram of the postprocessing part of neo2tools

4.3.1 Description of classes

ReconPlot The `ReconPlot` class is designed to plot the data from reconstruction runs. The reconstruction run is a special mode of NEO-2 on which an already performed run is started again with the same physical and numerical settings and is explained in Subsection 4.1.3.

This thesis tries to simplify the usage of NEO-2. In the `ReconPlot` class a previously developed NEO-2 interface [8] was integrated. The mentions of the name NEO-2 interface, in the following paragraphs refer to this.

The NEO-2 interface exports the distribution function $f_{1,k}$, of a Point of Interest (PoI) along the magnetic field line. The NEO-2 interface itself needs an input file. Furthermore, the desired PoI or the list of PoI have to be transferred to the interface. The points are transmitted in Boozer coordinates (s, ϑ, φ) together with a labeling tag. The NEO-2 interface returns then the distribution function $f_{1,k}$ for all PoI.

The `ReconPlot` class manages all the steps interacting with the NEO-2 interface. The involved methods and attributes are now explained. Figure 4.7 shows all public and hidden attributes and methods of the `ReconPlot` class. A leading underscore denotes the hidden attributes and methods.

If the NEO-2 interface's required files are incomplete, the hidden method `_fill_req_files_names()` fills them in the corresponding attributes `req_files_names` and `req_files_paths`.
The `_plot_write()`, `_createfiles()` and the `_run_dentf()` methods fill the NEO-2 interface input file and call the NEO-2 interface.

The attributes `_plotdir, _rundir, _templatepath` are set during the instantiation.

To simplify the user interaction, the class merges two other classes, as displayed by the arrows in Figure 4.6 and by the name of the composite class in Figure 4.7.

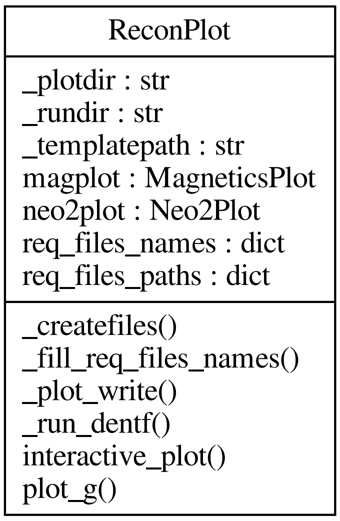


Figure 4.7: Classdiagram of the ReconPlot class

The attribute `magplot` of the `ReconPlot` class contains an instance of the class `MagneticsPlot`. The `MagneticsPlot` class only needs the magnetic field's directory as initialization parameter. Because each NEO-2 run contains a magnetic field file, the run directory's information will be passed at the initialization of `magplot`.
Under the attribute `neo2plot`, the second compositing class, `Neo2Plot` is held. This is useful, as it simplifies the plotting procedure. With the use of autocompletion, the possible plotting parameters are displayed and easily selected. An illustrative example is shown in Subsection 4.3.2. Further explanations are made in the `Neo2Plot` class description.

The distribution functions $f_{1,k}$ are renamed to g_k for consistency with the NEO-2 interface. Plotting the distribution function g_k offers an insight to the velocity distribution of the particles. The plot method `plot_g()` offers this possibility to plot the distribution function of selected PoI on the magnetic field line. Additionally, derivatives with respect to the direction of the particle motion can be plotted.

At the instantiation of the `Reconplot` class, the NEO-2 run directory `rundir`, and the plot directory `plotdir`, are required as input parameters. The parameter `templatepath` is optional. Also one of the distribution functions g_1, g_2 or g_3 has to be chosen. This is done by setting the parameter `index` to the wanted index k of g_k:

```
neo2tools.ReconPlot(rundir,plotdir,templatepath,index).
```

Upon initiating the class instance, the run directory will be checked if it is a valid NEO-2 run directory. The plot directory can be set as an absolute path or otherwise, it is set as a relative path to the run directory.

The method `plot_g()` is for plotting g_k. In contrast to the way, the PoI are transmitted to the NEO-2 interface with Boozer coordinates (s, ϑ, φ), `plot_g()` uses the parameter `poi` as a one-dimensional distance along the magnetic field line:

```
ReconPlot.plot_g(typ,poi,tags,subplot).
```

By default, the reconstructed distribution functions of all chosen PoI are plotted. This can be limited, by passing the desired PoI's labels as a list to the parameter `tags`. Also, the derivative parallel and perpendicular to the direction of the magnetic field line are plotted and controlled through the parameter `typ`. If the boolean parameter `subplot` is set to true, separate plots are generated for each, of the derivatives and of the field line. To limit the number of plotted distribution functions to specific tags, these tags can be chosen and passed as list to the parameter `tags`.

Calling of `interactive_plot()` needs no input parameter.
The `interactive_plot()` method uses the Jupyter Notebook functionalities to internally call the `plot_g()` method. In this case, only one PoI can be chosen. This point can be changed interactively with the text field or with the slider. To limit large magnetic field lines, also the range of this field line can be adjusted. With a drop-down menu, either

the distribution function or one of the derivatives can be plotted. The interactive mode is illustrated in Subsection 4.3.4.

Neo2Plot The `Neo2Plot` class is built upon the matplotlib module. The `Neo2Plot` class enriches the plot functionality of matplotlib with a default x-axes. In addition to the known properties of matplotlib, there are some meta-information saved, like where the data is located. The main purpose of this class is to plot the results of NEO-2 runs.

At the instantiation of the `Neo2Plot` class a hdf5 file handle has to be passed to the parameter `h5file`. hdf5 is a storage efficient file format. A default x-Axes with the parameter name `def_x` can be passed as well:

```
neo2tools.Neo2Plot(h5file, def_x).
```

If `def_x` is not set in the instantiation, the default x-Axes is automatically set with `boozer_s`. The hdf5 file handle and the default x-Axes are saved as the attributes `file` and `def_x` and is displayed in the class diagram in Figure 4.8.

The plot method `plot()` is usually called on its own and is mostly used by other classes, e.g. the `Neo2Scan` class. The method `plot()` is internally called when the name of the parameter to plot is appended to the instance like an attribute and executed. An example is shown in Subsection 4.3.2. The speciality of the `Neo2Plot` class and the integrated `plot()` method is, that it caches the possible plot parameters, depending on the chosen default x-Axes. This is managed due to processing the results of NEO-2() runs and validating it, through checking the plot compatibility with the x-Axes. The `_get_valid_keys()` method checks this plot compatibility and saves the allowed plot parameters in the attribute `_valid_keys`.

The method `plot_overview()` plots all possible plot parameters of the `_valid_keys` list in one overview plot. Furthermore, the `Neo2Plot` class also uses a key feature of Jupyter Notebooks: the autocompletion. If the user presses the Tabulator key, a list of possible plot parameters is shown. With the integration of the method `_ipython_key_completion()`,

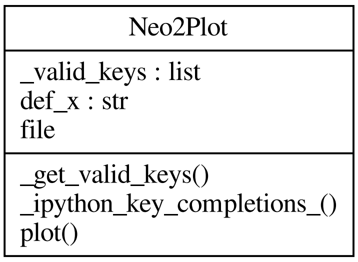
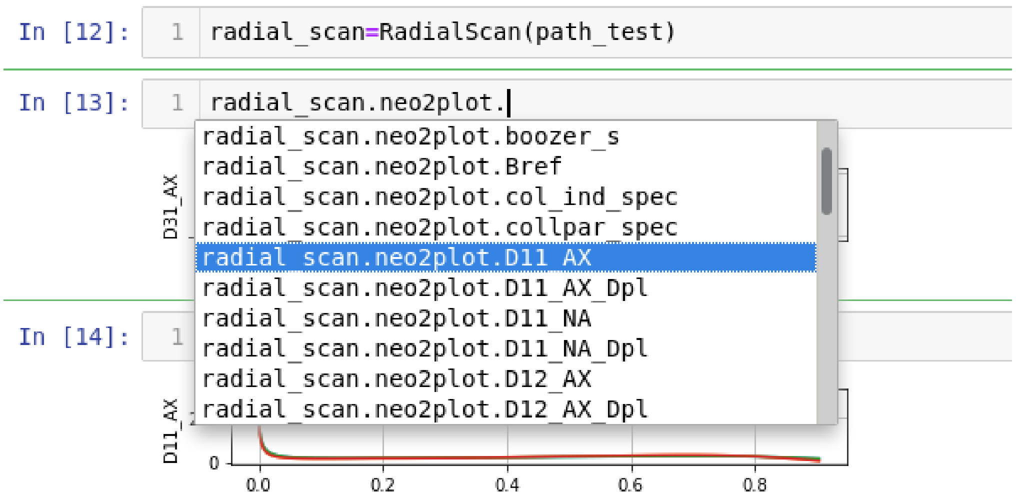


Figure 4.8: Classdiagram of the Neo2Plot class

pressing the tabulator key, offers the possibilities to plot of the NEO-2 results. Figure 4.9 shows the popped up list after pressing the tabulator key.



```
In [12]: radial_scan=RadialScan(path_test)

In [13]: radial_scan.neo2plot.
         radial_scan.neo2plot.boozer_s
         radial_scan.neo2plot.Bref
         radial_scan.neo2plot.col_ind_spec
         radial_scan.neo2plot.collpar_spec
         radial_scan.neo2plot.D11_AX
         radial_scan.neo2plot.D11_AX_Dpl
         radial_scan.neo2plot.D11_NA
         radial_scan.neo2plot.D11_NA_Dpl
         radial_scan.neo2plot.D12_AX
         radial_scan.neo2plot.D12_AX_Dpl

In [14]:
```

Figure 4.9: IPython key completion

The possible plot parameters depend on the NEO-2 Version used. The `neo2plot` attribute is an instance of the `Neo2Plot` class and therefore, the autocompletion to the possible plot parameters works.

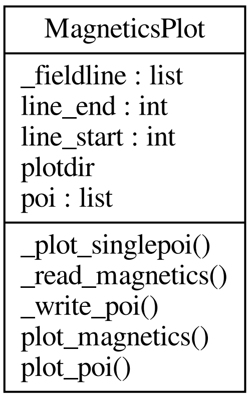


Figure 4.10: Classdiagram of the MagneticsPlot class

MagneticsPlot The `MagneticsPlot` class provides methods to plot the magnetic field of a given flux surface. Each NEO-2 run covers the flux surface with one magnetic field line. With the `MagneticsPlot` class, it is possible to plot this magnetic field line on top of the flux surface. The class diagram with the corresponding attributes and methods is shown in Figure 4.10.

The instantiation of `MagneticsPlot` needs only the directory with the NEO-2 run. This directory is passed with the parameter `rundir`:

```
neo2tools.MagneticsPlot(rundir).
```

The flux surface is plotted with the method `plot_magnetics()`. This method uses internally the `_read_magnetics()` method to read the magnetic field files:

```
neo2tools.MagneticsPlot.plot_magnetics(length,startphi,
                                        starttheta).
```

The correct usage of the `plot_magnetics()` method is shown in Subsection 4.3.3. The parameters `length`, `startphi`, `starttheta`, define the field line. The magnetic field line's start and end position are saved in the line_start and line_end attributes of the MagneticsPlot class.

The method plot_magnetics() plots also the magnetic field line. The field line is saved in the _fieldline attribute. The method plot_poi() places PoI on top of the magnetic field line. This is also useful to determine the position of the PoI at the flux surface. The PoI are saved in the attribute poi. The method _write_poi() has been implemented to pass the points. The ReconPlot class can then plot the distribution function of these points if the reconstruction run has been done. The _plot_singlepoi() method is utilized from the interactive_plot() method of the ReconPlot class.

MultipleReconPlot The MultipleReconPlot class is derived from the ReconPlot class. As explained in Section 4.1, meaningful results needs various NEO-2 runs. Therefore, it is useful to extend the functionality of investigating PoI, from single NEO-2 runs to multiple NEO-2 runs. The MultipleReconPlot class enables this extension.

The instantiation of the MulitpleReconPlot class is similiar to the ReconPlot class. The only change is the parameter rundirs replaces the rundir parameter:

```
neo2tools.MulitpleReconPlot(rundirs,plotdir,templatepath,
                            index).
```

The different NEO-2 directories have to be passed as list to the parameter rundirs. The rest of the instantiation parameter are explained in the ReconPlot class at page 52.

Due to the different structures of the directories, the plot_g() method is adapted and overwritten to perform correctly. Consequently, it is displayed in Figure 4.11. The methods plot_g() and plot_g_anti() for plotting the different distribution functions is demonstrated in Subsection 4.3.3. Because the data based for the plots is not a single run, but multiple runs the attribute _runs handles the different run directories. The syntax of plot_g() stays the same. Also plot_g_anti() has the

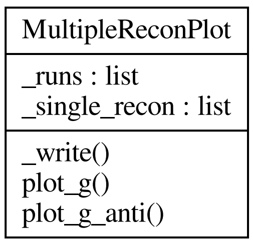


Figure 4.11: Classdiagram of the MultpleReconPlot class

same syntax. The method `_write()` ensures that the correct distributions functions are selected. For manual use of single `ReconPlot` instances, the `_single_recon` attribute saves these instances in a list.

The parameter `poi` defines the wanted PoI as a list. This is normally done by the `MagneticsPlot` class. For an alternative choice, `_write()` sets them:

```
neo2tools.MulitpleReconPlot._write(poi).
```

4.3.2 Data collection

The data collection can be seen as the reverse process of the preprocessing part, which is explained in Section 4.1. Every single NEO-2 run provides only limited information. To get sufficient information about diffusion coefficients, or torque, data must be collected from multiple runs. How data from different runs are collected depends on what kind of information should be plotted. One example is a radial scan (Figure 4.12), where data from all magnetic flux surfaces are needed. Another one is a scan for different collisionalities (Figure 4.13). The bootstrap coefficient λ_{bb} from Figure 4.12 and Figure 4.13 is often of interest because it can be compared to analytical theory [25].

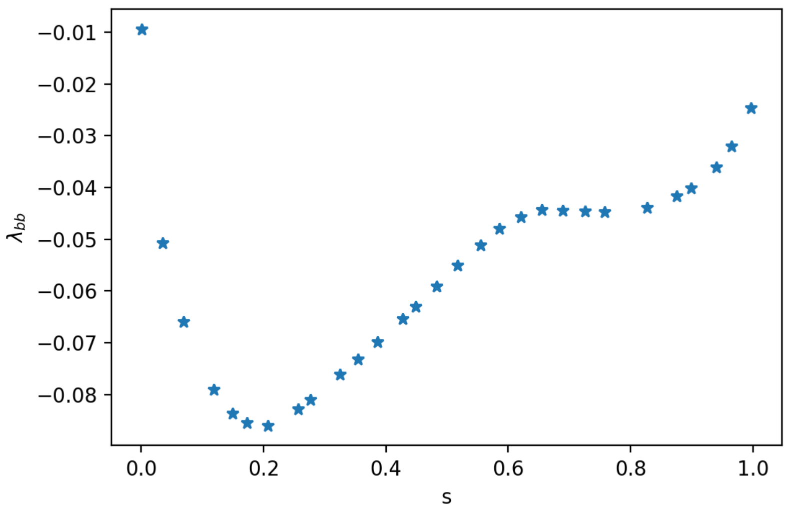


Figure 4.12: Radial scan

The type of scan is usually already defined in the preprocessing part. In the preprocessing, different input parameters for different NEO-2 runs are prepared to scan along a parameter. In the data collection, the relevant data as a function of the scanned parameter are collected. Therefore, also the `Neo2Scan` class and especially the `loadfiles()` method together with the `Neo2Plot` class is utilised. Continuing with the `Job_scan` instance of Subsection 4.1.2, the required files are loaded with the `loadfiles()` method.

A dictionary of the possible single runs used is generated. The dictionary is called `singlerun_names` and is an attribute of the `Neo2Scan` class. The keys of the generated dictionary are the paths to each single run. The values of `singlerun_names` contain the information, how the input file for each single NEO-2run is altered. For the data collection, only the keys of the `singlerun_names` are relevant. The keys are used to identify each NEO-2 run. The `loadfiles()` method searches for the output of the NEO-2 runs in the respective directory. This output is then sorted by the scanned parameter. Because of the high dimensionality of the output, it is rearranged to enable easier data access.

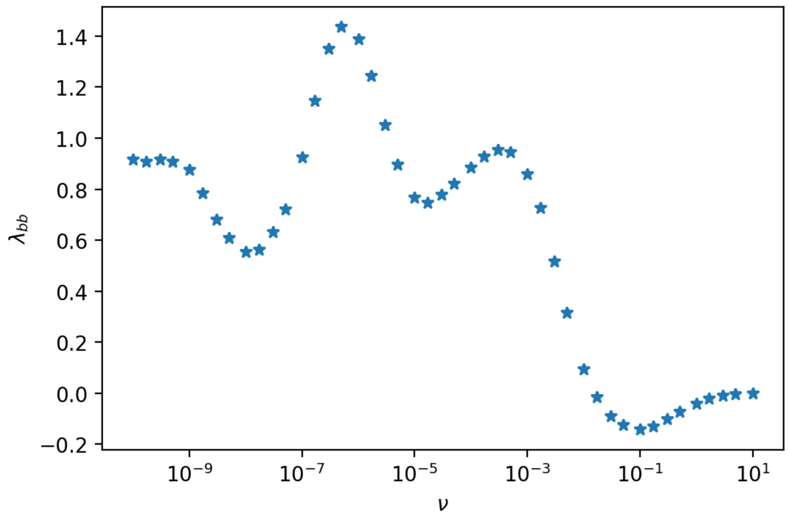


Figure 4.13: Collisionallity scan

The `Neo2Plot` class can access this collected data to produce the graphical output. The `Neo2Scan` class has the `neo2plot` attribute, which is an instance of the `Neo2Plot` class.
Therefore, also the `_ipython_key_completion()` method works as shown in Figure 4.9. Depending on the version and the exact input parameters of the performed NEO-2 run, there are a lot of possible output parameters to plot. The `plot_overview()` method plots all possible plot parameters in an overview plot. Figure 4.14 shows the overview plot of an NEO-2-QL multispecies scan run. From the 98 subplots in this example Figure 4.14 shows only a cropped view .

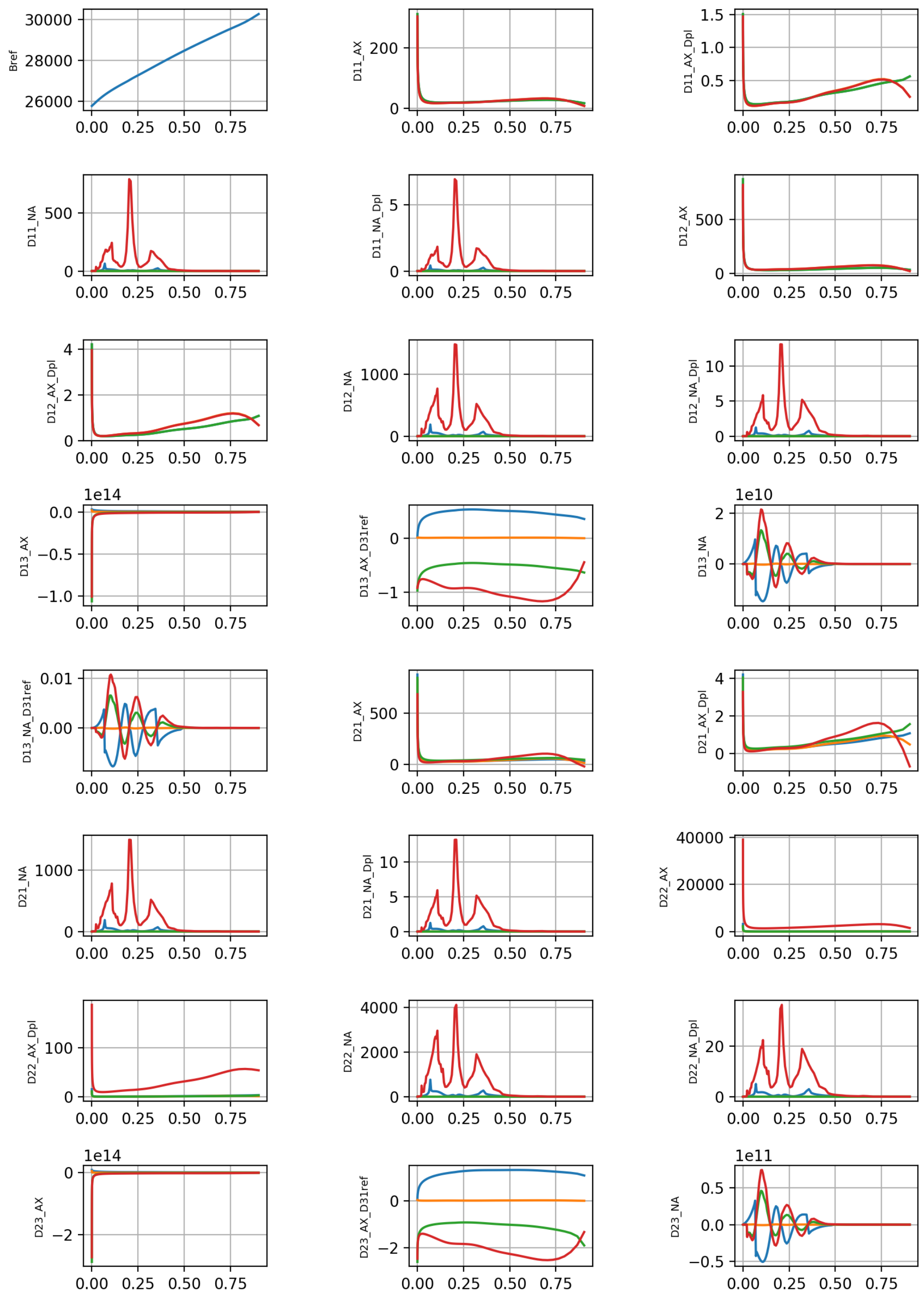


Figure 4.14: Overview plot of a NEO-2-QL multispecies scan run

4.3.3 Graphical data representation

This subsection, focused on the graphical data representation of the neo2tools package, is divided into three parts. The first part is about plotting the flux surface. The second part covers the magnetic field line; the third shows the distribution function for the selected PoI. In the first and the second part, examples of both stellarator and tokamak geometries are presented.

Plotting the flux surface

```
Job_surf=neo2tools.MagneticsPlot(rundir="/temp/singlerun/")
Job_surf.plot_magnetics()
```

Listing 4.1: Plotting of the flux surface

Listing 4.1 shows how to use the `MagneticsPlot` class without integration in another class. The first line in Listing 4.1 shows the instantiation with the required NEO-2 directory. The second line plots the flux surface.

The illustration of the flux surface in Boozer coordinates provides an overview of the relevant magnetic field for the NEO-2 run. Figure 4.15 and Figure 4.16 show examples of this flux surfaces in Boozer coordinates. The x-axis describes the toroidal angle φ and the y-axis the poloidal angle ϑ- The z-axis is showed in color representation and covers the magnetic field strength for each φ and ϑ pair.

Figure 4.15 shows the surface of a tokamak. The magnetic field in a tokamak does not depend on the φ direction. Because the coordinate system is designed that $\varphi \in [0, 2\pi)$ and $\vartheta \in [0, 2\pi)$, the whole flux surface is plotted. The data shown in Figure 4.15 originates from Asdex Upgrade in Garching. The magnetic field strength on the surface shown is between 1.7 T and 2.3 T.

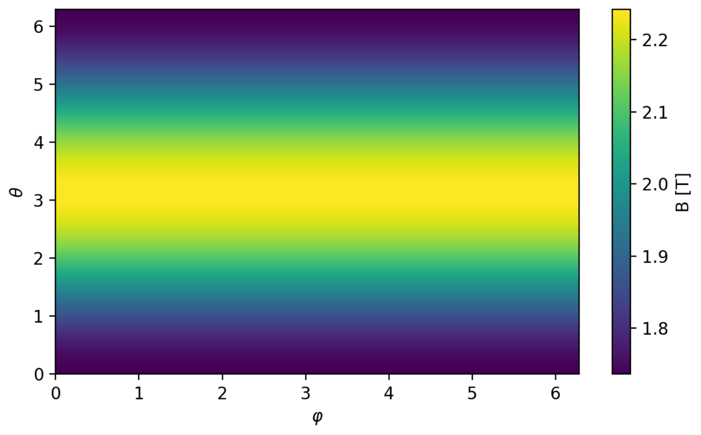


Figure 4.15: Flux surface of a tokamak

Figure 4.16 shows the flux surface of a stellarator. In contrast to the magnetic field in a tokamak, the magnetic field in a stellarator in not axisymmetric and does therefore dependent on φ. The data shown in Figure 4.16 is from the Wendelstein 7-x in Greifswald. Because the stellarator has a five-fold recurrence of the magnetic field, the x-axis only goes to $\frac{2\pi}{5}$. This symmetry is retrieved from the magnetic field input file for NEO-2. The magnetic field varies between 2.8 T and 3.5 T.

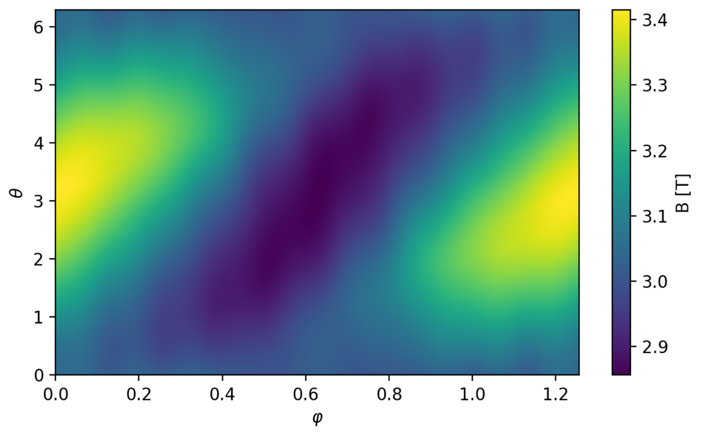


Figure 4.16: Flux surface of a stellarator

On top of the surface plot described above, the magnetic field line can be plotted. Because of the equilibrium of the magnetic field, the safety factor is a fixed value for each surface. The start tuple (φ, ϑ) of magnetic field line can be freely chosen. The endpoints will be calculated with the length of the line. Listing 4.2 shows how to pass the three parameters `length`, `startphi` and `starttheta` to the method `plot_magnetics()`.

```
Job_surf.plot_magnetics(length=15,startphi=0.5,starttheta=1)
```

Listing 4.2: Plotting of the flux surface with magnetic field line

Figure 4.17 shows the magnetic field line plotted on top of the flux surface of a stellarator. The start point is denoted with the blue, filled circle. The trajectory of the red magnetic field line demonstrates the periodic nature of the coordinate system. If the field line hits the boundary at 2π in ϑ or $\frac{2\pi}{5}$ in φ the line continues with the angle of 0 in the respective coordinate. The other coordinate remains unchanged.

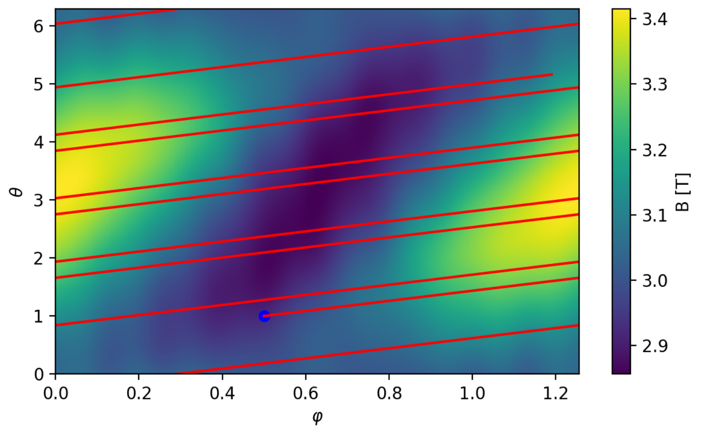


Figure 4.17: Flux surface and the magnetic field line

In Figure 4.18 the length of the field line is significantly raised. The longer the field line is, the closer every point can be approached by the field line. The `RadialScan` class described in Subsection 4.1.1 takes care, that the magnetic field does not close too early.

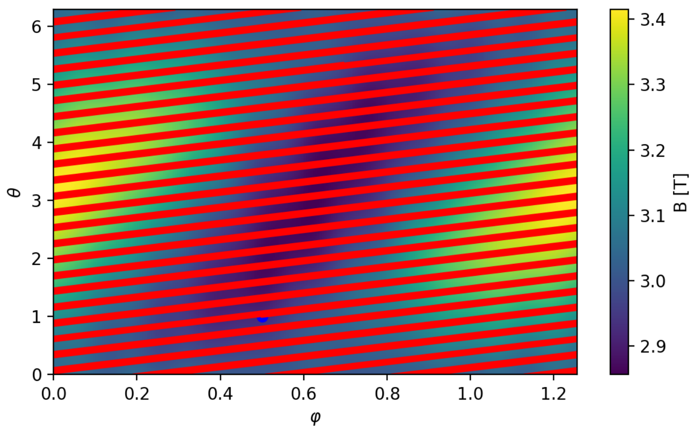


Figure 4.18: Flux surface covered by a long magnetic field line

Plotting of the magnetic field along the magnetic field line

This subsection shows the magnetic field following the magnetic field line on the flux surface, as discussed in the previous sites. NEO-2 saves the whole magnetic field line in the file `magnetics.h5`. To plot this magnetic field line, the class `MagneticsPlot` is used. In Listing 4.3 the plotting routine is shown. The parameter `rundir` has to point to a NEO-2 directory.

```
run_mag = "/temp/singlerun/"
Job_mag=MagneticsPlot(rundir=run_mag)
Job_mag.plot_poi(poi=[3,4,5],write=True)
```

Listing 4.3: Plotting the magnetic field line

The first line in Listing 4.3 defines the path to the NEO-2run directory. The second line shows the instantiation. In the `ReconPlot` class this is done automatically to the attribute `magplot`.

Figure 4.19 shows an example of a magnetic field line plot with three PoI. The x-axis of this plot is φ_s. This is the distance along the magnetic field line. This is the same length parameter as in Listing 4.2. In contrast to

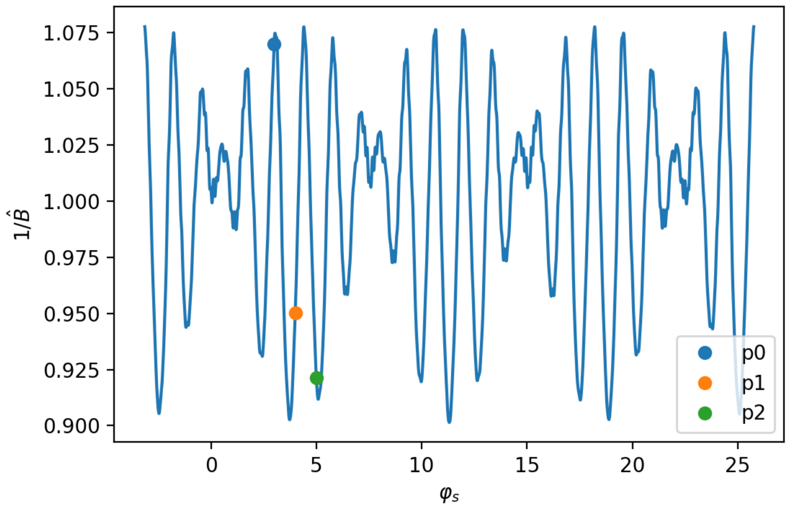


Figure 4.19: Magnetic field module along the magnetic field line with inserted Point of Interests

the flux surface plots, the magnetic strength is now plotted in $\frac{1}{\hat{B}}$, where $\hat{B}$ is the magnetic field module. In Figure 4.19 can be seen, that the full magnetic field line is closed. The start and the endpoint have the same value.

One purpose of plotting the magnetic field line, is defining PoI for plotting the distribution function. PoI are passed as values of the length along the magnetic field line φ_s. By default, plotting the PoI also converts these points for the reconstruction input file. If there is already an input file, this will be overwritten. To change this writing behaviour to the reconstruction input file, the parameter `write` has to set to false. In Section 4.3.3 it is explained how to move one PoI interactively.

Plotting of the distribution function

Plotting the distribution function g_k offers an insight to the velocity distribution of the particles. The plot method `plot_g()` offers this possibility to plot the distribution function of selected PoI on the magnetic

Table 4.4: Explanation of parameter type for plot_g()

plot type	description
g	distribution function
gpa	parallel derivative of the distribution function
gtr	orthogonal derivative of the distrbution function

field line. Additionally, derivatives with respect to the direction of the particle motion can be plotted. The plot type has to be passed at the first position to the method `plot_g()`. Table 4.4 explains the possible plot types.

The instantiation is done by assigning the parameter `rundir` the desired NEO-2 run. The `plotdir` parameter is the name were the plots will be saved. The parameter `index` defines the chosen distribution function.

```
Job_recon=neo2tools.ReconPlot(rundir="/temp/singlerun/",
                              plotdir="Plot",index=1)
Job_recon.plot_g("g")
```

Listing 4.4: Plotting of the distribution function g_1

Listing 4.4 shows the command for plotting the distribution function for the PoI in Figure 4.19. As the distribution function is a four-dimensional function and through fixing the spatial position, it is only a one-dimensional function in velocity space left. Figure 4.20 shows the distribution function g_1 plotted over $\lambda = \frac{v_\parallel}{v}$. g_1 is plotted, because the parameter `index` was set to 1 at the instantiation of `Job_recon`.

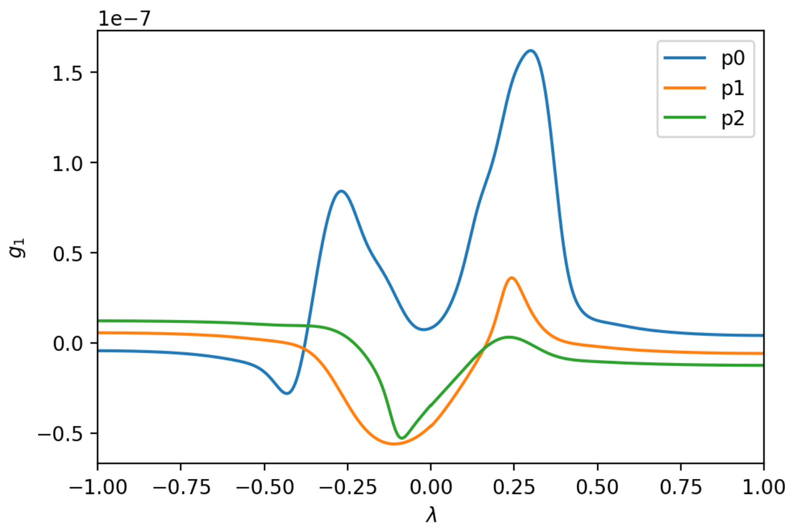


Figure 4.20: Distribution function g_1 for three different Point of Interests

The derivative of the generalized Spitzer function in the field line direction is from special interest. Depending on the spatial point, where the microwave hits the plasma, the current generations is constructive or destructive. Therefore, g_3 is plotted together with $\frac{\partial g_3}{\partial v_\parallel}$. Figure 4.21 shows g_3 and $\frac{\partial g_3}{\partial v_\parallel}$ of the three spatial points from Figure 4.19. The commands is stated in Listing 4.5.

A new instance of `Reconplot` is needed, because the index of g changed to 3. The first line in Listing 4.5 shows this. The second line displays the passing of the two types `"g"` and `"gpa"` as string to the `plot_g()` method. If more than one type of distribution function is plotted, (see Table 4.4), the different lines can be grouped with setting the parameter `subplot` to `True` as it is displayed.

```
Job_spitz=neo2tools.ReconPlot(run1,plot1,template1,index=3)
Job_spitz.plot_g("g","gpa",subplot=True)
```

Listing 4.5: Plotting the distribution function with subplots

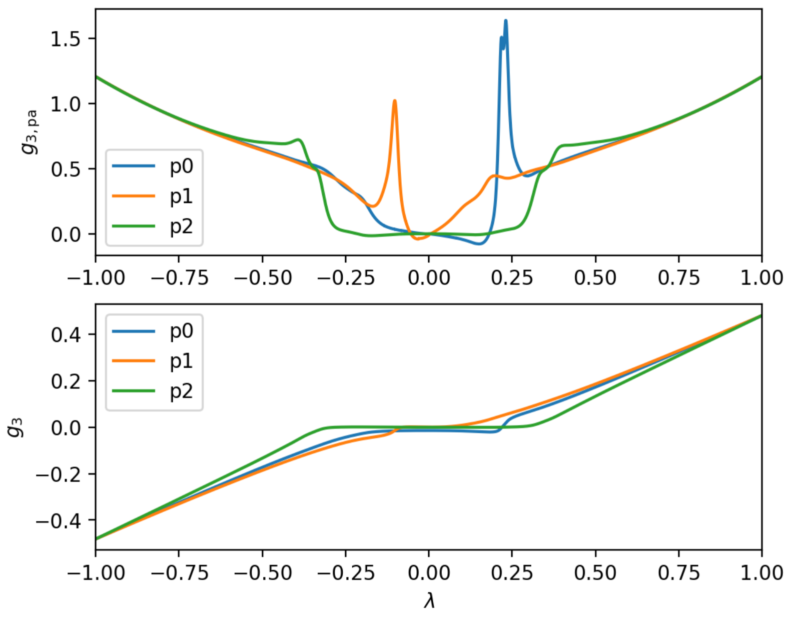


Figure 4.21: Distribution function g_3 and the derivative $\frac{\partial g_3}{\partial v_\parallel}$ for three different Point of Interests

Another way to compare different distribution functions is to use the `MultipleReconPlot` class. With this class, one PoI has to be chosen. The second input parameter are the different NEO-2 runs.

The bootstrap coefficient λ_{bb} from Figure 4.12 is based on the gradient driven distribution function g_1. Of special interest is g_1 at spatial points next to the trapped-passing boundary. With the selection of one PoI next to a local maxima, the effect of different collisionalities to g_1 is illustrated.

The result for different collisionalities of Listing 4.6 is shown in Figure 4.22.

```
Job_mult=neo2tools.MultipleReconPlot(rundirs,index=1)
Job_mult.magplot._write_poi([1,745])
Job_mult.plot_g("g")
```

Listing 4.6: MultipleReconPlot

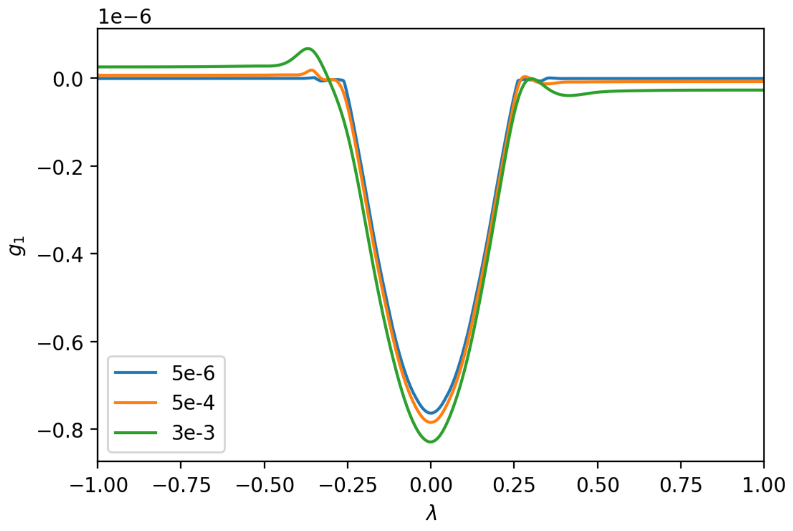


Figure 4.22: Distribution function g_1 for three different collisionalities

Because of the symmetric nature of g_1 near local maxima, the second method `plot_g_anti()` plots the antisymmetric part $g^a = (g^+ - g^-)/2$ of the distribution function, where the upper index is the sign of the velocity. In addition, in `plot_g_anti()` the x-axes is η defined in Equation 3.3. The command for this plot routine is stated in Listing 4.7.

```
Job_mult.plot_g_anti("g")
```

Listing 4.7: Plotting with the antisymmetric plot function

The graphical output of the antisymmetric distribution function is shown in Figure 4.23. The broadening of the distribution function with higher collisionalities can be clearly seen.

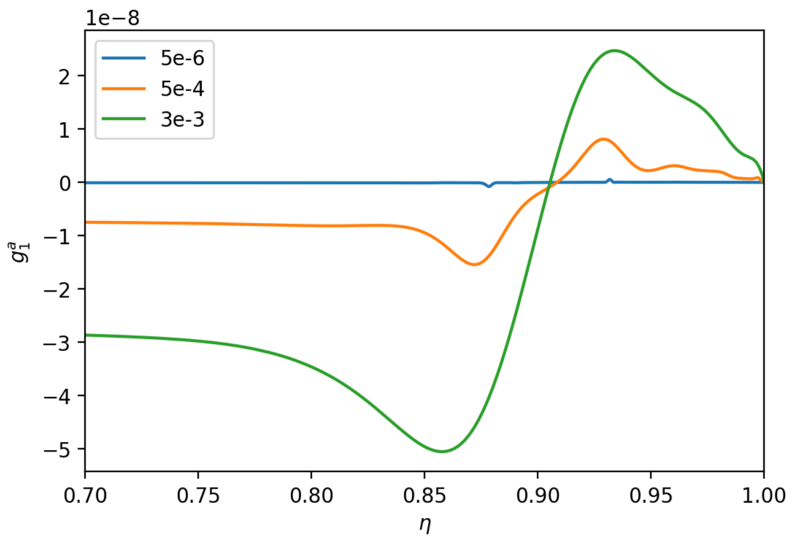


Figure 4.23: Antisymmetric part of the distribution function g_1

4.3.4 Interactive Jupyter elements

Alternatively to the method `plot_g()`, the distribution function's plot can also be generated in an interactive mode. Listing 4.8 shows how the interactive mode is started.

```
Job_recon.interactive_mode()
```

Listing 4.8: Interactive mode of plot_g

Figure 4.24 shows an example of the graphical interface produced, upon starting the interactive mode. On the left side, there is the magnetic field line and on the right side, the distribution function. There are four interactive input elements in the interface. There is one text input cell. Two sliders and one dropdown menu.

The text field on the upper left is directly connected with the slider below. With these, the PoI can be changed along the magnetic field line. Directly connected to the position of the PoI is the plot on the right side.

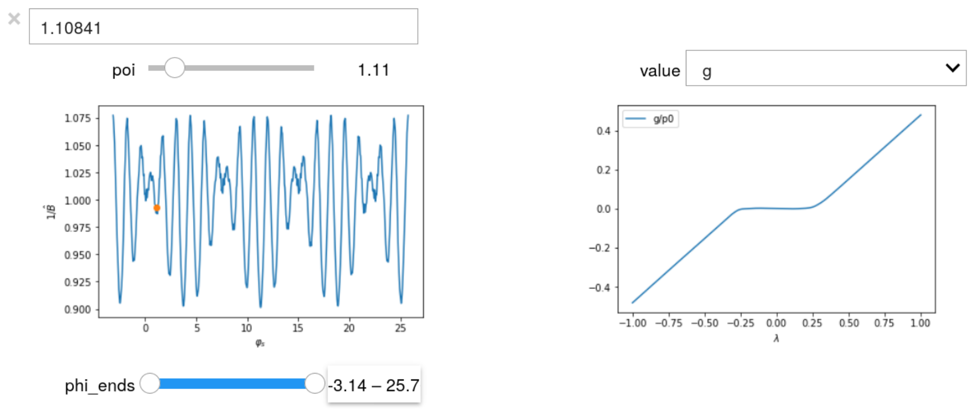


Figure 4.24: Graphical output of the interactive mode

If the PoI changes, the distribution function of the right plot updates, according to the new position of the PoI.

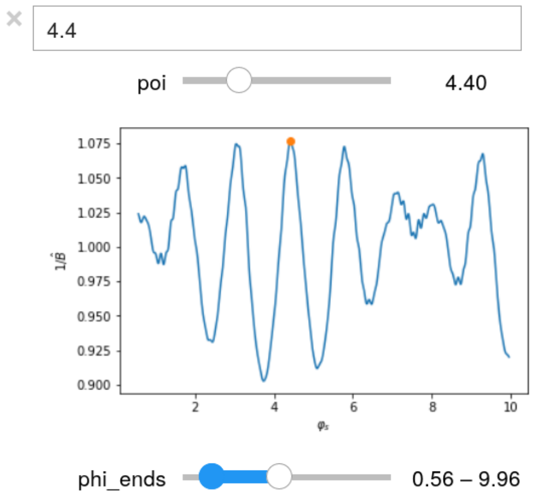


Figure 4.25: Graphical output of the interactive mode, Range Slider

Figure 4.25 shows a zoom to the left side of the interactive interface. It displays the magnetic field line. The range slider on the bottom controls how much of the magnetic field line is displayed in the plot above. The slider's both end positions are the start and the end position of the magnetic field line. The current values for them are shown, on the right side of the slider.

In Figure 4.26 the expanded dropdown menu is shown.

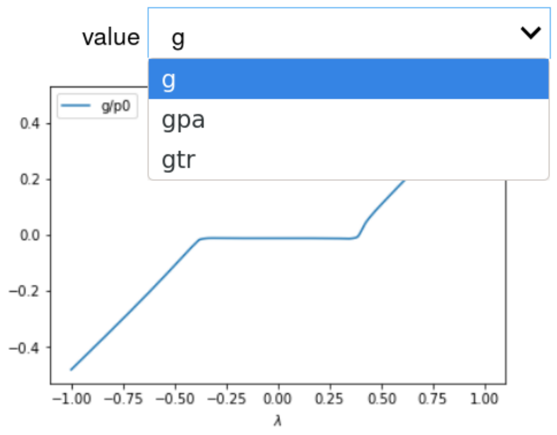


Figure 4.26: Graphical output of the interactive mode, Dropdown menu

5 Summary

This thesis aims to provide different tools to facilitate the application of the NEO-2 package. Basic concepts of the drift kinetic equation, neoclassical transport and the coordinate system used are introduced.

The drift kinetic equation solver NEO-2 is explained in more detail. The origin of the matrix elements is shown and their precomputation is described. In addition, Python, its interactive version IPython and the development environment Jupyter Notebooks are explained.

The Python toolbox neo2tools developed in this thesis is explained. The application procedure is divided into three sections, covering the preprocessing, the runtime optimization and the postprocessing of NEO-2 runs. The preprocessing part is for setting up single and multiple runs for NEO-2. Depending on the branch used, different prerequisites are required. Therefore, multiple classes were constructed. To simplify the application of the toolbox, a common class for all branches was created. Derived from this common class, different classes for different use cases were implemented. A special radial scanning class ensures that the setting of the correct flux surfaces is done correctly. A special mode of NEO-2, the reconstruction run, which enables saving the distribution function, is also explained. Examples are shown to demonstrate the application of the classes in Jupyter Notebook.

With the precomputation of the matrix elements, the calculation of the matrix elements is separated from solving the drift kinetic equation and therefore it can save expensive computation time. It can also be utilized by multiple NEO-2 runs.

Furthermore, the application procedure and exemplary results of different visualization tools are shown. These tools can be used for plotting the results of NEO-2 runs and NEO-2 reconstruction runs, configured and deployed with the methods explained before. The predefined plot routines enable the visualization of flux surfaces, the magnetic field line along the flux surfaces, neoclassical transport coefficients or the selected distribution functions. A special mode to visualize the distribution function from the reconstruction run is the interactive mode provided by Jupyter Notebook. In the interactive mode, one point of interest can be moved along the magnetic field line. The distribution function, corresponding to the selected point, is updated automatically.

Bibliography

[1] IPCC Climate Change et al. *The physical science basis*. 2007.

[2] Per Helander and Dieter J. Sigmar. *Collisional Transport in Magnetized Plasmas*. Cambridge University Press, 2002. ISBN: 978-0-521-80798-2.

[3] Winfried Kernbichler et al. "Recent Progress in NEO2 — A Code for Neoclassical Transport Computations Based on Field Line Tracing." In: *Plasma and Fusion Research* 3 (Jan. 2008). DOI: 10.1585/pfr.3.S1061.

[4] W. D. D'haeseleer, W. N. G. Hitchon, J. D. Callen, J. L. Shohet. *Flux Coordinates and Magnetic Field Structure. A Guide to a Fundamental Tool of Plasma Theory*. Springer Berlin Heidelberg, 1991. ISBN: 978-3-642-75597-2.

[5] Jeffrey P. Freidberg. *Plasma Physics and Fusion Energy*. Cambridge University Press, 2007. ISBN: 978-0-521-85107-7.

[6] Harold Grad. "Toroidal Containment of a Plasma." In: *The Physics of Fluids* 10.1 (1967), pp. 137–154. DOI: 10.1063/1.1761965.

[7] Robert G. Littlejohn. "Variational principles of guiding centre motion." In: *Journal of Plasma Physics* 29.1 (1983), pp. 111–125. DOI: 10.1017/S002237780000060X.

[8] Gernot Kapper. "Impact of finite plasma collisionality on the current drive efficiency in tokamaks and stellarators." PhD thesis. Institute of Theoretical and Computational Physics, Graz University of Technology, 2017.

[9] S. P. Hirshman et al. “Plasma transport coefficients for nonsymmetric toroidal confinement systems.” In: *The Physics of Fluids* 29.9 (1986), pp. 2951–2959. DOI: 10.1063/1.865495.

[10] Andreas Martitsch. “Modeling of Plasma Rotation in Tokamaks.” PhD thesis. Institute of Theoretical and Computational Physics, Graz University of Technology, 2016.

[11] V. V. Nemov et al. “Evaluation of 1/ neoclassical transport in stellarators.” In: *Physics of Plasmas* 6.12 (1999), pp. 4622–4632. DOI: 10.1063/1.873749.

[12] W Kernbichler et al. “Solution of drift kinetic equation in stellarators and tokamaks with broken symmetry using the code NEO-2.” In: *Plasma Physics and Controlled Fusion* 58.10 (Aug. 2016), p. 104001. DOI: 10.1088/0741-3335/58/10/104001.

[13] E. Strumberger, S. Günter, and C. Tichmann. “MHD instabilities in 3D tokamaks.” In: *Nuclear Fusion* 54.6 (May 2014), p. 064019. DOI: 10.1088/0029-5515/54/6/064019.

[14] Georg O. Leitold. “Computation of neoclassical transport coefficients and generalized Spitzer functions in toroidal fusion plasmas.” PhD thesis. Institute of Theoretical and Computational Physics, Graz University of Technology, 2010.

[15] H. Maaßberg, C. D. Beidler, and N. B. Marushchenko. “Electron cyclotron current drive modelling with parallel momentum correction for tokamaks and stellarators.” In: *Physics of Plasmas* 19.10 (2012), p. 102501. DOI: 10.1063/1.4751436.

[16] Fernando Pérez and Brian E. Granger. “IPython: a System for Interactive Scientific Computing.” In: *Computing in Science and Engineering* 9.3 (May 2007), pp. 21–29. ISSN: 1521-9615. DOI: 10.1109/MCSE.2007.53. URL: https://ipython.org.

[17] Helen Shen. “Interactive notebooks: Sharing the code.” In: *Nature* 515.7525 (2014), pp. 151–152. DOI: 10.1038/515151a.

[18] Gernot Kapper et al. "Electron cyclotron current drive simulations for finite collisionality plasmas in Wendelstein 7-X using the full linearized collision model." English. In: *Physics of plasmas* 23.11 (Nov. 2016), p. 112511. ISSN: 1070-664X. DOI: 10.1063/1.4968234.

[19] *Jupyter, Mathematica, and the Future of the Research Paper*. https://paulromer.net/jupyter-mathematica-and-the-Future-of-the-Research-Paper. Accessed: 2020-09-07.

[20] *Gravitational Wave Open Science Center 2020 GWOSC tutorials*. www.gw-openscience.org/tutorials/. Accessed: 2020-07-13.

[21] R. Abbott et al. (LIGO Scientific Collaboration and Virgo Collaboration). *Open data from the first and second observing runs of Advanced LIGO and Advanced Virgo*. 2019. arXiv: 1912.11716 [gr-qc].

[22] Marshall L. Ward. "f90nml - A Python module for Fortran namelists." In: *Journal of Open Source Software* 4.38 (2019), p. 1474. DOI: 10.21105/joss.01474.

[23] Douglas Thain, Todd Tannenbaum, and Miron Livny. "Distributed computing in practice: the Condor experience." In: *Concurrency - Practice and Experience* 17.2-4 (2005), pp. 323–356.

[24] *HTCondor Manual*. Accessed: 2020-10-20. 2020. URL: https://htcondor.readthedocs.io/en/latest/.

[25] S. Kasilov et al. "Evaluation of the parallel current density in a stellarator using the integration technique along the magnetic field line." In: *27th EPS Conference on Contr. Fusion and Plasma Phys. Budapest, 12-16 June* (2000).