Computational Science and Engineering
(International Master's Program)

Technische Universität München

Master's Thesis

Parallel gyrokinetic simulations with Python

Emily Bourne

Computational Science and Engineering
(International Master's Program)

Technische Universität München

Master's Thesis

Parallel gyrokinetic simulations with Python

Author:	Emily Bourne
1st examiner:	Univ.-Prof. Dr. Eric Sonnendrücker
Assistant advisor:	Dr. rer. nat. Yaman Güçlü
Submission Date:	28th September, 2018

I hereby declare that this thesis is entirely the result of my own work except where otherwise indicated. I have only used the resources given in the list of references.

28th September, 2018 Emily Bourne

Acknowledgments

I would like to thank Yaman Güçlü for his help and support throughout this project. I am also grateful to Ahmed Ratnani and Saïd Hadjout for their support with all pyccel related queries. Finally I would like to thank Eduardo Zoni for providing the scaling results and timings for the original Fortran implementation of the problem.

Abstract

Pyccel, a python to human-readable Fortran translator, is used to test the feasibility of writing large parallel simulations in python without significant loss of runtime speed as compared to writing in Fortran directly. A parallel simulation of a plasma using gyrokinetic theory is written for this purpose. The simulation uses a field-aligned semi-lagrangian method. Cylindrical (screw-pinch) geometry is used to approximate the shape of the domain. The acceleration obtained using Pyccel is very large. Although the speed of the pure Fortran version is not quite reached, the decrease in the amount of development time required largely offsets this small loss.

Contents

Chapter 1

Introduction

As an interpreted language, Python runs significantly slower than other languages such as C or Fortran. However code can usually be developed much faster in Python. As a result it is preferred by many developers for smaller projects. Although the resulting program runs slower, the trade-off is acceptable as the overall time spent on the project is lower.

The situation is more problematic for large projects. In this case the runtime can be several days. A program written in python often runs as much as 100 times slower than its equivalent in C or Fortran. This would therefore lead to a runtime of close to one year. This is especially problematic as the simplicity of python means that many scientists rarely use lower level languages and are thus out of practice when they need to use them.

Multiple solutions exist to try and improve this situation which consist of accelerating python code so that it runs at speeds which more closely approach those seen in C or Fortran. One type of solution is just-in-time compilers such as PyPy and numba. Pypy is especially powerful as it requires no changes to the code. It can thus adapt the function to the types that are used when it is called, however many packages are not supported and the entire code must be run using pypy. In contrast numba does not need to be used everywhere, but cannot be used immediately. It requires the types used by the function to be declared in a decorator. This however means that it can reuse the compiled code making it faster when calls are reused a lot.

Another alternative is to use ahead-of-time compilers such as cython or pythran. Numba also has an ahead-of-time compiler. These tools can be more cumbersome. The type of the arguments must always be given, however they can potentially lead to large speed-ups when used. These tools each work in different ways, for example numba requires only a decorator added to the function which will be compiled. It then compiles the code directly and generates a shared object file. In contrast cython translates the code to C which can then be compiled and called from python. The advantage of generating code in another language is that it can be compiled with different compilers. It is also possible to modify the resulting code if there are improvements possible which cannot be mimicked in python. Unfortunately the code generated by cython is rarely human readable so this second advantage is lost.

In this project another ahead-of-time compiler will be used: pyccel. Like cython, Pyccel translates the python code into another language, in this case

Fortran. However, unlike for cython, the code generated by pyccel is human-readable. It is also simple to use, requiring only that a decorator be added to the functions to be translated. Its simplicity, the scale of the speed-up that can be achieved, and the fact that the generated code can be modified to give improve the functions further when required, make pyccel a very useful tool for rapidly developing projects.

The magnitude of the difference in speed between Python and C or Fortran is however so large that it is important to examine the effect that pyccel can have on all aspects of the development of a project in order to determine whether the speed-up achieved is sufficient to allow the development of scientific code in python. In this project a gyrokinetic plasma simulation is developed in python. It is then accelerated using pyccel and the resulting program is compared with a Fortran version of the same simulation.

The simulation which will be explored was first developed by Latu et al. [1] in their paper Field-Aligned Interpolation for Semi-Lagrangian Gyrokinetic Simulations. In plasma simulations, many points are needed in order to accurately represent all of phase space. This is very costly. As a result many methods have been developed allowing the number of points needed, to be reduced. Gyrokinetic theory allows the problem to be reduced from six dimensions to five (or four for collisionless models). The field-aligned interpolation method explored in the paper by Latu et al. allows fewer points to be used along the toroidal axis

Chapter 2

Gyrokinetics

2.1 Introduction

Particles in a plasma follow a complicated movement determined by electric and magnetic fields. The combination of the fields makes the particles move in a spiral motion around a guiding centre as shown by figure 2.1.

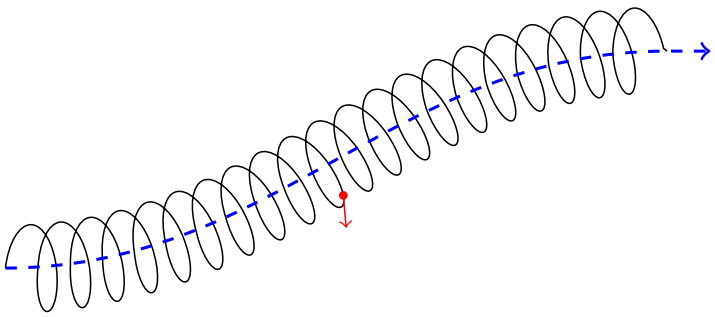


Figure 2.1: Path of a particle in a magnetic field and the guiding centre of that particle (shown in dashed blue)

This gyro-rotation is problematic for simulations as this motion means that the phase-space coordinates of the particle change rapidly. As a result, small time steps are required to obtain an accurate solution. Fortunately the high frequency gyro-rotation does not have an important effect on the low frequency physics demonstrated by the plasma, as a result if this motion can be rigorously removed from the equations of motion then the resulting equations which are less constrained can be used to analyse low frequency behaviour. In addition, the removal of this high frequency motion allows the reduction of the 6-D problem to a 5-D problem. This is the aim of gyrokinetics.

There are multiple methods used for finding the equations used in gyrokinetics (for example as described by Sugama et al [2], Brizard [3], and others) which include some kind of coordinate transformations. The result of these transformations is that the coordinates are $(\boldsymbol{x}, v_\parallel, \mu)$ where $\boldsymbol{x}$ is the position of the gyro centre, $v_\parallel$ is the velocity parallel to the time-independent background magnetic field B_0, and μ is the conserved magnetic moment. Normally there

would also be a coordinate θ representing the gyroangle which is necessary to handle time-dependent fluctuations in the magnetic field, however the gyrokinetic transformations move these dependencies so that they are averaged which then removes them from the final equations. As a result the problem is reduced to five dimensions.

2.2 Continuity Equation

In the simulations, the plasma is described by a distribution function. This function describes the positions of a collection of identical particles of charge $q \neq 0$ and mass $m > 0$, immersed in a static magnetic field B(x). This value of this function is therefore conserved along particle trajectories [4]:

$$f\left(\boldsymbol{Z}\left(\boldsymbol{Z}_0, t\right), t\right) = f\left(\boldsymbol{Z}_0, t_0\right) \tag{2.1}$$

The corresponding conservation equation is therefore defined as follows:

$$\frac{d}{dt} f\left(\boldsymbol{Z}\left(\boldsymbol{Z}_0, t\right), t\right) = \frac{\partial}{\partial t} f\left(\boldsymbol{Z}, t\right) + \sum_{\alpha} \frac{d\boldsymbol{Z}_\alpha}{dt} \frac{\partial}{\partial \boldsymbol{Z}} f\left(\boldsymbol{Z}, t\right) = 0 \tag{2.2}$$

For the gyro-centre distribution function which is used for gyrokinetic theory, $f \in \mathbb{R}_+$ and is a function of time $t \in \mathbb{R}_+$ and of the phase space coordinates $\left(\boldsymbol{x}, v_\parallel, \mu\right)$. This yields the following function:

$$\frac{\partial}{\partial t} f + \dot{\boldsymbol{x}} \cdot \vec{\nabla} f + \dot{v}_\parallel \frac{\partial f}{\partial v_\parallel} + \dot{\mu} \frac{\partial f}{\partial \mu} = 0 \tag{2.3}$$

The values of $\boldsymbol{u} = \dot{\boldsymbol{x}}$, $a_\parallel = \dot{v}_\parallel$, $M = \dot{\mu}$ can be determined using the Euler-Lagrange equations:

$$\frac{d}{dt}\left(\frac{\partial \mathcal{L}}{\partial \dot{\boldsymbol{Z}}_\alpha}\right) - \frac{\partial \mathcal{L}}{\partial \boldsymbol{Z}_\alpha} = 0 \tag{2.4}$$

where $\mathcal{L}$ is the Lie-transformed low-frequency particle Lagrangian obtained using gyrokinetic theory, defined (in SI units) as follows [4]:

$$L = \left(e\boldsymbol{A} + m v_\parallel \hat{b}\right) \cdot \dot{\boldsymbol{x}} + \frac{m\sqrt{4\pi}}{e\mu_0^{\frac{3}{2}}} \mu \dot{\theta} - H\left(\boldsymbol{x}, v_\parallel\right) \tag{2.5}$$

Although θ is not one of the gyrokinetic coordinates, it is a phase-space coordinate and as such can also be used to obtain useful equations. The Euler-Lagrange equation for θ is as follows:

$$\frac{d}{dt}\left(\frac{m\sqrt{4\pi}}{e\mu_0^{\frac{3}{2}}}\mu\right) + 0 = 0 \tag{2.6}$$

$$\dot{\mu} = 0 \tag{2.7}$$

This implies that $M = \dot{\mu} = 0$.

The Euler-Lagrange equation for $v_\parallel$ is as follows:

$$\frac{d}{dt}\Big(0\Big) - m\hat{b}\cdot\dot{\boldsymbol{x}} + \frac{\partial H}{\partial v_\parallel} = 0 \tag{2.8}$$

$$\hat{b}\cdot\dot{\boldsymbol{x}} = \frac{1}{m}\frac{\partial H}{\partial v_\parallel} \tag{2.9}$$

The Euler-Lagrange equation for $\boldsymbol{x}$ is as follows:

$$\frac{d}{dt}\left(e\boldsymbol{A} + mv_\parallel\hat{b}\right) - e\vec{\nabla}\left(A\cdot\dot{\boldsymbol{x}}\right) - mv_\parallel\vec{\nabla}\left(\hat{b}\cdot\dot{\boldsymbol{x}}\right) + \vec{\nabla}H = 0 \tag{2.10}$$

$$e\mathcal{J}_{\boldsymbol{A}}\dot{\boldsymbol{x}} + ma_\parallel\hat{b} + mv_\parallel\mathcal{J}_{\hat{b}}\dot{\boldsymbol{x}} - e\mathcal{J}_{\boldsymbol{A}}^T\dot{\boldsymbol{x}} - mv_\parallel\mathcal{J}_{\hat{b}}^T\dot{\boldsymbol{x}} + \vec{\nabla}H = 0 \tag{2.11}$$

$$\dot{v_\parallel}\hat{b} = \frac{e}{m}\left(\mathcal{J}_{\boldsymbol{A}}^T - \mathcal{J}_{\boldsymbol{A}}\right)\dot{\boldsymbol{x}} + v_\parallel\left(\mathcal{J}_{\hat{b}}^T - \mathcal{J}_{\hat{b}}\right)\dot{\boldsymbol{x}} - \frac{1}{m}\vec{\nabla}H \tag{2.12}$$

We now simplify the equations by introducing the following definitions:

$$\boldsymbol{A}^* = \boldsymbol{A} + \frac{mv_\parallel}{e}\hat{b} \tag{2.13}$$

$$\boldsymbol{B}^* = \vec{\nabla}\times\boldsymbol{A}^* = \boldsymbol{B} + \frac{mv_\parallel}{e}\vec{\nabla}\times\hat{b} \tag{2.14}$$

Which leaves us with the equation:

$$\dot{v_\parallel}\hat{b} = \frac{e}{m}\left(\mathcal{J}_{\boldsymbol{A}^*}^T - \mathcal{J}_{\boldsymbol{A}^*}\right)\dot{\boldsymbol{x}} - \frac{1}{m}\vec{\nabla}H \tag{2.15}$$

This equation can be simplified by remarking that $\left(\mathcal{J}_A^T - \mathcal{J}_A\right)\boldsymbol{C} = \boldsymbol{C}\times\left(\vec{\nabla}\times\boldsymbol{A}\right)$ for all vector $\boldsymbol{C}$:

$$\begin{aligned}\left[\boldsymbol{C}\times\left(\vec{\nabla}\times\boldsymbol{A}\right)\right]_i &= \varepsilon_{ijk}c_j\varepsilon_{klm}\partial_l a_m\\ &= \varepsilon_{kij}\varepsilon_{klm}c_j\partial_l a_m\\ &= \left(\delta_{il}\delta_{jm} - \delta_{im}\delta_{jl}\right)c_j\partial_l a_m\\ &= c_j\partial_i a_j - c_j\partial_j a_i\\ &= \left(\mathcal{J}_{A\,ij}^T - \mathcal{J}_{A\,ij}\right)c_j\\ &= \left[\left(\mathcal{J}_A^T - \mathcal{J}_A\right)\boldsymbol{C}\right]_i\end{aligned}$$

Equation 2.15 is then written as follows:

$$\dot{v_\parallel}\hat{b} = \frac{e}{m}\dot{\boldsymbol{x}}\times\left(\vec{\nabla}\times\boldsymbol{A}^*\right) - \frac{1}{m}\vec{\nabla}H \tag{2.16}$$

$$= \frac{e}{m}\dot{\boldsymbol{x}}\times\boldsymbol{B}^* - \frac{1}{m}\vec{\nabla}H \tag{2.17}$$

This expression can now be used to find a definition of $a_\parallel = \dot{v_\parallel}$. This is done by taking the dot product of the equation and $\hat{b}$:

$$\dot{v_\parallel}\boldsymbol{B}^* \cdot \hat{b} = \frac{e}{m}\boldsymbol{B}^* \cdot (\dot{\boldsymbol{x}} \times \boldsymbol{B}^*) - \frac{1}{m}\boldsymbol{B}^* \cdot \vec{\nabla} H \tag{2.18}$$

$$\dot{v_\parallel} = -\frac{\boldsymbol{B}^*}{mB_\parallel^*} \cdot \vec{\nabla} H \tag{2.19}$$

where $B_\parallel^* = \hat{b} \cdot \boldsymbol{B}^*$.

The definition of $\boldsymbol{u} = \dot{\boldsymbol{x}}$ can also be found from equations 2.9 and 2.17 by taking the cross product of equation 2.17 and $\hat{b}$:

$$\dot{v_\parallel}\hat{b} \times \hat{b} = \frac{e}{m}\hat{b} \times (\dot{\boldsymbol{x}} \times \boldsymbol{B}^*) - \frac{1}{m}\hat{b} \times \vec{\nabla} H \tag{2.20}$$

$$0 = \frac{e}{m}\left(\left(\hat{b} \cdot \boldsymbol{B}^*\right)\dot{\boldsymbol{x}} - \left(\hat{b} \cdot \dot{\boldsymbol{x}}\right)\boldsymbol{B}^*\right) - \frac{1}{m}\hat{b} \times \vec{\nabla} H \tag{2.21}$$

$$\frac{eB_\parallel^*}{m}\dot{\boldsymbol{x}} = \frac{e}{m^2}\frac{\partial H}{\partial v_\parallel}\boldsymbol{B}^* + \frac{1}{m}\hat{b} \times \vec{\nabla} H \tag{2.22}$$

$$\dot{\boldsymbol{x}} = \frac{1}{mB_\parallel^*}\frac{\partial H}{\partial v_\parallel}\boldsymbol{B}^* + \frac{1}{eB_\parallel^*}\hat{b} \times \vec{\nabla} H \tag{2.23}$$

The equations obtained using the Euler Lagrange equations (equations 2.7 2.19 2.23) can now be substituted into equation 2.3 to define the problem:

$$\frac{\partial}{\partial t}f + \boldsymbol{u} \cdot \vec{\nabla} f + a_\parallel \frac{\partial f}{\partial v_\parallel} + \boldsymbol{M}\frac{\partial f}{\partial \mu} = 0 \tag{2.24}$$

$$\boldsymbol{u} = \frac{1}{mB_\parallel^*}\frac{\partial H}{\partial v_\parallel}\boldsymbol{B}^* + \frac{1}{eB_\parallel^*}\hat{b} \times \vec{\nabla} H \tag{2.25}$$

$$a_\parallel = -\frac{\boldsymbol{B}^*}{mB_\parallel^*} \cdot \vec{\nabla} H \tag{2.26}$$

$$\boldsymbol{M} = 0 \tag{2.27}$$

This expression depends on the Hamiltonian of the system which is defined using gyrokinetic theory as follows (in SI units) [1]:

$$H = \frac{1}{2}mv_\parallel^2 + \mu B(\boldsymbol{x}) + e\langle\phi\rangle_g(t, \boldsymbol{x}) \tag{2.28}$$

This leaves the following definitions for the advection coefficients:

$$\boldsymbol{u} = \frac{v_\parallel \boldsymbol{B}^*}{B_\parallel^*} + \frac{1}{eB_\parallel^*}\hat{b} \times \left(\mu\vec{\nabla}B(\boldsymbol{x}) + e\vec{\nabla}\phi\right) \tag{2.29}$$

$$a_\parallel = -\frac{\boldsymbol{B}^*}{mB_\parallel^*} \cdot \left(\mu\vec{\nabla}B(\boldsymbol{x}) + e\vec{\nabla}\phi\right) \tag{2.30}$$

2.2.1 Cylindrical Geometry

Generally the definition of the magnetic field used will be that used by Latu et al [1]:

$$\boldsymbol{B} = \frac{B_0 R_0}{R}\left(\zeta(r)\hat{\theta} + \hat{\varphi}\right) \qquad \zeta(r) = \frac{r}{q(r)R_0} \tag{2.31}$$

where R_0 is the major radius, B_0 is the toroidal magnetic field at the magnetic axis, $R(r,\theta) = R_\theta + r\cos\theta$, and $q(r)$ is the classical safety factor in the large aspect ratio limit ($\frac{r}{R_0} \to 0$).

In the simplified case of a straight periodic cylinder, the toroidal angular variable $\hat{\varphi}$ is replaced by a straight variable $\hat{z} = R_0\hat{\varphi}$, and R is taken to be equal to R_0. This leaves the following expression for the magnetic field:

$$\boldsymbol{B} = B_0\left(\zeta(r)\hat{\theta} + \hat{z}\right) \qquad \zeta(r) = \frac{\iota(r) r}{R_0} \tag{2.32}$$

where $\iota(r) = \frac{1}{q(r)}$.

The unit vector $\hat{b}$ is therefore defined as follows:

$$\hat{b} = \begin{cases} b_r = 0 \\ b_\theta = \frac{s\zeta}{\sqrt{1+\zeta^2}} \\ b_z = \frac{s}{\sqrt{1+\zeta^2}} \end{cases} \tag{2.33}$$

where $s = \operatorname{sgn}(B_0)$.

This formulation has the advantage of simplifying equations 2.29 and 2.30 as $\vec{\nabla} B = \mathbf{0}$. This can be proven using the fact that $B = \|\boldsymbol{B}\|$ and that it only depends on r. This means that $\left(\vec{\nabla} B\right)_\theta = \left(\vec{\nabla} B\right)_z = 0$ and $\left(\vec{\nabla} B\right)_r$ is determined below:

$$\begin{aligned}
\left(\vec{\nabla} B\right)_r =& \partial_r \sqrt{\sum_i B_i(r)^2} \\
=& \frac{1}{B}\sum_i B_i(r)\partial_r B_i(r) \\
=& \frac{1}{B}\left(B_\theta \partial B_\theta + B_z \partial_r B_z\right) \\
=& \frac{1}{B}\left(\frac{s\zeta}{(1+\zeta^2)^{\frac{1}{2}}}\frac{s\partial_r\zeta}{(1+\zeta^2)^{\frac{3}{2}}} + \frac{s}{(1+\zeta^2)^{\frac{1}{2}}}\frac{-s\zeta\partial_r\zeta}{(1+\zeta^2)^{\frac{3}{2}}}\right) \\
=& \frac{1}{B}\left(\frac{\zeta\partial_r\zeta}{(1+\zeta^2)^2} - \frac{\zeta\partial_r\zeta}{(1+\zeta^2)^2}\right) \\
=& 0
\end{aligned}$$

Equations 2.29 and 2.30 can therefore be written as follows:

$$\boldsymbol{u} = \frac{v_\parallel \boldsymbol{B}^*}{B_\parallel^*} + \frac{1}{B_\parallel^*}\hat{b} \times \vec{\nabla}\phi \tag{2.34}$$

$$a_\parallel = -\frac{e\boldsymbol{B}^*}{mB_\parallel^*} \cdot \vec{\nabla}\phi \tag{2.35}$$

These equations are expressed in the orthogonal basis $(\hat{r}, \hat{\theta}, \hat{z})$. However for field-aligned expressions we are interested in objects which are parallel to the magnetic field. It is therefore important to identify these objects by including the magnetic unit vector $\hat{b}$ in the equations. In order to do this, the equations will now be expressed on the non-orthogonal basis $(\hat{r}, \hat{\theta}, \hat{b})$. These expressions can be found using the following definitions for an arbitrary vector $\boldsymbol{v}$:

$$\boldsymbol{v} = v_1\hat{r} + v_2\hat{\theta} + v_3\hat{z} \qquad = v_1\hat{r} + w_1\hat{\theta} + w_2\hat{b}$$

$$\begin{aligned} w_1 =& \frac{v_2\hat{b}\times\hat{\theta} + v_3\hat{b}\times\hat{z}}{\hat{b}\times\hat{\theta}} \\ =& v_2 - v_3\frac{b_\theta}{b_z} \\ =& v_2 - v_3\zeta \\ w_2 =& \frac{v_3\hat{\theta}\times\hat{z}}{\hat{\theta}\times\hat{b}} \\ =& \frac{v_3}{b_z} \end{aligned}$$

In order to express $a_\parallel$ in the non-orthogonal coordinates, $\boldsymbol{B}^*(x, v_\parallel)$ must first be expressed. In order to express $\boldsymbol{B}^*(x, v_\parallel)$, $\vec{\nabla}\times\hat{b}$ is also needed:

$$\begin{aligned} \vec{\nabla}\times\hat{b} =& -\partial_r b_z\hat{\theta} + \frac{1}{r}\partial_r(rb_\theta)\hat{z} \\ =& \left(\frac{s\zeta\zeta'}{(1+\zeta^2)^{\frac{3}{2}}}\right)\hat{\theta} + \left(\frac{s\zeta}{r\,(1+\zeta^2)^{\frac{1}{2}}} + \frac{s\zeta'}{(1+\zeta^2)^{\frac{3}{2}}}\right)\hat{z} \\ =& \left(\frac{s\zeta\zeta'}{(1+\zeta^2)^{\frac{3}{2}}} - \frac{s\zeta^2}{r\,(1+\zeta^2)^{\frac{1}{2}}} - \frac{s\zeta'\zeta}{(1+\zeta^2)^{\frac{3}{2}}}\right)\hat{\theta} \\ &+ \left(\frac{s\zeta}{r\,(1+\zeta^2)^{\frac{1}{2}}} + \frac{s\zeta'}{(1+\zeta^2)^{\frac{3}{2}}}\right)\frac{1}{b_z}\hat{b} \\ =& -\frac{s\zeta^2}{r\,(1+\zeta^2)^{\frac{1}{2}}}\hat{\theta} + \left(\frac{\zeta}{r} + \frac{\zeta'}{1+\zeta^2}\right)\hat{b} \end{aligned}$$

where $\zeta' = \partial_r\zeta(r)$.

$\boldsymbol{B}^*(x, v_\parallel)$ can now be expressed:

$$\begin{aligned} \boldsymbol{B}^*(x, v_\parallel) =& \boldsymbol{B}(x) + \frac{m}{e}v_\parallel\vec{\nabla}\times\hat{b}(x) \\ =& |B_0|\sqrt{1+\zeta^2}\hat{b}(x) + \frac{m}{e}v_\parallel\left[\frac{s\zeta\zeta'}{(1+\zeta^2)^{\frac{3}{2}}}\hat{\theta}\right. \\ &\left. + \frac{1}{r}\left(\frac{s\zeta}{(1+\zeta^2)^{\frac{1}{2}}} + \frac{rs\zeta'}{(1+\zeta^2)^{\frac{3}{2}}}\right)\hat{z}\right] \end{aligned}$$

$$
\begin{aligned}
=&|B_0|\sqrt{1+\zeta^2}\hat{b}(x)+\frac{m}{e}v_\parallel\left[-\frac{s\zeta^2}{r\left(1+\zeta^2\right)^{\frac{1}{2}}}\hat{\theta}+\left(\frac{\zeta}{r}+\frac{\zeta'}{1+\zeta^2}\right)\hat{b}\right]\\
=&-\frac{mv_\parallel}{e}\frac{s\zeta^2}{r\left(1+\zeta^2\right)^{\frac{1}{2}}}\hat{\theta}+\left[|B_0|\sqrt{1+\zeta^2}+\frac{mv_\parallel}{e}\left(\frac{\zeta}{r}+\frac{\zeta'}{1+\zeta^2}\right)\right]\hat{b}
\end{aligned}
$$

This leaves the following expression for $a_\parallel$:

$$
\begin{aligned}
a_\parallel =&-\frac{e\boldsymbol{B}^*}{mB_\parallel^*}\cdot\vec{\nabla}\phi\\
=&-\frac{e}{mB_\parallel^*}\left[-\frac{mv_\parallel}{e}\frac{s\zeta^2}{r^2\sqrt{1+\zeta^2}}\partial_\theta\phi+\left(|B_0|\sqrt{1+\zeta^2}\right.\right.\\
&\left.\left.+\frac{mv_\parallel}{e}\left(\frac{\zeta}{r}+\frac{\zeta'}{1+\zeta^2}\right)\right)\hat{b}\cdot\vec{\nabla}\phi\right]
\end{aligned}
\tag{2.36}
$$

In order to also express $\boldsymbol{u}$, $\hat{b}\times\vec{\nabla}\phi$ must also be expressed:

$$
\begin{aligned}
\hat{b}\times\vec{\nabla}\phi=&\left(b_\theta\partial_z\phi-b_z\frac{\partial_\theta\phi}{r}\right)\hat{r}+b_z\partial_r\phi\hat{\theta}-b_\theta\partial_r\phi\hat{z}\\
=&\left(b_\theta\partial_z\phi-b_z\frac{\partial_\theta\phi}{r}\right)\hat{r}+\left(b_z\partial_r\phi+\frac{b_\theta^2}{b_z}\partial_r\phi\right)\hat{\theta}-\frac{b_\theta}{b_z}\partial_r\phi\hat{b}\\
=&\left[\frac{b_\theta}{b_z}\left(\hat{b}\cdot\vec{\nabla}\phi-b_\theta\frac{\partial_\theta\phi}{r}\right)-b_z\frac{\partial_\theta\phi}{r}\right]\hat{r}+\left(\frac{b_z^2+b_\theta^2}{b_z}\right)\partial_r\phi\hat{\theta}-\frac{b_\theta}{b_z}\partial_r\phi\hat{b}\\
=&\left[\frac{b_\theta}{b_z}\hat{b}\cdot\vec{\nabla}\phi-\frac{1}{b_z}\frac{\partial_\theta\phi}{r}\right]\hat{r}+\frac{1}{b_z}\partial_r\phi\hat{\theta}-\frac{b_\theta}{b_z}\partial_r\phi\hat{b}
\end{aligned}
$$

This leaves the following expression for $\boldsymbol{u}$:

$$
\begin{aligned}
\boldsymbol{u}=&\frac{v_\parallel\boldsymbol{B}^*}{B_\parallel^*}+\frac{1}{B_\parallel^*}\hat{b}\times\vec{\nabla}\phi\\
=&\frac{1}{B_\parallel^*}\left[\left(\zeta\hat{b}\cdot\phi-\frac{s}{\sqrt{1+\zeta^2}}\frac{\partial_\theta\phi}{r}\right)\hat{r}\right.\\
&+\left(\frac{s}{\sqrt{1+\zeta^2}}\partial_r\phi-\frac{mv_\parallel^2}{e}\frac{s\zeta^2}{r\sqrt{1+\zeta^2}}\right)\hat{\theta}\\
&\left.+\left(|B_0|\sqrt{1+\zeta^2}v_\parallel+\frac{mv_\parallel^2}{e}\left(\frac{\zeta}{r}+\zeta'\frac{\zeta^2}{1+\zeta^2}\right)-\zeta\partial_r\phi\right)\hat{b}\right]
\end{aligned}
\tag{2.37}
$$

In order to complete the definition, $B_\parallel^*$ is also expressed explicitly:

$$
\begin{aligned}
B_\parallel^* =&B_\theta^* b_\theta+B_z^*\\
=&|B_0|\sqrt{1+\zeta^2}+\frac{mv_\parallel}{e}\left(-\frac{\zeta^3}{r\left(1+\zeta^2\right)}+\frac{\zeta}{r}+\frac{\zeta'}{1+\zeta^2}\right)\\
=&|B_0|\sqrt{1+\zeta^2}+\frac{mv_\parallel}{e}\left(\frac{\zeta+\zeta' r}{r\left(1+\zeta^2\right)}\right)
\end{aligned}
\tag{2.38}
$$

For a torus to be effectively approximated by a straight cylinder, the major radius must be very large. As a result we let $\zeta(r)$ and $\zeta'(r)$ tend to 0. Applying this condition to equations 2.36, 2.37, and 2.38 gives the following expressions:

$$\boldsymbol{u} = -\frac{1}{B_0}\frac{\partial_\theta \phi}{r}\hat{r} + \frac{1}{B_0}\partial_r \phi \hat{\theta} + v_\parallel \hat{b} \tag{2.39}$$

$$a_\parallel = -\frac{e}{m}\hat{b}\cdot\vec{\nabla}\phi \tag{2.40}$$

Thus equation 2.3 can be written as follows:

$$\partial_t f + \left(-\frac{1}{B_0}\frac{\partial_\theta \phi}{r}\hat{r} + \frac{1}{B_0}\partial_r\phi\hat{\theta} + v_\parallel \hat{b}\right)\cdot\vec{\nabla} f - \frac{e}{m}\hat{b}\cdot\vec{\nabla}\phi\,\partial_{v_\parallel} f = 0 \tag{2.41}$$

$$\partial_t f - \frac{1}{B_0}\frac{\partial_\theta\phi}{r}\partial_r f + \frac{1}{B_0}\partial_r\phi\frac{\partial_\theta\phi}{r} + v_\parallel\vec{\nabla}_\parallel f - \frac{e}{m}\vec{\nabla}_\parallel\phi\,\partial_{v_\parallel} f = 0 \tag{2.42}$$

To simplify the expression a Poisson bracket notation is introduced, defined as follows:

$$\{\phi, f\} = -\frac{\partial_\theta\phi\partial_r f}{rB_0} + \frac{\partial_r\phi\partial_\theta f}{rB_0} \tag{2.43}$$

This leaves the following final expression:

$$\partial_t f + \{\phi, f\} + v_\parallel\vec{\nabla}_\parallel f - \frac{e}{m}\vec{\nabla}_\parallel\phi\,\partial_{v_\parallel} f = 0 \tag{2.44}$$

2.3 Quasi-Neutrality Equation

Tronko et al. [5] state the following (re-written in SI units):

$$-\sum_{sp}\int dW\frac{1}{B^*_\parallel}\vec{\nabla}_\perp\cdot\left[B^*_\parallel f_0\frac{m}{B^2}\vec{\nabla}_\perp\phi_1\right] = \sum_{sp} q_s\int dW\mathcal{J}_0^{gc} f_1 \tag{2.45}$$

$B^*_\parallel$ is defined in equation 2.38 for cylindrical geometry. As above we take the limit $\zeta(r)\to 0$ which leads $B^*_\parallel$ to converge towards the constant $|B_0|$. Given the dependencies of each element the equation can therefore be rewritten as follows:

$$-\sum_{sp}\vec{\nabla}_\perp\cdot\left[\left(\int dW f_0\right)\frac{m}{B^2}\vec{\nabla}_\perp\phi_1\right] = \sum_{sp} q_s\int dW\mathcal{J}_0^{gc} f_1 \tag{2.46}$$

$$-\sum_{sp}\vec{\nabla}_\perp\cdot\left[n_0\frac{m}{B^2}\vec{\nabla}_\perp\phi_1\right] = \sum_{sp} q_s n_{1s} \tag{2.47}$$

Note that $n_{1s} = n_s - n_0$ and for adiabatic electrons $n_e = n_0\exp\left(-q_e\frac{\phi-\langle\phi\rangle_r}{\kappa T_e}\right)$. The equation can therefore be written as:

$$-\vec{\nabla}_\perp\cdot\left[\frac{\rho_{m0}}{B^2}\vec{\nabla}_\perp\phi\right] = q_i(n_i - n_0) + q_e n_0\left(\exp\left(-q_e\frac{\phi-\langle\phi\rangle_r}{\kappa T_e}\right) - 1\right) \tag{2.48}$$

where ρ_{m0} is the equilibrium mass density.

The linearisation of the exponential term leads to the following equation:

$$-\vec{\nabla}_\perp \cdot \left[\frac{\rho_m}{B^2}\vec{\nabla}_\perp\phi\right] = q_i(n_i - n_0) - q_e^2 n_0 \frac{\phi - \langle\phi\rangle_r}{\kappa T_e} \tag{2.49}$$

Dividing by ε_0 we obtain the following equation:

$$-\vec{\nabla}_\perp \cdot \left[\frac{\rho_{m0}}{\varepsilon_0 B^2}\vec{\nabla}_\perp\phi\right] + \frac{q_e^2 n_0}{\varepsilon_0 \kappa T_e}\left[\phi - \langle\phi\rangle_f\right] = \frac{1}{\varepsilon_0}\rho_{c1} \tag{2.50}$$

$$-\vec{\nabla}_\perp \cdot \left[\frac{\rho_{m0}}{\varepsilon_0 B^2}\vec{\nabla}_\perp\phi\right] + \frac{1}{\lambda_D^2}\left[\phi - \langle\phi\rangle_f\right] = \frac{1}{\varepsilon_0}\rho_{c1} \tag{2.51}$$

where ρ_{c1} is the charge perturbation density due to the ions and $\lambda_D = \sqrt{\frac{\varepsilon_0 \kappa T_e}{q_e^2 n_0}}$ is the electron Debye length.

In the case of non-adiabatic electrons, the electron contribution would take a similar form to the ionic contribution. This would be written as:

$$-\vec{\nabla}_\perp \cdot \left[\frac{\rho_{m0}}{\varepsilon_0 B^2}\vec{\nabla}_\perp\phi\right] + \frac{1}{\lambda_D^2}\left[\phi - \langle\phi\rangle_f\right] = \frac{1}{\varepsilon_0}\rho_{c1} \tag{2.52}$$

as the electron contribution is absorbed by ρ_{c1}. The equation can also be simplified by removing the term $\langle\phi\rangle_f$. The final expression used will therefore be the following:

$$-\vec{\nabla}_\perp \cdot \left[\frac{\rho_{m0}}{\varepsilon_0 B^2}\vec{\nabla}_\perp\phi\right] + \frac{1}{\lambda_D^2}\left[\phi - \chi \cdot \langle\phi\rangle_f\right] = \frac{1}{\varepsilon_0}\rho_{c1} \tag{2.53}$$

where χ is an optional parameter equal to either 0 or 1.

Chapter 3

Advection Equations

3.1 Introduction

Advection equations describe the transport of a conserved quantity. They are therefore necessary to describe the movement of particles in a plasma. The derivation of the advection equations is described in detail in chapter 2.

There are many methods for solving advection equations. In this project a semi-lagrangian method will be used with Strang splitting.

3.2 Semi-Lagrangian Method

The semi-Lagrangian method is a method of solving advection equations of the form:

$$\partial_t f(\boldsymbol{x}, t) + \boldsymbol{c}(\boldsymbol{x}, t) \cdot \vec{\nabla} f(\boldsymbol{x}, t) = 0 \tag{3.1}$$

As described by Garbet et al. [6], it is a combination of the Eulerian and Lagrangian methods of solving advection equations. The Eulerian method consists of discretising the phase space on a fixed grid, and applying a grid based method such as finite differences, finite volumes, and Fourier transforms. This however leads to restrictions on the time step that can be used due to the CFL condition and can lead to dissipation.

The Lagrangian method takes advantage of the fact that advection equations arise from conservation equations to remove the restrictions on the time step. Conserved quantities are quantities whose values remain constant along a trajectory and thus obey the following rule:

$$\frac{d}{dt} f(\boldsymbol{x}(t), t) = 0 \tag{3.2}$$

$$\frac{\partial}{\partial t} f(\boldsymbol{x}(t), t) + \vec{\nabla} f(x(t), t) \cdot \frac{\partial}{\partial t} \boldsymbol{x}(t) = 0 \tag{3.3}$$

Evidently this trajectory can therefore be described by the following equation:

$$\frac{\partial}{\partial t} x(t) = c(x, t) \tag{3.4}$$

If this equation can be solved trivially then the exact solution can be easily determined. For example, in the simplified case of a 1-D constant advection problem, the exact solution can be defined as follows:

$$f(x,t) = f_0(x - ct) \tag{3.5}$$

where $f_0(x) = f(x, 0)$.

The Lagrangian method exploits these properties to create a method which does not necessarily have any restrictions on the size of the time step. Indeed the time step has no restrictions if the trajectory can be defined exactly. Otherwise it must be approximated via another method to a sufficient degree of accuracy. The method consists of following specific particles along their trajectories. These particles are chosen using statistical methods. This means that at the beginning of the simulation they should be approximately evenly spaced, however it is not guaranteed that this spacing is maintained. This can lead to areas which at a given time contain very few of the chosen particles. This means that the approximation in that area is not very precise.

The semi-Lagrangian method tries to combine the Eulerian and Lagrangian methods to benefit from the advantages of each. The method is grid-based but still uses trajectory integration. At each time step the trajectory defined by equation 3.4 is followed back from each grid point to its position at the start of the time step. The value of the function at the starting point is then approximated and the value is saved at the grid point.

There are multiple ways of approximating the value at the grid point. Here splines will be used.

3.3 Strang Splitting

Advection equations can be expressed as:

$$\partial_t f = \sum_i L_i(f) \tag{3.6}$$

where $L_i(f)$ is an operator. If the operators can be expressed as matrices then the exact solution can be defined as follows:

$$f(t) = e^{(\sum_i L_i)t} f_0 \tag{3.7}$$

Of course in the case of an advection equation the operator is a derivative not a matrix. However, the definition of the derivative means that it can be represented as a matrix of infinite dimensions. Therefore the approximation is appropriate.

If the matrices are commutative then the solution can be expressed as:

$$f(t) = \prod_i e^{L_i t} f_0 \tag{3.8}$$

This means that the numerical equation can be split into multiple equations:

$$f(t) = e^{L_n t} f_{n-1}(t) \tag{3.9}$$
$$f_i(t) = e^{L_i t} f_{i-1}(t) \tag{3.10}$$
$$f_1(t) = e^{L_1 t} f_0 \tag{3.11}$$

This is useful as the resulting equations are much simpler and can be solved on contiguous data sets.

However the matrices representing the derivatives are not necessarily commutative which means that the equation $e^{A+B} = e^A e^B$ does not necessarily hold. This means that at each time step there is an error due to this supposition. In the simplest splitting method, Lie splitting, this error is $\mathcal{o}(\tau)$. The Strang splitting method reduces this error to $\mathcal{o}(\tau^2)$. The process is as follows:

1. One half time step of the least expensive operators
2. One time step of the most expensive operator
3. One half time step of the least expensive operators

The proof of the order of the error is given below, where operators A and B are supposed to be relatively inexpensive compared to operator C:

$$\begin{aligned}
&e^{A\frac{\tau}{2}} e^{B\frac{\tau}{2}} e^{C\tau} e^{B\frac{\tau}{2}} e^{A\frac{\tau}{2}} \\
=&\left(\mathbb{1} + A\frac{\tau}{2} + \frac{A^2\tau^2}{8} + \frac{A^3\tau^3}{48} + \mathcal{O}\left(\tau^4\right)\right) \cdot \\
&\left(\mathbb{1} + B\frac{\tau}{2} + \frac{B^2\tau^2}{8} + \frac{B^3\tau^3}{48} + \mathcal{O}\left(\tau^4\right)\right) \cdot \\
&\left(\mathbb{1} + C\tau + \frac{C^2\tau^2}{2} + \frac{C^3\tau^3}{6} + \mathcal{O}\left(\tau^4\right)\right) \cdot \\
&\left(\mathbb{1} + B\frac{\tau}{2} + \frac{B^2\tau^2}{8} + \frac{B^3\tau^3}{48} + \mathcal{O}\left(\tau^4\right)\right) \cdot \\
&\left(\mathbb{1} + A\frac{\tau}{2} + \frac{A^2\tau^2}{8} + \frac{A^3\tau^3}{48} + \mathcal{O}\left(\tau^4\right)\right) \\
=&\mathbb{1} + (A+B+C)\,\tau + (A+B+C)^2 \frac{\tau^2}{2} + \mathcal{O}\left(\tau^3\right) \\
=&e^{(A+B+C)} + \mathcal{O}\left(\tau^3\right)
\end{aligned}$$

3.4 Screw-Pinch model - Numerical Methods

As explained in chapter 2, the particle distribution function $f(t, r, \theta, z, v_\parallel)$ must satisfy equation 2.44:

$$\partial_t f + \{\phi, f\} + v_\parallel \nabla_\parallel f - \nabla_\parallel \phi \; \partial_{v_\parallel} f = 0$$

As described in section 3.3, this equation can be split into multiple advection equations. These equations are:

$$\partial_t f + v_\parallel \nabla_\parallel f = 0 \tag{3.12}$$
$$\partial_t f + \nabla_\parallel \phi \, \partial_{v_\parallel} f = 0 \tag{3.13}$$
$$\partial_t f + \{\phi, f\} = 0 \tag{3.14}$$

where $\{\phi, f\}$ is a Poisson bracket, defined as follows:

$$\{\phi, f\} = -\frac{\partial_\theta \phi}{r B_0} \partial_r f + \frac{\partial_r \phi}{r B_0} \partial_\theta f \tag{3.15}$$

3.4.1 Flux surface advection operator

The flux surface advection operator defined by equation 3.12 is a two-dimensional semi-lagrangian operator. The advection coefficient has no relation to the flux surface, therefore the advection is constant on each surface and the trajectory used by the semi-lagrangian method can be defined exactly.

The value of the particle distribution function outside of the grid points is approximated using a combination of a one-dimensional cubic spline interpolation in the θ direction and a 5-th order Lagrange interpolation (field-aligned) in the z direction. The six grid points on the z-axis, nearest to the final position, are determined. The theta coordinates of the flux-aligned line which passes through the final position, are calculated at these z values. The values of the particle distribution function at these six points are then approximated using the one-dimensional cubic spline interpolation in the θ direction. The six values are finally used to construct a 5-th order Lagrange interpolation (field-aligned) in the z direction and the value at the final position is determined from this interpolation.

3.4.2 V-Parallel surface advection operator

The $v_\parallel$ surface advection operator defined by equation 3.13 is a one-dimensional semi-lagrangian operator which uses a cubic spline interpolation to approximate the value of the particle distribution function outside of the grid points. The parallel gradient of phi depends only on the spatial coordinates and is therefore constant along the $v_\parallel$ surface. As a result the trajectory used by the semi-lagrangian method can be defined exactly.

The parallel gradient of phi is computed using 6th order finite differences (field-aligned) in the z direction. This is calculated as described by Latu et al. [1] using the following equation:

$$\nabla_\parallel \phi(r_i, \theta_j, z_k) \approx \frac{b_z(r_i)}{\Delta z} \sum_{l=-3}^{3} w_l \tilde{\phi}\left(r_i, \text{fieldline}_\theta\left(\theta_j, \frac{z_k}{R_0}, k+l\right), z_{k+l}\right) \tag{3.16}$$

where the fieldline is defined as follows:

$$\text{fieldline}_\theta(\theta^\star, \varphi^\star, j^\star + k) = \theta^\star + \iota(r_0)\left(\varphi_{j^\star+k} - \varphi^\star\right) \tag{3.17}$$

and the weights are:

$$w_0 = 0, \quad w_1 = -w_{-1} = \frac{3}{4}, \quad w_2 = -w_{-2} = -\frac{3}{20}, \quad w_3 = -w_{-3} = \frac{1}{60}$$

3.4.3 Poloidal surface advection operator

The poloidal surface advection equation as defined by equation 3.14 can be rewritten as follows:

$$\partial_t f - \frac{\partial_\theta \phi}{rB_0}\partial_r f + \frac{\partial_r \phi}{rB_0}\partial_\theta f = 0 \tag{3.18}$$

The operator used is a two-dimensional semi-lagrangian operator which uses a 2-D tensor-product cubic spline interpolation in the (r, θ) plane to approximate the value of the particle distribution function outside of the grid points.

The trajectory used by the semi-lagrangian operator is found using the second order Runge method. This can either be explicit or implicit. The explicit form (also known as Heun's method) is defined as follows:

$$f^{n+1} = f^n - \frac{\tau}{2}\left(\{\phi, f^n\} + \{\phi, f^n - \tau\{\phi, f^n\}\right) \tag{3.19}$$

The implicit form is an iterative method:

$$f_{i+1}^{n+1} = f^n - \frac{\tau}{2}\left(\{\phi, f^n\} + \{\phi, f_i^{n+1}\}\right) \tag{3.20}$$

It is considered to have converged when $\|f_{i+1}^{n+1} - f_i^{n+1}\| < \varepsilon$ for a given tolerance ε.

3.4.4 Strang Splitting

As mentioned in section 3.3, expensive operators will only be used once per time-step while relatively inexpensive operators will be used twice.

Operator	Time per step [s]	Time per grid [s]
Flux surface advection	$2.85 \cdot 10^{-3}$	1.23
V-Parallel surface advection	$3.08 \cdot 10^{-4}$	2.78
Poloidal surface advection	$3.41 \cdot 10^{-2}$	12.7

Table 3.1: Time required for one time step of each advection operator averaged over 1000 steps. 20 points are used in each direction on the grid

Table 3.1 shows the time required for one step of each operator. As the poloidal surface advection is approximately ten times more expensive than the other operators, this operator will only be used once per time step.

The algorithm is therefore:

1. Use the operator described in equation 3.12 with a time step of $\frac{\tau}{2}$
2. Use the operator described in equation 3.13 with a time step of $\frac{\tau}{2}$
3. Use the operator described in equation 3.14 with a time step of τ
4. Use the operator described in equation 3.13 with a time step of $\frac{\tau}{2}$
5. Use the operator described in equation 3.12 with a time step of $\frac{\tau}{2}$

3.5 Convergence

3.5.1 V-Parallel surface advection

The $v_\parallel$ surface advection operator defined by equation 3.13 can be written more simply as follows:

$$\partial_t f(v_\parallel) + c\partial_{v_\parallel} f = 0 \tag{3.21}$$

this formulation allows us to test the advection operator alone.

As the trajectory for the v-parallel surface advection can be calculated exactly, the convergence order ought to be equal to the order of the spline approximation. Here the function will be approximated by 3rd degree splines.

The convergence is tested using the following function:

$$f_0(v_\parallel) = \cos(0.1\pi v_\parallel)^4 \tag{3.22}$$

over the domain $\Omega = [-5, 5]$. The boundary conditions are such that $f_0(\mathbb{R}\backslash\Omega) = 0$

This function is chosen as on the boundary $\delta\Omega$, $f_0(\delta\Omega) = f_0'(\delta\Omega) = f_0''(\delta\Omega) = f_0'''(\delta\Omega) = 0$. This is important as it ensures that the spline approximation is $\mathcal{C}^3$ at the boundary.

$\mathbf{N}_x$	$\mathbf{L}^2$ **norm**	**Order**	$\mathbf{L}^\infty$ **norm**	**Order**
64	$1.49 \cdot 10^{-5}$		$8.52 \cdot 10^{-6}$	
128	$1.63 \cdot 10^{-6}$	3.20	$9.36 \cdot 10^{-7}$	3.19
256	$1.94 \cdot 10^{-7}$	3.07	$1.12 \cdot 10^{-7}$	3.06
512	$2.39 \cdot 10^{-8}$	3.02	$1.38 \cdot 10^{-8}$	3.02
1024	$2.97 \cdot 10^{-9}$	3.01	$1.71 \cdot 10^{-9}$	3.01
2048	$3.69 \cdot 10^{-10}$	3.01	$2.12 \cdot 10^{-10}$	3.01
4096	$4.60 \cdot 10^{-11}$	3.01	$2.67 \cdot 10^{-11}$	2.99

Table 3.2: $v_\parallel$ advection with c = 2, $N_x \cdot dt = 0.32$, endTime = 1s, with initial conditions described by equation 3.22

The results can be seen in table 3.2. We see that the convergence is of 3-rd order as expected.

The convergence of the parallel gradient of phi is tested using the following equation:

$$\phi(\theta, z) = \cos(\theta)^2 + \sin(0.1\pi z)^2 \tag{3.23}$$

The result should therefore be the following:

$$\vec{\nabla}_\parallel \phi(\theta, z) = 0.2\pi b_z \sin(0.1\pi z)\cos(0.1\pi z) - 2b_\theta \cos(\theta)\sin(\theta) \tag{3.24}$$

The convergence order is tested in both the θ and z directions. In the θ direction, the convergence should be due to the spline and should therefore be of order p+1 where p is the order of the spline. In the tests third order splines are used. Table 3.4 shows that the convergence is similar to what is expected. In the z direction the convergence should be of the same order as the finite differences method used (6th order). Table 3.3 shows that this is indeed the case. Both the L_2 and L_∞ norms converge to 6th order. The L_∞ norm shows one erroneous result for $N_z = 512$ however this is probably due to the fact that the results approach machine precision at this point.

$\mathbf{N}_z$	$\mathbf{L}^2$ **norm**	**Order**	$\mathbf{L}^\infty$ **norm**	**Order**
8	$1.80 \cdot 10^{-1}$		$2.08 \cdot 10^{-2}$	
16	$3.00 \cdot 10^{-3}$	5.91	$4.67 \cdot 10^{-4}$	5.48
32	$3.68 \cdot 10^{-5}$	6.35	$7.99 \cdot 10^{-6}$	5.87
64	$4.17 \cdot 10^{-7}$	6.46	$1.28 \cdot 10^{-7}$	5.97
128	$4.63 \cdot 10^{-9}$	6.49	$2.01 \cdot 10^{-9}$	5.99
256	$5.12 \cdot 10^{-11}$	6.50	$3.18 \cdot 10^{-11}$	5.98
512	$7.50 \cdot 10^{-13}$	6.09	$9.43 \cdot 10^{-13}$	5.08

Table 3.3: derivative of phi with $N_\theta = 1024$

$\mathbf{N}_\theta$	$\mathbf{L}^2$ **norm**	**Order**	$\mathbf{L}^\infty$ **norm**	**Order**
8	$1.01 \cdot 10^{-5}$		$1.50 \cdot 10^{-4}$	
16	$7.62 \cdot 10^{-7}$	3.74	$7.59 \cdot 10^{-6}$	4.31
32	$6.46 \cdot 10^{-8}$	3.56	$4.49 \cdot 10^{-7}$	4.08
64	$5.65 \cdot 10^{-9}$	3.52	$2.77 \cdot 10^{-8}$	4.02
128	$4.98 \cdot 10^{-10}$	3.50	$1.72 \cdot 10^{-9}$	4.00
256	$4.39 \cdot 10^{-11}$	3.50	$1.08 \cdot 10^{-10}$	4.00
512	$3.89 \cdot 10^{-12}$	3.50	$6.79 \cdot 10^{-12}$	3.99

Table 3.4: derivative of phi with $N_z = 1024$

3.5.2 Poloidal advection

The poloidal advection operator as described in section 3.4.3 should be tested using a self-consistent potential ϕ such that the exact trajectory can be found and defined simply but which is sufficiently complex for a second order scheme to be required. In addition the boundary conditions must be periodic in the θ direction.

The chosen function is the following:

$$\phi(r, \theta) = -5r^2 + \sin(\theta) \tag{3.25}$$

Using equation 3.18, the trajectory can be described by the following equations:

$$\begin{aligned} \frac{\partial}{\partial t} r(t) &= -\frac{\partial_\theta \phi}{r B_0} & &= -\frac{\cos(\theta)}{r B_0} \\ \frac{\partial}{\partial t} \theta(t) &= \frac{\partial_r \phi}{r B_0} & &= \frac{-10r}{r B_0} = -\frac{10}{B_0} \end{aligned}$$

From this, it is trivial to find the explicit expression which describes the coordinate θ:

$$\theta(t) = \theta_0 - \frac{10}{B_0} t \tag{3.26}$$

The expression which describes the coordinate r is slightly more complicated but can also be simply determined:

$$\frac{\partial}{\partial t} r(t) = -\frac{\cos\left(\theta_0 - \frac{10}{B_0} t\right)}{r B_0}$$

$$\int \frac{\partial}{\partial t}\left(\frac{1}{2} r(t)^2\right) dt = -\frac{1}{B_0} \int \cos\left(\theta_0 - \frac{10}{B_0} t\right) dt$$

$$r(t)^2 = \frac{2}{B_0} \sin\left(\theta_0 - \frac{10}{B_0} t\right) \frac{B_0}{10} + C$$

$$r(t) = \sqrt{\frac{1}{5} \sin\left(\theta_0 - \frac{10}{B_0} t\right) + C}$$

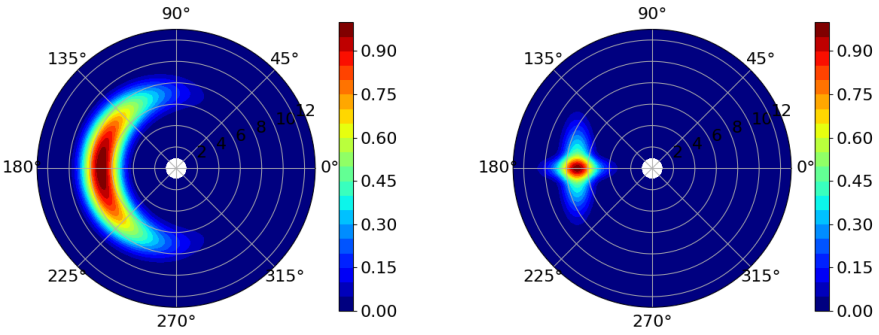
The initial conditions can then be used to define the unknown quantities

$$r(0) = r_0 = \sqrt{\frac{1}{5} \sin(\theta_0) + C}$$

$$C = r_0^2 - \frac{1}{5} \sin(\theta_0)$$

$$r(t) = \sqrt{\frac{1}{5}\left(\sin\left(\theta_0 - \frac{10}{B_0} t\right) - \sin(\theta_0)\right) + r_0^2} \tag{3.27}$$

The trajectories can therefore be used to find the exact solution which is used to determine the error for the convergence studies.



(a) Equation 3.28 (b) Equation 3.29

Figure 3.1: Initial conditions f_0 for the poloidal advection convergence tests as described by equations 3.28 and 3.29

Two different initial conditions will be used. The first will be an ellipsis in the coordinates (r,θ). The second will not be dependent on the coordinates and will have a more irregular shape as shown in figure 3.1. The conditions are

defined as follows:

$$f_0(r,\theta) = B(r_1(r,\theta), a_1) \tag{3.28}$$
$$f_0(r,\theta) = B(r_2(r\cos(\theta), r\sin(\theta), a_2) + B(r_3(r\cos(\theta), r\sin(\theta)), a_2) \tag{3.29}$$
$$B(r,a) = \begin{cases} \cos\left(\frac{\pi r}{2a}\right)^4 & \text{if } r \le a \\ 0 & \text{otherwise} \end{cases}$$
$$r_1(x,y) = \sqrt{(x-7)^2 + 2*(y-\pi)^2}$$
$$r_2(x,y) = \sqrt{(x+7)^2 + 8*y^2}$$
$$r_3(x,y) = \sqrt{4*(x+7)^2 + 0.5*y^2}$$
$$a_1 = 4$$
$$a_2 = 6$$

over the domain $\Omega = [0,20] \times [0,2\pi]$. The boundary conditions are periodic in θ and such that $f_0(\mathbb{R}\backslash\Omega) = 0$.

$\mathbf{N}_x$	$\mathbf{N}_y$	$\mathbf{L}^2$ **norm**	**Order**	$\mathbf{L}^\infty$ **norm**	**Order**
32	32	$1.79 \cdot 10^{-3}$		$3.05 \cdot 10^{-4}$	
64	64	$2.19 \cdot 10^{-4}$	3.03	$3.71 \cdot 10^{-5}$	3.04
128	128	$2.72 \cdot 10^{-5}$	3.00	$4.61 \cdot 10^{-6}$	3.01
256	256	$3.41 \cdot 10^{-6}$	3.00	$5.75 \cdot 10^{-7}$	3.00
512	512	$4.26 \cdot 10^{-7}$	3.00	$7.19 \cdot 10^{-8}$	3.00

Table 3.5: Spatial convergence for poloidal advection, $N_x \cdot dt = 0.032$ s, t_{end}=0.2s, f_0 is given by equation 3.28

$\mathbf{N}_x$	$\mathbf{N}_y$	$\mathbf{L}^2$ **norm**	**Order**	$\mathbf{L}^\infty$ **norm**	**Order**
32	32	$2.01 \cdot 10^{-1}$		$6.00 \cdot 10^{-2}$	
64	64	$1.32 \cdot 10^{-1}$	0.61	$6.34 \cdot 10^{-2}$	-0.08
128	128	$1.99 \cdot 10^{-2}$	2.73	$9.89 \cdot 10^{-3}$	2.68
256	256	$2.22 \cdot 10^{-3}$	3.17	$1.03 \cdot 10^{-3}$	3.27
512	512	$2.60 \cdot 10^{-4}$	3.09	$1.23 \cdot 10^{-4}$	3.06
1024	1024	$3.20 \cdot 10^{-5}$	3.03	$1.51 \cdot 10^{-5}$	3.02

Table 3.6: Spatial convergence for poloidal advection, $N_x \cdot dt = 0.032$ s, t_{end}=0.05s, f_0 is given by equation 3.29

To determine the order of convergence the spatial and temporal convergence orders must be treated separately. In order to determine the spatial convergence the calculated trajectory is used instead of the second order scheme. This means that all errors are due to the interpolation. As a result the convergence order ought to be equal to the order of the spline approximation. Here the function will be approximated by 3rd degree splines.

Tables 3.5 and 3.6 confirm this convergence order. Although it should be noted that shapes which are not dependent on the coordinate system require more points in order to achieve a given accuracy and convergence is only seen once the error is sufficiently small.

In order to determine the temporal convergence order a large number of spatial points is used to ensure that the error is dominated by the errors due to the trajectory calculation. The convergence should therefore be second order. This is confirmed by table 3.7

$\mathbf{N}_t$	dt	$\mathbf{L}^2$ norm	Order	$\mathbf{L}^\infty$ norm	Order
10	0.1	$3.81 \cdot 10^{-3}$		$9.19 \cdot 10^{-4}$	
20	0.05	$9.43 \cdot 10^{-4}$	2.01	$2.28 \cdot 10^{-4}$	2.01
40	0.025	$2.35 \cdot 10^{-4}$	2.00	$5.71 \cdot 10^{-5}$	2.00
80	0.0125	$5.88 \cdot 10^{-5}$	2.00	$1.43 \cdot 10^{-5}$	2.00
160	0.00625	$1.47 \cdot 10^{-5}$	2.00	$3.57 \cdot 10^{-6}$	2.00

Table 3.7: Temporal convergence for poloidal advection, with $\mathrm{N}_x = \mathrm{N}_y = 100$, endTime = 1s, f_0 is given by equation 3.28

3.5.3 Flux-surface advection

The flux-surface advection operator is defined by equation 3.12:

$$\partial_t f(v_\parallel) + c\vec{\nabla}_\parallel f = 0 \tag{3.30}$$

As the trajectory for the flux-surface advection can be calculated exactly, the convergence order ought to depend on a combination of the error due to the spline approximation and the error due to the Lagrange interpolation. A combination of different ordered convergences can be difficult to identify. Therefore for the purpose of the tests a 3-rd degree Lagrange interpolation will be used in place of the 5-th order interpolation. The function will still be approximated by 3rd degree splines. The convergence therefore ought to be of 3-rd order.

The convergence is tested using the following function:

$$\begin{aligned} f_0(\theta, z) &= B(r_1(r,\theta), a_1) \\ B(r,a) &= \begin{cases} \cos\left(\frac{\pi r}{2a}\right)^4 & \text{if } r \leq a \\ 0 & \text{otherwise} \end{cases} \\ r_1(x,y) &= \sqrt{(x-10)^2 + 2*(y-\pi)^2} \\ a_1 &= 4 \end{aligned} \tag{3.31}$$

over the domain $\Omega = [0, 2\pi] \times [0, 20]$. The boundary conditions are periodic in θ and such that $f_0(\mathbb{R}\backslash\Omega) = 0$.

This function is chosen as it is $\mathcal{C}^3$ everywhere.

The results can be seen in table 3.2. We see that the convergence is of 3-rd order as expected.

$\mathbf{N}_\theta$	$\mathbf{N}_z$	**dt**	$\mathbf{L}^2$ **norm**	**Order**	$\mathbf{L}^\infty$ **norm**	**Order**
32	32	0.1	$8.26 \cdot 10^{-2}$		$4.96 \cdot 10^{-2}$	
64	64	0.05	$1.29 \cdot 10^{-2}$	2.67	$7.97 \cdot 10^{-3}$	2.64
128	128	0.025	$1.70 \cdot 10^{-3}$	2.93	$1.08 \cdot 10^{-3}$	2.88
256	256	0.0125	$2.14 \cdot 10^{-4}$	2.99	$1.36 \cdot 10^{-4}$	2.99
512	512	0.00625	$2.68 \cdot 10^{-5}$	3.00	$1.71 \cdot 10^{-5}$	3.00

Table 3.8: Spatial convergence for the flux advection operator with $t_{\text{end}} = 1\text{s}$, f_0 as given by equation 3.31, 3rd degree Lagrange interpolation

Chapter 4

Poisson Equation

4.1 Introduction

Plasma is quasi-neutral. This means that although locally there may be charged regions, it is neutral overall. This condition is expressed by a Poisson equation. It is therefore important to have a robust and accurate method for solving Poisson equations in order to determine the electric potential.

The independence of the equations in the z-direction will be used to solve multiple equations in parallel. In addition, the method will use Fourier transforms to exploit the periodicity of the solution in the θ-direction in order to obtain further parallelisation.

4.2 Quasi-Neutrality Equation

As discussed in chapter 2, the quasi-neutrality condition can be expressed using the following equation:

$$-\vec{\nabla}_\perp \cdot \left[\frac{\rho_{m0}}{\varepsilon_0 B^2}\vec{\nabla}_\perp \phi\right] + \frac{1}{\lambda_D^2}\left[\phi - \chi \cdot \langle\phi\rangle_f\right] = \frac{1}{\varepsilon_0}\rho_{c1} \tag{4.1}$$

where $\vec{\nabla}_\perp = \left(\vec{\nabla} - \hat{b}\left(\hat{b}\cdot\vec{\nabla}\right)\right)$, $\hat{b}$ is the magnetic unit vector as defined in equation 2.33, ρ_{m0} is the equilibrium mass density, ε_0 is the permittivity of free space, B is the intensity of the equilibrium background magnetic field, ϕ is the electrostatic potential, λ_D is the electron Debye length, $\langle\phi\rangle_f$ is the flux-surface average of ϕ, and ρ_{c1} is the charge perturbation density due to the ions.

As discussed previously, for a torus to be effectively approximated by a straight cylinder, the major radius must be very large. As a result we let $\zeta(r)$ tend to 0. In this limit $\hat{b}$ tends towards $\hat{z}$. This means that the perpendicular gradient can be defined as follows:

$$\begin{aligned}\vec{\nabla}_\perp\phi = \vec{\nabla}\phi - \hat{b}\left(\hat{b}\cdot\vec{\nabla}\phi\right) \rightarrow &\vec{\nabla}\phi - \hat{z}\left(\hat{z}\cdot\vec{\nabla}\phi\right)\\ =&\partial_r\phi\hat{r} + \frac{1}{r}\partial_\theta\phi\hat{\theta} + \partial_z\phi\hat{z} - \partial_z\phi\hat{z}\\ =&\partial_r\phi\hat{r} + \frac{1}{r}\partial_\theta\phi\hat{\theta}\end{aligned} \tag{4.2}$$

Defining $g(r) = \frac{\rho_{m0}(r)}{\varepsilon_0 B^2}$, the first term in equation 4.1 can now be expressed:

$$\begin{aligned}\vec{\nabla}_\perp \left(g\vec{\nabla}_\perp\phi\right) &= \left(\vec{\nabla} - \hat{b}\left(\hat{b}\cdot\vec{\nabla}\right)\right)\cdot\left[g\vec{\nabla}\phi\right] \\ &\rightarrow \left(\vec{\nabla} - \hat{z}\left(\hat{z}\cdot\vec{\nabla}\right)\right)\cdot\left[g\partial_r\phi\hat{r} + \frac{g}{r}\partial_\theta\phi\hat{\theta}\right] \\ &= \frac{1}{r}\partial_r(rg\partial_r\phi) + \frac{1}{r}\partial_\theta(\frac{g}{r}\partial_\theta\phi) \\ &= \frac{g}{r}\partial_r\phi + \partial_r(g\partial_r\phi) + \frac{g}{r^2}\partial_\theta^2\phi \\ &= \frac{g}{r}\partial_r\phi + \partial_r g\partial_r\phi + g\partial_r^2\phi + \frac{g}{r^2}\partial_\theta^2\phi) \end{aligned} \tag{4.3}$$

Substituting equation 4.3 and the definition of the charge perturbation density due to the ions into equation 4.1 and dividing by g then leaves us with the following equation:

$$-\left[\partial_r^2 + \left(\frac{1}{r} + \frac{\partial_r g}{g}\right)\partial_r + \frac{1}{r^2}\partial_\theta^2\right]\phi + \frac{1}{g\lambda_D^2}\left[\phi - \chi\cdot\langle\phi\rangle_\theta\right] = \frac{1}{g\varepsilon_0}\rho_{c1} \tag{4.4}$$

where $\rho_{c1} = q_i\int(f - f_{eq})dv$ is the charge perturbation density due to the kinetic species.

This provides a simple differential equation which will be solved using the methods detailed in this chapter.

4.3 Fourier Transform

Fourier transforms decompose functions into the frequencies of which they are composed. This makes them practical for analysing periodic functions. The Fourier transform of a function is defined as followed:

$$\hat{f}(k) = \mathcal{F}\left[f(x)\right] = \int_{-\infty}^{\infty} f(x)e^{-ikx}dx \tag{4.5}$$

while the inverse Fourier transform is defined as follows:

$$f(x) = \mathcal{F}^{-1}\left[\hat{f}(k)\right] = \int_{-\infty}^{\infty} \hat{f}(k)e^{ikx}dk \tag{4.6}$$

In addition Fourier transforms have several useful properties. Notably the Fourier transform of a derivative is defined as follows:

$$\widehat{\frac{df}{dx}}(k) = ik\hat{f}(k)$$

The proof of this is shown below:

$$\begin{aligned}\widehat{\frac{df}{dx}}(k) &= \mathcal{F}^{-1}\left[\frac{d}{dx}\int_{-\infty}^{\infty}\hat{f}(k)e^{ikx}dx\right] \\ &= \mathcal{F}^{-1}\left[\int_{-\infty}^{\infty}\hat{f}(k)\frac{d}{dx}e^{ikx}dx\right] \\ &= \mathcal{F}^{-1}\left[\int_{-\infty}^{\infty}\hat{f}(k)ike^{ikx}dx\right] \\ &= ik\hat{f}(k)\end{aligned}$$

This along with the linearity of the Fourier transform allows equation 4.4 to be rewritten as follows:

$$\begin{aligned}&\mathcal{F}_\theta\left[-\left[\partial_r^2+\left(\frac{1}{r}+\frac{\partial_r g}{g}\right)\partial_r\phi+\frac{1}{r^2}\partial_\theta^2\right]\phi+\frac{1}{g\lambda_D^2}\left[\phi-\chi\cdot\langle\phi\rangle_\theta\right]\right] \\ =&-\partial_r^2\mathcal{F}_\theta\left[\phi\right]-\left(\frac{1}{r}+\frac{\partial_r g}{g}\right)\partial_r\mathcal{F}_\theta\left[\phi\right]-\frac{1}{r^2}(ik)^2\mathcal{F}_\theta\left[\phi\right]+\frac{1}{g\lambda_D^2}\left[\mathcal{F}_\theta\left[\phi\right]-\chi\delta_{k0}\mathcal{F}_\theta\left[\phi\right]\right] \\ =&-\partial_r^2\mathcal{F}_\theta\left[\phi\right]-\left(\frac{1}{r}+\frac{\partial_r g}{g}\right)\partial_r\mathcal{F}_\theta\left[\phi\right]+\frac{k^2}{r^2}\mathcal{F}_\theta\left[\phi\right]+\frac{1}{g\lambda_D^2}\left[\mathcal{F}_\theta\left[\phi\right]-\chi\delta_{k0}\mathcal{F}_\theta\left[\phi\right]\right] \\ =&\frac{1}{g\varepsilon_0}\mathcal{F}_\theta\left[\rho_{c1}\right]\end{aligned}$$

This leaves us with the following simplified equation which is a one-dimensional differential equation which can be solved using the finite elements method:

$$\left[-\partial_r^2-\left(\frac{1}{r}+\frac{\partial_r g}{g}\right)\partial_r+\frac{k^2}{r^2}+\frac{1}{g\lambda_D^2}(1-\chi\delta_{k0})\right]\hat{\phi}=\frac{\hat{\rho}}{g\varepsilon_0} \tag{4.7}$$

4.4 Finite Elements

The finite elements method is a numerical method for solving various equations including Poisson equations. It consists of using the representation of functions on a given basis to formulate a matrix equation which can be solved to find the solution to the original equation.

All functions can be represented exactly on an infinite basis however this approach clearly doesn't work in a numerical setting. Instead the solution will be restricted to a subset of functions, with the finite elements method allowing the best approximation in this subset to be found. In this project the basis used will be B-Splines (see section 4.5). For a given mode m and a given value of the z coordinate, the solution can therefore be expressed as follows:

$$\hat{\phi}(r,m,z)=\sum_{i=0}^{N}c_i\varphi_{i,m,z}(r) \tag{4.8}$$

where $\varphi_{i,m,z}(r)$ is the i-th B-Spline basis for the values of the m and z coordinates, and c_i is the associated coefficient.

The weak form of an equation consists of multiplying it by a test function and integrating over the relevant domain. Any function satisfying the original

equation should also satisfy the weak form of the equation for all test functions. If the test functions are chosen such that they form the basis of a function space then the equation stands for any test function in the function space. For this reason the test functions are often chosen to be the basis of the function space on which the solution is approximated.

The weak form of equation 4.7 is shown below:

$$\begin{aligned} -\int_\Omega \partial_r^2\hat{\phi}(r,m)\,\psi(r)r\,dr \\ -\int_\Omega \left(\frac{1}{r}+\frac{\partial_r g(r)}{g(r)}\right)\partial_r\hat{\phi}(r,m)\,\psi(r)r\,dr \\ +\int_\Omega \left(\frac{k^2}{r^2}+\frac{1}{g(r)\lambda_D(r)^2}(1-\chi\delta_{k0})\right)\hat{\phi}(r,m)\,\psi(r)r\,dr = \int_\Omega \frac{\hat{\rho}(r,m)}{g\varepsilon_0}\,\psi(r)r\,dr \end{aligned} \tag{4.9}$$

where Ω is the relevant domain, namely $[r_{min}, r_{max}]$, $\psi(r)$ is the test function, and r appears in the integral due to the Jacobian of the cylindrical coordinates.

The first term of this equation can be re-expressed as follows:

$$\begin{aligned} -\int_\Omega \partial_r^2\hat{\phi}(r,m)\,\psi(r)r\,dr = -\partial_r\hat{\phi}(r,m)\,\psi(r)r\Big|_{\partial\Omega} + \int_\Omega \partial_r\hat{\phi}(r,m)\,\partial_r\psi(r)r\,dr \\ +\int_\Omega \partial_r\hat{\phi}(r,m)\,\psi(r)dr \end{aligned} \tag{4.10}$$

This avoids the use of high order derivatives which would decrease the accuracy of the solution. It also introduces a boundary term which allows the boundary conditions to be imposed.

In our simulations two types of boundary conditions will be used. These are homogeneous Dirichlet boundary conditions (where the boundary value is fixed at zero) and homogeneous Neumann boundary conditions (where the derivative at the boundary is fixed at zero). In the case of Neumann boundary conditions it is clear that the boundary term is equal to zero. In the case of Dirichlet boundary conditions the boundary term is also zero as long as the function space is restricted to functions which are also equal to zero on the boundary.

Inserting equation 4.10 and the basis representation described by equation 4.8 into equation 4.9 gives the following equation:

$$\begin{aligned} \sum_j \Bigg[\int_\Omega \partial_r\varphi_j(r)\,\partial_r\psi_i(r)r\,dr \\ +\int_\Omega \partial_r\varphi_j(r)\,\psi_i(r)dr \\ -\int_\Omega \left(\frac{1}{r}+\frac{\partial_r g(r)}{g(r)}\right)\partial_r\varphi_j(r)\,\psi_i(r)r\,dr \\ +\int_\Omega \left(\frac{k^2}{r^2}+\frac{1}{g(r)\lambda_D(r)^2}(1-\chi\delta_{k0})\right)\varphi_j(r)\,\psi_i(r)r\,dr\Bigg]c_j = \int_\Omega \frac{\hat{\rho}(r,m)}{g(r)\varepsilon_0}\,\psi_i(r)r\,dr \end{aligned} \tag{4.11}$$

The solution to the problem is therefore the coefficients c_j. Everything else can be expressed using the definitions of the B-Splines and a quadrature scheme. The quadrature scheme used in this project is Gauss-Legendre quadrature which

is exact for polynomials of degree 2n-1 when n is the number of different points required to calculate the quadrature. This equation can also be expressed in matrix notation, in this case the integral is contained within a matrix $\underline{A} \in \mathbb{R}^{N\text{x}N}$, known as the stiffness matrix, and defined as follows:

$$\begin{aligned}\underline{A}_{ij} = &\int_\Omega \partial_r\varphi_j(r)\,\partial_r\psi_i(r) r\,dr + \int_\Omega \partial_r\varphi_j(r)\,\psi_i(r) dr \\ &- \int_\Omega \left(\frac{1}{r} + \frac{\partial_r g(r)}{g(r)}\right) \partial_r\varphi_j(r)\,\psi_i(r) r\,dr \\ &+ \int_\Omega \left(\frac{k^2}{r^2} + \frac{1}{g(r)\lambda_D(r)^2}(1-\chi\delta_{k0})\right) \varphi_j(r)\,\psi_i(r) r\,dr \end{aligned} \tag{4.12}$$

The right-hand side is a vector $\boldsymbol{b} \in \mathbb{R}^N$:

$$\boldsymbol{b}_i = \int_\Omega \frac{\hat{\rho}(r,m)}{g(r)\varepsilon_0}\,\psi_i(r) r\,dr$$

As mentioned previously $\hat{\rho}$ is the Fourier transform of ρ_{c1} which is defined as follows:

$$\rho_{c1} = q_i \int (f - f_{eq}) dv \tag{4.13}$$

The values will therefore be calculated at each point for which the value of the distribution function f is stored. These points are the Greville points of a spline function. The result is therefore an approximation of $\hat{\rho}$ in the function space of the splines. As a result it can also be expressed using b-splines:

$$\hat{\rho}(r,m,z) = \sum_{i=0}^{N} p_i \varphi_{i,m,z}(r) \tag{4.14}$$

The vector $\boldsymbol{b}$ can therefore be rewritten as:

$$\boldsymbol{b}_i = \sum_j p_j \int_\Omega \frac{1}{g(r)\varepsilon_0} \psi_j(r)\,\psi_i(r) r\,dr$$

The vector can therefore also be expressed in matrix notation. In this case the integral is contained within a matrix $\underline{M} \in \mathbb{R}^{N\text{x}N}$, known as the mass matrix, and defined as follows:

$$M_{ij} = \int_\Omega \frac{1}{g(r)\varepsilon_0} \psi_j(r)\,\psi_i(r) r\,dr \tag{4.15}$$

The coefficients c_j are therefore the elements of the vector $\boldsymbol{c} \in \mathbb{R}^N$, solution of the matrix equation $\underline{A}\boldsymbol{c} = \underline{M}\boldsymbol{p}$

4.5 B-Spline

B-splines are a basis of the function space containing splines. A spline $s(x)$ is a piece-wise polynomial function. It is defined by a degree d, and knots $x_0, \ldots, x_m$ such that $a_i \leq a_{i+1}$ and $s(x)$ is (d-r) times differentiable at any knot which appears r times [7]. In this project, the knots for n-th degree splines will

always be chosen such that the spline is $\mathcal{C}^{d-1}$. This means that repeated knots can only appear on the boundary of the considered domain.

Every function space can have multiple different bases. B-splines are a specific basis defined as follows:

$$N_i^0(x) = \begin{cases} 1 & \text{if } x \in [x_i, x_{i+1}[\\ 0 & \text{otherwise} \end{cases} \tag{4.16}$$

$$N_i^n = \alpha_i^{n-1} N_i^{n-1}(x) + \left(1 - \alpha_{i+1}^{n-1}\right) N_{i+1}^{n-1}(x) \tag{4.17}$$

$$\alpha_i^{n-1} = \begin{cases} \frac{u - x_i}{x_{i+n} - x_i} & \text{if } x_{i+n} \neq x_i \\ 0 & \text{otherwise} \end{cases} \tag{4.18}$$

The b-spline basis for 3-rd degree splines can be seen in figure 4.1. Note that the central splines cover d+1=4 elements $[a_i, a_{i+1}[$. Therefore d+2 knots are required for each basis function. The additional knots required for the functions nearer to the edge of the domain are repeated knots on the boundary; thus $a_0 = a_1 = \cdots = a_d$, and $a_m = a_{m-1} = \cdots = a_{m-d}$.

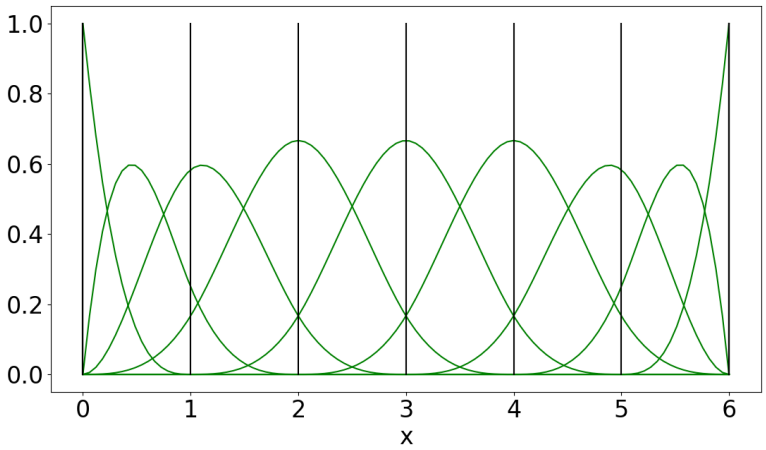


Figure 4.1: 3rd degree B-Spline with additional knots found on the boundary

The d-th degree splines, which make up the function space to which the solution of the Finite Elements problem will belong, can therefore be expressed as follows:

$$s(x) = \sum_i c_i N_i^d(x) \tag{4.19}$$

The shape of the B-Splines therefore means that the mass and stiffness matrices will be band matrices.

4.6 Implementation Details

The aforementioned schemes will therefore be used to implement equation 4.4:

$$-\left[\partial_r^2 + \left(\frac{1}{r} + \frac{\partial_r g}{g}\right)\partial_r + \frac{1}{r^2}\partial_\theta^2\right]\phi + \frac{1}{g\lambda_D^2}\left[\phi - \chi \cdot \langle\phi\rangle_\theta\right] = \frac{1}{g\varepsilon_0}\rho_{c1} \tag{4.20}$$

Note that $g\lambda_D^2 = \frac{m\kappa T_e(r)}{B^2 q_e^2}$, $\frac{\partial_r g}{g} = \frac{\partial_r n_0}{n_0}$ and $g\varepsilon_0 = \frac{m n_0(r)}{B^2}$. For implementation purposes, it is supposed that m=q=1. Thus the following parameters must be provided to complete the problem:

- $n_0(r)$
- $\partial_r n_0(r)$ (in practice $\frac{\partial_r n_0(r)}{n_0(r)}$ may be provided if it can be expressed more simply)
- B
- $\kappa T_e(r)$
- χ

In addition an additional parameter will be requested in order to signify whether electrons are considered to be a kinetic or adiabatic species. In the kinetic case it will be assumed that $\frac{1}{\lambda_D^2} = 0$.

The above equation is somewhat complicated and it is therefore preferable to implement the following equation:

$$a \cdot \partial_r^2 \phi + b \cdot \partial_r \phi + c \cdot \phi + d \cdot \partial_\theta^2 \phi = e \cdot \rho \tag{4.21}$$

where a is a constant and b,c,d and e are any functions dependant on r, in order to be able to test each element of the equation individually. This makes it easier to locate any errors in the scheme.

The following matrices will then be built:

$$A_{ij} = \int_\Omega a \left(\partial_r \phi_j \partial_r \psi_i r + \partial_r \phi_j \psi_i\right) dr \tag{4.22}$$

$$B_{ij} = \int_\Omega b(r) \partial_r \phi_j \psi_i r \, dr \tag{4.23}$$

$$C_{ij} = \int_\Omega c(r) \phi_j \psi_i r \, dr \tag{4.24}$$

$$D_{ij} = \int_\Omega d(r) \phi_j \psi_i r \, dr \tag{4.25}$$

$$E_{ij} = \int_\Omega e(r) \phi_j \psi_i r \, dr \tag{4.26}$$

The solution will therefore be the solution $\boldsymbol{\phi}$ to the following matrix equation:

$$\left(-\underline{A} + \underline{B} + \underline{C} - k^2 \underline{D}\right) \boldsymbol{\phi} = \underline{E} \boldsymbol{\rho} \tag{4.27}$$

As mentioned in section 4.4 Dirichlet and Neumann homogeneous boundary conditions will be used. Dirichlet boundary conditions will be imposed by restricting the function space of both the test functions and the solution. Specifically this is done by reducing the basis so that it no longer includes the function which is not zero at the boundary. This means that the solution is no longer expressed as in equation 4.8, but is instead expressed as follows:

$$\hat{\phi}(r, m, z) = \sum_{i=1}^{N-1} c_i \varphi_{i,m,z}(r) \tag{4.28}$$

Care should be taken with this formulation as the right hand side is still expressed as in equation 4.14. This means that the mass matrix is a rectangular matrix.

The boundary conditions used at $r = r_{\max}$ are always homogeneous Dirichlet conditions. At $r = r_{\min}$ the boundary conditions are homogeneous Dirichlet conditions for all modes except the 0-th mode where homogeneous Neumann conditions are used.

4.7 Convergence

The convergence order of the scheme described above will be determined by testing the following equation:

$$\left[-\partial_r^2 - \left[\frac{1}{r} - \kappa_{n_0}\left(1 - \tanh\left(\frac{r - r_p}{\delta_{r_{n_0}}}\right)^2\right)\right]\partial_r + \frac{1}{T_e(r)} - \frac{1}{r^2}\partial_\theta^2\right]\phi = \rho \quad (4.29)$$

The function ρ will be determined by choosing a solution and substituting this into equation 4.29. The solution chosen will be the following:

$$\phi(r,\theta) = \cos\left(\frac{3\pi(r - r_{\min})}{2(r_{\max} - r_{\min})}\right)^4 \sin(\theta)^3 \quad (4.30)$$

The boundary conditions will be Neumann at $r = r_{\min}$ and Dirichlet at $r = r_{\max}$. This solution can be seen in figure 4.2.

Tests have shown that the Fourier transform is exact to machine precision for this function even for small values of N_θ. The results therefore show only the convergence as a function of the number of finite elements in the r direction. We expect that a convergence order of d+1 when splines of degree d are used.

The results are shown in table 4.1, and figure 4.4. It can be seen that the 1st degree spline approximation converges with the expected order, while the

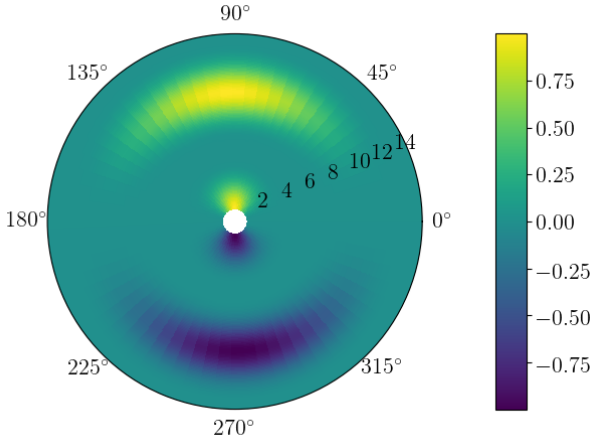


Figure 4.2: The form of equation 4.30, the chosen solution of equation 4.29 used to test convergence. $N_r = 128$, $N_\theta = 64$

Spline degree	**N_{elem}**	**L^2 norm**	**Order**	**L^∞ norm**	**Order**
1	16	$1.27 \cdot 10^{-1}$		$4.59 \cdot 10^{-2}$	
1	32	$3.28 \cdot 10^{-2}$	1.95	$1.43 \cdot 10^{-2}$	1.69
1	64	$8.28 \cdot 10^{-3}$	1.99	$3.91 \cdot 10^{-3}$	1.87
1	128	$2.08 \cdot 10^{-3}$	2.00	$1.01 \cdot 10^{-3}$	1.95
1	256	$5.19 \cdot 10^{-4}$	2.00	$2.56 \cdot 10^{-4}$	1.98
1	512	$1.30 \cdot 10^{-4}$	2.00	$6.44 \cdot 10^{-5}$	1.99
1	1024	$3.25 \cdot 10^{-5}$	2.00	$1.61 \cdot 10^{-5}$	2.00
1	2048	$8.12 \cdot 10^{-6}$	2.00	$4.03 \cdot 10^{-6}$	2.00
2	16	$9.98 \cdot 10^{-3}$		$7.40 \cdot 10^{-3}$	
2	32	$9.56 \cdot 10^{-4}$	3.38	$8.11 \cdot 10^{-4}$	3.19
2	64	$9.30 \cdot 10^{-5}$	3.36	$6.92 \cdot 10^{-5}$	3.55
2	128	$9.41 \cdot 10^{-6}$	3.30	$5.75 \cdot 10^{-6}$	3.59
2	256	$1.05 \cdot 10^{-6}$	3.16	$5.45 \cdot 10^{-7}$	3.40
2	512	$1.26 \cdot 10^{-7}$	3.06	$6.02 \cdot 10^{-8}$	3.18
2	1024	$1.56 \cdot 10^{-8}$	3.02	$7.09 \cdot 10^{-9}$	3.09
2	2048	$1.94 \cdot 10^{-9}$	3.00	$8.60 \cdot 10^{-10}$	3.04
3	16	$2.38 \cdot 10^{-3}$		$8.76 \cdot 10^{-4}$	
3	32	$2.00 \cdot 10^{-4}$	3.57	$1.51 \cdot 10^{-4}$	2.54
3	64	$2.50 \cdot 10^{-5}$	3.00	$2.21 \cdot 10^{-5}$	2.77
3	128	$2.30 \cdot 10^{-6}$	3.45	$2.20 \cdot 10^{-6}$	3.33
3	256	$1.74 \cdot 10^{-7}$	3.73	$1.75 \cdot 10^{-7}$	3.65
3	512	$1.19 \cdot 10^{-8}$	3.87	$1.24 \cdot 10^{-8}$	3.82
3	1024	$7.78 \cdot 10^{-10}$	3.93	$8.23 \cdot 10^{-10}$	3.91
3	2048	$4.99 \cdot 10^{-11}$	3.96	$5.32 \cdot 10^{-11}$	3.95
4	16	$4.41 \cdot 10^{-4}$		$1.76 \cdot 10^{-4}$	
4	32	$1.88 \cdot 10^{-5}$	4.55	$1.61 \cdot 10^{-5}$	3.45
4	64	$1.04 \cdot 10^{-6}$	4.18	$9.28 \cdot 10^{-7}$	4.12
4	128	$3.45 \cdot 10^{-8}$	4.91	$3.19 \cdot 10^{-8}$	4.86
4	256	$8.17 \cdot 10^{-10}$	5.40	$7.63 \cdot 10^{-10}$	5.38
4	512	$1.62 \cdot 10^{-11}$	5.66	$1.53 \cdot 10^{-11}$	5.64
4	1024	$3.00 \cdot 10^{-13}$	5.75	$2.79 \cdot 10^{-13}$	5.78

Table 4.1: Convergence of the Finite Elements method for equation 4.29 with $N_\theta = 8$

2nd and 3rd degree spline approximations converge to the correct order. The 4th degree spline approximation has approximately the correct order however it is hard to gain conclusive evidence as the error quickly approaches machine precision which causes errors which are not due to the method.

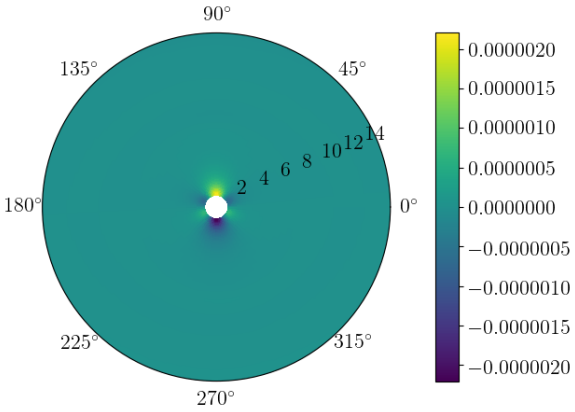


Figure 4.3: Error of the solution of equation 4.29, where the exact solution is specified by equation 4.30. Note that most errors are found near the Neumann boundary.

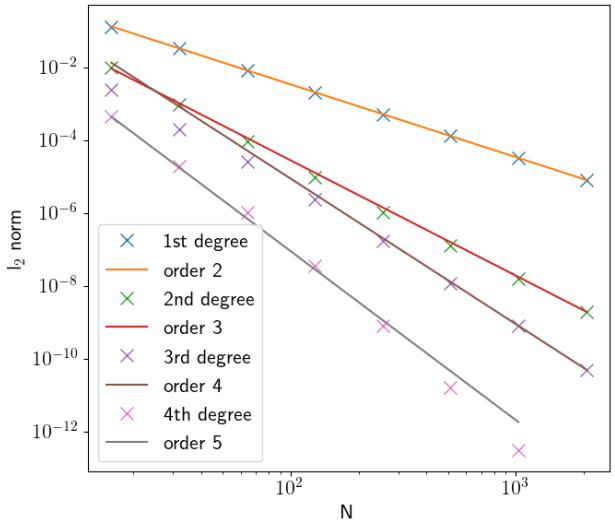


Figure 4.4: Convergence of the Finite Elements method for equation 4.29. Exact values can be found in table 4.1

Chapter 5

Parallel Set-up

5.1 Introduction

Gyrokinetic equations are defined in at least four dimensions. This means that even if a small number of points is used in each dimension, the resulting grid will have a very large number of points. In addition the non-linear nature of the equation means that for most dimensions, an accurate solution cannot be obtained from a very small number of points. Operations on such a high volume of data can be very time consuming. It is therefore important to ensure that parallelisation is used wherever possible.

Parallelisation allows independent calculations to be carried out simultaneously. This decreases the time required to calculate the final solution as the solution at multiple points in the data can be calculated simultaneously. For this method to be effective, it is important that as much of the calculation as possible can be carried out independently. If data stored on a different process is required to complete the calculation then the exchange of data will dramatically increase the overhead, slowing down the calculation time. This can negate the improvement achieved by parallelisation and should therefore be avoided wherever possible. This means that if a calculation requires values from multiple points along one dimension, then ideally that dimension should not be distributed.

This highlights a second consideration: the organisation of the data in memory. Different configurations of the data can lead to different memory latency. If data is accessed contiguously then the loading of cache lines and prefetching (which can be carried out easily by the system as the access pattern is easily recognisable) will ensure that the subsequently required data is already found on a cache. This is important as cache misses can be very costly. The further the data must travel before it can be used, the greater the latency. Optimal memory access is especially important as calculations are often memory-bound rather than compute-bound due to the increase in CPU power following Moore's law and the lack of improvement in memory latency.

It is therefore important to manage access to data structures in an efficient way to ensure that calculations can be carried out as quickly and easily as possible. This should be carried out using parallelisation as well as an optimal arrangement of the data in memory.

In this chapter effective ways of managing the data to fulfil these criteria will be examined.

5.2 Choosing layouts

The arrangement of the data in memory depends on the requirements of the calculations. The screw-pinch model with cylindrical geometry is characterised by 3 operators which arise from the following three advection equations:

$$\partial_t f + v_\parallel \nabla_\parallel f = 0 \tag{5.1}$$

$$\partial_t f + \nabla_\parallel \tilde{\phi} \partial_{v_\parallel} f = 0 \tag{5.2}$$

$$\partial_t f + \{\tilde{\phi}, f\} = 0 \tag{5.3}$$

Equation 5.1 leads to an operator which acts on the particle distribution function f, along the flux surface. This surface is defined on η_2 and η_3 so the memory access for these variables ought to be contiguous for this operator. Equation 5.2 leads to an operator which acts on the distribution function along the velocity or η_4. Therefore the memory access for η_4 ought to be contiguous for this operator. Equation 5.3 leads to an operator which acts on the distribution function along the poloidal surface. This surface is defined on η_1 and η_2 so the memory access for these variables ought to be contiguous for this operator.

Evidently it is not possible to find a structure which is optimal for all three of these operators, however given the amount of calculation involved with each step it is reasonable to change layout between each operator. This has been done previously by Latu et al, 2007 [8]. A set-up must therefore be found such that the following requirements are met:

Requirements

1. The number of processes that can be used is maximised
2. The dimensions used by the operator are not distributed
3. The dimensions used by the operator are contiguous
4. The number of MPI messages sent and received is minimised
5. The memory required is minimised
6. After set-up no memory is allocated
7. As little copying of data as possible is used

With no other requirements, the maximum number of processes possible is equal to the number of grid points. However requirement 2 inhibits requirement 1. Thus the maximum number of processes possible for a given layout is the product of the number of points in each dimension not being used by the operator. In order to achieve a good load balance, the number of processes used in each operation should be the same. Thus the maximum number of processes that can be used at any one time is:

Dimension	Number of points
η_1	256
η_2	512
η_3	32
η_4	128

Table 5.1: Example configuration of data points

$$max\_procs = \min\{n_{\eta_1} \cdot n_{\eta_4},\ \ n_{\eta_1} \cdot n_{\eta_2} \cdot n_{\eta_3},\ \ n_{\eta_3} \cdot n_{\eta_4}\}$$

Using the example set-up in table 5.1, it can be seen that in this situation the maximum number of processes would be $n_{\eta_3} \cdot n_{\eta_4} = 4096$. This shows that it does not make sense to distribute this example in more than two dimensions. Even if we tried to take advantage of being able to distribute in three dimensions for equation 5.2, it would be unlikely that this would significantly increase the total number of processes that could be used, as in general $n_{\eta_4} \ll n_{\eta_1} \cdot n_{\eta_2}$. In addition it is often the case that $n_{\eta_4} < n_{\eta_1}$ and $n_{\eta_4} < \cdot n_{\eta_2}$ which would mean that no gains in the total number of processes would be made. Thus the increase in complexity does not justify the possible gains.

The example set-up in table 5.1 also shows that it makes sense to aim for a distribution in two dimensions. The maximum number of processes that can be used in this case ($n_{\eta_3} \cdot n_{\eta_4} = 4096$) is significantly larger than the maximum number of processes that could be used if the data was only distributed in one dimension ($\eta_4 = 128$). Therefore the two dimensional distribution is optimal, unless it causes very large overhead thus violating requirement 4.

This therefore leaves us with the possible configurations shown in table 5.2. This ordering is such that non-distributed dimensions are always in the final indices, thus satisfying requirement 3.

Accessing scheme	Possible Ordering
Flux surface	$(\eta_1, \eta_4, \eta_2, \eta_3)$
	$(\eta_4, \eta_1, \eta_2, \eta_3)$
	$(\eta_1, \eta_4, \eta_3, \eta_2)$
	$(\eta_4, \eta_1, \eta_3, \eta_2)$
V-parallel surface	$(\eta_1, \eta_2, \eta_3, \eta_4)$
	$(\eta_1, \eta_3, \eta_2, \eta_4)$
	$(\eta_2, \eta_1, \eta_3, \eta_4)$
	$(\eta_2, \eta_3, \eta_1, \eta_4)$
	$(\eta_3, \eta_1, \eta_2, \eta_4)$
	$(\eta_3, \eta_2, \eta_1, \eta_4)$
Poloidal surface	$(\eta_3, \eta_4, \eta_1, \eta_2)$
	$(\eta_4, \eta_3, \eta_1, \eta_2)$
	$(\eta_3, \eta_4, \eta_2, \eta_1)$
	$(\eta_4, \eta_3, \eta_2, \eta_1)$

Table 5.2: Plausible orderings of dimensions in the three different layouts

In order to simplify the problem, additional restrictions are placed on the set-up. Firstly it is assumed that η_2 is never distributed, as in any case there is only

one set-up in which it could be distributed. This has no effect on the preceding discussion. In addition it is assumed that η_2 will always be in the same position in the ordering. As η_4 must be contiguous for the operator associated with equation 5.2, η_2 must therefore always be in the third position. This leaves two possible orderings. The choice between the two is arbitrary (the only difference is that the ordering in the first and second positions are exchanged). The choice made here can be seen in table 5.3.

Accessing scheme	Ordering
Flux surface	$(\eta_1, \eta_4, \eta_2, \eta_3)$
V-parallel surface	$(\eta_1, \eta_3, \eta_2, \eta_4)$
Poloidal surface	$(\eta_4, \eta_3, \eta_2, \eta_1)$

Table 5.3: The chosen ordering for the three different layouts

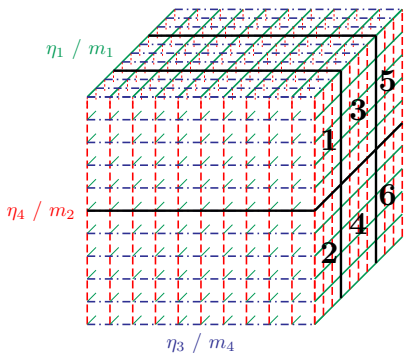
Using this ordering, it is important to find a method for changing layouts which respects requirements 5, 6, 7, and especially 4.

5.3 Moving between layouts

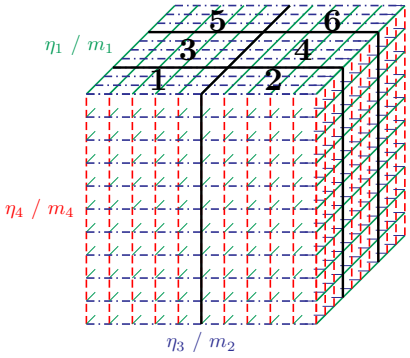
It is clear from the ordering in table 5.3 that at each change of layout: η_2 is untouched, one dimension must become distributed, one dimension must become contiguous and one dimension must not be touched unless the number of processes along that axis changes. The problem can therefore be simplified further by assuming that this will not happen. In other words, in a given distribution direction, at any given moment, the data will be distributed over the same number of processes in that direction. The only difference will be which dimension is stored along that axis in memory (ie. for the ordering in table 5.3, η_1 or η_4 on the first storage axis will each be distributed over the same number of processes). As η_2 does not vary the problem can therefore be visualised in three dimensions as shown in figure 5.1. It should be noted that these decisions have removed some flexibility from our system. In particular, it should be noted that it is impossible to move from a poloidal surface to a flux surface without either first changing to a $v_\parallel$ surface or changing the ordering of the variables in the flux surface.

Figure 5.1 shows three different layouts and allows the viewer to visualise the movement of data between processes during a layout change. Consider the change from the flux-surface layout (figure 5.1a) to the $v_\parallel$ surface layout (figure 5.1b). Process 1 and process 2 must exchange data with one another, however no data is required from any of the other processes. This is also the case for processes 3 and 4, and processes 5 and 6. Similarly when changing from the $v_\parallel$ surface layout to the poloidal surface layout (5.1c) processes 1, 3, and 5 must exchange data only with one another. Indeed, with the conditions imposed each three dimensional layout change is equivalent to multiple independent two dimensional layout changes.

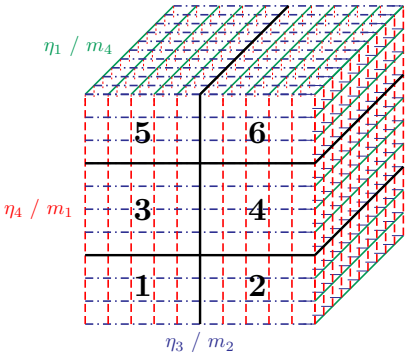
This property can be exploited using an MPI Cartesian grid. This allows the number of messages sent to be greatly reduced thus fulfilling requirement 4. If all processes exchange data amongst themselves, then p^2 messages are exchanged, where p is the total number of processes. In the set-up described here, for a



(a) Flux surface - contiguous in z



(b) $v_\parallel$ surface - contiguous in $v_\parallel$



(c) Poloidal surface - contiguous in r

Figure 5.1: 3-D visualisation of the layout change problem. The axis along which the data is stored in memory is indicated by m_1, m_2, m_4 (m_3 is not represented as it always contains all values of η_2). Note that the storage axis only changes when the distribution pattern along that axis changes.

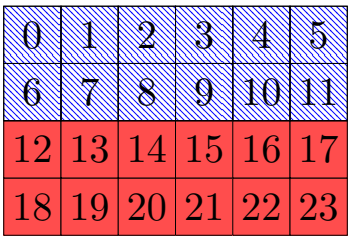
grid distributed along a $n \times m$ grid of processes ($nm = p$), changing the variable represented by the first storage axis requires mn^2 messages, and changing the variable represented by the second storage axis requires nm^2 messages.

In addition the reduced problem can be expressed in one MPI collective command on a sub-communicator: Alltoall. This allows us to use the algorithms provided by MPI rather than relying on multiple point to point messages. This will almost always result in a better optimisation of the code as the MPI implementation can be optimised for hardware.

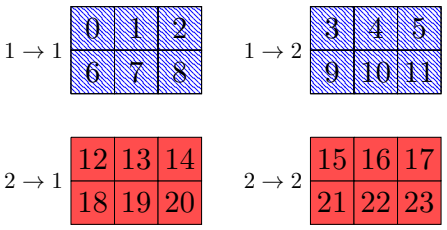
5.3.1 Practical Implementation

The different layouts each have different memory requirements. This would appear to mean that in order to fulfil requirement 6 (after set-up no memory is allocated), one array must be allocated for each layout. However this is not ideal as indicated by requirement 5. In order to improve this situation, views on numpy arrays are used to access the data. Numpy arrays consists of a 1D

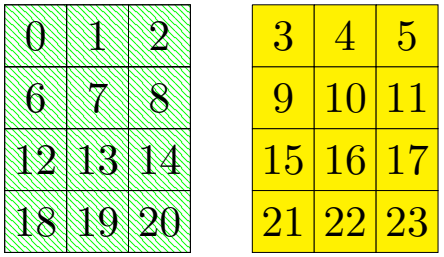
array storage and a set of values representing the step-size between consecutive elements in the same dimension. A numpy view is a numpy array where the data has been stored by a different array. Using views therefore allows the allocation of a single one-dimensional array containing the maximum amount of memory required by any given layout. The view then points to a subset of this memory and allows it to be accessed as a multi-dimensional array.



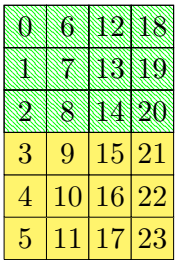
(a) Layout 1



(b) Blocks to be sent. These must be copied to a buffer in order to be contiguous by block in memory



(c) Blocks that are received. These are contiguous but the dimensions are not correctly ordered



(d) Layout 2

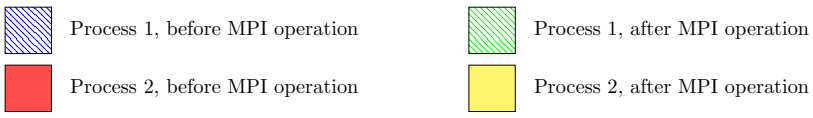
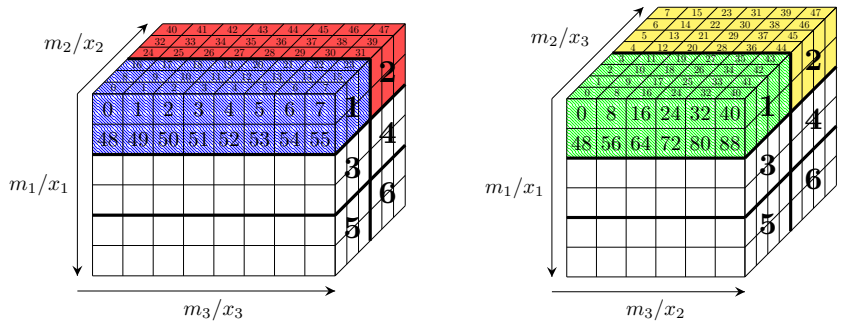


Figure 5.2: 2-D visualisation of the layout change algorithm.

When the layout is changed a second array is required for the data to be received. With careful management no additional memory should be required, thus fulfilling requirement 5.

Note that the numpy views used for each layout will be different as the dimensions represented by each storage axis change. This can be seen in figure 5.1, but is shown more clearly in figure 5.3.

The remaining requirement is requirement 7 regarding the copying of data. In an ideal implementation the data involved would only be copied once, when it is sent. However unfortunately this is not possible with the layouts described above. The reason for this is illustrated in figure 5.2. The data is stored in such a way that it is contiguous in memory (figure 5.2a). However this means that the data which will be moved to a different process is not contiguous (figure 5.2b). As a result a first copy is required before the data can be transmitted.



(a) Layout 1 (b) Layout 2

Figure 5.3: Different layout arrangements in memory. Note that when the layout changes the shape of the data in memory changes (Colouring is relevant for figure 5.4)

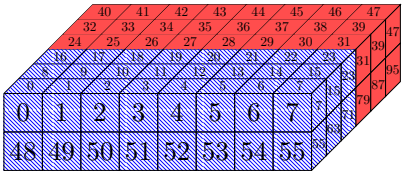
In addition on arrival the data is not ordered correctly (figure 5.2c). In two dimensions, this simply means that a transpose is required (figure 5.2d), however in more dimensions the situation is more complicated. The data must be concatenated along the correct storage axis as well as being transposed to the correct shape. The concatenation step is however not required if the dimension to be concatenated is stored along the first storage axis. This is due to the way in which the data is stored in memory.

The final algorithm utilises this fact. The data is split along the final storage axis and reordered such that the dimension to be concatenated is first. Thus after the MPI operation the only step required is the reordering of the dimensions.

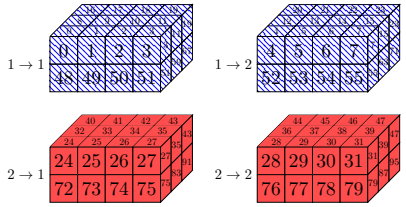
If the dimension to be concatenated is already stored on the first axis then only the first and last storage axes play a role. As a result the algorithm can be visualised in two dimensions as in figure 5.2. The reader can convince themselves of this fact by imagining a third axis on figure 5.2 which does not play a role. This shows that the 2-D case can be extended to 3-D if the second dimension plays no role. The N-D case can be summarised to the 3-D case by representing the (N-2) central dimensions by one variable.

If the dimension to be concatenated is not stored on the first axis then the algorithm must be visualised in three dimensions as in figure 5.4. As previously the data which will be moved to a different process is not contiguous (figure 5.4b). During the first copy, which ensures contiguity for sending, the dimensions are also reordered to position the dimension to be concatenated on the first storage axis (figure 5.4c). The data can then be sent in the same manner as described in the 2-D case. In both cases, the data is not ordered correctly on arrival (figures 5.2c and 5.4d). The data must then be reordered in order to be stored correctly (figure 5.4e).

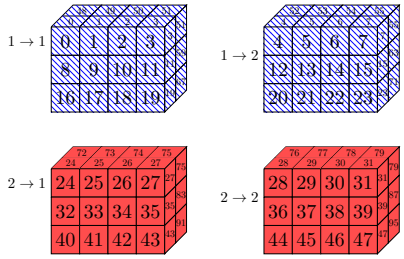
This 3-D visualisation can be used to understand N-D examples for which the dimension to be concatenated is not stored on the first axis. As before, the dimensions on the storage axes between the axis containing the dimension to be concatenated and the final axis play no role. A logic comparable to that used for the 2-D case can therefore be used to show that these axes do not need to be visualised in order to understand the problem. By using a single variable



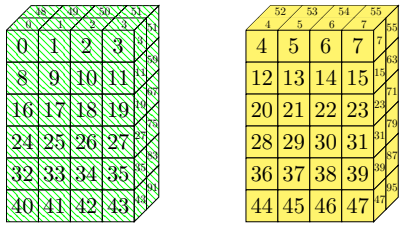
(a) Layout 1



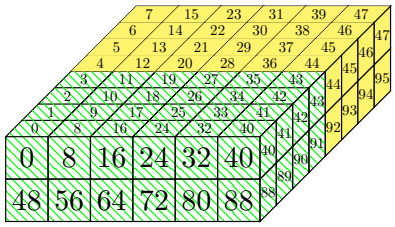
(b) Blocks to be sent. These must be copied to a buffer in order to be contiguous by block in memory



(c) Before sending, the blocks are rearranged so that the axis which was previously distributed becomes the axis with the largest step size (the first axis)



(d) Blocks that are received. These are contiguous but the dimensions are not correctly ordered



(e) Layout 2

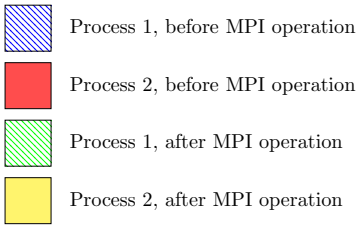


Figure 5.4: 3-D visualisation of the layout change algorithm. The figure shows the steps for moving the data on processes 1 and 2 from the layout described in figure 5.3a to the layout described in figure 5.3b

to represent the dimensions stored on the axes up to the axis containing the dimension to be concatenated, the N-D case can therefore be summarised to the 3-D case.

The algorithm is summarised in algorithm 1 and demonstrated in examples 5.1 and 5.2.

5.3.2 Restrictions

The method described above leads to a few restrictions on the method. Firstly the data must always be distributed over the same two-dimensional grid of processes. This is not problematic as there is little to be gained from changing the organisation of the full data set on the processes.

Secondly, the change of layout can only change two elements in the ordering at any one time. This means that some layout changes cannot be carried out in one step. In the example in table 5.3 it is not possible to change from the flux surface layout to the poloidal surface layout directly. In this case it is however possible to change between all layouts by using two orderings as shown in table 5.4. For example, when ordering 1 of the flux surface layout is used, it is possible to change to ordering 2 of the poloidal layout.

Accessing scheme	Ordering 1	Ordering 2
Flux surface	$(\eta_1, \eta_4, \eta_2, \eta_3)$	$(\eta_4, \eta_1, \eta_2, \eta_3)$
V-parallel surface	$(\eta_1, \eta_3, \eta_2, \eta_4)$	$(\eta_3, \eta_1, \eta_2, \eta_4)$
Poloidal surface	$(\eta_4, \eta_3, \eta_2, \eta_1)$	$(\eta_3, \eta_4, \eta_2, \eta_1)$

Table 5.4: Ordering for the three different layouts which allows moving from any given layout to any other

This will however decrease the total number of processes that can be used. The maximum number of processes that can be used at any one time would then be:

$$max\_procs = \cdot \min\{n_{\eta_1},\ n_{\eta_3},\ n_{\eta_4}\}^2$$

In the example configuration described in table 5.1 it would now only be possible to use a maximum of 1024 processes.

This approach should therefore only be used when it is absolutely necessary to change between all layouts. This is not the case here as Strang splitting will be used. The layouts will therefore be used in the following order:

1. Flux surface
2. V-parallel surface
3. Poloidal surface
4. V-parallel surface
5. Flux surface

Algorithm 1 Algorithm describing the change of layout

Input : A start layout *startLayout* and an end layout *endLayout*
A memory block *Arr* containing the distribution function evaluated in the start layout
An additional memory block *Buf*

Output : A memory block *Buf* containing the distribution function evaluated in the end layout

1: Find the axes (ax_{concat},$ax_{distrib}$) in *startLayout* which are exchanged in *endLayout*
2: Define ax_1, the first axis in the *startLayout*
3: Store views on slices of *Arr*, obtained by splitting along $ax_{distrib}$
4:
5: **for** each (non-contiguous) view **do**
6: Permute the axes of the view using the permutation: (ax_1, ax_{concat})
7: Write the result (contiguously) into *Buf*
8: **end for**
9:
10: **Send** the data from *Buf* using Alltoall
11: **Receive** the data into *Arr*
12: Permute the axes of a view on the data, using the permutation: (ax_1,$ax_{distrib}$,ax_{concat})
13: Write the transposition into *Buf*

Example 5.1 Example demonstrating algorithm 1 by showing the change from a flux surface layout to a v-parallel layout

Arr = f evaluated at $(\eta_1[i:j], \eta_4[k:l], \eta_2[\star], \eta_3[\star])$
$startLayout = (\eta_1, \eta_4, \eta_2, \eta_3)$
$endLayout = (\eta_1, \eta_3, \eta_2, \eta_4)$
1: $(ax_{concat}, ax_{distrib}) \leftarrow (\eta_4, \eta_3)$
2: subBlocks $\leftarrow Arr\big[\eta_1[i:j], \eta_4[k:l], \eta_2[\star], \eta_3[m:p]\big]$
3: $ax_1 \leftarrow \eta_1$
4:
5: for s[$\diamond$,$\square$,$\star$,$*$] in subBlocks:
6: permutation = (η_1,η_4)
7: $Buf[v:w] \leftarrow$ s[$\square$,$\diamond$,$\star$,ast]
$\big(= Arr\big[\eta_4[k:l], \eta_1[i:j], \eta_2[\star], \eta_3[m:p]\big] \big)$

11: $Arr \leftarrow (\eta_4[\star], \eta_1[i:j], \eta_2[\star], \eta_3[m:p])$
12: permutation = (η_1,η_3,η_4)
13: $Buf \leftarrow (\eta_1[i:j], \eta_3[m:p], \eta_2[\star], \eta_4[\star])$

Example 5.2 Example demonstrating algorithm 1 by showing the change from a v-parallel layout to a poloidal layout

Arr = f evaluated at $(\eta_1[i:j], \eta_4[k:l], \eta_2[\star], \eta_3[\star])$
$startLayout = (\eta_1, \eta_3, \eta_2, \eta_4)$
$endLayout = (\eta_4, \eta_3, \eta_2, \eta_1)$
1: $(ax_{concat}, ax_{distrib}) \leftarrow (\eta_1, \eta_4)$
2: subBlocks $\leftarrow Arr\big[\eta_1[i:j], \eta_3[k:l], \eta_2[\star], \eta_4[m:p]\big]$
3: $ax_1 \leftarrow \eta_1$
4:
5: for s[$\diamond$,$\square$,$\star$,$*$] in subBlocks:
6: permutation = $(\eta_1,\eta_1) = (\eta_1)$
7: $Buf[v:w] \leftarrow$ s[$\diamond$,$\square$,$\star$,ast]
$\big(= Arr\big[\eta_1[i:j], \eta_3[k:l], \eta_2[\star], \eta_4[m:p]\big] \big)$

11: $Arr \leftarrow (\eta_1[\star], \eta_3[k:l], \eta_2[\star], \eta_4[m:p])$
12: permutation = $(\eta_1,\eta_4,\eta_1) = (\eta_1,\eta_4)$
13: $Buf \leftarrow (\eta_4[m:p], \eta_3[k:l], \eta_2[\star], \eta_1[\star])$

5.4 Applicability for different models

The discussion above is relevant to a model which has been simplified using the assumption $\mu = 0$ as described by Latu et al, 2018 [1]. However the approach can easily be adapted to a model which does not use this assumption. In this case there are not four but five parameters, with the fifth being the magnetic moment μ. In a collision-less model the magnetic moment only appears in the equations as a parameter. As a result η_5 can simply be added as an additional distributed axis as shown in table 5.5. In addition this would result in an increase of the number of processes that could be used for the simulation:

$$max\_procs = n_{\eta_5} \cdot \min\{n_{\eta_1} \cdot n_{\eta_4},\ \ n_{\eta_1} \cdot n_{\eta_2} \cdot n_{\eta_3},\ \ n_{\eta_3} \cdot n_{\eta_4}\}$$

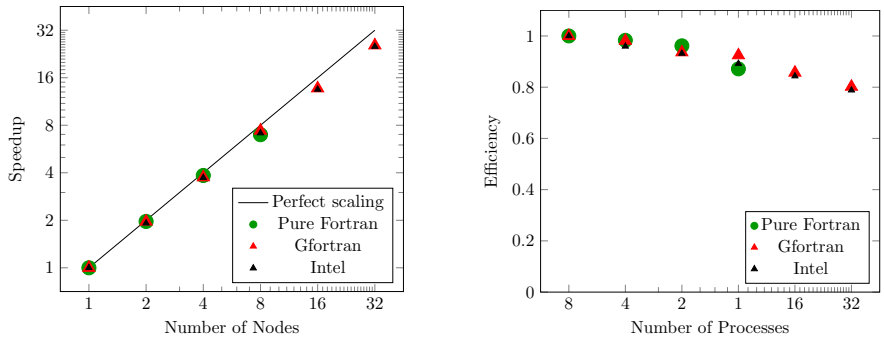
Accessing scheme	Ordering
Flux surface	$(\eta_5, \eta_1, \eta_4, \eta_2, \eta_3)$
V-parallel surface	$(\eta_5, \eta_1, \eta_3, \eta_2, \eta_4)$
Poloidal surface	$(\eta_5, \eta_4, \eta_3, \eta_2, \eta_1)$

Table 5.5: A possible ordering for the three different layouts for a toroidal geometry

Unfortunately in a model in which collisions are considered the magnetic moment is no longer simply a parameter In this case the solution proposed above would still be applicable but different layouts would have to be determined.

5.5 Scaling Results

The parallel set-up described in this chapter was tested using the simulation described in chapter 7. The higher the ratio of calculation to communication



(a) Speedup when increasing the number of nodes used

(b) Efficiency with different numbers of nodes

Figure 5.5: Scaling results for the pyccelised version of the code using different compilers

the better the scaling will be. Therefore in order to effectively test the method described it is important to take the version of the code where the calculations can run as fast as possible. For this reason the results presented in figure 5.5 are not the results for a pure python code, but for the code which has been accelerated using pyccel as described in chapter 6. The scaling results are shown for the code compiled with two different compilers. They are compared with the pure python implementation used by Latu et al. for their paper [1].

The parallel implementations used in this project and by Latu et al are similar but not the same. Latu et al also use a system of layouts, however the layouts are only distributed in one dimension. This limits the number of processes that can be used to 256 (8 nodes) whereas the implementation described here can use 4096 (128 nodes).

We note that the scaling for both implementations is very good. The size of the simulation means that the calculation time dominates. The implementation described here appears to be slightly better, however this is to be expected as the entire code runs three times slower, in the case of gfortran, or four times slower, in the case of intel, than the pure Fortran implementation. It would be more interesting to test this method using a code which runs at speeds more closely approaching those of the pure Fortran implementation. As discussed at the end of chapter 6 this may be possible after improvements to pyccel or by manually improving the generated Fortran code.

Furthermore it is difficult to compare the two implementations as the parallel overhead does not begin to dominate for the maximum number of processes possible on the pure Fortran implementation. It is possible that this point could be seen in this implementation, especially if it is tested using the code accelerated to speeds approaching that of the pure Fortran implementation, however this has not yet been tested due to time restrictions and restrictions on the maximum number of processes for the supercomputer used for these tests.

Chapter 6

Acceleration with Pyccel

6.1 Introduction

As mentioned in chapter 1 python runs much slower than compiled languages. Tests on the code produced thus far at first did not yield results. Once the total size of the simulation was reduced by a factor of eight, it was found that five time steps could be run on one node containing 32 processes in three hours. This means that the full simulation would take more than a year and a half to run on one node. Assuming perfect scaling, it would take nearly 5 days on the theoretical maximum number of processes (4096 processes, 128 nodes), and nearly 20 days on the maximum number of processes allowed by the supercomputer used (1024 processes, 32 nodes). These times are unreasonable, especially if the simulation must be rerun. It is therefore important to find a way to accelerate the code.

Pyccel [9] is one tool that can be used to accelerate python code. It translates the python code into human readable Fortran code. This code can then be compiled and f2py can be used to allow the accelerated version to be called from python. This allows bottlenecks to be run at faster speeds without the code needing to be written in Fortran. It also means that the majority of the code is still written in python which can make it easier to understand. In addition the readability of the Fortran code allows the generated files to be easily modified to exploit any advantages of Fortran which cannot be mimicked in python.

6.2 Using Pyccel

A function can be pyccelised if it contains only basic python and numpy methods[1]. As classes cannot be pyccelised, functions must be created containing the pyccelisable content. These can then be called from the class to mask the larger interface that this creates.

Once the functions to be pyccelised have been selected, the pyccelisation can begin. All functions which will be called directly should be placed in one file. All functions which are used by those functions are then placed in a second file

[1] Not all methods are currently supported by pyccel

whose name must have the form "mod_[XX].py", where [XX] can be anything. The "mod_[XX].py" is used by epyccel, the interactive version of pyccel, as a context, enabling it to generate a shared object file. All functions must also be given a header specifying the Fortran types of the arguments. The header can either take the form of a comment or a function decorator. Function decorators are preferable as they can be imported interactively with the function. The syntax is as follows:

```
#$ header function my_function(double,int,double[:],int[:,:])
@types('double','int','double[:]','int[:,:]')
```

Once the functions have been assembled in the correct files and given the appropriate headers, the context must be compiled. To do this one can use the command:

```
pyccel mod_[XX].py --fflags ' -O3 -fPIC'
```

The Fortran flags must be specified to ensure that the resulting files can be converted with f2py. If one wishes to generate the pyccel file from a different folder, then this can be done with the following command

```
pyccel folder1/folder2/mod_[XX].py --output folder1/folder2
  --fflags ' -O3 -fPIC'
```

This allows all files to be generated simply from one location. This is the version which is used in this project.

Alternatively if the generated Fortran code may be modified then the chosen pyccel command should be given the flag '-t'. The generated code must then be compiled. For example the following command must be used with the gfortran or intel compiler respectively:

```
gfortran -O3 -fPIC -c folder1/folder2/mod_[XX].f90 -o
   folder1/folder2/mod_[XX].o -J folder1/folder2/
ifort -O3 -fPIC -c folder1/folder2/mod_[XX].f90 -o
   folder1/folder2/mod_[XX].o -module folder1/folder2/
```

Once the context file is prepared then a shared object file can be generated using epyccel. In order to do this the module to be pyccelised and the functions from the context file must be imported. A ContextPyccel object should then be created containing all the functions required, and the types of their arguments. If the pyccelisation is carried out from a folder which does not contain the modules then the position of the module in the python hierarchy must be specified using the context_folder keyword. The following code, showing the commands required to generate the shared object for the spline module, serves as an example:

```
from pyccel.epyccel import ContextPyccel
from pyccel.epyccel import epyccel

import pygyro.splines.spline_eval_funcs
from pygyro.splines.mod_context_1 import
    find_span, basis_funs, basis_funs_1st_der

spline_context = ContextPyccel(name='context_1',
    context_folder='pygyro.splines')
spline_context.insert_function(find_span,
    ['double[:]','int','double'])
spline_context.insert_function(basis_funs,
    ['double[:]','int','double','int','double[:]'])
```

```
spline_context.insert_function(basis_funs_1st_der,
    ['double[:]','int','double','int','double[:]'])

spline_eval_funcs = epyccel(pygyro.splines.spline_eval_funcs,
    context=spline_context)
```

6.3 Spline Acceleration

In order to reap maximum benefits from the acceleration it is important to accelerate the functions which take the most time. These act as a bottleneck for the program. In order to determine which functions these are, the python profiler cProfile will be used on a personal laptop to examine the performance of the program using a grid with 10 grid points in each dimension. Three processes are used and the results are averaged over each process. The results for the pure python implementation are shown in table 6.1.

Function	Total time excluding sub functions [s]	Number of calls	Time per call [s]	Total time [s]
method 'Alltoall' from mpi4py	28.193	250	0.113	28.193
numpy.core.multiarray.array	16.070	5689347	0.000	16.070
bisplev from scipy	10.009	1006666	0.000	37.819
method scipy.interpolate._fitpack._bispev	8.503	1006666	0.000	8.503
atleast_1d from numpy	7.990	2915166	0.000	23.820
splev	7.334	901833	0.000	18.883
reshape from numpy	6.590	3397927	0.000	6.590

Table 6.1: The results of profiling the pure python implementation

We note firstly that the slowest function is mpi4py's Alltoall function. This is due to the small size of the grid which means that calculations can run faster, and the fact that the tests were run on 3 processes. This leads to a poor data balance as 2 processes have 3x10x10x10 sized grids and the third has a 4x10x10x10 sized grid. As a collective operation Alltoall therefore is obliged to wait until all processes have reached the same point. These problems will not apply when the full simulation is run so this function will not be accelerated.

The second bottleneck is due to internal numpy functions. There is nothing that can be done about this directly although it can be hoped that accelerating other functions will lead to less reliance on numpy which will therefore eliminate this bottleneck.

The next four bottlenecks all arise due to the implementation of the splines. Bisplev is a scipy function which calls bispev in order to evaluate a 2d spline while splev is its 1d equivalent. The reshape command is used to flatten the 2D array containing the coefficients for the bisplev function, as well as during all layout changes. The function atleast_1d is only used explicitly during the initialisation, which is not profiled. This means that the function must be called implicitly, probably by one of the scipy functions. The eval functions

of the classes Spline1D and Spline2D are therefore the functions which will be accelerated first.

In order to accelerate these two functions a mod_context_1.py file will be created. This file will contain the functions find_span, basis_funs, and basis_funs_1st_der. These functions are obtained from the SPL library [10]. In addition a spline_eval_funcs.py file will be created. The file will contain the functions eval_spline_1d_scalar, eval_spline_1d_vector, eval_spline_2d_scalar, and eval_spline_2d_cross which mimic the behaviour of the scipy functions splev and bisplev. Furthermore a function eval_spline_2d_vector will be added to help avoid unaccelerated Python loops. The scipy function calls will then be replaced by an if statement which verifies whether the input has a length, and a call to the necessary function. This implementation ensures that calls elsewhere in the program do not need to be modified. It may however slow the program somewhat as it adds additional function calls which are unnecessary if the type of the data is known.

The results of the acceleration can be seen in table 6.2. It can be seen that although the spline calls remain some of the most costly function calls, the total time spent in each function is significantly reduced. For example in the case of the 2D spline evaluation, previously 37.8 seconds were spent in the function bisplev and its sub-functions, in contrast now only 12.3 seconds are spent in eval in total. Similarly the 1D spline evaluation previously required 18.8 seconds and now takes only 6.3 seconds.

Function	Total time excluding sub functions [s]	Number of calls	Time per call [s]	Total time [s]
method 'Alltoall' from mpi4py	10.461	250	0.042	10.461
numpy.core.multiarray.array	9.340	1624947	0.000	9.340
eval in Spline2D	7.225	1006666	0.000	12.345
step in PoloidalAdvection	5.423	3333	0.002	35.266
eval in Spline1D	4.948	901833	0.000	6.319
step in FluxSurfaceAdvection	4.744	5000	0.001	9.527
_vectorize_call	4.403	383333	0.000	25.365

Table 6.2: The results of profiling the implementation after the acceleration of the spline functions

6.4 Initialisation Acceleration

The profiling results will once more be used to determine which functions should be accelerated. The slowest functions are the eval functions of the Spline1D and Spline2D classes, the poloidal advection and the flux surface advection. These functions call the eval functions. As part of the acceleration they can call the previously accelerated functions directly.

The advection steps use the expression for the equilibrium distribution at the boundaries. As a result, the initialisation functions must first be accelerated in order to accelerate the advection equations.

Unfortunately, unlike the splines, the initialisation cannot be accelerated while keeping the old file access patterns. The initialisation takes advantage of the flexibility of numpy which can handle both scalar and vector functions however Fortran does not have this flexibility. In order to avoid loosing speed the files will therefore be organised such that a file called mod_initialiser_funcs.py will be created containing all scalar functions and a file called initialiser_func.py will be created to handle the vector case by looping over the scalar function. mod_initialiser_funcs.py will also be used as a context for the advection.

Function	Total time excluding sub functions [s]	Number of calls	Time per call [s]	Total time [s]
method 'Alltoall' from mpi4py	20.373	250	0.081	20.373
numpy.core.multiarray.array	9.581	1872347	0.000	9.581
eval in Spline2D	7.423	1006666	0.000	12.722
step in PoloidalAdvection	5.639	3333	0.002	42.126
eval in Spline1D	5.211	910400	0.000	6.655
step in FluxSurfaceAdvection	4.802	5000	0.001	9.772
_vectorize_call	4.538	383333	0.000	25.898

Table 6.3: The results of profiling the implementation after the acceleration of the spline and initialisation functions

As the accelerated functions are used in the initialisation which is not profiled we do not expect to see a change in the results after these improvements. The new results of profiling are shown in table 6.3 and are as expected.

6.5 Advection Acceleration

We are now able to accelerate the advection functions. We begin by accelerating the poloidal advection. The files written for the splines and the initialisation are used to provide a context for the accelerated advection file. Unfortunately a context cannot have its own context. Thus a file mod_spline_eval_funcs.py must be created containing the contents of mod_context_1.py and spline_eval_funcs.py.

Once this step has been accomplished successfully the pyccelisable parts of the v-parallel and flux surface advections can be added to the accelerated advection file very simply.

The results of the profiling after this pyccelisation can be seen in table 6.4. Although the advection steps are still the slowest steps we note that while previously 42.1 seconds were spent in the PoloidalAdvection step and its sub-function, now only 3.1 seconds are required. Similarly while previously 9.8 seconds were spent in the FluxSurfaceAdvection step and its sub-functions, now only 3.4 seconds are required.

The remaining functions are harder to accelerate. They each contain very little code, but call 'eval' from Spline1D multiple times. This allows for a small improvement as it is possible to call the accelerated function directly without using Spline1D. This avoids the type check which decides whether to use the scalar or vector version of the accelerated function. An additional improvement

Function	Total time excluding sub functions [s]	Number of calls	Time per call [s]	Total time [s]
method 'Alltoall' from mpi4py	4.361	250	0.017	4.361
step in FluxSurfaceAdvection	1.874	5000	0.000	3.428
step in PoloidalAdvection	1.489	3333	0.000	3.135
method 'solve' from scipy 'SuperLU' object	1.330	96666	0.000	1.330
getPerturbedRho in ParallelGradient	1.221	101	0.012	5.407
parallel_gradient in ParallelGradient	1.213	1000	0.001	2.348
_solve_system_nonperiodic in SplineInterpolator1D	1.109	123700	0.000	1.109

Table 6.4: The results of profiling the implementation after the acceleration of the spline, initialisation, and advection functions

was found in 'getPerturbedRho' by calculating the value of the distribution function at equilibrium in the initialisation, thus avoiding recalculating it at each step.

The profile after these final improvements can be seen in table 6.5. Note that the improvements have slowed the parallel_gradient function as the calls are more complicated but the overall time spent in this function and its subfunctions is still greatly reduced. Note also that the bottleneck due to the numpy internal functions has also disappeared as hoped.

6.6 Results

Having accelerated the functions on a personal laptop, it is important to verify that the results hold for the full grid on the supercomputer. Table 6.6 show the averaged results of profiling the full 256x512x32x128 grid on one node with 32 processes. The results are very similar to those seen in table 6.4. This shows that this method was sufficient for finding the bottlenecks.

Note that the mpi 'Alltoall' command no longer dominates the simulation. This is due to better balancing and a better calculation to communication ratio. Instead the advection steps are by far the biggest bottleneck. If any further improvements are required, they would therefore ideally be found in these functions.

The performance of the generated code depends on the compiler used. For the profiling, gfortran was always used, however on the supercomputer the results can also be tested using the intel compiler. The improvements obtained are shown in table 6.7.

The pure Fortran implementation referenced is that used by Latu et al. [1] for their work on the same problem. Although the best pyccel version is still three times slower than the pure Fortran version, it still runs at an acceptable speed. The speed of development also makes this increase easily justifiable. In addition

Function	Total time excluding sub functions [s]	Number of calls	Time per call [s]	Total time [s]
method 'Alltoall' from mpi4py	2.735	250	0.011	2.735
step in FluxSurfaceAdvection	1.826	5000	0.000	3.355
parallel_gradient in ParallelGradient	1.492	1000	0.001	1.903
step in PoloidalAdvection object	1.420	3333	0.000	3.004
method 'solve' from scipy 'SuperLU'	1.294	96666	0.000	1.294
getPerturbedRho in ParallelGradient	1.022	101	0.010	2.307
_solve_system_nonperiodic in SplineInterpolator1D	0.977	123700	0.000	0.977

Table 6.5: The results of profiling the implementation after the acceleration of the spline, initialisation, and advection functions

there is still scope for improvement. The generated code could be improved, for example by indicating which functions are 'pure', or by using more compiler flags. The algorithms could also be modified to increase the speed. For example the implementation of the spline evaluation is very general however the points are equidistant in all cases, except for points close to the boundary along an axis which does not have periodic boundary conditions. This knowledge can be exploited to produce faster code, as is done in the pure Fortran version.

Version	Time per step [s]	Improvement
Pure Python	17748[2]	-
Pyccel accelerated (intel)	487.6	36.4
Pyccel accelerated (gfortran)	342.5	51.8
Modified pyccel accelerated (gfortran)	153.2[3]	115.8
Pure Fortran	122.7	144.7

Table 6.7: Time required to carry out one calculation step on one node containing 32 processes for various versions of the simulation, and the improvement that this represents compared to the pure python version

In addition the parallelisation method allows more processes to be used in the python version than in the original Fortran version. While the Fortran version is limited to 256 processes, the python version can use up to 4096 processes. Figure 6.1 shows the timings for the various versions and shows that this advantage

[2] Approximated from the timings using a smaller grid and only 5 timesteps
[3] Approximated from the timings using only 5 timesteps

Function	Total time excluding sub functions [s]	Number of calls	Time per call [s]	Total time [s]
step in FluxSurfaceAdvection	522.026	15360	0.034	579.579
step in PoloidalAdvection	510.948	1280	0.399	559.224
getPerturbedRho in Parallel-Gradient	184.991	11	16.817	231.574
method 'Alltoall' from mpi4py	108.146	78	1.386	108.146
step in VParallelAdvection	102.033	1966080	0.000	131.171
method 'solve' from scipy 'SuperLU' object	82.426	849920	0.000	82.426
_solve_system_nonperiodic in SplineInterpolator1D	50.279	4109824	0.000	50.279

Table 6.6: The results of profiling the implementation on one node of the supercomputer after accelerating the functions

means that the pyccel accelerated code, accelerated using gfortran can run faster than the Fortran code if it uses 1024 processes. Unfortunately results cannot be shown on more processes as the available supercomputers either did not have the required packages or did not allow programs running on more than 32 nodes.

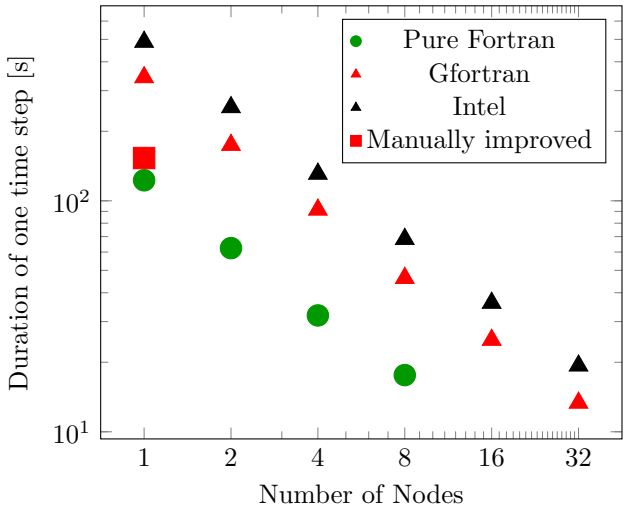


Figure 6.1: Comparison of time required to carry out one calculation step on different numbers of nodes, each containing 32 processes

A first attempt was made to manually improve the code generated by pyccel by avoiding unnecessary calls to the allocate routine and by marking functions as pure. Unfortunately due to time restrictions the results of this improvement cannot be fully presented. However figure 6.1 and table 6.7 are still able to present a first estimation of the effect that these improvements would have. We see that the resulting code is now only 1.2 times slower than the pure

Fortran code. This is an excellent result which clearly demonstrates that this development method is a good alternative to developing in Fortran. In addition the changes made were relatively simple, it is therefore hoped that a more mature version of pyccel would be able to generate code with these changes already made.

The new code was also profiled on a personal laptop, and the results can be seen in table 6.8. We note that the bottleneck is now the parallel_gradient function which has so far received little attention. This may imply that there is scope for further improvements.

Function	Total time excluding sub functions [s]	Number of calls	Time per call [s]	Total time [s]
method 'Alltoall' from mpi4py	2.295	250	0.009	2.295
parallel_gradient in ParallelGradient	1.546	1000	0.002	2.005
method 'solve' from scipy 'SuperLU'	1.324	88000	0.000	1.324
step in FluxSurfaceAdvection	1.083	4500	0.000	2.645
_solve_system_nonperiodic in SplineInterpolator1D	1.036	111330	0.000	1.036
compute_interpolant in SplineInterpolator1D	0.983	199330	0.000	4.275
step in VParallelAdvection	0.942	45000	0.000	1.735

Table 6.8: The results of profiling the implementation after manually improving the code generated by pyccel

Chapter 7

Simulation

7.1 Method Summary

Once all the steps have been written the complete algorithm can be assembled. Memory is allocated for the distribution function f, the electric potential *phi* and the perturbed charge density *rho*. Each of these objects has associated layouts. The layouts used for the distribution function can be seen in table 7.1. The layouts used for *phi* and *rho* can be seen in table 7.2.

Accessing scheme	Ordering
flux_surface	$(r, v_\parallel, \theta, z)$
v_parallel	$(r, z, \theta, v_\parallel)$
poloidal	$(v_\parallel, z, \theta, r)$

Table 7.1: The chosen ordering for f (the distribution function) with (n_1,n_2,1,1) processes used for the respective dimension

The 3D grids have layouts that are distributed in both one and two dimensions. When the data is distributed in one dimension the data is copied across the processes. It is therefore simple to change from a layout distributed in one dimension to a similar layout distributed in two dimensions as it is simply a matter of restricting the data to the area of interest. In the opposite direction a MPI allgather command is required.

Accessing scheme	Ordering	Number of processes per dimension
v_parallel_2d	(r, z, θ)	(n_1,n_2,1)
mode_solve	(θ, z, r)	(n_1,n_2,1)
poloidal	(z, θ, r)	(n_2,1,1)
v_parallel_1d	(r, z, θ)	(n_1,1,1)

Table 7.2: The ordering for *phi* (the electric displacement) and *rho* (the perturbed charge density). n_1 and n_2 are the number of processes along which the first and second dimension of the distribution function are distributed

The resulting algorithm is as follows:

1. Compute $\tilde{\phi}$ from f^n by solving the quasi-neutrality equation 2.53
   (a) Find the charge perturbation density ρ_1 by integrating the perturbed density function with respect to $v_\parallel$: $\int (f - f_{eq}) dv_\parallel$
   (b) Get the Fourier transform of the charge perturbation density ρ_1 : $\hat{\rho}_1$
   (c) Set the layout of *rho* and *phi* to 'mode_solve'
   (d) Solve the quasi-neutrality equation using the finite elements method in Fourier space to obtain the Fourier transform of the electric potential : $\mathcal{F}\left[\tilde{\phi}\right]$
   (e) Set the layout of *rho* and *phi* to 'v_parallel_2d'
   (f) Get the inverse Fourier transform of the electric potential $\mathcal{F}\left[\tilde{\phi}\right]$: $\tilde{\phi}$
2. Set the layout of f^n to 'flux_surface'
3. Save the values in f^n to a buffer
4. Compute $f^{n+\frac{1}{2}}$ using Lie splitting
   (a) Carry out a flux surface advection half step
   (b) Set the layout of f^n to 'v_parallel' and the layout of *phi* to 'v_parallel_1d'
   (c) Carry out a v parallel advection half step
      i. For each point along r:
         A. Calculate the parallel gradient of $\tilde{\phi}$: $\vec{\nabla}_\parallel \tilde{\phi}$ and save the spline representation of the result
         B. Carry out a v parallel advection half step at each point with the given value of r
   (d) Set the layout of f^n and *phi* to 'poloidal'
   (e) Carry out a poloidal advection half step
      i. For the first value in the $v_\parallel$ direction:
         A. Calculate and store the spline representation of $\tilde{\phi}$ for each point along the z direction
         B. Carry out a poloidal advection half step for each point with the given value of $v_\parallel$
      ii. For the remaining values in the $v_\parallel$ direction : Use the stored spline representations to carry out a poloidal advection half step for each point with the given value of $v_\parallel$
5. Set the layout of $f^{n+\frac{1}{2}}$ to 'v_parallel'
6. Compute $\tilde{\phi}$ from $f^{n+\frac{1}{2}}$ by solving the quasi-neutrality equation again using the method in point 1
7. Restore the values of f^n from the buffer thus reverting to layout 'flux_surface'
8. Compute f^{n+1} using Strang splitting
   (a) Carry out a flux surface advection half step

(b) Set the layout of f^n to 'v_parallel' and the layout of *phi* to 'v_parallel_1d'

(c) Carry out a v parallel advection half step as described in point 4c

(d) Set the layout of f^n and *phi* to 'poloidal'

(e) Carry out a **full** poloidal advection step as described in point 4e

(f) Set the layout of f^n to 'v_parallel'

(g) Carry out a v parallel advection half step

i. For each point along r: Carry out a v parallel advection half step at each point with the given value of r using the saved value of $\vec{\nabla}_{\parallel}\tilde{\phi}$

(h) Set the layout of f^n to 'flux_surface'

(i) Carry out a flux surface advection half step

7.2 Numerical Results

The parameters used are the same as those in the paper by Latu et al. [1], namely:

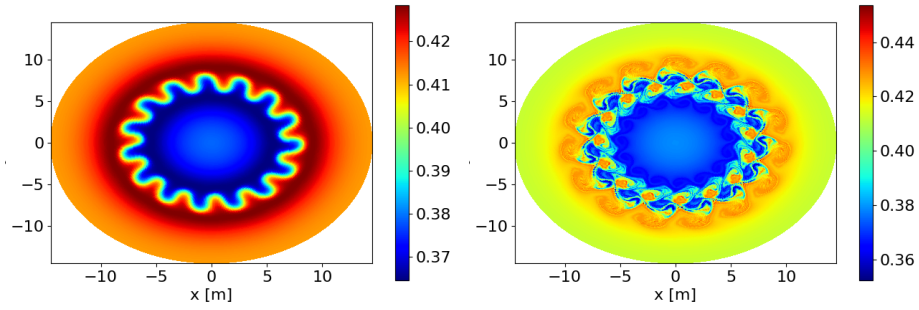
$$B_0 = 1, \qquad R_0 = 239.8081535, \qquad r_{\min} = 0.1, \qquad r_{\max} = 14.5,$$
$$r_p = \frac{r_{\min} + r_{\max}}{2}, \qquad \epsilon = 10^{-6}, \qquad \kappa_{n_0} = 0.055, \qquad \kappa_{T_i} = \kappa_{T_e} = 0.27586,$$
$$\delta_{r_{T_i}} = \delta_{r_{T_e}} = \frac{\delta_{r_{n_0}}}{2} = 1.45, \qquad \delta_r = \frac{4\delta_{r_{n_0}}}{\delta_{r_{T_i}}}, \qquad v_{\max} = 7.32$$

The grid size is as follows:

$$N_r = 256 \qquad N_\theta = 512 \qquad N_z = 32 \qquad N_{v_\parallel} = 128$$

The simulation also uses the following parameters:

$$\iota = 0 \qquad m = 15 \qquad n = 1$$



(a) T=4000 (b) T=6000

Figure 7.1: Poloidal cut of the distribution function at different times at z=0 and $v_\parallel \approx 0$

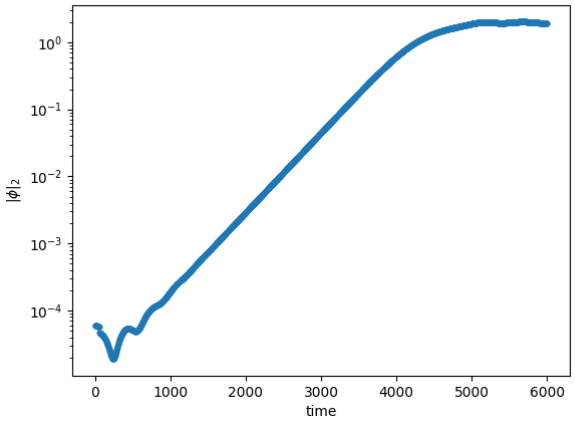


Figure 7.2: The time evolution of the discrete L^2-norm of the electric potential

The equilibrium function as well as the radial profiles $\{T_i\ ,\ T_e\ ,\ n_0\ \}$ are defined in the paper by Latu et al. [1].

The simulation results can be seen in figure 7.1.

In addition to the poloidal slice the time evolution of the discrete L^2-norm of the electric potential is used as a diagnostic quantity. This norm is defined as follows:

$$|\phi|_2(t) := \sqrt{\int_{r=r_{\min}}^{r=r_{\max}} \int_{\theta=0}^{\theta=2\pi} \int_{z=0}^{z=2\pi R_0} \phi(t,r,\theta,z)^2 r\, dr\, d\theta\, dz} \tag{7.1}$$

A trapezoidal rule is used to approximate the value. The results can be seen in figure 7.2.

We see that the L^2 norm demonstrates a linear phase as expected. Unfortunately the gradient seen in this region is not what is expected. In addition while the poloidal planes demonstrate a similar shape to those seen by Latu et al. there appears to be some missing turbulence. We therefore conclude that the code still requires some debugging. Unfortunately time restrictions did not allow this step to be completed in time. The results so far are however close enough to the expected result to imply that it is possible to generate an accurate scientific code using this method.

Chapter 8

Conclusion

Over the course of six months a complete gyrokinetic simulation was written. The code comprised of multiple advection steps, a poisson solver and a parallel implementation allowing the use of up to 4096 processes. This development time is much shorter than might be expected for such a code written in a lower level language such as C or Fortran and thus demonstrates the advantages of coding in python. Unfortunately the final code did not provide a completely accurate physical model, however the results are encouraging enough to indicate that with more time for debugging an accurate model could be produced. The resulting python code, as expected, was far too slow to have a practical use. This case was therefore a good choice of problem to test the capabilities of pyccel.

The code generated by pyccel was much better than the original python code however it was still at best three times slower than the equivalent pure Fortran implementation of the problem. This is sufficiently fast for the code to be useful however it is sub-optimal, especially in cases where the code will be reused multiple times. Simple improvements to the generated code however increased the speed so that the resulting implementation was only 1.2 times slower than the equivalent pure Fortran implementation. Further improvements of this type such as using the 'elemental' keyword may provide even greater gains. Although it is a shame that these pyccel was incapable of providing these improvements, their addition was a very simple process and it is possible that they may appear in a more mature version of pyccel. This means that the results from this method are very much a valid example of how to accelerate python code in a practical manner.

The parallel performance seen in this project was also very good. The maximum number of processes is very large which is very practical for large simulations with many points such as plasma simulations. In the tests conducted thus far the scaling seen using this method was also excellent, with 80% efficiency being seen when 32 nodes were used. Although it is important to revisit these results when the code is running at the increased speed the first results are very promising.

The final code presented here runs sufficiently quickly when compared to the equivalent pure Fortran implementation of the problem, however there is much scope for improvement. In addition to the improvements to the Fortran code already mentioned, compiler flags could be used to improve the speed of

the generated code. Algorithmic changes could also be made to improve the method used or to increase the speed by making it more specific. For example the spline implementation is very general and handles non-uniform domains, however in the case examined here the grid is uniform except near the boundaries in dimensions which do not have periodic boundary conditions. This knowledge can be exploited to increase the speed. Finally it would also be possible to pyccelise other parts of the code. For the moment only code containing pure python and numpy functions has been pyccelised, however as pyccel contains implementations for blas and MPI routines other more complicated sections could also be accelerated. This includes the spline interpolation calculation which is currently one of the main bottlenecks. Accelerating the MPI routines would also help avoid additional parallel overhead which may impact the scaling once the rest of the code is sufficiently accelerated.

In conclusion the method presented in this report generates code which is sufficiently fast to be used for real world applications, although some manual improvement is necessary to get the best possible results. In addition the parallelisation method used shows good scaling, allows for a very large number of processes and can be adapted to be used in other cases, including those where the problem has more dimensions.

Bibliography

[1] G. Latu, M. Mehrenberger, Y. Güçlü, M. Ottaviani, and E. Sonnendrücker, "Field-aligned interpolation for semi-lagrangian gyrokinetic simulations," *Journal of Scientific Computing*, vol. 74, pp. 1601–1650, March 2018.

[2] H. Sugama, "Gyrokinetic field theory," *Physics of Plasmas*, vol. 7, no. 2, pp. 466–480, 2000. [Online]. Available: https://doi.org/10.1063/1.873832

[3] A. J. Brizard, "Variational principle for nonlinear gyrokinetic vlasov–maxwell equations," *Physics of Plasmas*, vol. 7, no. 12, pp. 4816–4822, 2000. [Online]. Available: https://doi.org/10.1063/1.1322063

[4] A. Bottino and E. Sonnendrücker, "Monte carlo particle-in-cell methods for the simulation of the vlasov-maxwell gyrokinetic equations," *Journal of Plasma Physics*, vol. 81, no. 5, p. 435810501, October 2015.

[5] N. Tronko, A. Bottino, and E. Sonnendrücker, "Second order gyrokinetic theory for particle-in-cell codes," *Physics of Plasmas*, vol. 23, no. 8, p. 082505, 2016. [Online]. Available: https://doi.org/10.1063/1.4960039

[6] X. Garbet, Y. Idomura, L. Villard, and T. H. Watanabe, "Gyrokinetic simulations of turbulent transport," *Nuclear Fusion*, vol. 50, p. 043002, 2010.

[7] H. Prautzsch, W. Boehm, and M. Paluszny, *Bézier and B-Spline Techniques*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002. [Online]. Available: https://doi.org/10.1007/978-3-662-04919-8_5

[8] G. Latu, N. Crouseilles, V. Grandgirard, and E. Sonnendrucker, "Gysela 5d, a gyrokinetic semilagrangian parallel code," July 2018.

[9] "Pyccel," https://github.com/pyccel/pyccel.

[10] "Spl python library," https://github.com/pyccel/spl.git.