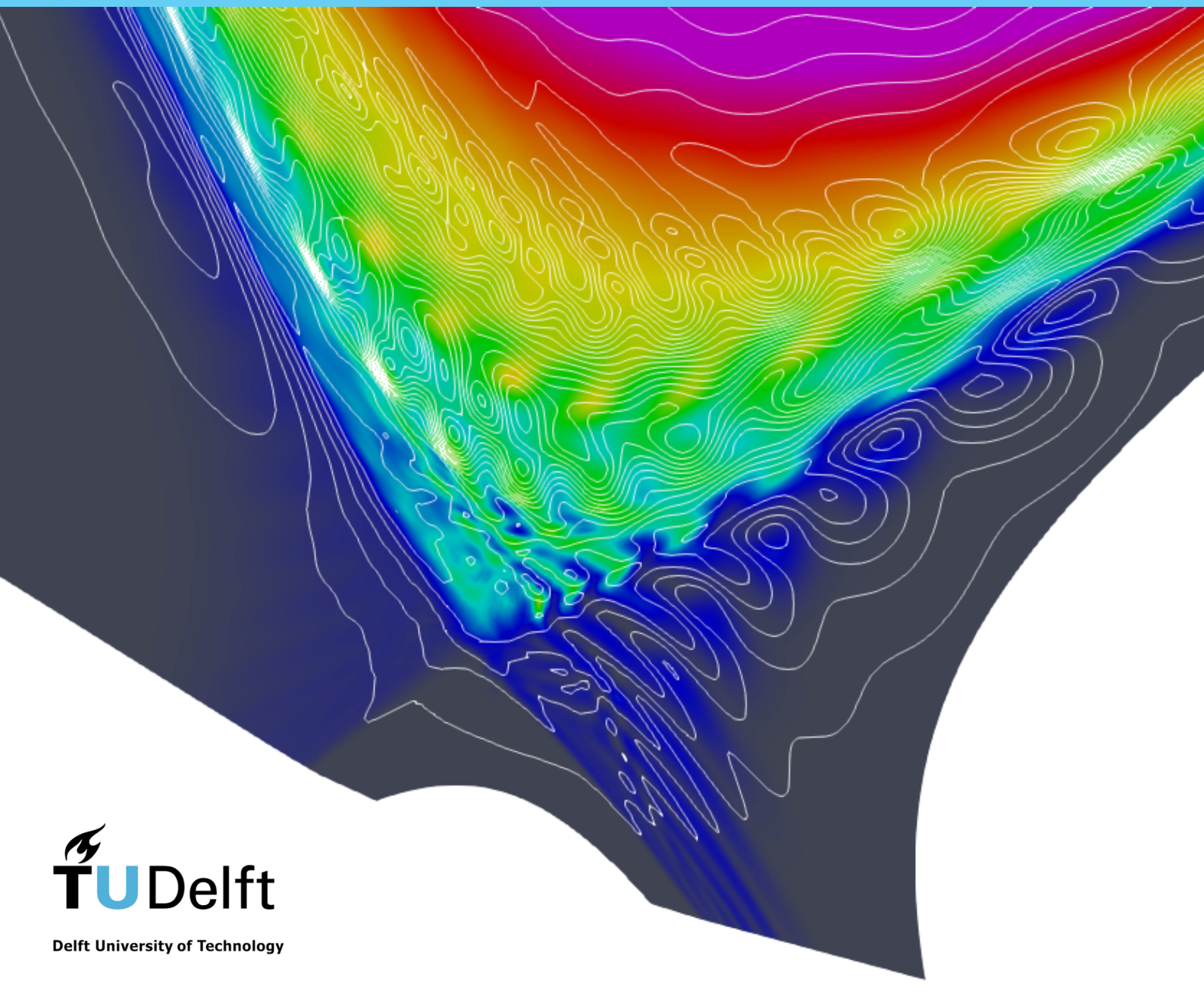
Towards Robust Numerical Solvers for Nuclear Fusion Simulations Using JOREK

Alexander Quinlan
Supervisor: Vandana Dwarka, PhD



TU Delft
Delft University of Technology

Towards Robust Numerical Solvers for Nuclear Fusion Simulations Using JOREK

MSc Thesis report

by

Alexander Quinlan

to obtain the degree of Master of Science
at the Delft University of Technology
to be defended publicly on September 29, 2023 at 13:00.

Part of this thesis was written into an article on Arxiv and
can be found at https://doi.org/10.48550/arXiv.2308.16124.

An electronic version of this thesis is available at
http://repository.tudelft.nl/

Thesis committee:	
Chair:	Dr. Kees Vuik
Supervisors:	Dr. Vandana Dwarka
External examiner:	Dr. Hai Xiang Ling
Place:	EEMCS, Delft
Project Duration:	November, 2022 - September, 2023
Student number:	5563275

Faculty of Electrical Engineering,
Mathematics, and Computer Science · Delft University of Technology

Abstract

One of the most well-established codes for modeling non-linear Magnetohydrodynamics (MHD) for tokamak reactors is JOREK, which solves these equations with a Bézier surface based finite element method. This code produces a highly sparse but also very large linear system. The main solver behind the code uses the Generalized Minimum Residual Method (GMRES) with a physics-based preconditioner. Even with the preconditioner there are issues with memory and computation costs and the solver doesn't always converge well. This work contains the first thorough study of the mathematical properties of the underlying linear system, enabling us to diagnose and pinpoint the cause of hampered convergence. In particular, analyzing the spectral properties of the matrix and the preconditioned system with numerical linear algebra techniques will open the door to research and investigate more performant solver strategies, such as projection methods.

Contents

1 Introduction

Extreme global warming and turbulent geopolitics have made it clear that our society needs new clean energy sources. One approach under active development is controlled nuclear fusion. In this approach, magnetic fields confine a super-hot plasma, mimicking to some extent the conditions that power stars. However, the inherent chaos of plasma dynamics and the high cost of reactor construction underscore the need for robust numerical modeling of plasma dynamics and reactor behavior.

The leading design for controlled fusion is the tokamak, which is a torus lined with superconducting magnets that drive a toroidal magnetic field. Fuseable elements such as hydrogen and deuterium are injected into the device and heated until ionization occurs, forming a plasma. The magnetic field then confines these elements within the walls of the device, where at sufficient temperatures and pressures, they can fuse into larger elements and release energy that can be captured.

ITER, the world's most expensive science experiment currently being built, is a 30-meter tall tokamak fusion reactor that aims to demonstrate a significant energy release from the fusion process and spur a generation of commercializable fusion power plants. Its design and analysis require immense computational resources, and one of the codes used to model plasma physics inside ITER is JOREK, developed by an international community including the Max Planck Institute for Plasma Physics (IPP) in Garching, Germany.

JOREK is an advanced and widely-established code used to simulate plasma behavior in tokamaks. It is designed to model plasma instabilities that can shut down a plasma or damage the walls of reactors. JOREK uses a finite element model over Bézier surfaces and a toroidal Fourier decomposition to model toroidal plasmas. It produces a very large system of equations, and has several approaches to solve these efficiently. However, a lot of work still remains to be done to improve the solver efficiency, as the nonlinear model converges slowly. The purpose of this project is to analyze this system of equations with respect to the physics of the model, and to develop ways to improve solver convergence. Primarily, we intend to build on JOREK's preconditioner, and to develop heuristics by which alternative solvers or alternative models could be suggested to the user.

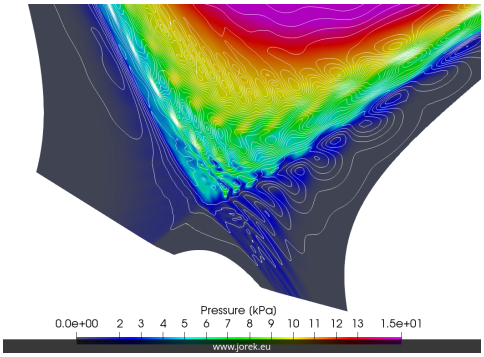


Figure 1.1: Plasma pressure during an edge-localized-mode in the Joint European Torus plasma. [Futatani et al., 2019])

2

Plasma Physics and Magnetohydrodynamics

2.1. Introduction

Plasmas are the most abundant form of matter in the universe. From our sun, to the interstellar medium, lightning and even the glow of neon signs, plasmas are everywhere. Better understanding of them is key to our understanding of all these phenomena. Plasmas are fluids of conducting particles that create and interact with electromagnetic fields. The strong interaction between the plasma and electromagnetic fields means that they are significantly more complicated to understand and model than ordinary liquids and gasses. There are many mathematical models used to study plasmas, but the most well-known of them is Magnetohydrodynamics, often shortened to MHD.

MHD is a set of equations describing the mechanics of electrically conducting fluids such as plasmas or liquid metals. As opposed to particle-based "kinetic" plasma models, MHD is a fluid-flow model that ignores individual particle behavior, instead focusing on the fluid in aggregate. It is similar to the famous Navier-Stokes Equations for fluid flow, but with additional terms for electromagnetic effects. Though less complicated than other plasma models such as Vlasov or the two-fluid model, MHD models are especially well-suited to situations where magnetic forces confine the plasma, such as our usecase in tokamak fusion reactors [Bellan, 2006].

2.2. Tokamak Dynamics

The leading candidate for fusion power generation is the tokamak design, which is a magnetic confinement device designed to confine a plasma in the shape of a torus. The basic idea behind magnetic-confinement fusion is to confine the charged particles of a plasma at high temperature for the duration necessary to have a high fusion probablility before a particle escapes the system. To achieve this, a strong magnetic field is applied to the plasma. Charged particles traveling in a magnetic field will curve and rotate around magnetic field lines due to the Lorentz force that acts perpendicular to both the magnetic field and the particle's direction of motion. The stronger the field, the tighter the radius of these circles (known as the "cyclotron radius").[Bellan, 2006, Freidberg et al., 2015]. Early plasma confinement devices were linear, but since linear confinements did not prevent particles from escaping out of the ends of the device, the linear configuration was abandoned in favor of a closed torus. For topological reasons (proven by Poincaré in 1885), a vector field such as a magnetic field cannot "comb" a sphere flat without points of 0-field at the poles. So in order to prevent particles escaping at the poles, a toroidal design is used.

In a simple toroidal magnetic field, the lines circle the device, which keeps charged particles somewhat confined. Since magnetic field lines always connect back to each other, the circular geometry of the device means that the magnetic field will be stronger towards the inside of the torus and weaker on the outside. This gradient in the magnetic field causes charged particles to slowly drift laterally, leaving the device. In the tokamak design, currents are added that twist the magnetic field lines helically around the torus, which leads the particle drift to cancel itself out[Wesson, 1999].

2.3. MHD Derivation

When neglecting certain particle effects and collisionality, plasmas can be described as a fluid. Therefore, to start the derivation, we can start with physical properties of fluids. The easiest place to start is with quantities that will be conserved: mass, momentum, energy, and magnetic flux.

Plasmas conserve mass, so the amount of mass within a volume can only change due to a flux of the mass through the volume. This leads us to the continuity equation:

$$\partial_t \rho + \nabla \cdot (\rho \mathbf{V}) = 0 \tag{2.1}$$

Where $\mathbf{V}$ is fluid flow and ρ is mass density.

Particles in a plasma also conserve momentum, so we adopt the momentum balance equation from Navier Stokes, but with an additional $\mathbf{J} \times \mathbf{B}$ component from the Lorentz force. Here, p is pressure:

$$\rho \partial_t \mathbf{V} = \mathbf{J} \times \mathbf{B} - \nabla p - \mathbf{V} \cdot \nabla \mathbf{V} \tag{2.2}$$

The current comes from moving current, which itself induces a magnetic field

$$\rho \mathbf{V} = \mathbf{J} = \frac{1}{\mu_0} \nabla \times \mathbf{B} \tag{2.3}$$

Taking the cross product of the current and the magnetic field yields

$$\begin{aligned} \mathbf{J} \times \mathbf{B} &= \frac{1}{\mu_0} (\nabla \times \mathbf{B}) \times \mathbf{B} \\ &= -\frac{1}{2\mu_0} \nabla \mathbf{B}^2 + \frac{1}{\mu_0} (\mathbf{B} \cdot \nabla) \mathbf{B} \end{aligned} \tag{2.4}$$

From here, the similarities to fluid equations start to diverge. Charged particles are influenced by the Lorentz force,

$$\mathbf{F} = \rho_e \mathbf{E} + \rho_e \mathbf{V} \times \mathbf{B} \tag{2.5}$$

Where ρ_i is the ion charge density, ρ_e is the electron charge density, and ρ is the total charge density. Similarly for pressure:

$$\begin{aligned} \rho &= \rho_i - \rho_e \\ p &= p_i + p_e \end{aligned} \tag{2.6}$$

From the ideal gas law, pressure relates to density and pressure as $p = \rho T$ with a constant term (μ_0) that drops out due to normalization.

In the stationary case where pressure is balanced by electric current, what follows is a generalized Ohm's law, where η is the electrical resistivity [Krebs, 2012].

$$\mathbf{E} + \mathbf{V} \times \mathbf{B} = \eta \mathbf{J} + \frac{1}{en} (\mathbf{J} \times \mathbf{B} - \nabla p) - \frac{m_e}{e} \frac{d\mathbf{u}}{dt} \tag{2.7}$$

Additional terms come from Maxwell's Equations

$$\begin{cases} \nabla \cdot \mathbf{E} = \rho / \epsilon_0 \\ \nabla \times \mathbf{E} = -\partial_t \mathbf{B} \\ \nabla \times \mathbf{B} = \mu_0 (\mathbf{J} + \epsilon_0 \partial_t \mathbf{E}) \\ \nabla \cdot \mathbf{B} = 0 \end{cases} \tag{2.8}$$

One can take certain assumptions if one is only interested in phenomona larger than certain plasma phenomena such as the Debye length (the length scale at which electric fields are screened) and slower than certain frequencies such as the electron cyclotron and plasma frequencies (respectively related to the speed at which electrons orbit magnetic field lines and screen out electric fields). For this "Ideal MHD" model, one makes a number of assumptions. If one assumes that the plasma is quasi-neutral on macroscopic scales and that electrons can quickly displace to balance out charge inequalities, Gauss's Law can be ignored. The resistivity term η drops out, as does the whole right side of the generalized Ohm's law, ending up with $\mathbf{E} + \mathbf{V} \times \mathbf{B} = 0$. Since electrons shield electric fields, the $\partial_t \mathbf{E}$ term of Ampere's law disappears.

For normalization, we normalize with respect to the central mass density ρ_0, and the vacuum permeability μ_0. Time is normalized time according to the Alfvén time ($\tau_A = a\sqrt{\mu_0\rho_0}/B_0$), which is the time needed for an Alfvén wave to travel one radian toroidally.

This is a simplified model, and real-world applications need additional extensions, such as finite resistivity, anisotropic heat transport, or two-fluid effects where the electrons and the ions have different properties.

2.4. Reduced MHD

Starting from the ideal MHD formulation above, we re-add some resistivity to the current:

$$\mathbf{E} + \mathbf{V} \times \mathbf{B} = \eta \mathbf{J} \tag{2.9}$$

We then add heat conductivity (κ), viscosity (μ), diffusivity (D), and source terms (S_H, S_ρ for heat and particle sources) to the continuity, momentum, and energy equations:

$$\begin{aligned} &\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \boldsymbol{v}) = \nabla \cdot (D \nabla \rho) + S_\rho, \\ &\rho \frac{d\boldsymbol{v}}{dt} = \boldsymbol{J} \times \boldsymbol{B} - \nabla p + \mu \nabla^2 \boldsymbol{v}, \\ &\frac{d}{dt}\left(\frac{p}{\rho^\gamma}\right) = \nabla \cdot (\kappa \nabla p) + S_H, \end{aligned} \tag{2.10}$$

The reduced MHD model is a subset of this "resistive and diffusive" MHD model and is designed to reduce the computation costs of the model. It does so by making assumptions that are reasonable for a tokamak configuration. Since it is designed for tokamak plasmas, it uses a cylindrical coordinate system R, Z, ϕ, along with an associated poloidal coordinate system r, θ.

We start by converting the above equations to cylindrical coordinates, with

$$\begin{cases} X = R\cos\phi \\ Y = -R\sin\phi \\ Z = Z \end{cases} \tag{2.11}$$

Since $\nabla \cdot \mathbf{B} = 0$, we can break the magnetic field into poloidal and toroidal components $\mathbf{B} = \mathbf{B}_\phi + \mathbf{B}_\theta$. We make the following assumptions: The magnetic field is dominated by its toroidal component, and the poloidal component is relatively weak ($\mathbf{B}_\phi >> \mathbf{B}_\theta$). Additionally, the toroidal component of the magnetic field is assumed to be constant in time. The ansatz for the magnetic field is:

$$\mathbf{B} = \frac{F_0}{R}\mathbf{e}_\phi + \frac{1}{R}\nabla\psi \times \mathbf{e}_\phi \tag{2.12}$$

where F_0 is a constant, and $\mathbf{e}_\phi$ is the toroidal basis vector. ψ is the poloidal magnetic flux.

These assumptions eliminate "fast magnetosonic waves," which are the fastest waves in the system. This allows for larger timesteps as the timestep size depends on the shortest relevant timescales. Additionally, there are fewer unknowns to compute and store[Hoelzl et al., 2021].

Eventually, (see [Franck, Emmanuel et al., 2015] or [Pamela, 2010] for the full derivation) this comes out to:

$$
\begin{aligned}
\frac{\partial \rho}{\partial t} =& -\nabla \cdot (\rho \boldsymbol{v}) + \nabla \left(D_\perp \nabla_\perp \rho\right) + S_\rho \\
R\nabla \cdot \left[R^2 \rho \nabla_\perp \left(\frac{\partial u}{\partial t}\right)\right] =& \left[R^4 \rho W, u\right] - \frac{1}{2}\left[R^2 \rho, R^4 \left|\nabla_\perp u\right|^2\right] \\
& - \left[R^2, p\right] + [\psi, j] - \frac{F_0}{R}\frac{\partial j}{\partial \phi} + \mu R \nabla^2 W \\
\rho F_0^2 \frac{dv_\parallel}{dt} =& F_0 \frac{\partial p}{\partial \phi} - R[\psi, p] + \mu_\parallel \nabla^2 v_\parallel \\
\rho \frac{\partial T}{\partial t} =& -\rho \boldsymbol{v} \cdot \nabla T - (\gamma - 1) p \nabla \cdot \boldsymbol{v} + \nabla \cdot \left(\kappa_\perp \nabla_\perp T + \kappa_\parallel \nabla_\parallel T\right) + S_T \\
\frac{\partial \psi}{\partial t} =& \eta\left(j - j_A\right) + R[\psi, u] - \frac{\partial u}{\partial \phi}
\end{aligned}
\tag{2.13}
$$

where the velocity $\boldsymbol{v}$ and the toroidal vorticity W , as well as the magnetic field $\mathbf{B}$ and the toroidal current j are defined, respectively, by

$$
\begin{aligned}
\boldsymbol{v} &= \boldsymbol{v}_\parallel + \boldsymbol{v}_\perp = v_\parallel \boldsymbol{B} + R^2 \nabla\phi \times \nabla u \\
W &= \nabla\phi \cdot (\nabla \times \boldsymbol{v}_\perp) = \nabla_\perp^2 u \\
\boldsymbol{B} &= F_0 \nabla\phi + \nabla\psi \times \nabla\phi \\
j &= -R^2 \nabla\phi \cdot \boldsymbol{J} = \frac{1}{\mu_0}\Delta^* \psi
\end{aligned}
\tag{2.14}
$$

where Δ^* is the Grad-Shafranov operator ($\Delta^*\psi = \nabla^2\psi + 2\mathbf{B}_\theta$), u is the electric potential, and pressure is defined as $p = \rho T$. Note that Poisson brackets have been used, with the definition $[a, b] = \boldsymbol{e}_\phi \cdot (\nabla a \times \nabla b)$.

2.5. Model Extensions

JOREK can also execute a full MHD model. However, this is not considered here as the reduced model captures key physics very well under most conditions for less computation cost.

JOREK can extend its model to include additional effects such as relativistic electrons, impurities, a neutral gas, or kinetic particles. One can separate certain concepts between electrons and ions such as their temperatures in order to form a "two-fluid" model.

3

Finite Element Methods

Finite Element Methods (FEM) are a technique for solving partial differential equations numerically. The basic idea is to convert an infinitely-dimensional partial differential equation into a linear problem that can be solved with numerical linear algebra techniques.

The FEM technique involves subdividing a continuous system into a finite set of smaller components using a mesh, and then transforming the differential problem on these sub-elements into an algebraic equation that can be solved numerically. The process is essentially:

1. Establish the strong form of the problem
2. Establish a weak form of the problem
3. Break the problem into a finite set of elements, made from compact polynomials
4. Establish the weights of those polynomial elements.

[Niels Saabye Ottosen, 1992]

3.1. Background and Derivation

The process underlying FEM is to first cast the problem up into its "weak formulation," which allows us to use linear algebra to solve arbitrary partial differential equations.

Given a system to solve

$$Au = f \tag{3.1}$$

finding the solution $u \in V$ is equivalent to finding $u \in V$ such that for all "test functions" $v \in V$,

$$(Au)(v) = f(v) \tag{3.2}$$

Then approximating the weak form of the problem with a finite-dimensional problem by replacing the subspace V of the weak form with a subspace of functions of small, compact, low-degree polynomial "elements" over the domain. Then selecting a subspace V in L^2 and putting the problem in its bilinear Galerkin form [Saad, 1996, Hughes, 1987, Landstorfer et al., 2021]:

From there,

$$\text{Find } u \in V \text{such that } a(u, v) = F(v),\ \forall\ v \in V \tag{3.3}$$

Our computers can only solve finite-dimensional problems, so we perform a dimension reduction:

$$\text{Find } u_n \in V_n \text{ such that } a(u_n, v_n) = F(v_n),\ \forall\ v_n \in V_n \tag{3.4}$$

This is called the Galerkin equation, and it is a projection of (3.3) onto V_n. From our finite dimensional problem, we now extract a linear system of equations. Since V_n is finite-dimensional, there exists a basis $(\phi_1, ..., \phi_n)$ in V_n that can construct our solution u_n:

$$u_n = \sum_{i=1}^{n} \alpha_i \phi_i \tag{3.5}$$

Due to bilinearity,

$$a(u_n, v_n) = \sum_{i=1}^{n} \alpha_i a(\varphi_i, v_n) \overset{v_n=\varphi_j}{\Longrightarrow} a\left(u_n, \varphi_j\right) = \sum_{i=1}^{n} \alpha_i a\left(\varphi_i, \varphi_j\right). \tag{3.6}$$

.

This allows us to translate the problem into a linear system we can solve with numerical linear algebra techniques:

$$A_n \alpha = \begin{bmatrix} a(\varphi_1, \varphi_1) & \cdots & a(\varphi_n, \varphi_1) \\ \vdots & \ddots & \vdots \\ a(\varphi_1, \varphi_n) & \cdots & a(\varphi_n, \varphi_n) \end{bmatrix} \begin{bmatrix} \alpha_1 \\ \vdots \\ \alpha_n \end{bmatrix} = \begin{bmatrix} F(\varphi_1) \\ \vdots \\ F(\varphi_n) \end{bmatrix} \tag{3.7}$$

3.2. Bézier Surfaces

For the finite element method, we need a test function that satisfies certain properties. It needs to be compact, The first version of the JOREK code used "generalized h-p refinable finite elements"[Hoelzl et al., 2021], but in practice, mesh refinement was impractical. For the second version of JOREK, a new finite element formulation was proposed and implemented that was based on G^1 continuous 2D isoparametric cubic Bézier surfaces[Hoelzl et al., 2021].

Bézier surfaces were developed in the 1960s by Paul Bézier to design automobile bodies [Czarny and Huysmans, 2008]. They are based on interpolating Bernstein polynomials developed by Sergei Bernstein based on the following formulae:[Czarny and Huysmans, 2008]

$$\begin{cases} B_i^n(s) = C_n^i s^i (1-s)^{n-i}, \\ C_n^i = \frac{n!}{i!(n-i)!}, \\ 0 \le i \le n \end{cases} \tag{3.8}$$

This gives us a set of polynomials with some useful properties:

1. B_i^n is a basis of P^n, the set of polynomials of degree $\le n$.
2. $0 \le B_i^n(s) \le 1, \forall s \in [0;1]$.
3. $\sum_{i=0}^{n} B_i^n(s) = 1, \forall s \in [0;1]$.

In our case, we are using cubic Bézier surfaces, which are polynomials of degree 3 (shown plotted in Figure 3.1) [Czarny and Huysmans, 2008]:

$$\begin{cases} B_0^3(s) = (1-s)^3, \\ B_1^3(s) = 3s(1-s)^2, \\ B_2^3(s) = 3s^2(1-s), \\ B_3^3(s) = s^3. \end{cases} \tag{3.9}$$

These can be extended into rectangular patches [Czarny and Huysmans, 2008], and used as a basis for our FEM model:

$$P(s,t) = \sum_{i=0}^{3} \sum_{j=0}^{3} P_{i,j} B_i^3(s) B_j^3(t), \ 0 \le s,t \le 1 \tag{3.10}$$

Here, s and t are local coordinates where $0 \le s,t \le 1$, and $P_{i,j}$ are the 16 control points for the surface. Of these, four correspond to the corners of the patch, and the rest correspond to the tangents

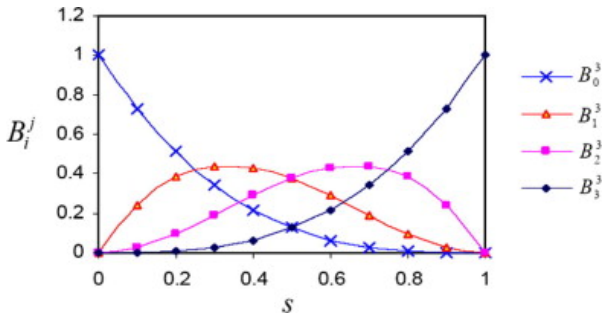


Figure 3.1: Cubic Bézier polynomials [Czarny and Huysmans, 2008].

($\frac{\partial P}{\partial s}$, $\frac{\partial P}{\partial t}$) and the cross derivatives ($\frac{\partial^2 P}{\partial t \partial s}$) at the corners. The patches can be organized (as in JOREK) into an unstructured mesh. In our formulation, Bézier patches are G^1 continuous, meaning that where two patches share a common edge, they also share a common angle or tangent at that edge.

Bézier surfaces were chosen for JOREK because of several properties. They require only four degrees of freedom per node, which is an advantage over Lagrangian formulations. They react well to mesh-refinement (unlike pure Hermite formulations or the JOREK I formulation). Bézier surfaces can also be aligned well with the magnetic-fields present, which is advantageous as the physics parallel to the magnetic field differ from the physics perpendicular to the fields [Czarny and Huysmans, 2008, Krebs, 2012].

Bézier surfaces are used to construct the basis for the weak formulation of equations used in Finite Elements. This treatment is discussed later, in section 5.2.

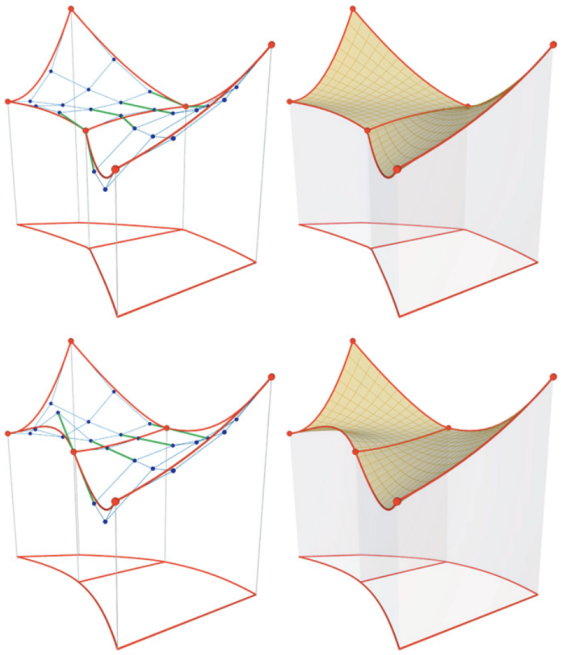


Figure 3.2: Example Bézier surfaces, showing control points on the left and the resulting surface on the right. The figures at the top are G^0 continuous, while the figures at the bottom are G^1 continuous (as used by JOREK) [Hoelzl et al., 2021].

4

Numerical Solution Methods

FEM simulations generate linear systems with massive number of terms. A whole field has arisen to solve these systems quickly and efficiently, and there are now many popular techniques, that have advantages and disadvantages in certain cases.

MHD problems produce very large linear systems, due to the multitude of important plasma physics that have effects over many orders-of-magnitude in both time and space. For example, typical fusion plasma dimensions are of the order of a few meters, but the resistive skin depth of the plasma is typically on the order of sub-millimeters, leading to a scale separation of four orders of magnitude [Hoelzl et al., 2021]. This kind of dynamic forces the simulation of large volumes to use a relatively very fine mesh. As a result, this can produce very large linear systems that can be difficult to solve. For the solution to have good performance and stability, a thoughtful application of numerical linear algebra techniques is required.

Solving a system $\mathbf{A}x = b$, where nonsingular $\mathbf{A} \in \mathbb{C}^{n\times n}$, takes on the order of n^3 operations if done naively. Taking advantage of certain features of the problem, one can speed this up considerably. With about one in 3000 entries nonzero, the system that JOREK solves is highly sparse. This sparsity makes it a good candidate for certain numerical techniques.

4.1. Condition Number

Relevant for discussing the numerical behavior for most algorithms is the "Condition number" $\kappa_p(\mathbf{A})$. Given $y = f(x)$, the condition number represents the error Δy induced by a small error Δx in x. This is very important for numerical solvers, as many operations will be needed to solve a large system, and every floating point operation will carry with it a small error due to the maximum machine precision.

For a function generally, the condition number looks like [Vuik and Lahaye, 2023, Liesen, 2020]

$$\kappa_f(x) = \frac{||f(x+\Delta x) - f(x)||_Y}{||f(x)||_Y} \frac{||x||_X}{||\Delta x||_X} \tag{4.1}$$

For a differentiable function f, this rearranges to form

$$\kappa_f(x) = ||f'(x)|| \frac{||x||}{||f(x)||} \tag{4.2}$$

For a linear system $f(x) = \mathbf{A}x$ and a norm $||\cdot||$, this comes out to

$$\kappa_f(x) = ||\mathbf{A}|| \frac{||x||}{||\mathbf{A}x||} \tag{4.3}$$

If A is nonsingular, this gives the bound

$$\kappa_f(x) \leq ||\mathbf{A}^{-1}|| \cdot ||\mathbf{A}|| \tag{4.4}$$

Thus we define the condition number for a matrix $\mathbf{A}$ and p-norm as

$$\kappa_p(\mathbf{A}) = ||\mathbf{A}||_p||\mathbf{A}^{-1}||_p \tag{4.5}$$

For the 2-norm, this is equivalent to

$$\kappa_2(\mathbf{A}) = \frac{\sqrt{\lambda_{max}(\mathbf{A}^T\mathbf{A})}}{\sqrt{(\lambda_{min}(\mathbf{A}^T\mathbf{A})}} \tag{4.6}$$

thus showing one way the spectrum of a problem can impact numerical behavior.

As an example, consider the following linear problem from [Vuik and Lahaye, 2023]

$$\begin{bmatrix} 1 & 1 \\ 1 & .999 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 2 \\ 1.999 \end{bmatrix} \tag{4.7}$$

It is nearly rank-deficient, meaning the smallest eigenvalue is very close to 0. Moreover, if there is inaccuracy in our system and the right-hand side is changed to $(2\ 2)^T$, the solution changes drastically from $(x\ y)^T = (1\ 1)^T$ to $(x\ y)^T = (2\ 0)^T$.

4.2. Direct Methods

The basic idea behind direct methods is to decompose the problem into a simpler subproblem. For example, the system

$$\mathbf{A}x = b \tag{4.8}$$

can be decomposed into its lower and upper triangular components

$$\mathbf{A} = \mathbf{LU} \tag{4.9}$$

which turns the problem into

$$\mathbf{LU}x = b \tag{4.10}$$

yielding two simpler problems with triangular matrices that are therefore easy to solve

$$\mathbf{L}y = b,\ \mathbf{U}x = y \tag{4.11}$$

Other direct methods may use different decompositions, but the basic idea is the same.

A disadvantage with direct methods is that until the algorithm is finished, one does not have any partial solution and the system cannot be analyzed prematurely for information on the state of the solution. They can also have high memory consumption for sparse problems due to fill-in, such as the sparse systems solved by JOREK. JOREK uses direct solvers only for small problems and to solve blocks of its preconditioner.

4.3. Iterative Methods

As opposed to direct methods, iterative methods progressively approximate a solution over numerous iterations. When the approximate solution is satisfactory, the iterations are stopped.

These methods start with some initial guess x_0, and iteratively improve that guess until an acceptable level of accuracy is reached, as measured by the residual $r_k = b - \mathbf{A}x_k$.

These have some improvements over direct solvers. Iterative methods parallelize better, However, iterative methods can fail to converge. Performance can be highly susceptible to the condition number of the system, often requiring preconditioning.

4.4. Krylov subspace methods

Krylov subspace methods are a class of iterative methods that attempt to solve $\mathbf{A}x = b$ by iteratively searching within a limited region of the Krylov subspace of $\mathbf{A}$.

These are a subclass of so-called projection methods, which look for the approximate solution of the form $x_k = x_0 + \mathcal{S}_k$, where $\mathcal{S}_k$ is a k-dimensional subspace called the "search space." With k degrees of

freedom in x_k, we also need k constraints. We impose these on the residual ($r_k = b - Ax_k$) with $r_k \perp C_k$ where C_k is some space that we have taken as our "constraints space."

Krylov subspace methods use the Krylov subspace as the search space $\mathcal{S}_k = \mathcal{K}_n$. Given an initial guess x_0 with an approximate solution x_m. Given $\mathbf{A} \in \mathbb{C}^{nxn}$ and $r_0 \in \mathbb{C}^n$, where the residual $r_0 = b - \mathbf{A}x_0$ [Saad, 1996], then the Krylov subspace is

$$\mathcal{K}_n(\mathbf{A}, r_0) \coloneqq \text{span}\{r_0, \mathbf{A}r_0, \mathbf{A}^2 r_0, ..., \mathbf{A}^{n-1} r_0\} \tag{4.12}$$

It can be shown that an iterated Krylov subspace includes an approximate solution[Bai, 2015]:
Given $x_{x+1} = x_k + \omega(b - \mathbf{A}x_k)$, where ω is a relaxation parameter and $k = 0, 1, 2, ...$, then

$$x_k = (I - \omega\mathbf{A})x_{k-1} + \omega b = \sum_{j=0}^{k-1} (I - \omega\mathbf{A})^j \omega b \in \mathcal{K}_k(\mathbf{A}, b) \tag{4.13}$$

Provided $x_0 = 0$ and the sequence x_k is convergent, then the solution

$$x_* \in \mathcal{K}_\infty(\mathbf{A}, b) \tag{4.14}$$

It can even be shown that the solution exists in the Krylov subspace of degree k, where k is the minimum degree polynomial of $\mathbf{A}$

Let $\phi_k(\mathbf{A})$ be the minimum degree polynomial of $\mathbf{A}$, therefore

$$\phi_k(\mathbf{A}) = \alpha_0\mathbf{I} + \alpha_1\mathbf{A} + ... + \alpha_k\mathbf{A}^k \tag{4.15}$$

Since $\mathbf{A}$ is nonsingular, $\alpha_0 \neq 0$, and therefore

$$\begin{aligned} \mathbf{A}\left(\alpha_1 I + \alpha_2\mathbf{A} + \cdots + \alpha_k\mathbf{A}^{k-1}\right) &= -\alpha_0 I, \\ \therefore \mathbf{A}^{-1} &= -\frac{1}{\alpha_0}\left(\alpha_1 I + \alpha_2\mathbf{A} + \cdots + \alpha_k\mathbf{A}^{k-1}\right) \end{aligned} \tag{4.16}$$

Therefore,

$$\begin{aligned} x_* = \mathbf{A}^{-1}b &= -\frac{1}{\alpha_0}\left(\alpha_1 I + \alpha_2 A + \cdots + \alpha_k\mathbf{A}^{k-1}\right) b \\ &\in \text{span}\left\{b, \mathbf{A}b, \ldots, \mathbf{A}^{k-1}b\right\} = \mathcal{K}_k(\mathbf{A}, b). \end{aligned} \tag{4.17}$$

JOREK makes use of several different Krylov subspace methods in its solver – namely Restarted GMRES and BiCGSTAB, which will be discussed in section 4.7 and section 4.8.

4.5. Preconditioning

Numerical methods to solve linear systems of equations can be made more efficient by transforming the problem into one that is better solved by that given iterative method[Saad, 1996]. This process is called "preconditioning," and is extremely important to iterative methods and numerical linear algebra in general.

For example, given a system $\mathbf{A}x = b$, one wants to find a preconditioner $\mathbf{M}^{-1}$ that is easily invertible and similar to $\mathbf{A}^{-1}$. Then the system $\mathbf{M}^{-1}\mathbf{A}x = \mathbf{M}^{-1}b$ is easy to solve as $\mathbf{M}^{-1}\mathbf{A}$ is close to the identity, and the system has a condition number close to 1. Of course, finding a matrix close to $\mathbf{A}^{-1}$ is not easy, as finding $\mathbf{A}^{-1}$ is essentially the whole problem to solve.

The strategy applies whether you are multiplying the preconditioner from the left or from the right. There is a third type called "split preconditioner" that is preferred for symmetric system. It works by modifying the system as:

$$\mathbf{M} = \mathbf{L}\mathbf{U}, \tag{4.18}$$

$$\mathbf{L}^{-1}\mathbf{A}\mathbf{U}^{-1}u = \mathbf{L}^{-1}b, \quad x = \mathbf{U}^{-1}u. \tag{4.19}$$

Unless dealing with a symmetric system and using a split preconditioner, left and right preconditioners are largely equivalent. However some preconditioners perform better as right preconditioners for Krylov subspace methods, since these methods are based on the norm of the residual of the preconditioned system [Ghai et al., 2016]. If the preconditioner is ill-conditioned, ($\kappa(\mathbf{M}) \gg 1$), then the preconditioned residual may be inflated compared to the true residual, and therefore the algorithm will take additional iterations.

4.6. Arnoldi Method

The Arnoldi method is an algorithm for producing an orthonormal basis of vectors. It is used, particularly in other numerical solvers, because it can be made to produce an orthonormal basis of the Krylov subspace. It was originally made in 1951 in order to reduce a matrix into a quasi-triangular form known as the Hessenberg form[Saad, 1996].

Let $\mathbf{A} \in \mathbb{C}^{n\times n}$ and $v \in \mathbb{C}^n \backslash \{0\}$ be of grade d with respect to $\mathbf{A}$. Then there exists $\mathbf{V} \in \mathbb{C}^{n\times d}$ with orthonormal columns and an unreduced upper Hessenberg matrix $\mathbf{H} = \left[h_{ij}\right] \in \mathbb{C}^{d\times d}$, i.e., $h_{ij} = 0$ for $i > j + 1$ and $h_{i+1,i} \neq 0$ for $i = 1, \ldots, d-1$, such that [Liesen, 2020]

$$\mathbf{AV} = \mathbf{VH} \tag{4.20}$$

The Arnoldi method builds this relation iteratively, via the relation $\mathbf{AV}_m = \mathbf{V}_{m+1}\mathbf{H}_m$, where $\mathbf{V_m} \in \mathbb{C}^{n\times m}$, $\mathbf{H_m} \in \mathbb{C}^{(m+1)\times m}$. One variant of the algorithm is:

Algorithm 1: Arnoldi [Saad, 1996]

Compute: Choose a vector v_1 of norm 1
for $j \in 1, 2, ..., m$ **do**
  Compute: $h_{ij} = (Av_j, v_i)$ for $i = 1, 2, ..., j$
  Compute: $w_j := Av_j - \sum_{i=1}^{j} h_{ij}v_i$
  $h_{j+1}, j = ||w_j||_2$;
  if $h_{j+1}, j0$ **then**
    Stop
  end
  $v_{j+1} = w_j/h_{j+1,j}$;
end

If one does not stop the algorithm before the m-th step, then the vectors $v_1, v_2, ..., v_m$ form an orthonormal basis of the Krylov subspace. This forms the basis for many Krylov subspace solvers such as GMRES.

4.7. GMRES

In 1975, the algorithm MINRES was developed to solve linear systems. It has the disadvantage of only working on symmetric matrices. GMRES is an iterative method based on MINRES, but that allows for nonsymmetric matrices. It uses Arnoldi's method to compute an orthonormal basis of the Krylov subspace.

For the GMRES method, we take constraints space, $C_m = \mathbf{A}\mathcal{K}_m$, so that $\mathbf{A}\mathcal{K}_m \perp b - \mathbf{A}x_m$[Saad, 1996]. Arnoldi's method is used to compute an orthonormal basis of the Krylov subspace.

This gives us the following algorithm:

Algorithm 2: GMRES [Liesen, 2020]

Compute: $r_0 = b - Ax_0$
for $k = 1, 2...$ **do**
  Perform the kth step of Arnoldi to generate $\mathbf{V}_k$ and $\mathbf{H}_{k+1,k}$;
  Update the QR factorization of $\mathbf{H}_{k+1,k}$;
  Compute the updated residual norm $\|r_0\|_2 \left(\mathbf{Q}_{k+1}^{\mathbf{H}} e_1\right)_{k+1}$;
  if *the residual norm is small enough* **then**
    Compute the least squares solution t_k ;
    Return: the approximate solution $x_k = x_0 + \mathbf{V}_k t_k$
  end
end

A major disadvantage of the method is that it needs to store the entire Krylov subspace for every iteration. As the number of iterations grow, the whole subspace must be stored in memory and can become a serious limitation. Note that if **A** is symmetric, our Hessenberg matrix is tridiagonal, and we only need a three-term recurrence, limiting our memory requirements. In general, however, there is a full recurrence and the Arnoldi iterations need increasing memory for every iteration. There are several forms of GMRES that truncate or restart in an attempt to minimize this memory burden [Vuik and Lahaye, 2023, Ghai et al., 2016]. However, these have their own disadvantages and may not converge as well. For some matrices that are poorly conditioned and not positive definite, the restarted GMRES algorithm–the main iterative method used by JOREK–can stagnate indefinitely.

4.8. BiCGSTAB

The JOREK team has recently started incorporating the BiCGSTAB algorithm in addition to GMRES. BiCGSTAB is another Krylov subspace iterative method. BiCGSTAB produces a residual vector of the form $r_j = \psi_j(\mathbf{A})\phi_j(\mathbf{A})r_0$, where ψ_j, ϕ_j are polynomials. ψ_j is defined recursively as $\psi_{j+1}(t) = (1 - \omega_j t)\psi_j(t)$ for some scalar ω_j. BiCGSTAB determines ω_j by minimizing $||r_j||$ with respect to ω_j.[Ghai et al., 2016, Saad, 1996]

Algorithm 3: BiCGSTAB [Saad, 1996]

Compute: $r_0 = b - \mathbf{A}x_0; r_0^* arbitrary;$
Let $p_0 \coloneqq r_0$;
for $j = 0, 1,$ *until convergence* **do**
  $\alpha_j \coloneqq (r_j, r_0^*)/(\mathbf{A}p_j, r_0^*)$;
  $s_j \coloneqq r_j - \alpha_j A p_j$;
  $\omega_j \coloneqq (\mathbf{A}s_j, s_j)/(\mathbf{A}s_j, \mathbf{A}s_j)$;
  $x_{j+1} \coloneqq x_j + \alpha_j p_j + \omega_j s_j$;
  $r_{j+1} \coloneqq s_j - \omega_j \mathbf{A}s_j$;
  $\beta_j \coloneqq \frac{(r_{j+1}, r_0^*)}{(r_j, r_0^*)} \times \frac{\alpha_j}{\omega_j}$;
  $p_{j+1} \coloneqq r_{j+1} + \beta_j(p_j - \omega_j \mathbf{A}p_j)$;
end

BiCGSTAB uses less memory than GMRES, but is less stable. Due to memory issues, it is often be an improvement over restarted GMRES[Ghai et al., 2016, Shadid and Tuminaro, 1994]. It is not as performant as GMRES in a well-preconditioned system. Without an effective preconditioner, though, restarted GMRES may stagnate or converge slowly. BiCGSTAB is also more well-suited than GMRES to handle non-symmetric matrices.

One downside is that the residual does not decrease monotonically, but variants such as QMRCGSTAB exist to handle this problem, at some computational cost.

4.9. Convergence

Several diagnostic tools are available to numerical analysts to determine convergence properties of the linear systems involved. These depend on properties of the underlying linear system and can be analyzed using numerical linear algebra techniques.

As a rule of thumb, one often looks at the condition number of an invertible matrix $\mathbf{A}$, which in the $p-$norm is defined

$$\kappa(\mathbf{A}) = ||\mathbf{A}||_p ||\mathbf{A}^{-1}||_p. \tag{4.21}$$

In case $\mathbf{A}$ is symmetric and positive definite (SPD), the upper expression in the 2-norm reduces to

$$\kappa_2(\mathbf{A}) = \frac{\lambda_{\max}(\mathbf{A})}{\lambda_{\min}(\mathbf{A})}, \tag{4.22}$$

where $\lambda_{\max}$ and $\lambda_{\min}$ denote the respective largest and smallest eigenvalue of the matrix $\mathbf{A}$. A widely propagated misconception is that GMRES convergence can be navigated by looking at the condition number. While this may be true for SPD matrices, this is in general not true for nonnormal ($\mathbf{A}\mathbf{A}^H \neq \mathbf{A}^H\mathbf{A}$) and indefinite matrices (matrices having both negative and positive eigenvalues). Here, $\mathbf{A}^H$ represents the complex conjugate of $\mathbf{A}$.

In general, for normal matrices, the distribution and clustering of the eigenvalues determine the convergence speed of Krylov subspace methods, in particular GMRES. If the eigenvalues are clustered near the point $(1, 0)$ in the complex plane, we generally expect fast convergence. For nonnormal matrices however, this may not be the case [Liesen and Tichý, 2004, Liesen, 2020]. Some numerical evidence has been gathered over the years to suggest that spectral analysis may still provide some notions which could outline convergence behavior [Dwarka and Vuik, 2020].

Especially for fusion simulations due to the complexity of the underlying mathematical operators, inadequate conditioning can be misleading in assessing what preconditioning strategies will perform better. Consequently, in this work we focus on unraveling these underlying mathematical properties to interpret the convergence behavior of the current solver in order to work towards acceleration strategies. For example, indefinite nonsymmetric matrices also arise in wave propagation problems, and the respective Krylov based solvers often show acceleration using projection and multigrid techniques (for an overview of examples and literature, see [Dwarka and Vuik, 2020, Dwarka and Vuik, 2022]).

4.9.1. GMRES Convergence

If the matrix is normal (that is to say $\mathbf{A}\mathbf{A}^H = \mathbf{A}^H\mathbf{A}$), then the worst-case convergence behavior is determined by the spectrum. If the matrix is nonnormal, the convergence behavior is not solely determined by the spectrum and other heuristics may be necessary for a complete analysis. According to [Liesen and Tichý, 2004], GMRES convergence analysis for nonnormal matrices remains an open problem.

For our analysis, note that any vector in the Krylov subspace can be written in polynomial form where p_n is a polynomial of degree $\leq n$. This means that the error $(x - x_n)$ and the residual (r_n) can be written as

$$x - x_n = p_n(\mathbf{A})(x - x_0), \quad r_n = p_n(\mathbf{A})r_0 \tag{4.23}$$

GMRES convergence is analyzed via its residual, $x_k \in x_0 + \mathcal{K}_k(A, r_0)$

$$\|r_k\|_2 = \|b - \mathbf{A}x_k\|_2 = \min_{z \in x_0 + \mathcal{K}_k(\mathbf{A}, r_0)} \|b - \mathbf{A}z\|_2 \tag{4.24}$$

In the case that our matrix is diagonalizable, $\mathbf{A} = \mathbf{X}\mathbf{\Lambda}\mathbf{X}^{-1}$. Our residual is bounded as:

$$\begin{aligned}
\|r_k\|_2 &= \min_{z \in \mathcal{K}_k(\mathbf{A}, r_0)} \|r_0 - \mathbf{A}z\|_2 \\
&= \min_{p \in \mathcal{P}_k(0)} \|p(\mathbf{A})r_0\|_2 \\
&\leq \min_{p \in \mathcal{P}_k(0)} \|p(\mathbf{A})\|_2 \|r_0\|_2 \\
&= \min_{p \in \mathcal{P}_k(0)} \left\|\mathbf{X}p(\mathbf{\Lambda})\mathbf{X}^{-1}\right\|_2 \|r_0\|_2 \\
&\leq \kappa(\mathbf{X}) \min_{p \in \mathcal{P}_k(0)} \|p(\mathbf{\Lambda})\|_2 \|r_0\|_2 \\
&= \kappa(\mathbf{X}) \|r_0\|_2 \min_{p \in \mathcal{P}_k(0)} \max_{1 \leq i \leq n} |p(\lambda_i)|
\end{aligned} \tag{4.25}$$

$$\therefore \quad \frac{\|r_k\|_2}{\|r_0\|_2} \leq \kappa(\mathbf{X}) \min_{p \in \mathcal{P}_k(0)} \max_{1 \leq i \leq n} |p(\lambda_i)| \tag{4.26}$$

For normal matrices, we can choose eigenvalues such that $\kappa(X) = 1$[Liesen, 2020, Liesen and Tichý, 2004]. In the case that the spectrum is contained within a disk in the complex plane that does not include the origin, say centered at $c \in \mathbb{C}$ and of radius r, such that $r < |c|$, then we can use the polynomial $p(z) = (1 - z/c)^k \in \mathcal{P}_k(0)$ and our bound becomes

$$\min_{p \in \pi_n} \max_k |p(\lambda_k)| \leq \left|\frac{r}{c}\right|^n \tag{4.27}$$

When the eigenvalues are tightly clustered and distant from the origin, then GMRES will converge quickly. However, if the opposite is true and the eigenvalues are not tightly clustered or close to the origin, this does not necessarily mean that GMRES converges slowly[Liesen, 2020].

Further refinement is possible in certain cases, such as for example, the symmetric part of A is positive definite.[Zou, 2023]

Overall, we end up with a bounds defined by a "worst-case" value $\psi_n(\mathbf{A})$ and what is called the "ideal" value $\phi_n(\mathbf{A})$:[Zou, 2023].

$$\frac{\|r_n\|}{\|r_0\|} \leq \psi_n(\mathbf{A}) \leq \phi_n(\mathbf{A}), \quad \psi_n(\mathbf{A}) = \max_{\|v\|=1} \min_{p \in \mathcal{P}_n} \|p(\mathbf{A})v\|, \quad \phi_n(\mathbf{A}) = \min_{p \in \mathcal{P}_n} \|p(\mathbf{A})\|. \tag{4.28}$$

The worst case is defined such that there exists an initial vector v such that $\|r_n\| = \psi_n(\mathbf{A})$. If $\mathbf{A}$ is normal, the ideal case $\phi_n(\mathbf{A}) = \psi_n(\mathbf{A})$.[Zou, 2023]

For GMRES, left, right, and split preconditioners can all be used. Unless $\mathbf{A}$ is highly symmetric or $\mathbf{M}$ is ill-conditioned, there is little difference between these forms in theory. The operators they produce ($\mathbf{M}^{-1}\mathbf{A}$, $\mathbf{A}\mathbf{M}^{-1}$, and $\mathbf{L}^{-1}\mathbf{A}\mathbf{U}^{-1}$) all have the same spectra, but as discussed, GMRES convergence is not purely due to spectrum.

The approximate solution obtained for GMRES by left or right conditioning is

$$x_m = x_0 + s_{m-1}(\mathbf{M}^{-1}\mathbf{A})z_0 = x_0 + \mathbf{M}^{-1}s_{m-1}(\mathbf{A}\mathbf{M}^{-1})r_0 \tag{4.29}$$

where $z_0 = \mathbf{M}^{-1}r_0$ and s_{m-1} is a polynomial of degree m that minimizes the residual norm $||b - \mathbf{A}x_m||_2$ if preconditioned from the right, and $||\mathbf{M}^{-1}(b - \mathbf{A}x_m)||_2$ if preconditioned from the left. As discussed above and in [Ghai et al., 2016], right preconditioning is generally preferred for Krylov subspace methods as the preconditioned residual can be enlarged if preconditioned from the left.

In practice, convergence in GMRES is not strictly consistent. It can pass through "superlinear" regions where the convergence speed improves per iteration, or can stagnate almost completely. While GMRES on an NxN system will always converge in at most N steps [Saad and Schultz, 1986] , the residual may flatten completely and only converge at the final step. In these cases, a restarted GMRES strategy will never converge. See [Zavorin et al., 2003] for certain conditions where this will occur, but the criteria are fairly complicated and the analysis is beyond the scope of this work.

4.9.2. BiCGSTAB Convergence

BiCGSTAB has an occasional tendency to breakdown or fail to converge, and some properties of our system can predict such cases.

The initial choice of x_0 is important with BiCGSTAB. A poor choice of r_0 can lead to a small iteration parameter $\alpha_i = (r_j, r_0)$ and therefore small or no improvement per iteration, and the algorithm may not converge. In that case, one must restart with a new r_0 or use another algorithm such as GMRES.

BiCGSTAB can converge poorly if for many consecutive steps, the angle between the residual r and $\mathbf{A}r$ is larger than 45 degrees. Additionally, no breakdown or near-breakdown of convergence will occur if the angle between the Krylov subspace and the "shadow" Krylov subspace ($\tilde{\mathcal{K}}_k := \mathcal{K}_k(A^T, r_0)$) is always less than $\pi/2$. In particular:

$$\inf_k \sup \left\{ \frac{|(r_k, \tilde{v})|}{\|r_k\| \, \|\tilde{v}\|} \mid \tilde{v} \in \tilde{\mathcal{K}}_{k+1} \right\} > 0 \tag{4.30}$$

4.10. Preconditioners for MHD Models

There are a variety of different viable strategies for constructing preconditioners, and similarly a large variety of MHD models and variants with different assumptions or extensions. There is some literature investigating preconditioner strategies for different MHD variants or formalisms [Laakmann, 2022], but much work remains to be done. Some of the more common preconditioner strategies used for MHD are as follows:

One approach is with multigrid methods, which creates a preconditioner based on solving a coarsened grid. However, these are known to only work well for systems with low Reynold's numbers and coupling [Laakmann, 2022].

The most common class of preconditioner used for MHD is the block preconditioner. Where the physics of the problem leads to a system of weakly coupled sub-systems that can be separated into blocks, these blocks can be solved separately for less computational effort than solving the whole larger system directly. If the blocks aren't too strongly coupled, then this is a good approximation of the solution and therefore a good preconditioner. These have been used in incompressible MHD models [Ma et al., 2016], and for the stationary MHD problem, for example[Laakmann, 2022] by breaking it into hydrodynamic (Navier-Stokes) and electromagnetic (Maxwell's) blocks and then using a multigrid to precondition these blocks separately . The JOREK code uses a block preconditioner that will be discussed in detail in section 5.4.

Another approach is to use an augmented Lagrangian preconditioner. By finding approximate solutions to a weaker and constrained form of the problem that is known as the augmented Lagrangian. This approximate solution can then be used to precondition our original problem. These have found some applicability Navier-Stokes models as well as incompressible, resistive Hall MHD models, highly coupled and high Reynolds Number plasmas, or anisothermal MHD models [Ma et al., 2016, Laakmann, 2022, Laakmann et al., 2022].

According to [Laakmann, 2022], no practical robust preconditioner yet exists for the problem in general. Certain subclasses of the problem are especially difficult, such as stationary MHD for highly coupled systems with high Reynold's numbers. It is unclear to what extent these different preconditioners will apply to the reduced MHD model and discretization scheme used by JOREK.

4.10.1. Time Discretization

Time integrating an equation of the form

$$\frac{\partial \mathbf{A}(\mathbf{u})}{\partial t} = \mathbf{B}(\mathbf{u}, t) \tag{4.31}$$

can be discretized in general as

$$\begin{aligned} &(1+\xi)\mathbf{A}^{n+1} - (1+2\xi)\mathbf{A}^{n} + \xi\mathbf{A}^{n-1} \\ &\quad = \Delta t \left[\theta \mathbf{B}^{n+1} + (1-\theta-\phi)\mathbf{B}^{n} - \phi \mathbf{B}^{n-1}\right] \end{aligned} \tag{4.32}$$

This is accurate to second order wherever $\phi + \theta - \xi = \frac{1}{2}$. Taking $\phi = 0$ and linearizing (with $\delta\mathbf{u^n} = \mathbf{u^{n+1}} - \mathbf{u^n}$, where n refers to the timestep) gives the equation

$$\left[(1+\xi)\left(\frac{\partial \mathbf{A}}{\partial \mathbf{u}}\right)^{n} - \Delta t\,\theta \left(\frac{\partial \mathbf{B}}{\partial \mathbf{u}}\right)^{n}\right] \delta\mathbf{u}^{n} = \Delta t \mathbf{B}^{n} + \xi \left(\frac{\partial \mathbf{A}}{\partial \mathbf{u}}\right)^{n} \delta\mathbf{u}^{n-1} \tag{4.33}$$

For the Crank-Nicolson scheme, $(\theta, \xi) = (1/2, 0)$, for second order BDF2 Gears scheme, $(\theta, \xi) = (1, 1/2)$, and for first order implicit Euler, $(\theta, \xi) = (1, 0)$ [Hoelzl et al., 2021]. After each timestep, we have a rather large linear problem that is then solved.

4.11. Eigenvalue Solvers

Since GMRES analysis convergence behavior is driven by the spectrum, our analysis will necessitate the computation of eigenvalues. Moreover, we will be dealing with very large sparse matrices, so a naive eigenvalue decomposition methods, will not be feasible, and we require methods that take advantage of our sparsity.

4.11.1. Krylov-Schur

The Krylov-Schur algorithm is a method used for finding extremal eigenvalues and corresponding eigenvectors of large, sparse matrices. It uses the Rayleigh-Ritz method to find approximate eigenvalue-eigenvector pairs.

In a typically Arnoldi process, the Ritz pairs converge quickly only in an optimal search direction. In practice, many iterations are needed, with more storage and more computation per iteration. To combat this, the algorithm can be restarted in a new initial search direction based on the computed Ritz vectors.

A Krylov-Schur decomposition is a special type of Krylov decomposition ($\mathbf{AV}_m = \mathbf{V}_m\mathbf{B}_m + v_{m+1}b_{m+1}$ but where the matrix $\mathbf{B}_m$ is in "real Schur form," with 1x1 or 2x2 diagonal blocks [Hernandez et al., 2007].

The algorithm operates by first forming an orthogonal basis for the Krylov subspace, using the Arnoldi process. Then, it performs a Schur decomposition on the resulting Hessenberg matrix. The resulting Krylov-Schur decomposition is reordered and truncated to order p, where $p < m$ is the number of resultant Ritz pairs (approximations of eigenvalue/eigenvector pairs). The truncation is performed based on how the Ritz values meet a specified convergence criterion. Then the subspace is extended, and the algorithm restarts from the second step. [Hernandez et al., 2007]

Our analysis will use this algorithm to find the extreme eigenpairs of our system in a computationally efficient manner.

5

The JOREK Code

JOREK is a code for the simulation of magnetic confinement fusion reactors. Its goal is to model the dynamics of major plasma disruptions and instabilities, so as to control or minimize them for existing reactors or for new reactors such as ITER[Holod et al., 2021]. It uses the Finite Element Method over Bézier surfaces to model plasma physics using a number of different models, such as full, reduced, or extended MHD equations. To simulate toroidal confinement devices, it uses a toroidal Fourier decomposition[Holod et al., 2021].

JOREK is written primarily in Fortran 90/95, with some libraries in C and C++ [Holod et al., 2021]. It is massively paralellelized via MPI and OpenMP, and is designed to be run on a high-performance supercomputing cluster such as Marconi-Fusion [Team, 2023]. The code is developed by an international community including the Max Planck Institute for Plasma Physics (IPP) in Garching, Germany. It consists of more than 250 thousand lines of code, contributed to by dozens of developers. More than 40 scientists world-wide use the code for their research [Hoelzl et al., 2021]. The code is hosted in a private git repository, and access is granted through the ITER organization. Source-code contributions from within the community are encouraged.

5.1. Motivation

JOREK is designed to model large-scale instabilities in tokamak plasmas. These instabilities can disrupt the plasma, thereby either limiting its power, or damaging the walls of the reactor and leading to downtime and expensive repairs [Hoelzl, 2022].

Better modeling of these phenomena allow more robust reactor design that can run at higher power, has higher uptime, and is less susceptible to damage. Active-control measures can also be taken to control disruptions and preserve the plasma or the life of the reactor[Hoelzl, 2022]. JOREK is designed to model the physics of these disruptions and to examine control strategies.[Team, 2023]

5.2. Weak Form of Equations

JOREK takes the MHD equations discussed above (in reduced, full, or extended forms), and discretizes them for FEM with Bézier surfaces as the basis. Our test function comes out of our Bézier basis described in section 3.2:

$$\mathcal{T}^* = B_{i,j}(s,t) S_{i,j} e^{in\phi} \tag{5.1}$$

$B_{i,j}$ is the polynomial Bézier basis, $S_{i,j}$ a scaling factor, $\psi_{i,j,n}$ the coefficients composing the variable ψ, and $e^{in\phi}$ belonging to the Fourier representation, where n refers to the toroidal harmonic[Pamela, 2010]. For the toroidal basis used in JOREK, the s and t variables are taken to be orthogonal to the ϕ basis.

Multiplying our reduced MHD equations (2.13) by $\mathcal{T}^*$ and integrating over the volume gives us our weak formulation, that we can then discretize.

Using $\mathbf{V}$ as our physical variables and $\mathbf{A}$ is the magnetic vector potential, $\nabla \times \mathbf{A} = \mathbf{B}$, a weak form of the MHD problem is be reformulated as: find $(\mathbf{A}, \mathbf{V}, \rho, p)$ in $\mathcal{V}_\mathbf{A} \times \mathcal{V}_\mathbf{V} \times \mathcal{V}_\rho \times \mathcal{V}_p$ such that, for any test functions $(\mathbf{A}^\star, \mathbf{V}^\star, \rho^\star, p^\star)$ in $\mathcal{T}^*_\mathbf{A} \times \mathcal{T}^*_\mathbf{V} \times \mathcal{T}^*_\rho \times \mathcal{T}^*_p$, we have [Hoelzl et al., 2021]:

$$
\begin{aligned}
\int \frac{\partial \mathbf{A}}{\partial t} \cdot \mathbf{A}^{\star} dV &= -\int \mathbf{E} \cdot \mathbf{A}^{\star} dV, \\
\int \rho \frac{\partial \mathbf{V}}{\partial t} \cdot \mathbf{V}^{\star} dV &= -\int \left(\rho \mathbf{V} \cdot \nabla \mathbf{V} + \nabla p - \mathbf{J}\right) dV \times \mathbf{B} - \nabla \cdot \underline{\tau} - \mathbf{S}_{\mathbf{V}} \cdot \mathbf{V}^{\star}, \\
\int \frac{\partial \rho}{\partial t} \rho^{\star} &= -\int \left(\nabla \cdot (\rho \mathbf{V}) - \nabla \cdot (\underline{D} \nabla \rho) - S_{\rho}\right) \rho^{\star} \\
\int \frac{\partial p}{\partial t} p^{\star} &= -\int \left(\mathbf{V} \cdot \nabla p + \gamma p \nabla \cdot \mathbf{V} - \nabla \cdot (\underline{\kappa} \nabla T)\right. \\
&\qquad \left. -(\gamma - 1) \underline{\tau} : \nabla \mathbf{V} - S_{p}\right) p^{\star}
\end{aligned}
\tag{5.2}
$$

We transform these into scalar equations by projecting the vectors onto our basis.

5.2.1. Boundary Conditions

Both Dirichlet and Neumann boundary conditions can be used for all relevant regions. Where the flux is parallel to the boundary, Dirichlet conditions are assumed. Where the flux intersects the boundary, sheath boundary conditions are used. Certain variables, such as the poloidal flux, current density, electric potential, and vorticity are kept fixed.

The boundary temperatures are constrained by the boundary condition for heat flux. This form assumes that electron and ion temperatures are the same, but these can be separated. γ_{sh} is the sheath transmission factor.

$$
\begin{aligned}
\mathbf{q} \cdot \mathbf{n} &\equiv \left(\frac{\rho}{2} \mathbf{V} \cdot \mathbf{V} + \frac{\gamma}{\gamma - 1} \rho T\right) \mathbf{V} \cdot \mathbf{n} - \frac{\underline{\kappa}}{\gamma - 1} \nabla T \cdot \mathbf{n} \\
&= \gamma_{\mathrm{sh}} \rho T_{\mathrm{e}} \mathbf{V} \cdot \mathbf{n},
\end{aligned}
\tag{5.3}
$$

Additional boundary conditions given are given, and are expressed in terms of $\underline{\kappa} \nabla T \cdot \mathbf{n} = -\left(c_{\mathrm{b}} - 1\right) \rho T \mathbf{V} \cdot \mathbf{n}$:

$$
\begin{cases}
c_{\mathrm{b,e}} = (\gamma - 1)\left(\gamma_{\mathrm{sh,e}} - 1\right) \\
c_{\mathrm{b,i}} = (\gamma - 1)\left(\gamma_{\mathrm{sh,i}} - \gamma - 1\right) \\
c_{\mathrm{b,total}} = (\gamma - 1)\left(\gamma_{\mathrm{sh}}/2 - \gamma/2 - 1\right)
\end{cases}
\tag{5.4}
$$

5.3. Model Geometry

The linear system is organized as a hierarchical block tri-diagonal structure. The matrix is organized blockwise, first in the radial direction, then blockwise in the poloidal direction, and then the individual variables.

When organized for the preconditioner (which is based on toroidal harmonics), the matrix is Fourier-decomposed in the toroidal direction, and then the hierarchy is reorganized so that the toroidal harmonics come first. This puts the matrix into a block diagonal structure starting with the toroidal modes (cosin and sin modes). The toroidal mode blocks are different sizes–the n=0 mode block is half the size of the subsequent blocks, since there is no sin component [Hoelzl et al., 2021].

We will be dealing with tokamak models that either do or do not have an x-point geometry. Without an x-point, a tokamak has a circular or elliptical poloidal cross-section, and all magnetic field lines connect. With an x-point, there is a region where field lines connect, a region where field lines are "open," and a point where field lines cross eachother that separates these regions and is known as the separatrix. This is especially significant for our study because it dictates rotational symmetry in the poloidal direction. Without the x-point, our system is cyclical in the toroidal direction, and cyclical in the poloidal direction. With the x-point, the poloidal cyclicity is broken. For our matrix, that means that the systems without an x-point are cyclical in the poloidal direction, but with an x-point this cyclicity is broken

5.4. The JOREK Solver and Preconditioner

For simpler problems that are small or axisymmetric, the system is solved with a direct solver. Usually the PaStiX or STRUMPACK software packages are used, but other solvers are available as well

[Hoelzl et al., 2021].

For more complex systems, JOREK uses a restarted GMRES solver. It must be preconditioned, due to the stiffness and poor-conditioning of the system[Hoelzl et al., 2021]. The preconditioner used is physics and geometry-based, based on the toroidal harmonics that take place in a tokamak structure.

The matrix problem is written in blocks corresponding to toroidal modes. The preconditioner assumes that toroidal modes are decoupled, so diagonal blocks corresponding to self-interaction are kept while off-diagonal blocks (which correspond to coupling between modes) are dropped [Hoelzl et al., 2021].

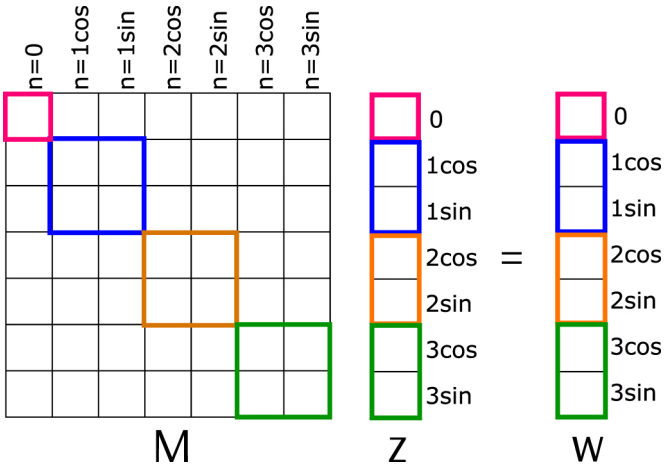


Figure 5.1: From [Holod et al., 2021]: "The matrix structure is shown for a simple example case with the toroidal modes $n = (0, 1, 2, 3)$. The color blocks outline the parts of the original matrix A used to form the preconditioner solution $z = M^{-1} w$. Each block represent individual toroidal Fourier mode."

For linear problems, this is a very effective assumption and the preconditioner behaves well. For highly non-linear problems, the mode-decoupling assumption underlying the preconditioner no longer holds and performance degrades unless a different preconditioner is used.

Current performance problems with the solver are associated with high memory consumption of the factorized preconditioner, poor parallelization of the direct solver used in preconditioning, and poor preconditioner behavior in non-linear cases with strong mode coupling [Hoelzl et al., 2021].

5.5. Recent Preconditioner Improvements

For problems with stronger coupling between modes, a newer preconditioner system is used that breaks the matrix into toroidal "mode groups" that overlap. The preconditioner solves these blocks separately, thus retaining interaction only within these groups. This relaxed assumption requires larger block-matrices to be solved, but the preconditioner matches the true physics of the system more closely and may improve GMRES convergence enough to improve performance overall.

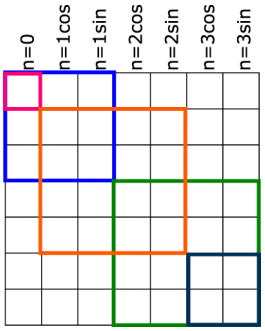


Figure 5.2: From [Holod et al., 2021]: "Schematic illustration for the Overlapping mode group approach. The color blocks outline the parts of the original matrix A used to form the preconditioner matrices which combine modes (0), (0,1), (1,2), (2,3), and (3) into diagonal blocks."

The implementation allows for arbitrary mode coupling, such as, for example, modes separated by a certain period that are believed to have strong coupling. This approach allows for modeling of reactor designs such as stellarators, which do not linearly decouple like tokamaks.

5.6. Relevant Problem Size

According to [Hoelzl et al., 2021], a "typical large problem" leads to a grid size of about 40 million, but the matrix is relatively sparse. About one out of every 3000 entries is nonzero. This "typical large problem" has "about 12 thousand non-zero entries in each matrix row and about 500 billion non-zero entries in the whole matrix, which requires about 4 TB of main memory for storing the double precision floating point numbers." [Hoelzl et al., 2021]. This matrix is generally not symmetric, but has symmetric sparsity. At this size, it is too large for a single compute node, so domain decomposition is used to divide the matrix into manageable sizes. The decomposed matrix is constructed in a parallelized manner on multiple compute nodes.

6

Research Plan

For some settings, the JOREK team sees poor convergence when trying to solve the linear system. The goal of this project is to analyze JOREK's linear system and solver so as to drive better overall performance. In particular, the team behind JOREK has asked us to:

1. Understand what limits the solver's convergence. What causes the bad convergence they occasionally see. Is this driven by e.g. a setting in the physics model or spatial feature? Is there a way to predict how a given run will perform without running the whole solver?
2. How does this behavior link to their numerical solvers? For what conditions does GMRES perform better or worse than other methods such as BICGSTAB?
3. Can we improve the preconditioner? Is there anything we can do to improve on memory usage, computation cost, or convergence? Is there anything about the matrix construction or physics model that could inform the construction of the preconditioner?

6.1. Research Plan

The JOREK team sends us data in the form of linear systems. These are systems and associated preconditioners, in a sparse format, and encoded in the HDF5 file format.

By studying several systems with a range of convergence behavior, we hope to understand and isolate features that negatively impact solver convergence. We will analyze these and determine if there is anything particular about them that leads to poor performance. Is the problem ill-conditioned? Does it have a particular spectrum that leads to poor performance for certain approaches?

Then we will attempt to see if there are any "easy wins" in terms of solving or preconditioning these problems. Is there a preconditioner we could use to solve these with any generality? The literature suggests other potential preconditioners for MHD, such as [Ma et al., 2016, Laakmann et al., 2022]. Are there any minor features to the matrix that can be adjusted? Are there other solvers we could use, such as BICGSTAB or variants of GMRES that would be preferred in some cases? For example, some GMRES algorithms that incorporate spectral information on restarts can overcome stalling or improve convergence in certain cases [Baker et al., 2004, Morgan, 1995].

6.2. Model Problems

The JOREK team has provided us with several example problems to analyze that have different physical and geometric properties, and different convergence behavior when trying to solve them.

These are further illustrated in chapter 7 and chapter 8

6.3. Solver Code

I was given access to a standalone solver by the JOREK team that was written in Fortran90 and C++ as a reference, but the code I used to analyze the systems was written by me in Julia [Bezanson et al., 2012]. Julia was chosen due to its performance in numerical computing, its readability and relative ease-of-use,

and a strong community that has built lots of useful extensions and libraries for sparse matrices and iterative solvers. I copied the algorithm used in the standalone solver as closely as possible,

For the GMRES and BiCGSTAB implementations, I used the implementations included in the Krylov.jl library [JuliaSmoothOptimizers, 2023].

For spectral decompositions, I used the eigsolve function from the KrylovKit.jl library [Jutho, 2023]. This uses an Arnoldi iteration method called Krylov-Schur to find extremal eigenvalues.

6.4. Hardware

For our analysis, we used the Marconi supercomputing cluster. The Marconi supercomputer has 3,188 nodes, with 2 24-core Intel Xeon 8160 (SkyLake) processors and 196 GB of ram per node [SCAI, 2023].

We only executed on one node at a time for our analysis, and made use of only 3-5 processors per node, depending on the system's construction.

6.5. Preconditioner Operator

It should be noted that the JOREK solver does not use an explicit preconditioner matrix, but instead preconditions by solving the LU decompositions of the preconditioner blocks against a solution vector. The literature on GMRES analysis only deals with explicit "M" matrices applied as a preconditioner, however we have no explicit preconditioner matrix. As a proof of concept, we created an explicit preconditioner for the smaller problems by inverting our preconditioner blocks, but this is computationally infeasible for the larger problems.

Thankfully, the solver and numerical analysis packages we used in Julia can also accept inputs in the form of generic untyped objects. By making an "Operator" object and attaching multiplication operations to it that behave identical to our preconditioning process, we have an object that behaves identical to an explicit preconditioner but does not require additional computational resources. For our analysis, we constructed an Operator out of our preconditioning algorithm and used that as though it was an explicit M^{-1} matrix. We verified that this object had identical solver behavior compared to our explicit preconditioner matrix.

6.6. Spectral Analysis

Since GMRES convergence behavior is driven not by the condition number, but by characteristics of the spectrum, a spectral analysis of our system is necessary (Refer to subsection 4.9.1).

Unfortunately, the system is too complicated to study analytically. As opposed to other often studied systems such as e.g. the Helmholtz Equation, we are not yet able to determine the eigenvalues analytically, and must study them numerically. For the smaller systems, it is computationally feasible to take the full spectrum. However, the larger cases are too large to calculate the complete spectrum. To save computational resources, we use a strategy that enables us to limit our calculations on the important regions of the spectrum.

In order to calculate the spectrum, we use the "eigsolve" function of the KrylovKit.jl software package [Jutho, 2023]. This uses the Krylov-Schur algorithm, which uses the Arnoldi method to build a Krylov subspace (see subsection 4.11.1. Using the Krylov-Schur method, we are able to take extremal eigenvalues without analyzing the entire system, thus saving computational resources. Using this method, we take in turns the "largest real", "smallest (most negative) real", "largest imaginary", and "largest magnitude" eigenvalues for each system. We could also get the "smallest (magnitude) imaginary" but this always yields a value with no imaginary component.

Since GMRES convergence is driven by the "radius" of the disk of eigenvalues and minimum distance to the origin, we can infer a complete picture of GMRES convergence performance from just the extremal eigenvalues without taking the complete spectrum. Convergence collapses in GMRES if we have a disk of eigenvalues that includes the origin. Because our preconditioned system consistently yields a spectrum centered at 1 with no imaginary component, we can develop a relatively effective picture of GMRES convergence by simply taking the most negative real-valued eigenvalue, and seeing if it is negative or comes close to a negative value.

- Take the LU decomposition of each of our preconditioner matrices. In parallel, each processor computes and stores the LU decomposition for a separate preconditioner block.

- Construct a linear Operator (as described above in section 6.5) out of these preconditioners such that when given a vector x, return a solution vector by solving against these preconditioner blocks, and then bringing these "local" solutions into the "global" ordering. This is the block-preconditioner as described by [Holod et al., 2021], and it behaves the same as an explicit $\mathbf{M}^{-1}$ preconditioner matrix.
- Take that operator and multiply it by our original $\mathbf{A}$ matrix to have a memory-efficient $\mathbf{M}^{-1}\mathbf{A}$ operator that does not require the direct computation of expensive inverses.
- Plugging our $\mathbf{M}^{-1}\mathbf{A}$ operator into common Julia eigensolvers that use Krylov-Schur gives us our spectrum or the extremal eigenvalues of our spectrum.

7

Simple Tearing Mode Case in Limiter Geometry

The first problem analyzed has a simple toroidal geometry with a circular cross-section without an X-point ("limiter geometry") with the major radius of the torus being 10 times higher than the minor radius of the circular cross section ("large aspect ratio"). As physics model, the reduced visco-resistive MHD model of JOREK without parallel velocity is used. The anisotropy of the heat conduction is low. The pressure of the plasma is so low that it does not contribute to the dynamics ("low beta"). This plasma develops a slowly growing so-called tearing mode instability dominated by the toroidal mode number n=1 that is destabilized by the radial profile of the plasma current and leads to the reconnection of magnetic field lines and the formation of magnetic islands (see Figure 7.1). The toroidal Fourier spectrum used to model the case includes only three toroidal modes with the mode numbers n=0, 1 and 2.

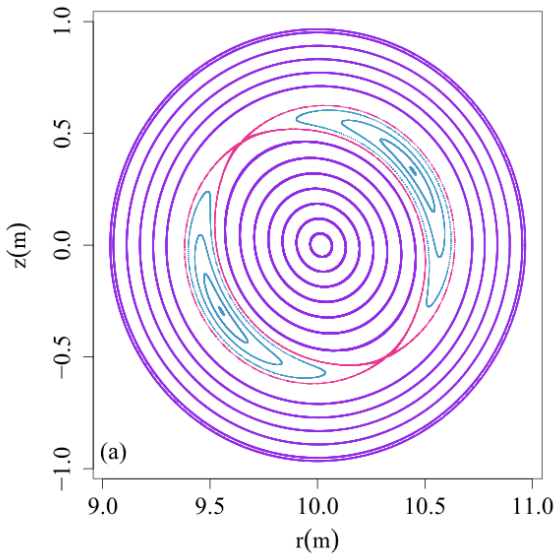


Figure 7.1: A Poincaré map of the poloidal plane of a circular tokamak in a similar simulation. Shown in blue is the magnetic island formed by the 2/1 tearing mode at the point where they have come to saturation, and are at their largest width. The separatrix is plotted in pink. The nearly-concentric surfaces of constant magnetic flux outside the magnetic island are plotted in purple. [Pratt et al., 2016]

For this model, we were given an h5 file describing a sparse square matrix of n=20,160, with 21,081,600 nonzero elements, and 3 preconditioner blocks corresponding to the three different toroidal modes n=0,1,2. As discussed in (section 5.4), the first preconditioner block corresponds to the first

toroidal mode, while subsequent blocks are of duplicate dimension, since they contain cosine and sine components. The first toroidal mode block is a sparse square matrix of n=4,032 with 843,264 nonzero elements. The following two blocks both have n=8,064 with 3,373,056 nonzero elements.

This reduced-size and simplified model is a "toy problem," and was primarily used to test and benchmark my analysis tools. It is not one of the problems that challenged the JOREK solver, but it can still be useful to analyze as a baseline.

7.1. Solver Behavior

Running GMRES without preconditioning, we get very poor convergence, as can be seen in Figure 7.2. The residual norm drops in the first two iterations, but then remains relatively flat.

With the block preconditioner described in (section 5.4), we have convergence to a residual of 10^{-9} in 10 iterations (Figure 7.2). The residual norm drops exponentially.

In general for this and the following problems, we are solving up to a residual with a relative norm of 10^{-12} and an absolute norm of 10^{-36}, where these are defined by the stopping criteria $||r_k|| \leq 10^{-36} + 10^{-12}||r_0||$. These were chosen for this analysis because they are the defaults in the JOREK solver code that we are analyzing.

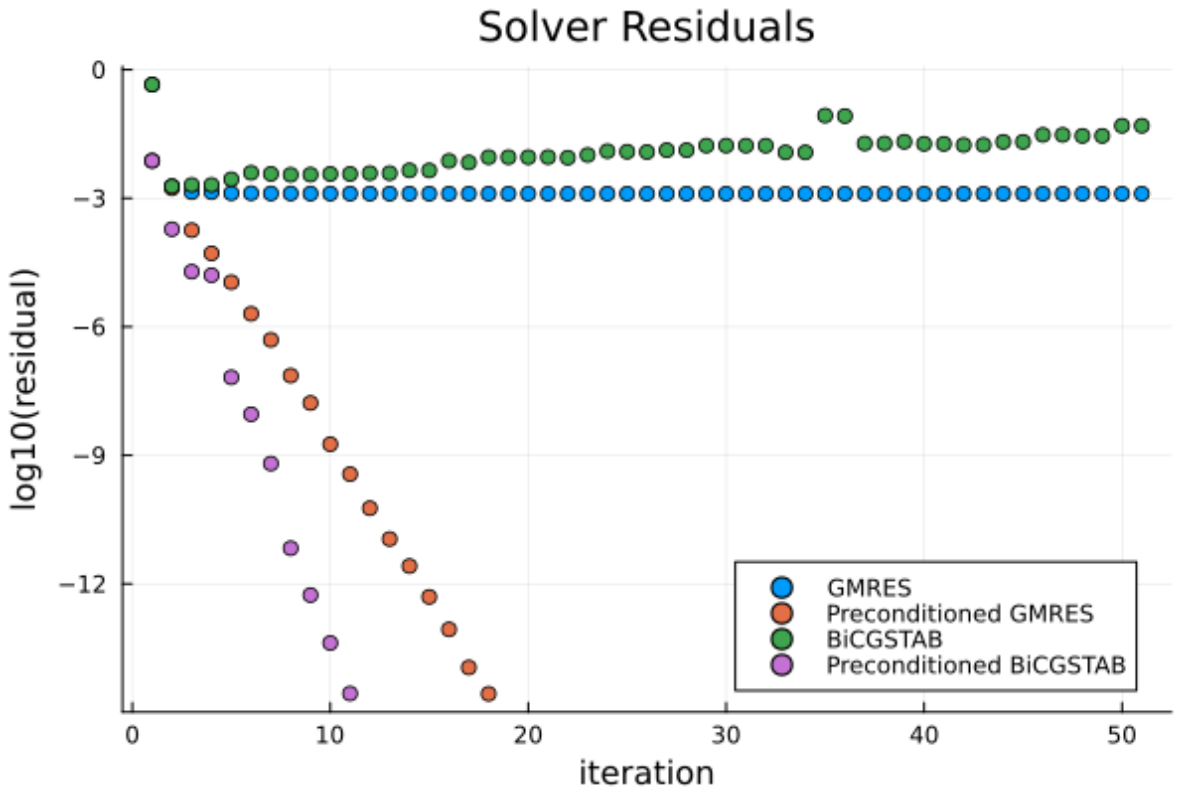


Figure 7.2: Residual per Iteration for several solver configurations, showing rapid convergence with preconditioning but poor convergence without.

Similar behavior is seen with BiCGSTAB. Without preconditioning, the residual norm actually increases slightly (Figure 7.2. With preconditioning, the residual norm drops exponentially to 10^{-9} after only 7 iterations (Figure 7.2).

7.2. Spectral Analysis

The system is not symmetric (using a fairly large tolerance to test corresponding floats for equality), but is symmetric in sparsity pattern. The system is also non-normal. The preconditioned small system is also non-symmetric.

Calculating the eigenvalues for the un-preconditioned problem, we see a wide range in the complex plane, stretching from approximately -10^5 to 10^{12} on the real number line, with values as small as

$\pm 5 \cdot 10^{-7}$, and as large as $\pm 140i$ in the imaginary.

This is an indefinite, and extremely poorly conditioned system. It is exactly the kind of system that would have poor performance in GMRES. It is real-valued, but not symmetric, so it does not lend itself to other more performant methods such as CG.

After preconditioning however, we obtain a spectrum centered at 1 with values ranging ± 0.2 in the real and $\pm 0.35i$ in the imaginary. As a system with tightly clustered eigenvalues, this spectrum explains the effectiveness of the preconditioning system used by JOREK. An effective preconditioner (one that roughly emulates $\mathbf{A}^{-1}$) leads to a system centered at 1 on the complex plane, with all values tightly clustered around 1 and all values greater than 0. This explains GMRES performance since GMRES convergence is driven by the radius and distance from the origin of the spectrum (Equation 4.27).

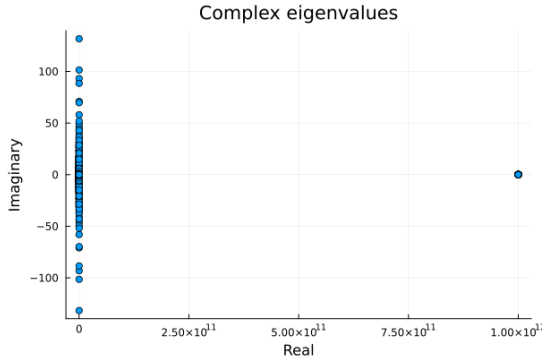


Figure 7.3: Spectrum for the non-preconditioned matrix $\mathbf{A}$

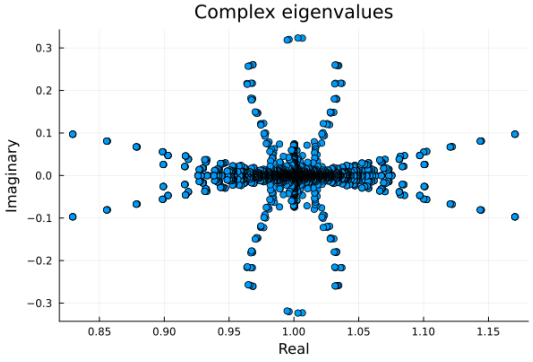


Figure 7.4: Spectrum for the preconditioned matrix $\mathbf{M}^{-1}\mathbf{A}$

8

Ballooning Mode Case in X-Point Geometry

The second cases we analyzed is considerably more realistic. It contains a plasma with an aspect ratio (major radius divided by the minor radius) that is typical for tokamak experiments, has an X-point, a plasma pressure that influences the dynamics, and a more realistic heat diffusion anisotropy. Furthermore, the visco-resistive reduced MHD model of JOREK including flows along magnetic field lines is used. Driven by the radial gradient of the pressure, the plasma develops a so-called ballooning mode instability with higher toroidal mode numbers than the previous case, a type of interchange instability that can be seen in Figure 8.1.

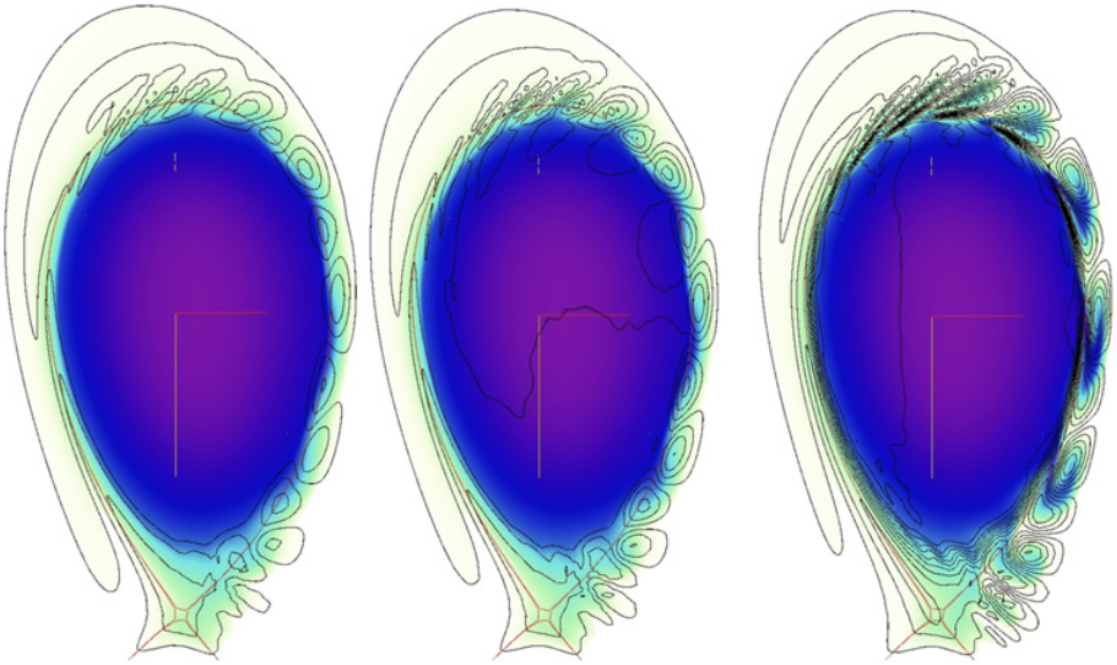


Figure 8.1: A poloidal cross-section of a similar model, showing density and vorticity streamlines at (a) $t = 2650\tau_A$, (b) $t = 2700\tau_A$ and (c) $t = 2890\tau_A$. The development of n=6 ballooning modes can be seen on the edges. [Huysmans and Czarny, 2007]

Three different variants of this case were given to us by the JOREK team: one artificially small case (44,667 rows/columns) with two toroidal modes, a similar case with higher FEM resolution (350,679 rows/columns), and another even larger case (584,465 rows/columns) that has three toroidal modes.

See Table 8.1 for more details. Significantly larger cases also exist, but for this analysis, we've chosen to focus on these.

Case	# of Toroidal Modes	n	Nonzero Elements	Toroidal Mode Size
Ballooning-Mode "Small"	3	44,667	32,219,901	14,889
Ballooning-Mode "Medium"	3	350,679	260,359,785	116,893
Ballooning-Mode "Large"	5	584,465	723,221,625	116,893

Table 8.1: System Sizes for the Ballooning-mode Cases. Toroidal Mode Size refers to the matrix size of our first toroidal mode block, and is double the size for subsequent blocks.

8.1. Solver Behavior

Similarly to the tearing mode case, convergence without preconditioning was poor. Using the simple block preconditioner however, the solution converges well. The "small" case reached a residual of 10^{-9} in 16 iterations. As in the previous case, we are solving up to a relative norm of out residual of 10^{-12}, and an absolute norm of 10^{-36}.

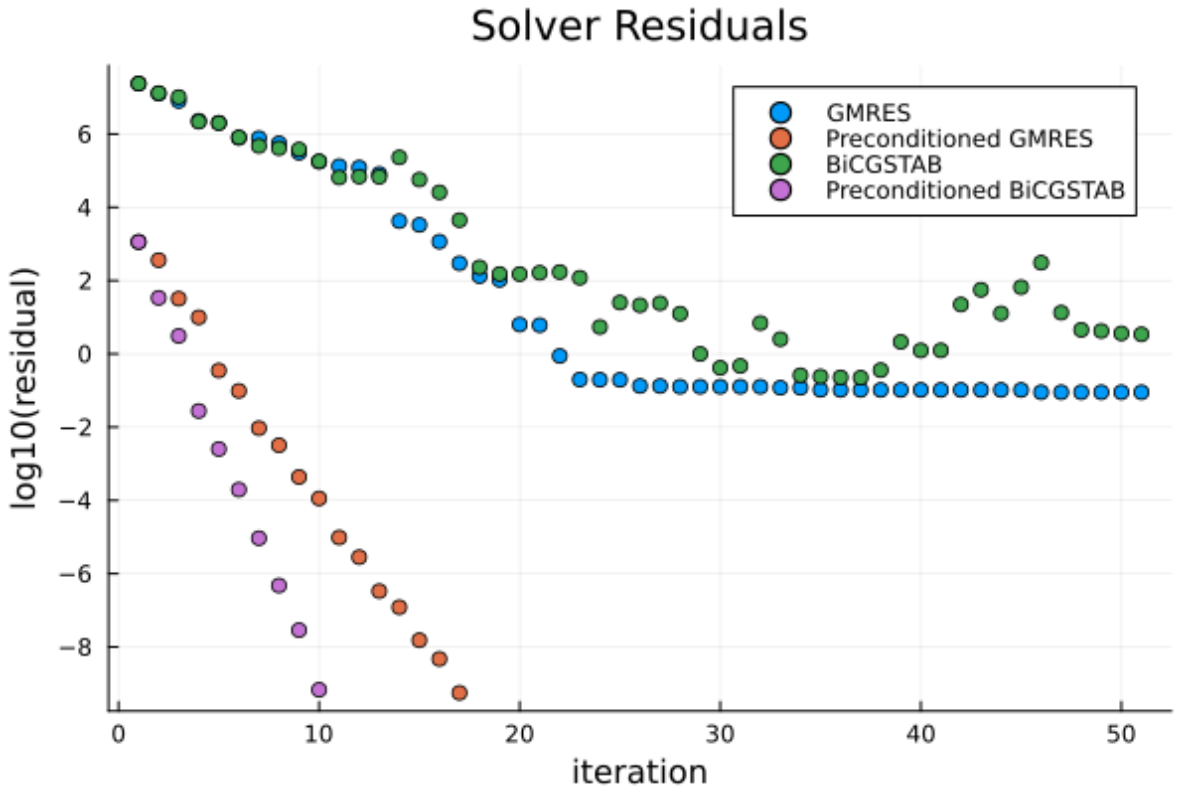


Figure 8.2: Residual per Iteration for several solver configurations in the "small" problem, showing rapid convergence with preconditioning but poor convergence without.

It is important to note in Figure 8.4, that the convergence was much worse for BiCGSTAB than for GMRES, also taking significantly longer per iteration (Figure 8.15).

8.2. Spectral Analysis

The systems are not symmetric (using a fairly large tolerance to test corresponding floats for equality), but are symmetric in sparsity pattern. They are also non-normal. The preconditioned small system is also non-symmetric. We did not do the analysis for the larger systems, but we see no reason this would not also be true.

For the smaller ballooning mode system, the spectrum shows similar patterns as for the tearing

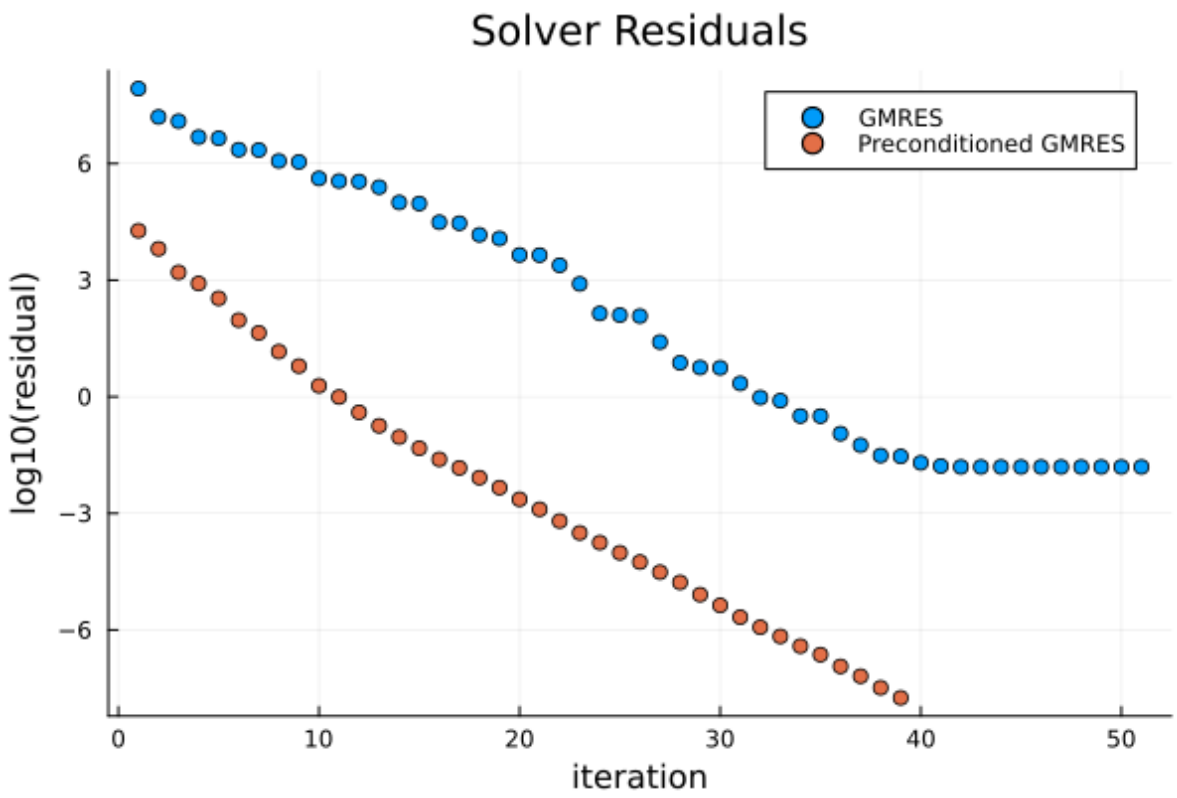


Figure 8.3: Residual per Iteration for several solver configurations in the "medium" problem, showing rapid convergence with preconditioning but poor convergence without.

mode case. The un-preconditioned state is highly indefinite, with eigenvalues tracking in the reals from $-3.9 * 10^{10}$ to $1 * 10^{12}$, and with eigenvalues as low as $1 * 10^{-7}$

Applying the preconditioner matrix, we now see a spectrum centered around $1 \pm (0.2 + 0.35i)$. These two spectra explain (as they did for the tearing mode case) the convergence behavior for GMRES in the preconditioned and un-preconditioned forms.

We were only able to fully analyze the smaller of the ballooning-mode systems. The larger systems, are too large to take the full spectrum, but an incomplete spectrum can still tell us about convergence behavior.

Using the Krylov-Schur method, we are able to take extremal eigenvalues without analyzing the entire system, saving lots of computation. We used the KrylovKit.jl Krylov-Schur "eigsolve" method to take in turns the "largest real", "smallest real", "largest imaginary", "smallest imaginary", and "largest magnitude" eigenvalues for each system.

Since GMRES convergence is driven by the "radius" of the disk of eigenvalues and minimum distance to the origin (Equation 4.27), we can infer a complete picture of GMRES convergence performance from just the extremal eigenvalues.

As can be seen in Figure 8.8 for the medium size case, the spectrum is again centered around 1 in a disk that does not include the origin.

However with a range of values from 0.37 to 1.63 in the reals and $\pm 0.81i$ in the imaginary, the spectral radius is larger than it is for the smaller case (Figure 8.6), which has a range of 0.86 to 1.14 in the reals and $\pm 0.25i$.

This larger spectral radius and closer distance to the origin explains the convergence performance seen in the medium case (38 iterations) compared to the small case (16 iterations).

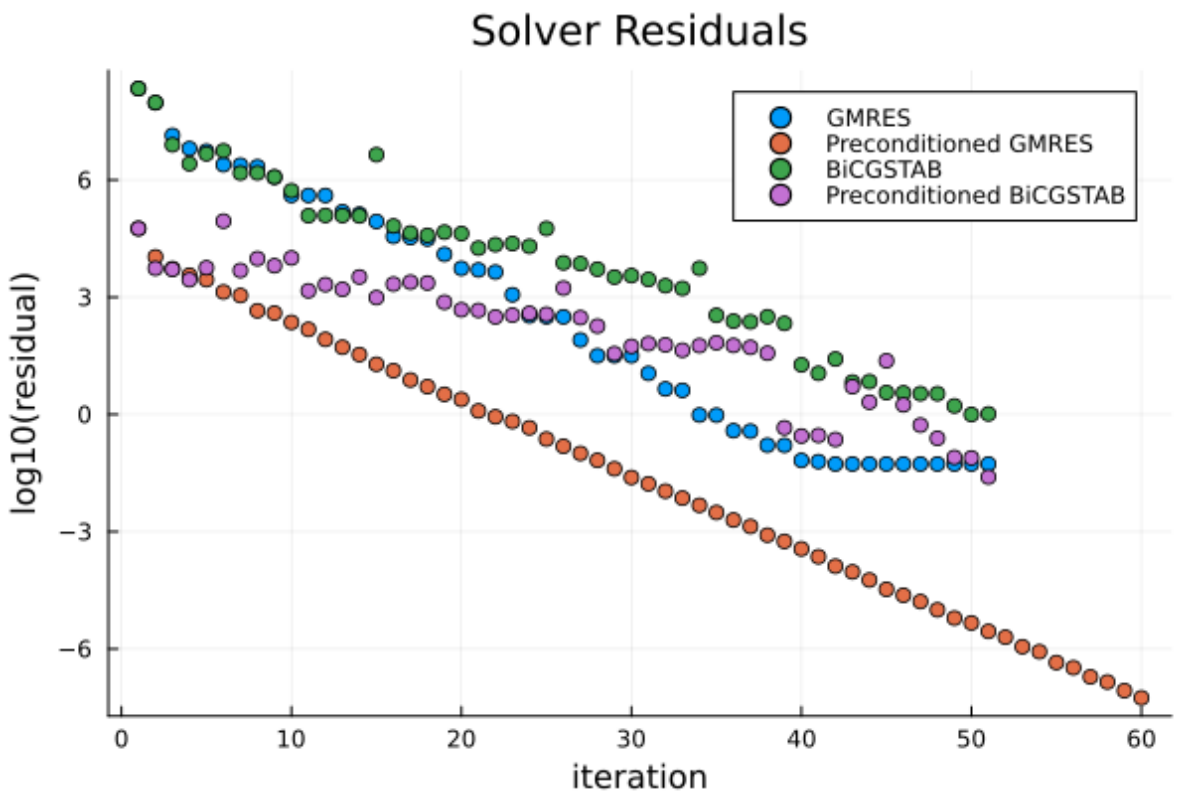


Figure 8.4: Residual per Iteration for several solver configurations in the "large" problem, showing rapid convergence with preconditioning but poor convergence without.

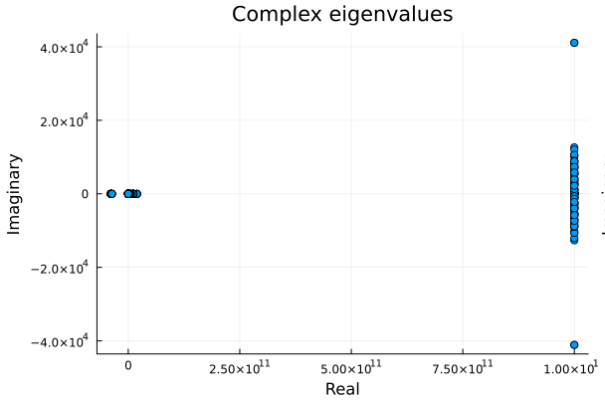


Figure 8.5: Complex spectrum for A in the small ballooning-mode case

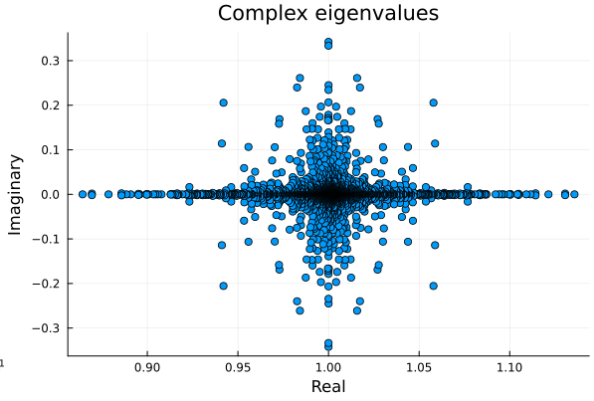


Figure 8.6: Preconditioned spectrum $M^{-1}A$ for the small ballooning-mode case

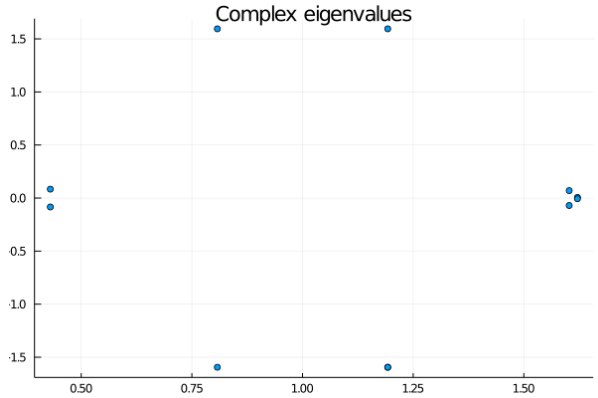


Figure 8.9: Extremal eigenvalues for the preconditioned spectrum $\mathbf{M}^{-1}\mathbf{A}$ for the large ballooning-mode case

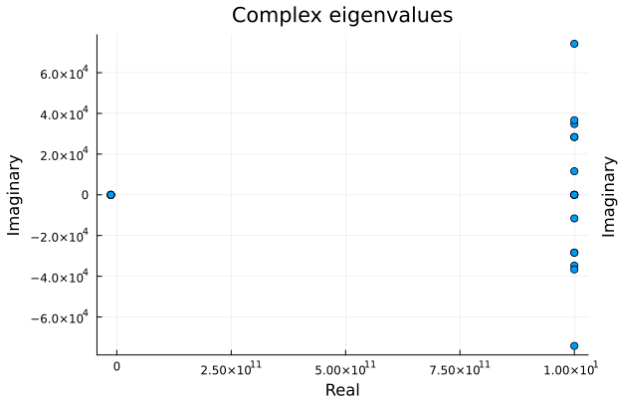


Figure 8.7: Extremal eigenvalues for $\mathbf{A}$ in the medium ballooning-mode case

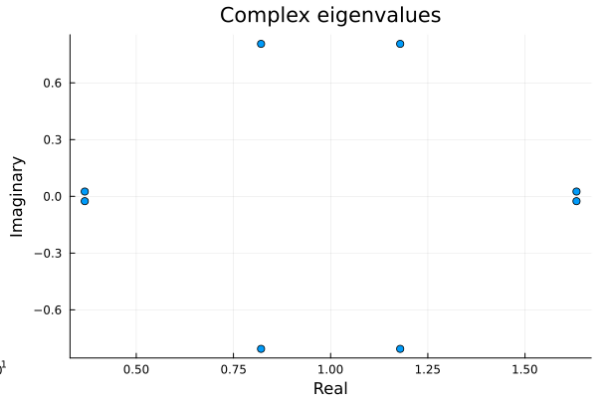


Figure 8.8: Extremal eigenvalues for the preconditioned spectrum $\mathbf{M}^{-1}\mathbf{A}$ for the medium ballooning-mode case

We obtain a very similar plot for the larger case. For a full table of the range of eigenvalues in the complex domain, see Table 8.3

Ballooning-Mode Case	Smallest Real	Largest Real	Largest Imaginary
"Small"	$-3.90 * 10^{10}$	$1.00 * 10^{12}$	$1.00 * 10^{12} \pm 6.75 * 10^{4}i$
"Medium"	$-1.30 * 10^{10}$	$1.00 * 10^{12}$	$1.00 * 10^{12} \pm 2.8 * 10^{4}i$
"Large"	$-1.30 * 10^{10}$	$1.00 * 10^{12}$	$1.00 * 10^{12} \pm 2.8 * 10^{4}i$

Table 8.2: Spectral properties for the ballooning-mode cases without preconditioning.

Ballooning-Mode Case	Iterations[1]	Smallest Real	Largest Real	Largest Imaginary
"Small"	16	0.86	$1.14 + 0.0i$	$1.06 + 0.25i$
"Medium"	38	$0.37 \pm 0.025i$	$1.63 \pm 0.025i$	$1.18 + 0.81i$
"Large"	59	$0.39 \pm 0.61i$	$1.65 \pm 0.56i$	$1.13 + 1.40i$

Table 8.3: Preconditioned solution behavior and spectral properties for the ballooning-mode cases.

8.3. Parameter Studies

The JOREK team has observed that several model parameters and settings can severely impact convergence, so they asked us to examine models for a few different systems. These new systems are otherwise identical to the "large" ballooning-mode system, which we can use here as a reference. All of the following experiments are performend on variations of our "large" model.

8.3.1. Stale Preconditioner

As the system evolves, the JOREK uses the same preconditioner since the LU decompositions required to update the preconditioner are computationally expensive. Over successive iterations, the effectiveness of the preconditioner drifts and GMRES convergence collapses.

We ran an experiment on a preconditioner that is 5 time-steps out-of-date. It had much worse convergence behavior (see Figure 8.11) and its spectrum was considerably worse (see Figure 8.10 and Table 8.4). The stale preconditioner is no longer a good approximation of the system. In the original preconditioned system, $\mathbf{M}^{-1}$ is the inverse of $\mathbf{A}$ with some assumptions about mode coupling. After several time-steps, one has evolved, and $\mathbf{M}$ no longer really approximates $\mathbf{A}$, and $\mathbf{M}^{-1}\mathbf{A}$ no longer approximates the identity. This leads to a divergence in the spectrum and a consequent impact on the convergence for GMRES. In the case of our 5 time-steps stale preconditioner, the spectrum has grown

[1]The number of iterations was slightly different between JOREK's code and the Julia code, and the JOREK solver was not written to handle the smallest case. Here I am listing the number of iterations in the Julia code.

from a width of 1.26 in the reals to a width of 4.57, resulting in a much larger spectral "radius" as used in our GMRES convergence relation Equation 4.27.

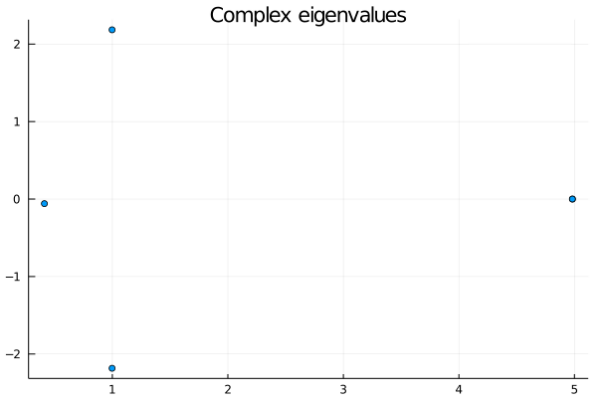


Figure 8.10: Extremal eigenvalues for the preconditioned spectrum $\mathbf{M}^{-1}\mathbf{A}$ for the large ballooning-mode case with a stale preconditioner.

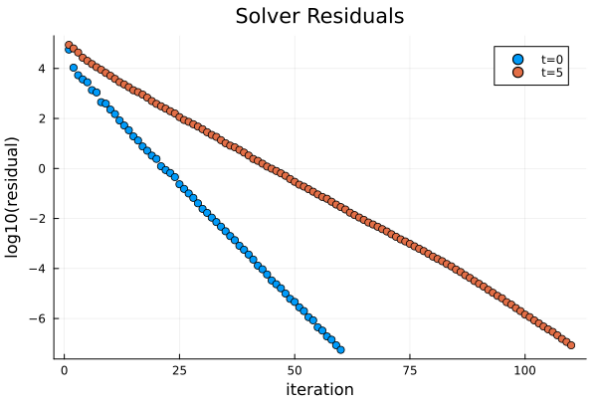


Figure 8.11: Residual per Iteration for GMRES for the "stale" Preconditioned Problem.

8.3.2. Parallel Thermal Conductivity

As the JOREK team believes the parallel thermal conductivity parameter (known as κ_{par}) could be a driver for poor convergence, we also ran a series of experiments with different values for κ_{par}, the parallel thermal conductivity. Due to the way charged particles move in magnetic fields, the thermal conductivity in a tokamak plasma is extremely anisotropic. Parallel thermal conductivity along magnetic field lines is much larger than perpendicular thermal conductivity, that moves across magnetic field lines. Since our ballooning instability is driven by the relative anisotropy, it makes sense that the anisotropy of thermal conductivity could impact model performance.

The JOREK team supplied us with two additional runs with κ_{par} set respectively at 100 and 200. The systems are otherwise identical to the "large" system, which has a κ_{par} value of 10, so we will use that system as our reference.

As can be seen in Figure 8.12, Figure 8.13, Figure 8.14 and Table 8.4, the convergence degrades considerably compared to our reference system. For both of these systems, the smallest eigenvalue is considerably closer to 0, which is where we would expect to see a complete collapse of GMRES convergence. This explains why we see such a large change in our required GMRES iterations, from 59 in the reference to 81 for $\kappa_{par} = 100$.

The difference in convergence and spectrum between the κ_{par} 100 and 200 systems though is fairly small. As expected, the $\kappa_{par} = 200$ system has slightly larger spectral radius, and slightly worse convergence behavior. It has the same spectral radius in the imaginary, but it grows in the reals from 0.89 to 0.94, and comes closer to the origin at 0.17 instead of 0.20. This is a relatively small difference, which reflects as a relatively small difference in the convergence behavior (87 vs 81 iterations). With such a small jump for a doubling of the parameter, this suggests that larger and larger values of κ_{par} only make a significant difference up to a certain value.

Ballooning-Mode Case	GMRES Iterations	Smallest Real	Largest Real	Largest Imaginary
Reference	59	$0.39 \pm 0.61i$	$1.65 \pm 0.56i$	$1.13 + 1.40i$
Stale Preconditioner	109	$0.412 \pm 5.9 * 10^{-2}i$	$4.98 \pm 4.18 * 10^{-9}i$	$1.00 \pm 2.18i$
$\kappa_{par} = 100$	81	$0.20 \pm 1.57 * 10^{-2}i$	$1.98 \pm 1.81 * 10^{-3}i$	$1.18 \pm 1.59i$
$\kappa_{par} = 200$	87	$0.165 \pm 7.39 * 10^{-3}i$	$2.05 \pm 2.76 * 10^{-3}i$	$1.18 \pm 1.59i$

Table 8.4: Preconditioned solution behavior and spectral properties for additional the ballooning-mode cases.

8.3.3. GMRES Restart

On every iteration, GMRES uses a larger Krylov subspace and therefore uses more and more memory. On large systems, one may periodically restart GMRES in order to control memory consumption.

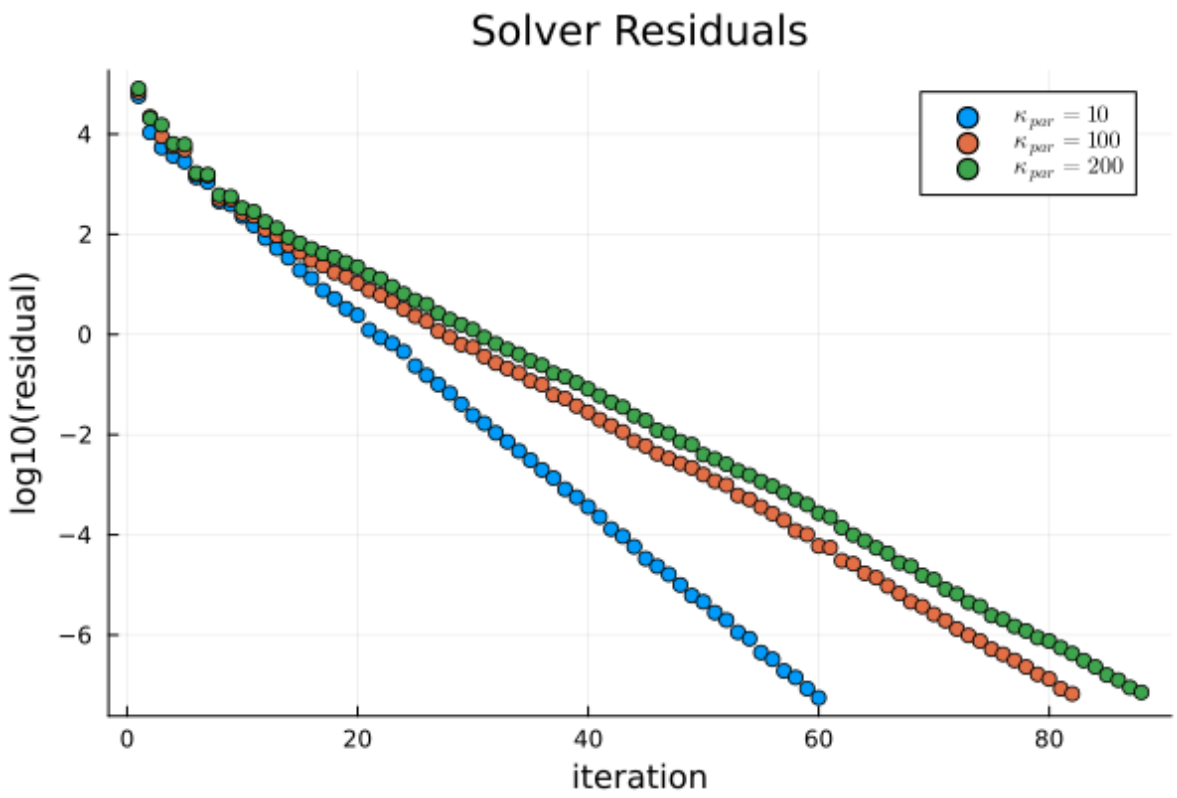


Figure 8.12: Residual per Iteration for GMRES for the Preconditioned Problem with different levels of kpar

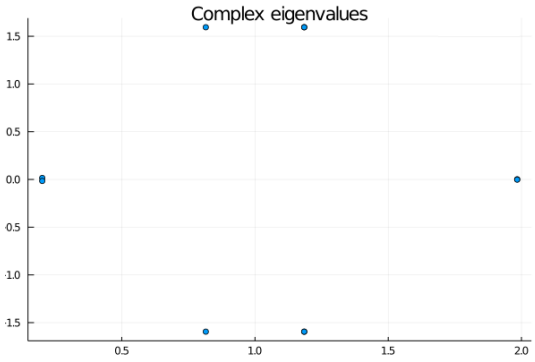


Figure 8.13: Extremal eigenvalues for the preconditioned spectrum $M^{-1}A$ for the large ballooning-mode case with $\kappa_{par} = 100$

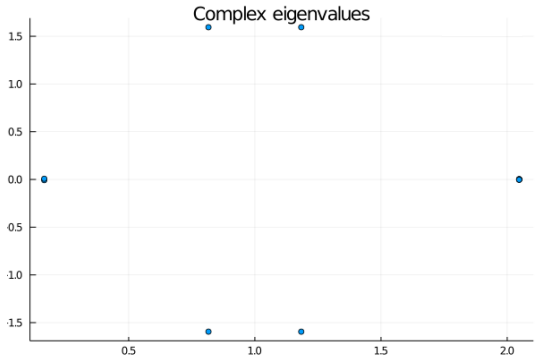


Figure 8.14: Extremal eigenvalues for the preconditioned spectrum $\mathbf{M}^{-1}\mathbf{A}$ for the large ballooning-mode case with $\kappa_{par} = 200$

The basic idea is to terminate GMRES after a certain number of iterations, say m, and then restart it, discarding the previously computed vectors to free up memory. While this strategy can often provide a good solution with fewer storage requirements, there is a trade-off. Restarts can harm the convergence behavior of GMRES, because the information collected from the Krylov subspace in previous iterations is lost at every restart. Therefore, choosing an appropriate restart frequency is vital. A small "m" can reduce memory usage but too small "m" can slow down the convergence. If "m" is too large, it may cause out-of-memory errors. The optimal choice of "m" usually depends on the specific problem and system characteristics.

We performed a small set of analysis with different restart frequencies against our "large" system. This analysis was somewhat artificial, as this system does not require restarts on the Marconi infrastructure. However, larger problems frequently encountered by the JOREK team require restarts, and a restart frequency of m=20 is currently in use. A difference in convergence was measured as expected, but with

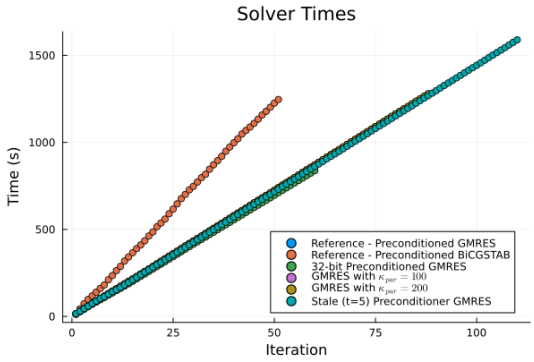


Figure 8.15: Timing data for various preconditioners and solvers.

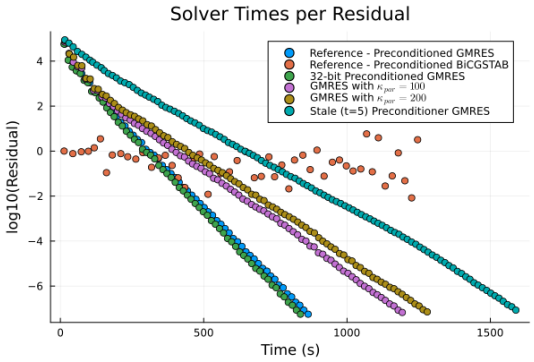


Figure 8.16: Residual per time data for various preconditioners and solvers.

a restart frequency of 10, only 12 extra iterations were needed. At the frequency used by JOREK of m=20, 67 iterations were used instead of 59 for the reference system.

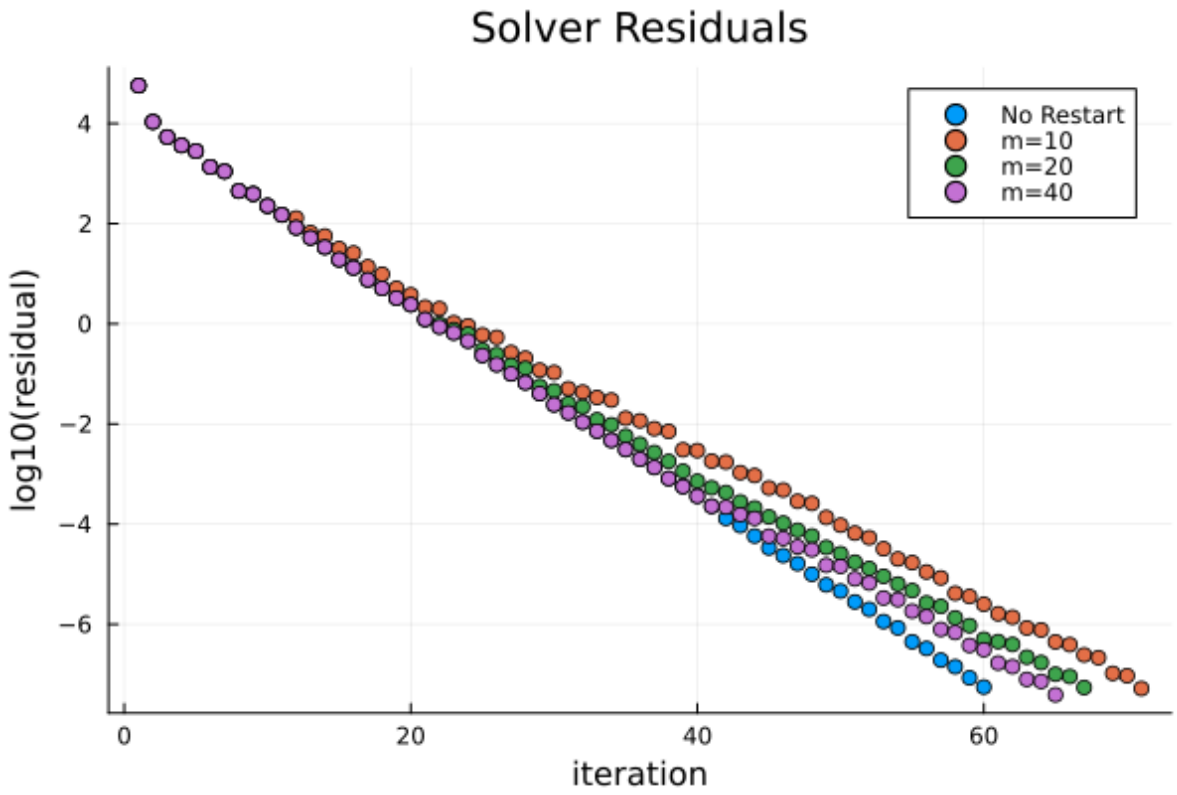


Figure 8.17: Residuals for different values of m.

8.3.4. 32-Bit Preconditioners

As an experiment, we also measured performance using preconditioners made up of both 64-bit and 32-bit floats. For the cases measured, there was no difference in terms of iteration count, but a small speedup was observed, taking 837s with the 32-bit preconditioner and 864s with the regular 64-bit preconditioner (see Figure 8.15).

16-bit floats were not possible for the preconditioner, as the values in the preconditioner blocks (approximately $\pm 10^{13}$) span a larger range than 16-bit floats can handle ($\pm 65,504$).

8.4. Conclusion

As our systems get less ideal, the "tails" of our spectrum grow along the real number line, reaching towards the y-axis on the left and higher values on the right. As they grow, they not only increase the spectral radius, but as the "tail" grows on the left towards 0, it also reduces the spectral disk's distance from the origin. These two factors drive GMRES convergence, as dictated by Equation 4.27. This relation to convergence can be seen in the growing number of GMRES iterations necessary to reach our required tolerance.

An interesting question of course appears, which is "what causes these tails to grow?" Our preconditioner is a simplified model wherein we assume that no coupling exists between different toroidal modes, and each mode can be solved independently. If this assumption were completely true, our preconditioner would be ideal, and all eigenvalues would exist at (1, 0) on the complex plane. Therefore, eigenvalues that drift away from this point imply the breakdown of this assumption. In effect, the spectral radius acts as a sort of quantifiable measure of the toroidal mode coupling, where a larger spread corresponds with more coupling. More complex models with a larger preconditioned spectral radius must have greater mode-coupling, however it still is not clear what exactly is causing this mode-coupling.

It would be very useful to determine what properties of the system drive the coupling that these eigenvalues correspond to, such as physical properties or geometry of the system. With such a complicated physical model, an analytical study of this is difficult and outside the scope of our analysis. Due to our numerical experiments with κ_{par}, we can see that this value has a coupling effect, as a larger κ_{par} results in more coupling.

The good news is that knowing about these problematic eigenvalues opens us up to deflation techniques that can remove the problematic parts of the spectrum, improving numerical performance. These deflation methods are a set of techniques that allow the removal of unwanted eigenvalues from a sub-problem that can then be solved with Krylov methods more effectively than solving the original system [Burrage et al., 1998].

9

Future Work

In this work we present the first spectral analysis from a numerical analysis point of view to thoroughly understand the convergence behavior of solvers used in fusion simulations. Discretized versions of the reduced MHD equations often lead to sparse, complex, indefinite and nonsymmetric systems. Consequently, the choice of numerical solvers is nontrivial as state-of-the-art solvers do not apply to these type of matrices. As a result, convergence can be slow and difficult to improve. By studying the underlying mathematical properties of the matrices, such as eigenvalues, we have diagnostic tools to interpret the convergence and find solutions to accelerate the simulation speed. The spectrum observed in these examples reiterate a classical point of view, often encountered in numerical analysis: as the model problems become more involved, the underlying eigenvalues start growing tails, leading to hampered convergence. Now that we have established this effect, we can use a set of intricate techniques to enhance the preconditioner, such as subspace projection methods, which can be addressed in future work.

Using deflation techniques, it may be possible to alter portions of the spectrum without sacrificing solution accuracy. The idea is to remove the smallest or most convergence-prohibitive eigenvalues from the system, in exchange for real positive eigenvalues that don't slow down convergence [Erhel et al., 1996]. However, further analysis for deflation for this problem is is outside the scope of this project, but it is an obvious follow-up area of research.

10

Failed Attempts

10.1. Python Implementation

Before the Julia implementation, I attempted to solve this problem with Python, since it was the language I was most familiar with, and was aware that libraries such as Numpy and Scipy are popular for linear algebra applications. Some problems I faced were that the Python implementation was much much slower, even just basic matrix manipulation using Numpy and Scipy. More importantly though, the direct solvers used by Scipy and Numpy for the LU decompositions gave me slightly different answers than I was getting from the JOREK Fortran code, for the same matrix. This is possibly due to a bug in my code, though none was found. Another possibility could be that due to the extreme ill-conditioning of these matrices, very minor variations in direct solver libraries caused large discrepancies between solutions. Normally this would be very unlikely, but the extreme ill-conditioning of these matrices means that minor differences in solvers could have large effects.

After verifying that the Julia LU decompositions were similar to the Fortran solutions, much more performant, and just as easy to write, I switched my development to Julia.

10.2. Explicit Preconditioner

As discussed in section 6.5, our first attempt was to create explicit $\mathbf{M}^{-1}$ matrices, but this was only feasible for very small model problems. However, since we without another way to calculate spectra, we made a new plan using our block matrices and the Gershgorin Circle Theorem to bound our $\mathbf{M}^{-1}]\mathbf{A}$ spectrum based on the spectra of our preconditioner blocks, which are individually small enough to run computations on, and our $\mathbf{A}$, which was manageable because it did not have to be inverted and could remain sparse.

The process would have been as follows:

1. Take the extremes of the spectrum of our $\mathbf{A}$ matrix.
2. Using either Gershgorin's Circle Theorem or Krylov-Schur, estimate the range of the eigenvalues for our preconditioner block matrices.
3. Since the eigenvalues of an inverted matrix are equal to the inverses of the eigenvalues of the original matrix, we use the eigenvalues of our $\mathbf{M_i}$ matrices to create a bounds on our $\mathbf{M}$ matrices.
4. Multiplying eigenvalues, we now have bounds on our the spectrum of our $\mathbf{M}^{-1}\mathbf{A}$ system.

This was abandoned for two reasons. First, this created impractically large bounds on our spectrum, generally around $\pm 10^{20}$ or so. Additionally, we developed a computationally practical approach to analyze our spectra. Using artificial "Operator" objects in Julia rather than fully dense matrices, we could successfully calculate eigenvalues for our system.

10.3. Nested GMRES

We attempted to implement a Nested GMRES system, similar to the preconditioners described in ([Van der Vorst and Vuik, 1994] [Saad, 1993]). The idea was to solve the preconditioner blocks with GMRES rather than an LU decomposition in the inner loop. However, GMRES convergence even for individual blocks is very poor, and this performed worse than direct solvers. There was no advantage compared to the current implementation. Our "nested GMRES" implementation did not turn out to be practical.

Bibliography

[Bai, 2015] Bai, Z.-Z. (2015). Motivations and realizations of krylov subspace methods for large sparse linear systems. *Journal of Computational and Applied Mathematics*, 283:71–78.

[Baker et al., 2004] Baker, A. H., Jessup, E. R., and Manteuffel, T. (2004). A technique for accelerating the convergence of restarted gmres. *SIAM Journal on Matrix Analysis and Applications*, 25(4).

[Bellan, 2006] Bellan, P. M. (2006). *Fundamentals of Plasma Physics*. Cambridge University Press.

[Bezanson et al., 2012] Bezanson, J., Karpinski, S., Shah, V. B., and Edelman, A. (2012). Julia: A fast dynamic language for technical computing. *arXiv preprint arXiv:1209.5145*.

[Burrage et al., 1998] Burrage, K., Erhel, J., Pohl, B., and Williams, A. (1998). A deflation technique for linear systems of equations. *SIAM Journal on Scientific Computing*, 19(4):1245–1260.

[Czarny and Huysmans, 2008] Czarny, O. and Huysmans, G. (2008). Bézier surfaces and finite elements for mhd simulations. *Journal of Computational Physics*, 227(16):7423–7445.

[Dwarka and Vuik, 2020] Dwarka, V. and Vuik, C. (2020). Scalable convergence using two-level deflation preconditioning for the helmholtz equation. *SIAM Journal on Scientific Computing*, 42(2):A901–A928.

[Dwarka and Vuik, 2022] Dwarka, V. and Vuik, C. (2022). Scalable multi-level deflation preconditioning for highly indefinite time-harmonic waves. *Journal of Computational Physics*, 469:111327.

[Erhel et al., 1996] Erhel, J., Burrage, K., and Pohl, B. (1996). Restarted gmres preconditioned by deflation. *Journal of Computational and Applied Mathematics*, 69(2):303–318.

[Franck, Emmanuel et al., 2015] Franck, Emmanuel, Hölzl, Matthias, Lessig, Alexander, and Sonnendrücker, Eric (2015). Energy conservation and numerical stability for the reduced mhd models of the non-linear jorek code. *ESAIM: M2AN*, 49(5):1331–1365.

[Freidberg et al., 2015] Freidberg, J. P., Mangiarotti, F. J., and Minervini, J. (2015). Designing a tokamak fusion reactor—how does plasma physics fit in? *Physics of Plasmas*, 22(7):070901.

[Futatani et al., 2019] Futatani, S., Pamela, S., Garzotti, L., Huijsmans, G., Hoelzl, M., Frigione, D., Lennholm, M., the JOREK Team, and Contributors, J. (2019). Non-linear magnetohydrodynamic simulations of pellet triggered edge-localized modes in jet. *Nuclear Fusion*, 60(2):026003.

[Ghai et al., 2016] Ghai, A., Lu, C., and Jiao, X. (2016). A comparison of preconditioned krylov subspace methods for large-scale nonsymmetric linear systems.

[Hernandez et al., 2007] Hernandez, V., Roman, J. E., Tomas, A., and Vidal, V. (2007). Krylov-schur methods in slepc. Technical Report STR-7, Universitat Politècnica de València. Available at https://slepc.upv.es.

[Hoelzl, 2022] Hoelzl, M. (2022). Violent transient plasma instabilities in magnetic confinement fusion plasmas and their control.

[Hoelzl et al., 2021] Hoelzl, M., Huijsmans, G., Pamela, S., Bécoulet, M., Nardon, E., Artola, F., Nkonga, B., Atanasiu, C., Bandaru, V., Bhole, A., Bonfiglio, D., Cathey, A., Czarny, O., Dvornova, A., Fehér, T., Fil, A., Franck, E., Futatani, S., Gruca, M., Guillard, H., Haverkort, J., Holod, I., Hu, D., Kim, S., Korving, S., Kos, L., Krebs, I., Kripner, L., Latu, G., Liu, F., Merkel, P., Meshcheriakov, D., Mitterauer, V., Mochalskyy, S., Morales, J., Nies, R., Nikulsin, N., Orain, F., Pratt, J., Ramasamy, R., Ramet, P., Reux, C., Särkimäki, K., Schwarz, N., Verma, P. S., Smith, S., Sommariva, C., Strumberger, E., van Vugt, D., Verbeek, M., Westerhof, E., Wieschollek, F., and Zielinski, J. (2021). The jorek non-linear extended mhd code and applications to large-scale instabilities and their control in magnetically confined fusion plasmas. *Nuclear Fusion*, 61(6):065001.

[Holod et al., 2021] Holod, I., Hoelzl, M., Verma, P. S., Huijsmans, G., Nies, R., and Team, J. (2021). Enhanced preconditioner for jorek mhd solver. *Plasma Physics and Controlled Fusion*, 63(11):114002.

[Hughes, 1987] Hughes, T. J. (1987). *The finite element method Linear static and dynamic finite element analysis*. Prentice-Hall International.

[Huysmans and Czarny, 2007] Huysmans, G. and Czarny, O. (2007). Mhd stability in x-point geometry: simulation of elms. *Nuclear Fusion*, 47(7):659.

[JuliaSmoothOptimizers, 2023] JuliaSmoothOptimizers (2023). Krylov.jl. `https://github.com/JuliaSmoothOptimizers/Krylov.jl`.

[Jutho, 2023] Jutho (2023). Krylovkit.jl. `https://github.com/Jutho/KrylovKit.jl`.

[Krebs, 2012] Krebs, I. (2012). Non-linear reduced mhd simulations of edge-localized modes in realistic asdex upgrade geometry. Master's thesis, Ludwig Maximilians Universitaet Muenchen.

[Laakmann, 2022] Laakmann, F. (2022). *Discretisations and Preconditioners for Magnetohydrodynamics Models*. PhD thesis, Mathematical Institute, University of Oxford.

[Laakmann et al., 2022] Laakmann, F., Farrell, P. E., and Mitchell, L. (2022). An augmented lagrangian preconditioner for the magnetohydrodynamics equations at high reynolds and coupling numbers. *SIAM Journal on Scientific Computing*, 44(4):B1018–B1044.

[Landstorfer et al., 2021] Landstorfer, D. M., Ullrich, J., Kruse, D. R., Voigt, D. M., and Peschka, D. D. (2021). Numerical mathematics ii for engineers.

[Liesen, 2020] Liesen, J. (2020). Numerical linear algebra i.

[Liesen and Tichý, 2004] Liesen, J. and Tichý, P. (2004). Convergence analysis of krylov subspace methods. *GAMM-Mitteilungen*, 27(2):153–173.

[Ma et al., 2016] Ma, Y., Hu, K., Hu, X., and Xu, J. (2016). Robust preconditioners for incompressible mhd models. *Journal of Computational Physics*, 316:721–746.

[Morgan, 1995] Morgan, R. B. (1995). A restarted gmres method augmented with eigenvectors. *SIAM Journal on Matrix Analysis and Applications*, 16(4):1154–1171.

[Niels Saabye Ottosen, 1992] Niels Saabye Ottosen, H. P. (1992). *Introduction to the Finite Element Method*. Prentice Hall.

[Pamela, 2010] Pamela, S. (2010). *Simulation Magnéto-Hydro-Dynamiques des Edge-Localised-Modes dans un tokamak*. PhD thesis, Université de Provence.

[Pratt et al., 2016] Pratt, J., Huijsmans, G. T. A., and Westerhof, E. (2016). Early evolution of electron cyclotron driven current during suppression of tearing modes in a circular tokamak. *Physics of Plasmas*, 23(10).

[Saad, 1993] Saad, Y. (1993). A flexible inner-outer preconditioned gmres algorithm. *SIAM Journal on Scientific Computing*, 14(2):461–469.

[Saad, 1996] Saad, Y. (1996). *Iterative Methods for Sparse Linear Systems*. PWS Publishing Company, Boston.

[Saad and Schultz, 1986] Saad, Y. and Schultz, M. H. (1986). Gmres: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM Journal on Scientific and Statistical Computing*, 7(3):856–869.

[SCAI, 2023] SCAI, C. (2023). Marconi | scai. `https://www.hpc.cineca.it/hardware/marconi`.

[Shadid and Tuminaro, 1994] Shadid, J. N. and Tuminaro, R. S. (1994). A comparison of preconditioned nonsymmetric krylov methods on a large-scale mimd machine. *SIAM Journal on Scientific Computing*, 15(2):440–459.

[Team, 2023] Team, J. (2023). *JOREK Wiki*.

[Van der Vorst and Vuik, 1994] Van der Vorst, H. A. and Vuik, C. (1994). Gmresr: a family of nested gmres methods. *Numerical Linear Algebra with Applications*, 1(4):369–386.

[Vuik and Lahaye, 2023] Vuik, C. and Lahaye, D. (2023). Scientific computing (wi4201).

[Wesson, 1999] Wesson, J. (1999). *The Science of JET*. "JET Joint Undertaking".

[Zavorin et al., 2003] Zavorin, I., O'Leary, D., and Elman, H. (2003). Complete stagnation of gmres. *Linear Algebra and its Applications*, 367:165–183.

[Zou, 2023] Zou, Q. (2023). GMRES algorithms over 35 years. *Applied Mathematics and Computation*, 445:127869.

Acknowledgements

I'd like to thank my supervisor, Dr. Vandana Dwarka, Drs. Matthias Hoelzl and Igor Holod at IPP, my colleague Ferhat Sindy, and Dr. Angelica Ferrara for encouraging me to pursue this.

A Code

A.1. Helper functions

```
# utils/helpers.jl
using LinearMaps
using Dates
using LinearAlgebra
using IterativeSolvers
using HDF5
using SparseArrays
using Plots
using KrylovKit
using Krylov
using LinearOperators
using ArnoldiMethod
using DelimitedFiles
using Plots
using Distributed


# Construct Preconditioner Operator
struct Preconditioner
end

opM = Preconditioner()

function load_run(run_name::String)
    # returns A, then PCs
    runs = Dict("199_small"=>(
                ["data/199_small/matA.h5"],
                ["data/199_small/pc_00.h5", "data/199_small/pc_01.h5", "
                    data/199_small/pc_02.h5"]
            ), "199_large"=>(
                ["data/199_large/matA_00.h5", "data/199_large/matA_01.h5"
                    ],
                ["data/199_large/pc_00.h5", "data/199_large/pc_01.h5", "
                    data/199_large/pc_02.h5"]
            ), "303_modded"=>(
                ["data/303_modded/matA_00.h5", "data/303_modded/matA_01.h5
                    "],
                ["data/303_modded/pc_00.h5", "data/303_modded/pc_01.h5"]
```

```
            ), "303_inxflow"=>(
                ["data/303_inxflow/matA_00.h5"],
                ["data/303_inxflow/pc_00.h5", "data/303_inxflow/pc_01.h5"]
            ), "303_inxflow_3"=>(
                ["data/303_inxflow_3_medium/matA_00.h5", "data/303
                    _inxflow_3_medium/matA_01.h5"],
                ["data/303_inxflow_3_medium/pc_00.h5", "data/303
                    _inxflow_3_medium/pc_01.h5"]
            ), "303_inxflow_5"=>(
                ["data/303_inxflow_5_medium/matA_00.h5", "data/303
                    _inxflow_5_medium/matA_01.h5", "data/303
                    _inxflow_5_medium/matA_02.h5"],
                ["data/303_inxflow_5_medium/pc_00.h5", "data/303
                    _inxflow_5_medium/pc_01.h5", "data/303_inxflow_5_medium
                    /pc_02.h5"]
            ), "303_reference" => (
                ["/marconi_work/FUA37_MHD/iholod00/for_Alex/case_230822_0/
                    matA_00.h5", "/marconi_work/FUA37_MHD/iholod00/for_Alex
                    /case_230822_0/matA_01.h5", "/marconi_work/FUA37_MHD/
                    iholod00/for_Alex/case_230822_0/matA_02.h5"],
                ["/marconi_work/FUA37_MHD/iholod00/for_Alex/case_230822_0/
                    pc_00.h5", "/marconi_work/FUA37_MHD/iholod00/for_Alex/
                    case_230822_0/pc_01.h5", "/marconi_work/FUA37_MHD/
                    iholod00/for_Alex/case_230822_0/pc_02.h5"]
            ), "303_zk100" => (
                ["/marconi_work/FUA37_MHD/iholod00/for_Alex/case_230822_1/
                    matA_00.h5", "/marconi_work/FUA37_MHD/iholod00/for_Alex
                    /case_230822_1/matA_01.h5", "/marconi_work/FUA37_MHD/
                    iholod00/for_Alex/case_230822_1/matA_02.h5"],
                ["/marconi_work/FUA37_MHD/iholod00/for_Alex/case_230822_1/
                    pc_00.h5", "/marconi_work/FUA37_MHD/iholod00/for_Alex/
                    case_230822_1/pc_01.h5", "/marconi_work/FUA37_MHD/
                    iholod00/for_Alex/case_230822_1/pc_02.h5"]
            ), "303_stale" => (
                ["c/matA_00.h5", "/marconi_work/FUA37_MHD/iholod00/
                    for_Alex/case_230822_2/matA_01.h5", "/marconi_work/
                    FUA37_MHD/iholod00/for_Alex/case_230822_2/matA_02.h5"],
                ["/marconi_work/FUA37_MHD/iholod00/for_Alex/case_230822_2/
                    pc_00.h5", "/marconi_work/FUA37_MHD/iholod00/for_Alex/
                    case_230822_2/pc_01.h5", "/marconi_work/FUA37_MHD/
                    iholod00/for_Alex/case_230822_2/pc_02.h5"]
            ), "303_zk200" => (
                ["/marconi_work/FUA37_MHD/iholod00/for_Alex/case_230828_0/
                    matA_00.h5", "/marconi_work/FUA37_MHD/iholod00/for_Alex
                    /case_230828_0/matA_01.h5", "/marconi_work/FUA37_MHD/
                    iholod00/for_Alex/case_230828_0/matA_02.h5"],
                ["/marconi_work/FUA37_MHD/iholod00/for_Alex/case_230828_0/
                    pc_00.h5", "/marconi_work/FUA37_MHD/iholod00/for_Alex/
                    case_230828_0/pc_01.h5", "/marconi_work/FUA37_MHD/
                    iholod00/for_Alex/case_230828_0/pc_02.h5"]
            )
        )
    return runs[run_name]
end

function load_A(run_name)
```

```
    A_fps, _ = load_run(run_name)
    A, rhs, _ = multi_load_sparse_matrix_from_hdf5(A_fps, false);
    n = size(A,1)
    return A, rhs, n
end

function load_preconditioners(run_name)
    _, pc_fps = load_run(run_name)

    pcs = Vector{SparseMatrixCSC{Float64,Int64}}()
    rhss = Vector{Vector{Float64}}()
    l2gs = Vector{Vector{Int32}}()

    for pc_fp in pc_fps
        pc, pc_rhs, pc_l2g = load_sparse_h5(pc_fp, true);
        push!(pcs, pc)
        push!(rhss, pc_rhs)
        push!(l2gs, pc_l2g)
    end

    return pcs, rhss, l2gs
end

function save_h5_dense(filename::String, A::Matrix)
    datasetname = "data"
    h5open(filename, "w") do file
        write(file, datasetname, A)
    end
end

function load_h5_dense(filenam::String)
    datasetname = "data"
    h5open(filename, "r") do file
        A = read(file, datasetname)
    end
    return A
end

function dummy_A()
    n = 41308
    A = spzeros(n,n)
    diags = 250
    for i in 1:diags
        println(i)
        A = A + spdiagm(i => rand(n - i))
        A = A + spdiagm(i*10 => rand(n - i*10))
        A = A + spdiagm(-i => rand(n - i))
        A = A + spdiagm(-i*10 => rand(n - i*10))
    end

    return A
end

function load_sparse_h5(filepath::String, loc2glob::Bool)
    val = nothing
    irn = nothing
```

```
    jcn = nothing
    rhs = nothing
    l2g = nothing
    # vals = Vector{Float64}()
    # irns = Vector{Int}()
    # jcns = Vector{Int}()
    # rhss = Vector{Float64}()
    # l2gs = Vector{Int}()

    h5open(filepath, "r") do h5file
        data = read(h5file)
        val = data["val"]
        irn = data["irn"]
        jcn = data["jcn"]
        rhs = data["rhs"]
        print(size(rhs))
        if loc2glob
            l2g = data["loc2glob"]
        end

    end

    # Just to make sure to use 64 bit indices
    irn = convert(Vector{Int64},irn)
    jcn = convert(Vector{Int64},jcn)

    m, n = maximum(irn), maximum(jcn)
    A = sparse(irn, jcn, val, m, n)
    return A, rhs, l2g
end

function multi_load_sparse_matrix_from_hdf5(filepaths, loc2glob)
    vals = Vector{Float64}()
    irns = Vector{Int}()
    jcns = Vector{Int}()
    rhss = Vector{Float64}()
    l2gs = Vector{Int}()

    for filepath in filepaths
        val = nothing
        irn = nothing
        jcn = nothing
        rhs = nothing
        l2g = nothing
        h5open(filepath, "r") do h5file
            data = read(h5file)
            val = data["val"]
            irn = data["irn"]
            jcn = data["jcn"]
            rhs = data["rhs"]
            print(size(rhs))
            if loc2glob
                l2g = data["loc2glob"]
            end

        end
```

```
174         append!(vals, val)
175         append!(irns, irn)
176         append!(jcns, jcn)
177 
178         rhss = rhs
179         if loc2glob
180             append!(l2gs, l2g)
181         end
182     end
183 
184     m, n = maximum(irns), maximum(jcns)
185     A = sparse(irns, jcns, vals, m, n)
186 
187     return A, rhss, l2gs
188 end
189 
190 function multi_load_sparse_matrix_from_hdf5(filepaths)
191     return multi_load_sparse_matrix_from_hdf5(filepaths, false)
192 end
193 
194 function collect_LUs(pcs::Vector{SparseMatrixCSC{T, Int64}}) where {T <:
    AbstractFloat}
195     lus = []
196     for pc in pcs
197         println("Taking LU of PC")
198 
199         push!(lus, lu(pc))
200     end
201 
202     return lus
203 end
204 
205 
206 function collect_LUs_dense(pcs::SparseMatrixCSC{T, Int64}) where {T <:
    AbstractFloat}
207     lus = []
208     for pc in pcs
209         println("Taking LU of PC")
210 
211         push!(lus, lu(Matrix(pc)))
212     end
213     return lus
214 end
215 
216 function global_to_locals(global_v, l2gs)
217     local_vs = []
218     for l2g in l2gs
219         push!(local_vs, global_v[l2g])
220     end
221     return local_vs
222 end
223 
224 function locals_to_global(local_vs, l2gs)
225     total_length = sum(length, local_vs)
226     global_v = similar(local_vs[1], total_length)
227 
```

```
228     for (local_v, l2g) in zip(local_vs, l2gs)
229         global_v[l2g] = local_v
230     end
231 
232     return global_v
233 end
234 
235 function solve_non_LU(x::Vector{AbstractFloat}, pcs::Vector{
        SparseMatrixCSC}, l2gs::Vector{Vector{Int32}})
236     local_solutions = []
237     for (pc, l2g) in zip(pcs, l2gs)
238         n = size(pc, 1)
239         x_local = x[l2g]
240         local_sol = pc \ x_local
241         push!(local_solutions, local_sol)
242     end
243 
244     global_solvec = locals_to_global(local_solutions, l2gs)
245     return global_solvec
246 end
247 
248 function solve_pcs(x, pc_LUs, l2gs)
249     # # TODO: LU type
250     local_solutions = []
251     for (pc_LU, l2g) in zip(pc_LUs, l2gs)
252         n = size(pc_LU, 1)
253         x_local = x[l2g]
254         local_sol = pc_LU \ x_local
255         push!(local_solutions, local_sol)
256     end
257 
258     global_solvec = locals_to_global(local_solutions, l2gs)
259     return global_solvec
260 end
261 
262 function load_eigs(filename::String)
263     return readdlm(filename, ',', Complex{Float64})
264 end
265 
266 function global_inverse_simple(n::Int64, pcs::Vector{SparseMatrixCSC{T,
        Int64}}, l2gs::Vector{Vector{Int32}}) where {T <: AbstractFloat}
267     global_inv = Array{Float64}(undef, n,n)
268     # This will take a while.
269     for (pc, l2g) in zip(pcs, l2gs)
270         println("Downscaling matrix")
271         down_pc = downscale_matrix(pc) # May be helpful
272         println("Taking block-wise sub-inverse...")
273         invpc = inv(Matrix(down_pc))
274 
275         global_inv[l2g,l2g] = invpc
276     end
277     return global_inv
278 end
279 
280 function global_inverse(n, pc_LUs, l2gs)
281     global_inv = Array{Float64}(undef, n,n)
```

```
    # This will take a while.
    for (pc_LU, l2g) in zip(pc_LUs, l2gs)
        println("Taking block-wise sub-inverse...")
        invpc = inv(pc_LU)

        global_inv[l2g,l2g] = invpc
    end
    return global_inv # AKA M^{-1}
end


function plot_residuals(residuals, output_dir)
    p = scatter(log10.(residuals), ylabel="log10(residual)", xlabel="
        iteration")

    savefig(p, output_dir * "residuals.png")
end

function save_eig_subset(M, type, typename::String, output_dir::String;
    nev=5)
    # eig_time = now()
    println("Eig decomp of $type")
    decomp, hist = partialschur(M, nev=nev, which=type)
    eigs = decomp.eigenvalues
    println("$typename eigs: $eigs")
    outname = output_dir * "$(typename)_eigs.txt"
    writedlm(outname, eigs)
    println("Saved to $outname")

    return outname
end


function save_eig_subset_kk(M, type, typename, output_dir; nev=NEV)
    println("Eig decomp of $type")
    n = size(M,1)

    eigs, vecs, info = KrylovKit.eigsolve(M, size(M,1), nev, type, Complex
        {Float64})#, type)
    println(info)
    vecs = nothing # Throw these away
    println("$typename eigs: $eigs")
    outname = output_dir * "$(typename)_eigs.txt"
    writedlm(outname, eigs)
    println("Saved to $outname")
    return outname
end

function downscale_pcs(pcs, new_size)
    small_pcs = (pc -> downscale_matrix(pc, new_size)).(pcs)
end

function downscale_matrix(matrix, new_size)
    return convert(SparseMatrixCSC{new_size,Int64}, matrix)
end

```

```
function dense_matrix_size(matrix)
    bytes = sizeof(eltype(matrix))
    return "$(size(matrix,1)^2 * bytes /(2^30)) GB"
end

function lu_decompose_parallel(pcs, l2gs)
    ## Construct LU decompositions:

    # Store each LU on respective processors.
    lu_confirmations = []
    workers_used = []
    println("Queueing LU storage")
    for (i, (pc, l2g)) in enumerate(zip(pcs, l2gs))
        conf = @spawnat i+1 lu_and_store(i, pc, l2g)
        push!(lu_confirmations, conf)
        push!(workers_used, i)
    end
    println("Only used these workers: ", workers_used)

    # (x -> wait(conf)).(lu_confirmations)
    for conf in lu_confirmations
        wait(conf)
    end
end


function lu_and_store(i::Int, pc::SparseMatrixCSC{T, Int64}, l2g::Vector{
    Int32}) where {T <: AbstractFloat}
    println(i, ", Storing lu")
    lu_pc = lu(pc)
    global lu_worker = lu_pc
    global l2g_worker = l2g
    println(i, "Stored")
    return true
end

function solve_pcs_parallel(x, pcs, l2gs)# where {T<: AbstractFloat, T1 <:
     Union{Vector{Float64}, SubArray{Float64, 1, Matrix{Float64}, Tuple{
    Base.Slice{Base.OneTo{Int64}}, Int64}, true}, Vector{AbstractFloat}}}
    local_solutions = []
    promises = Dict()
    i = 0
    for i in 1:length(pcs)
        local_sol = @spawnat i+1 solve_parallel_local(i, x)
        promises[i] = local_sol
    end

    # We now have workers working and promises.
    # Resolve promises.
    for i in 1:length(pcs)
        local_sol = fetch(promises[i])
        push!(local_solutions, local_sol)
    end

    global_solvec = locals_to_global(local_solutions, l2gs)
    return global_solvec
```

```
388 end
389
390 # function solve_parallel_local(i::Int, x::Vector{Float64})
391 function solve_parallel_local(i::Int, x)
392     # Knows its l2g, knows its lu
393     LU = lu_worker #decompositions[i]
394     n = size(LU, 1)
395     l2g = l2g_worker #l2gs[i]
396     x_local = x[l2g]
397     local_sol = LU \ x_local
398     return local_sol
399 end
400
401 function save_eigs(A::SparseMatrixCSC, output_dir::String, tolerance::
    Float64, all_eigs=true)
402     ## Spectrum
403     decomp, hist = partialschur(A, nev=5, which=LM())
404     eigs = decomp.eigenvalues
405     println("Largest eigs: $eigs")
406     writedlm(output_dir * "largest_eigs.txt", eigs)
407
408     decomp, hist = partialschur(A, nev=5, which=LR())
409     eigs = decomp.eigenvalues
410     println("Largest (real) eigs: $eigs")
411     writedlm(output_dir * "largest_real_eigs.txt", eigs)
412
413     decomp, hist = partialschur(A, nev=5, which=LI())
414     eigs = decomp.eigenvalues
415     println("Largest (im) eigs: $eigs")
416     writedlm(output_dir * "largest_im_eigs.txt", eigs)
417
418     decomp, hist = partialschur(A, nev=5, which=SR())
419     eigs = decomp.eigenvalues
420     println("Smallest (real) eigs: $eigs")
421     writedlm(output_dir * "smallest_real_eigs.txt", eigs)
422
423     decomp, hist = partialschur(A, nev=5, which=SI())
424     eigs = decomp.eigenvalues
425     println("Smallest (im) eigs: $eigs")
426     writedlm(output_dir * "smallest_im_eigs.txt", eigs)
427
428     eigs = nothing
429     GC.gc()
430
431     if all_eigs
432         println("Attempting to get all eigs:")
433         eigenvalues, vecs, info = KrylovKit.eigsolve(A, n - 1, tol=
               tolerance, krylovdim=n - 1, verbosity=2);#, nev=n-1, which=:SM)
434         println(info)
435         writedlm(output_dir * "all_eigvals$tolerance.txt", eigenvalues)
436
437         plot_eigs(eigenvalues, output_dir * "all_eigvals$tolerance")
438     end
439 end
440
441 function benchmark(start_time)
```

```
    elapsed = now() - start_time
    total_time = canonicalize(Dates.CompoundPeriod(elapsed))
    println("Total time: $total_time")

    return total_time
end

# Which of the following 2?
function plot_eigs(eigs, filename::String)
    x = real.(eigs)
    y = imag.(eigs)
    p1 = scatter(x, y, ylabel="Imaginary", xlabel="Real", legend=false)
    title!("Complex eigenvalues")
    fname1 = filename *"_scatter.png"
    savefig(p1, fname1)
    println("Saving plot to $(fname1)")

    # Compute the polar coordinates
    r = log10.(abs.(eigs)) # Radial coordinate is the log of the magnitude
        of eigs
    theta = angle.(eigs) # Angular coordinate is the angle of eigs

    # Create a scatter plot in polar coordinates
    p2 = scatter(theta, r, proj=:polar,)
    ticks = [-6,-4,-2,0,2,4,6,8,10,12] # TODO: Make dynamic.

    yticks!(ticks, string.(ticks), ylabel="log(|eigs|)", legend=false)

    title!("Complex Eigenvalues")


    fname2 = filename * "_polar.png"
    savefig(p2,fname2)
    println("Saving plot to $(fname2)")
end

function eigplot(eigs::Union{Vector{Complex}, Matrix{Complex}})
    # Compute the polar coordinates
    r = log10.(abs.(eigs)) # Radial coordinate is the log of the magnitude
        of eigs
    theta = angle.(eigs) # Angular coordinate is the angle of eigs

    # Create a scatter plot in polar coordinates
    p = scatter(theta, r, proj=:polar,)
    ticks = [-6,-4,-2,0,2,4,6,8,10,12] # TODO: Make dynamic.

    yticks!(ticks, string.(ticks), ylabel="log(|eigs|)")

    title!("Eigenvalues")
    return p
end
```

A.2. Loading our system

```
# utils/load_system.jl
include("helpers.jl")
```

```
start = now()

if myid() == 1
        # addprocs(3)
        println("Number of cores: ", nprocs())
        println("Number of workers: ", nworkers())

        for i in workers()
            id, pid, host = fetch(@spawnat i (myid(), getpid(),
                gethostname()))
            println(id, " " , pid, " ", host)
        end
end

println("Total memory: ", Sys.total_memory() / 2^20, " MB")
println("Free memory: ", Sys.free_memory() / 2^20, " MB")

runs = ["199_small", "199_large", "303_modded", "303_inxflow", "303
    _inxflow_3", "303_inxflow_5", "303_reference", "303_zk100", "303_stale"
    , "303_zk200"]
println(ARGS)
run_name = length(ARGS) >= 1 ? ARGS[1] : "199_small" # Default
@assert run_name in runs "ARGS[0] must be a valid run_name name ($runs).
    It is $ARGS"

output_dir = "output/julia/$run_name/"
isdir(output_dir) || mkdir(output_dir)
if length(ARGS) >= 2
        output_name = ARGS[2]
        output_dir = output_dir * "$output_name/"
        isdir(output_dir) || mkdir(output_dir)
end

println("Run: $run_name. Output: $output_dir")

if myid() == 1
        println("Loading matrices")
        A, rhs, n = load_A(run_name)
        pcs, rhss, l2gs = load_preconditioners(run_name)
        println()
end
```

A.3. Solver

```
# parallel_solve.jl
using Distributed
@everywhere include("utils/helpers.jl")
include("utils/load_system.jl")

# Run options:
no_pc_run = true
pc_run = true
downscaled_pc_run = true

# Run with -p 3 or thereabouts.
@assert nworkers() >= size(pcs,1)
```

```
TOL = 1e-12
rtol = 1e-12
atol = 1e-36
memory = 200
restart = false
if restart
    memory = 10
    output_dir = output_dir * "restart-$memory-"
end

struct Tee <: IO
    io1::IO
end

Base.write(t::Tee, x::UInt8) = (write(t.io1, x); write(kstdout, x))
Base.write(t::Tee, x::AbstractString) = (write(t.io1, x); write(kstdout, x
    ));

if myid() == 1
    ## Solve
    if no_pc_run
        println("Raw solve: (should fail to converge)")
        println("Krylov.jl")

        fp = open("$(output_dir)raw_gmres_run.txt", "w")
        io = Tee(fp)
        println("Saving to $(fp.name)")
        x, stats = Krylov.gmres(A, rhs, restart=restart, memory=memory,
            rtol=rtol, atol=atol, itmax=50, verbose=1, history=true,
            iostream=io)#, callback=should_terminate)
        close(fp)
        println(stats)
        writedlm("$(output_dir)raw_residuals.txt", stats.residuals)
        plot_residuals(stats.residuals, output_dir * "raw_")


        println("Krylov.jl BiCGSTAB")
        # io = open("$(output_dir)raw_bicgstab_run.txt")
        io = kstdout
        x, stats = Krylov.bicgstab(A, rhs, rtol=rtol, atol=atol, itmax=50,
             verbose=1, history=true, iostream=io)
        # close(io)
        println(stats)
        writedlm("$(output_dir)bicgstab_residuals.txt", stats.residuals)
        plot_residuals(stats.residuals, output_dir * "raw_bicgstab_")
    end

    if pc_run
        ######################################################
        # Preconditioner setup:
        lu_time = now()
        lu_decompose_parallel(pcs, l2gs)
        println("LU time:")
        benchmark(lu_time)

        # Must be globally scoped.
```

```
        function pc_fn(x::AbstractVector)
            solution = solve_pcs_parallel(x, pcs, l2gs)
            x .= solution # In place
        end

        function pc_fn!(out::AbstractVector, x::AbstractVector)
            out .= solve_pcs_parallel(x, pcs, l2gs)
        end

        LinearOperators.ldiv!(op::Preconditioner, x::AbstractVector) =
           begin pc_fn(x) end
        LinearOperators.ldiv!(y::AbstractVector, op::Preconditioner, x::
           AbstractVector) = begin pc_fn!(y, x) end


        Minv = LinearMap{eltype(A)}((y, x) -> pc_fn!(y, x), size(A,1),
           ismutating=true)

        ##################################################
        println("PC'd run:")

        println("Krylov.jl nil x0")
        fp = open("$(output_dir)pcd_gmres_run.txt", "w")
        io = Tee(fp)
        println("Saving to $(fp.name)")
        x, stats = Krylov.gmres(A, rhs, M=opM, iostream=io, ldiv=true,
           restart=restart, memory=memory, rtol=rtol, atol=atol, itmax
           =200, verbose=1, history=true)#, callback=should_terminate)
        close(fp)
        println(stats)

        writedlm("$(output_dir)pcd_residuals.txt", stats.residuals)
        writedlm("$(output_dir)solution_krylov.txt", x)
        plot_residuals(stats.residuals, output_dir * "PCd_")
        residual = rhs - A * x
        pc_residual = Minv * residual
        println("Residual: $(norm(residual))")
        println("PC residual: $(norm(pc_residual))")
        println("PC residual2: $(norm(Minv * rhs - Minv * A * x))")

        println("Krylov.jl BiCGSTAB")
        x, stats = Krylov.bicgstab(A, rhs, M=opM, ldiv=true, rtol=rtol,
           atol=atol, itmax=50, verbose=1, history=true)
        println(stats)
        writedlm("$(output_dir)pcd_bicgstab_residuals.txt", stats.
           residuals)
        plot_residuals(stats.residuals, output_dir * "PCd_bicgstab_")

    end

    if downscaled_pc_run
        ##########################################################
        small_bits = 32 # 16 or 32
        small_type = Float16
        if small_bits == 32
            small_type = Float32
```

```
        end
        println()
        println("$small_bits-bit PCs:")
        println(small_type)
        println()
        small_pcs = downscale_pcs(pcs, small_type)
        @everywhere lu_worker = nothing # reset just in case
        lu_time = now()
        lu_decompose_parallel(small_pcs, l2gs)
        println("LU time:")
        benchmark(lu_time)

        # Must be globally scoped.
        function small_pc_fn(x)
            solution = solve_pcs_parallel(x, pcs, l2gs)
            x .= solution # In place
        end

        function small_pc_fn!(out, x)
            out .= solve_pcs_parallel(x, pcs, l2gs)
        end

        LinearOperators.ldiv!(y::AbstractVector, op::Preconditioner, x::
            AbstractVector) = begin small_pc_fn!(y, x) end
        LinearOperators.ldiv!(op::Preconditioner, x::AbstractVector) =
            begin small_pc_fn(x) end

        println("Krylov.jl")
        fp = open("$(output_dir)32bit_pcd_gmres_run.txt", "w")
        println("Saving to $(fp.name)")
        io = Tee(fp)
        x, stats = Krylov.gmres(A, rhs, M=opM, iostream=io, ldiv=true,
            restart=restart, memory=memory, rtol=rtol, atol=atol, itmax
            =200, verbose=1, history=true)#, callback=should_terminate)
        println(stats)
        close(fp)
        writedlm("$(output_dir)$(small_bits)_bit_pcd_residuals.txt", stats
            .residuals)
        plot_residuals(stats.residuals, output_dir * "$(small_bits)
            _bit_PCd_")

        println("Krylov.jl BiCGSTAB")
        x, stats = Krylov.bicgstab(A, rhs, M=opM, ldiv=true, rtol=rtol,
            atol=atol, itmax=50, verbose=1, history=true)
        println(stats)
        writedlm("$(output_dir)$(small_bits)_bit_pcd_bicgstab_residuals.
            txt", stats.residuals)
        plot_residuals(stats.residuals, output_dir * "$(small_bits)
            _bit_PCd_bicgstab_")
    end

    benchmark(start)
end
```

A.4. Preconditioned spectra

```julia
# MA_eigs_parallel.jl
using Distributed
@everywhere include("utils/helpers.jl")
include("utils/load_system.jl")

# Run with -p 3 or thereabouts.
@assert nworkers() >= size(pcs,1)
TOL = 1e-9
NEV = 1

lu_decompose_parallel(pcs, l2gs)

function pc_fn!(out, x)
    out .= solve_pcs_parallel(x, pcs, l2gs)
end

M = LinearMap{eltype(A)}((y, x) -> pc_fn!(y, x), size(A,1), ismutating=
    true)

@everywhere function save_eig_subset_kk(M, type, typename, output_dir; nev
    =NEV)
    # eig_time = now()
    println("Eig decomp of $type")
    # x0 = randn(ComplexF64, n)
    eigs, vecs, info = KrylovKit.eigsolve(M, size(M,1), nev, type, Complex
        {Float64}, tol=TOL)#, type)
    println(info)
    vecs = nothing # Throw these away?
    println("$typename eigs: $eigs")
    outname = output_dir * "$(typename)_eigs.txt"
    writedlm(outname, eigs)
    println("Saved to $outname")
    return outname
end

function M_spectrums(M, output_dir)
        filenames = []
        types = [:LM, :LR, :LI, :SR, :SI]
        typenames = ["largest", "largest_real", "largest_im", "
            smallest_real", "smallest_im", "smallest"]
        for i in 1:length(types)
            type = types[i]
            typename = typenames[i]
                eig_time = now()
                fname = save_eig_subset_kk(M, type, typename, output_dir,
                    nev=NEV)

                benchmark(eig_time)
            println("-----------------------")
        println()
        push!(filenames, fname)
        end
        return filenames
end

MA = M*A
```

```
52
53 filenames = M_spectrums(MA, output_dir * "Minv_A_")
54
55 all_eigs = Vector{Complex}()
56 for fname in filenames
57         eigs = load_eigs(fname)
58         global all_eigs = vcat(all_eigs, eigs)
59 end
60
61 output_fname = joinpath(dirname(filenames[1]), "compound_eigs")
62 plot_eigs(all_eigs, output_fname)
63
64 benchmark(start)
```