Università degli Studi di Padova

Ghent University

JOINT RESEARCH DOCTORATE IN FUSION SCIENCE AND ENGINEERING

Cycle XXXIV

New architectures for real-time plasmas

Coordinator: Prof. Fabio Villone

Supervisor at Università degli Studi di Padova (Home University): Gabriele Manduchi

Supervisor at Università degli Studi di Padova (Home University): Prof. Paolo Bettini

Supervisor at Ghent University (Host University): Prof. Kristel Crombé

Ph.D. student: Nicolo Ferron

Padova, January 2022

Abstract

Theoretical and experimental research on magnetic confinement fusion showed that improved operational modes often rely on a large number of diagnostics that accurately reconstruct the plasma shape [1]. At the same time, an increasing number of models are required in the feedback loop to handle signals coming from the diagnostics [2]. As the number of tools to create a model increase, so do the variety of different architectures that a plasma system should be able to run. Models can be written in various ways. However, due to the real-time requirements, the plasma system is often built with different tools. A framework is then needed for the real-time system, and models are to be mapped in such a framework.

In this work new architectures will be built for a generic plasma real-time framework so as to increase the number of available interfaces, and their application to existing frameworks is shown [3]. New components will provide the framework with the tools required to interface with graphical, python and mathematical function models. Components are subsequently tested and added to the framework.

Furthermore, a plasma equilibrium and shape model [4] will be updated in order to improve performances and compatibility with the new framework.

A complete plasma framework is then built and test discharges are run in several modes.

Abstract

Theoretisch en experimenteel onderzoek naar magnetische opsluiting fusie heeft aangetoond dat verbeterde operationele modi vaak afhankelijk zijn van een groot aantal diagnostische gegevens die nauwkeurig de vorm plasmavorm [1]. Tegelijkertijd is er een toenemend aantal modellen nodig in de terugkoppellus om signalen te verwerken die afkomstig zijn van de diagnostiek [2].

In dit werk zullen nieuwe architecturen worden gebouwd voor een generiek plasma real-time raamwerk om het aantal beschikbare interfaces, en hun toepassing op bestaande raamwerken wordt getoond [3].

Bovendien, een plasma model [4] zal worden bijgewerkt om de prestaties en de compatibiliteit met het nieuwe raamwerk.

Een compleet plasma raamwerk wordt dan gebouwd en testontladingen worden in verschillende modi uitgevoerd.

Contents

1 Introduction **7**

1.1 Structure of this work 7
1.2 Energy production 8
1.2.1 Fusion energy 8
1.2.2 Triple product 10
1.3 Fusion concepts 12
1.3.1 Particle motion 12
1.3.2 Plasma current 14
1.3.3 Toroidal forces 16
1.3.4 Vertical forces 19
1.3.5 Ellipticity and triangularity 20
1.4 Magnetic system 20
1.4.1 Toroidal field coils 20
1.4.2 Poloidal field system 21

2 Interface components **25**

2.1 The need for interfaces 25
2.2 Talking with graphical models 26
2.2.1 A framework component to run the library 27
2.2.2 Component configuration 27
2.2.3 Component classes 29
2.2.4 Component functions 33
2.2.5 Cycle time 37
2.3 Talking with modules 39
2.3.1 A framework component to run an interpreted programme 39
2.3.2 Writing the interpreted programme 40
2.3.3 Component configuration 44

2.3.4 Component classes . . . 46
2.3.5 Component functions . . . 47
2.3.6 Cycle time . . . 51
2.4 Talking with mathematics . . . 51
2.4.1 A framework component to run mathematical functions . 52
2.4.2 Component configuration . . . 53
2.4.3 Component functions . . . 53
2.4.4 Cycle time . . . 55

3 Review of equilibrium and shape system 59
3.1 General outline of the model . . . 60
3.1.1 Model inputs . . . 60
3.1.2 Model outputs . . . 67
3.2 Equilibrium and shape elements . . . 67
3.2.1 Plasma current element . . . 67
3.2.2 Plasma position element . . . 73
3.2.3 Plasma shape element . . . 81
3.2.4 Additional elements . . . 83
3.3 Model library . . . 84
3.4 Updates to the model . . . 85
3.4.1 Circular and single-null configurations . . . 85
3.4.2 Structured inputs and outputs . . . 86
3.4.3 Model summary . . . 86

4 Final design of the equilibrium framework 87
4.1 Plasma system outline . . . 87
4.1.1 Components . . . 87
4.1.2 Inputs . . . 88
4.2 Outputs . . . 89
4.2.1 RFP circular discharge . . . 90
4.2.2 Single null discharges . . . 90

5 Conclusions 106

List of Figures

1.1 ΔE per nucleon . . . 9
1.2 Triple product . . . 11
1.3 Toroidal geometry . . . 13
1.4 Particle motion in toroidal field . . . 13
1.5 Mangetic field in toroidal geometry . . . 14
1.6 $\nabla_{\perp} B$ drift . . . 15
1.7 Electric field due to $\nabla_{\perp} B$ drift . . . 16
1.8 Toroidal field coils . . . 21
1.9 Poloidal field coils . . . 22
1.10 Magnetizing windings . . . 23
1.11 Interaction between poloidal field sources . . . 23

2.1 Graphical programming outline . . . 27
2.2 Example of simple model . . . 28
2.3 Simple model step time in framework and model . . . 38
2.4 Equilibrium and shape model step time in framework and model 39
2.5 Sum and multiplication module step time in framework and model 52
2.6 Step time in math function component 1 . . . 56
2.7 Step time in math function component 2 . . . 56
2.8 Step time in math function component 3 . . . 56
2.9 Step time in math function component 4 . . . 57
2.10 Step time in math function component 5 . . . 57

3.1 General outline of shape and equilibrium model . . . 60
3.2 Calculation of shell horizontal shift . . . 63
3.3 Outline of the plasma current management element . . . 68
3.4 Plasma position element . . . 74
3.5 Plasma position feedforward element . . . 75

3.6 Plasma position lead compensator element 76
3.7 Effect of lead compensator on I_P 77
3.8 Plasma position feedback element 78
3.9 Vertical shift stabilization 79
3.10 Effect of bias field of PI element 81
3.11 Ellipticity element 82
3.12 Triangularity element 82
3.13 FS voltage drop compensation 84

4.1 A plasma real-time system 88
4.2 Toroidal coil currents in RFP 91
4.3 FS coil voltages in RFP 92
4.4 SC coil currents in RFP 93
4.5 Toroidal coil currents in single null 94
4.6 FS coil voltages in single null 95
4.7 SC coil currents in single null 96
4.8 Toroidal coil currents in single null 97
4.9 FS coil voltages in single null 98
4.10 SC coil currents in single null 99
4.11 Toroidal coil currents in single null with $\Delta\beta_\theta$ 100
4.12 FS coil voltages in single null with $\Delta\beta_\theta$ 101
4.13 SC coil currents in single null with $\Delta\beta_\theta$ 102
4.14 Toroidal coil currents in single null with V_{MW} variation 103
4.15 FS coil voltages in single null with V_{MW} variation 104
4.16 SC coil currents with V_{MW} variation 105

List of Tables

2.1 Cycle times 38
2.2 Cycle times 38
2.3 Math function cycle times 58

3.2 Inputs and outputs of the equilibrium and shape model 62

Listings

2.1 Model wrapping component classes 29
2.2 component step function 36
2.3 Interpreted programme header 40
2.4 Adding elements 42
2.5 A `p_module` implementation 43
2.6 Module configuration example 45
2.7 Function wrapping module classes 46
2.8 Function wrapping module classes firstline 49
2.9 Step function 50

Chapter 1

Introduction

1.1 Structure of this work

The goals of this work are:

1. building a wrapper component for a plasma system that runs graphical models within the plasma system

2. building a wrapper component for a plasma system that interfaces with python modules and runs said modules within the plasma system

3. building a wrapper component for a plasma system that implements mathematical functions and runs said functions in the plasma system

4. updating an equilibrium and shape model [4] for circular and single null discharges

5. using the new components and the equilibrium and shape model to build a plasma system

The components are generalized so that they can run models with various number, type and dimension of input and output signals, constants, architectures and tools. Furthermore, they can be easily adapted to work *with most plasma frameworks* and have indeed already been used in such a way [3][5].

In *chapter 1* an introduction on magnetic confinement fusion is given. After a brief summary of energy sources and the purpose of fusion, basic fusion concepts are given and the main parts of a fusion machine are described.

The wrapper components are built in *chapter 2*, where why such interfaces are needed is also shown. Each component design, configuration and performances is described and examples of usage are given.

Chapter 3 focuses on the description of an equilibrium and shape model for circular and single null discharges [4]. The model is updated to reflect future improvements in the diagnostics [6].

Finally, *chapter 4* describes how all components and the equilibrium and shape model are put together to build a plasma system and test discharges are run to confirm the results.

1.2 Energy production

Though contribution of renewable sources (such as photovoltaic and wind, among many others) to energy sources is increasing, in the past decades a constant increase in the energy demand has been observed.

In the long-term, the development of an energy source that meets the following requirements is considered worthwhile:

1. abundance of the fuel required to power it
2. no production of CO_2

A good candidate is magnetic confinement fusion.

1.2.1 Fusion energy

Fusion features the merging of light elements, such as hydrogen (H) and deuterium (D). While this happens in stars due to gravity, making the same happen in a power plant is not easy.

In an element of mass number A the number of neutrons is denoted by N and the number of protons is denoted by Z, where $A = N + Z$. The mass of the nucleus m_A, however, is smaller than the sum of the masses of N and Z:

$$m_A < Nm_N + Zm_Z$$

The mass defect $\Delta m = Nm_N + Zm_Z - m_A$ is converted into the energy ΔE required to hold these particles together. The average amount of binding energy per nucleon is then $\frac{\Delta E}{A}$. This quantity is not constant and varies for each element (see fig. 1.1).

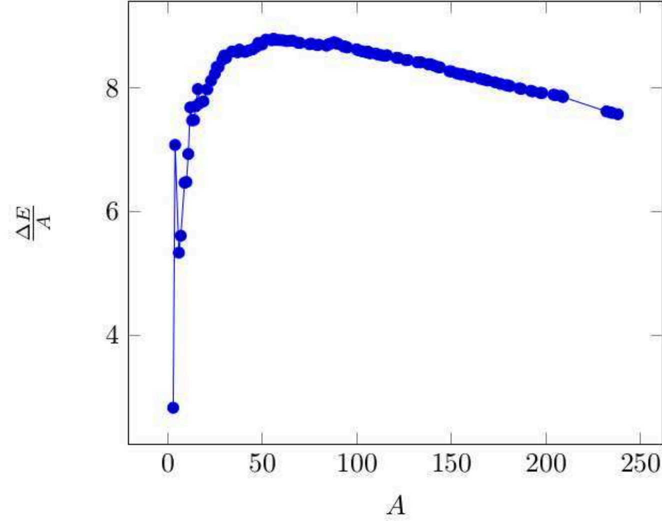

Figure 1.1: ΔE per nucleon ($\frac{\Delta E}{A}$) vs. mass number A.

The curve reaches a maximum around $A = 56$ (Fe). If elements with $\frac{\Delta E_1}{A_1}$ are merged to form an element with $\frac{\Delta E_2}{A_2} > \frac{\Delta E_1}{A_1}$ then energy is given off. This process can be described by an equation in the form:

$$A_1 + A_2 + \cdots + A_n \to B_1 + B_2 + \cdots + B_m + E$$

where $A_1, \ldots, A_n$ are the starting elements, $B_1, \ldots, B_m$ are the products and E is the energy that is given off in the process (in MeV).

The amount of energy can be calculated as:

$$\frac{E}{c^2} = (m_{A_1} + m_{A_2} + \cdots + m_{A_n}) - (m_{B_1} + m_{B_2} + \cdots + m_{B_m})$$

H has only one proton, so no ΔE can be obtained. The lightest element that can be considered for fusion is then D.

D-T merging is one of the most fitting due to the process being easy to initiate. The corresponding equation is:

$$\mathrm{D} + \mathrm{T} \to \mathrm{He}^4 + \mathrm{n} \tag{1.1}$$

Considering the masses of the involved elements:

$$
\begin{aligned}
m_D &= 3.342 \times 10^{-24}\,\text{kg} \\
m_T &= 5.007 \times 10^{-24}\,\text{kg} \\
m_{\text{He}} &= 6.645 \times 10^{-24}\,\text{kg} \\
m_n &= 1.675 \times 10^{-24}\,\text{kg}
\end{aligned}
\tag{1.2}
$$

the resulting masses can be calculated:

$$
\begin{aligned}
m_R &= m_D + m_T = 8.351 \times 10^{-24}\,\text{kg} \\
m_P &= m_{\text{He}} + m_n = 8.320 \times 10^{-24}\,\text{kg}
\end{aligned}
\tag{1.3}
$$

The mass defect is then $\Delta m = 0.031 \times 10^{-24}\,\text{kg}$, givin off $\Delta E = \Delta m \cdot c^2 = 2.75 \times 10^{-12}\,\text{J} = 17.2\,\text{MeV}$. This energy is given off in the form of kinetic energy associated to the end products. By assuming that the kinetic energy of the products is much greater than that of the starting elements, energy is inversely proportional to the masses. Since $m_{\text{He}} \approx 4m_n$:

$$
\begin{aligned}
E_{\text{He}} &= 3.6\,\text{MeV} \\
E_n &= 14.1\,\text{MeV}
\end{aligned}
\tag{1.4}
$$

the latter being the energy that can be used

1.2.2 Triple product

Let D and T be two elements of density n_D and n_T respectively. The number of interactions per unit volume per unit time is given by:

$$
g = n_D n_T < \sigma v > \tag{1.5}
$$

where $< \sigma v >$ is the product of cross section and velocity averaged on the particle velocity distribution. By using the energy in eq. 1.4 the power per unit volume can be calculated as:

$$
P_{\text{th}} = g \Delta E \tag{1.6}
$$

The maximum rate is achieved for $n_D = n_T = \frac{n}{2}$, where n is the total ion density. Thus

$$
P_{\text{th}} = \frac{n^2}{4} < \sigma v > \Delta E \tag{1.7}
$$

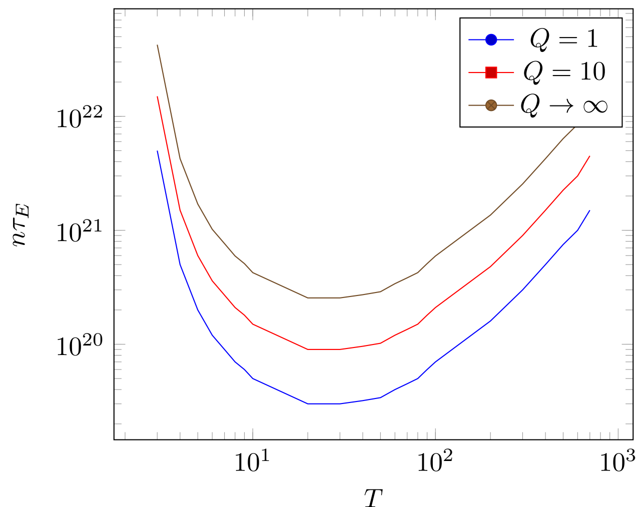

Figure 1.2: The curves describe the points of breakeven ($Q = 1$), the points of ignition ($Q \to \infty$) and an intermediate condition ($Q = 10$).

Charged particles remain inside the vessel thus contributing to plasma heating with $P_\alpha = \frac{n^2}{4} < \sigma v > \Delta E_{\mathrm{He}}$. Part of the heat P_{out} leaves the plasma due to heat transfert towards the outside of the vessel. The additional power that must be provided to the plasma is then:

$$P_{\mathrm{add}} = P_{\mathrm{out}} - P_\alpha \tag{1.8}$$

By considering only the power leaving the vessel due to transport (conduction and convection), calculated as $P_c = \frac{3nT}{\tau_E}$, P_{add} is:

$$P_{\mathrm{add}} = \frac{3nkT}{\tau_E} - \frac{n^2}{4} < \sigma v > \Delta E_{\mathrm{He}} \tag{1.9}$$

with τ_E the characteristic time of P_c. The *plasma energy gain factor* is then

$$Q = \frac{P}{P_{\mathrm{add}}} = \frac{P_n + P_\alpha}{P_{\mathrm{add}}} \tag{1.10}$$

Breakeven is achieved when $P_{\mathrm{add}} = P$ ($Q = 1$). If P_α is high $P_{\mathrm{add}} = 0$, and thus $Q \to \infty$ (ignition) (see fig. 1.2).

Eq. 1.9 is a function of n, τ_E and T only. Thus, a *triple product* can be defined as the product of these three quantities. For fusion to sustain itself and to have net energy production it must be:

$$n\tau_e T \geq 1.2 \times 10^{21}\,\mathrm{keVsm}^{-3} \tag{1.11}$$

or, in other words,

$$\begin{aligned} n &\approx 1 \times 10^{20}\,\mathrm{m}^{-3} \\ \tau_E &\approx 1\,\mathrm{s} \\ T &\approx 20\,\mathrm{keV} \end{aligned} \tag{1.12}$$

1.3 Fusion concepts

In magnetic confinement fusion the plasma (D and T) is expected to stay inside a vessel.

1.3.1 Particle motion

No material vessel is suitable to sustain temperatures such as that in eq. 1.12, since:

1. the vessel would be gradually worn away by the interaction with the plasma
2. the vessel, due to the same interaction with the plasma, would give off its coating into the plasma itself. This phenomenon is know as *sputtering* and causes the plasma to cool down.

The plasma is then magnetically enclosed, so that there is no interaction (or very few of it) with the vessel wall. The vessel itself is toroidally shaped so that field lines closes while still remaining inside the torus. The toroidal reference frame is shown in fig. 1.3.

Field coils are placed on the toroidal surface and a toroidal magnetic field is produced inside the torus. Plasma particles move along the magnetic field gyrating around its field lines due to the elecromagnetic force $\mathbf{F} = q\mathbf{v} \times \mathbf{B}$, thus remaining inside the toroidal vessel (see fig. 1.4). The gyro radius is defined as:

$$r_g = \frac{mv_\perp}{qB} \tag{1.13}$$

where m is the particle mass, $v_\perp$ is the particle velocity component perpendicular to the magnetic field B, and q is the particle charge.

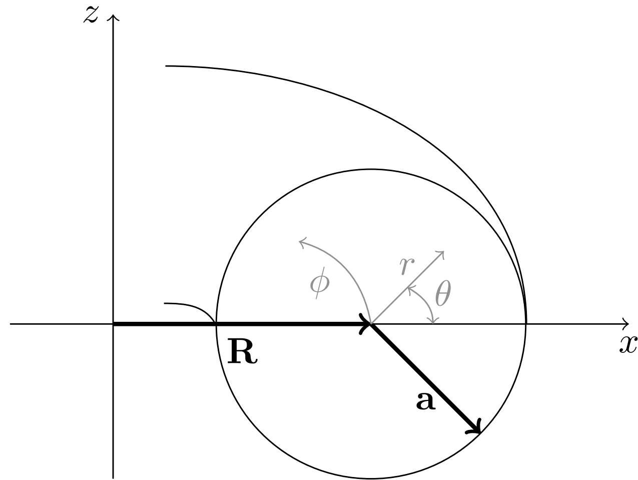

Figure 1.3: A toroidal reference frame (r, ϕ, θ). $\mathbf{R}$ is the *major radius*, $\mathbf{a}$ is the *minor radius*

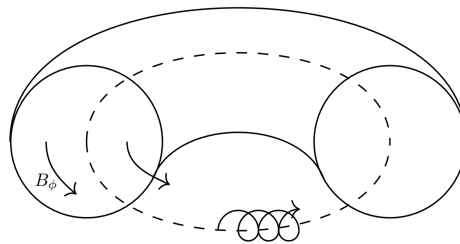

Figure 1.4: Particle motion in a toroidal magnetic field. The particle moves along a magnetic field gyrating around a field line with a radius proportional to its mass and inversely proportional to the magnetic field.

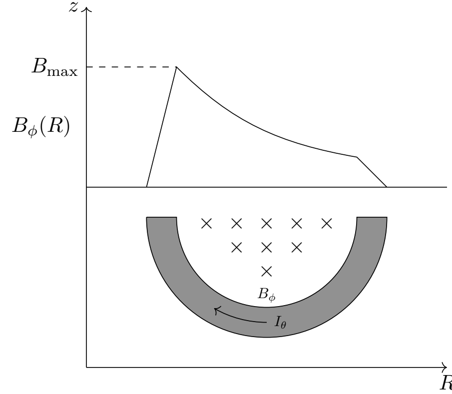

Figure 1.5: Outline of the toroidal magnetic field inside a toroidal coil.

The poloidal magnetic field B_ϕ inside a toroidal coil is described by eq. 1.14 ans shown in fig. 1.5.

$$B_\phi = \frac{\mu_0 N I}{2\pi R} \tag{1.14}$$

Since each particle is gyrating around a constant field line, the magnetic field is higher towards the centre of the torus and lower towards the outer side. Thus, the particle gyro radius is continually variating as shown in fig. 1.6, causing ions to drift towards the bottom of the vessel and electrons to drift towards the top of the vessel ($\nabla_\perp B$ *drift*). This generates in turn a vertical electric field (see fig. 1.7).

The addition of a perpendicular electric field adds a constant drift velocity $\mathbf{v}_D$ to the gyro motion, known as $\mathbf{E} \times \mathbf{B}$ *drift*:

$$\mathbf{v}_D = \frac{\mathbf{E} \times \mathbf{B}}{B^2} \tag{1.15}$$

This drift is not proportional to the particle mass or charge, is perpendicular to both **E** and **B** and makes particles shift towards the outer wall of the torus.

1.3.2 Plasma current

A solution is found in the induction of a current in the plasma. The current generates a poloidal magnetic field B_θ which, in combination with the toroidal field B_ϕ gives helicoidal field lines. The particles in their motion are thus going through both high and low sides of the field, averaging the gradient effect. There

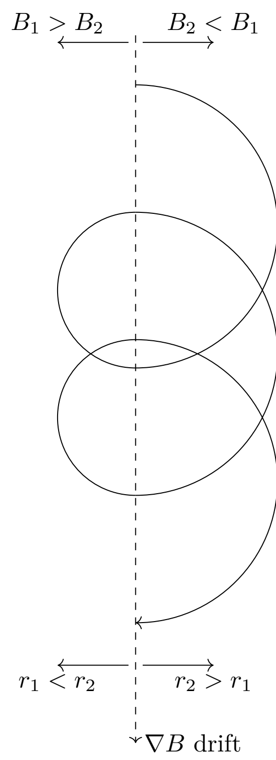

Figure 1.6: $\nabla_\perp B$ drift.

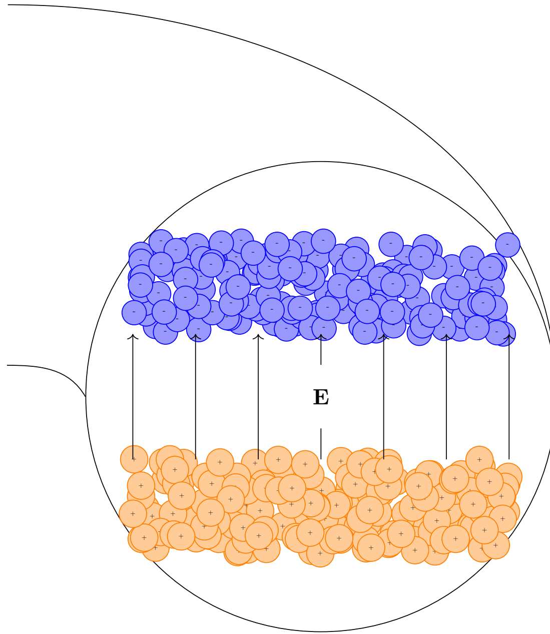

Figure 1.7: Electric field due to $\nabla_{\perp} B$ drift

is no build up of charge at the top and at the bottom of the plasma, and the $\mathbf{E} \times \mathbf{B}$ effect is greatly reduced.

The plasma current is generated with the *magnetizing windings* (see section 1.4.2.2).

Let $\Delta\theta_k$ be the angle between the poloidal position of a particle during transit k and $k+1$ of the same poloidal cross-section. The *rotational transform* ι is defined as:

$$\iota = \lim_{N \to \infty} \frac{1}{N} \sum_{k=1}^{N} \Delta\theta_k \tag{1.16}$$

The same concept is often referred as the *q factor*, defined as:

$$q = \frac{2\pi}{\iota} \tag{1.17}$$

1.3.3 Toroidal forces

In this section the 3 toroidal forces generated in a toroidal plasma are shown. These 3 forces are due to the toroidal shape and are all directed outward on the

horizontal plane of the torus. The toroidal forces are:

1. the *tire tube force*, due to the kinetic pressure
2. the *hoop* force, due to magnetic pressure
3. the $\frac{1}{R}$ *force*, due to magnetic pressure and toroidal field

The toroidal forces must be compensated by an additional force generated by a vertical magnetic field. The required magnetic field is then calculated.

1.3.3.1 Tire tube force

The same plasma pressure p is applied to both the inner and outer half of the plasma column (S_{in} and S_{out}). However, due to the toroidal shape of the plasma, $S_{\text{in}} < S_{\text{out}}$. The net force on each surface is then

$$F_{\text{in}} = pS_{\text{in}} < F_{\text{out}} = pS_{\text{out}} \tag{1.18}$$

resulting in a net force F_s pointing outward. The value of F_p applied to the whole plasma volume is

$$F_p =< p > \frac{\mathrm{d}V}{\mathrm{d}R_0} =< p > \frac{\mathrm{d}}{\mathrm{d}R_0}\left[(\pi a^2)(2\pi R_0)\right] =< p > (2\pi^2 a^2) \tag{1.19}$$

By definition it is $\beta_\theta = \frac{<p>}{\frac{B_\theta^2}{2\mu_0}}$, thus

$$F_p = (2\pi^2 a^2)\beta_\theta \frac{B_\theta^2}{2\mu_0} \tag{1.20}$$

When the plasma aspect ratio is $\frac{R_0}{a} >> 1$ then $\frac{B_\theta^2}{2\mu_0} = \frac{\mu_0 I_P^2}{8\pi^2 a^2}$, and the toroidal force is then

$$F_p = \beta_\theta \frac{\mu_0 I_P^2}{4} \tag{1.21}$$

1.3.3.2 Hoop force

This force points outwards and is produced by the current flowing in a circular loop. In this case the plasma current is circulating in the toroidal plasma column.

The magnetic flux Ψ produced by the plasma current around the plasma column is poloidally the same, but the associated field lines are denser on the

inner side of the torus, resulting in a greater magnetic field B_{in} on the inner side with respect to the magnetic field on the outer side B_{out}. The magnetic pressure $B_{\text{in}}S_{\text{in}}$, pointing outward, is then greater then the magnetic pressure $B_{\text{out}}S_{\text{out}}$ pointing inward, resulting in a net force F_θ pointing outwards.

The outward force due to a current flowing in a circular loop is

$$F_\theta = \frac{1}{2} I_P^2 \frac{\mathrm{d}L}{\mathrm{d}R_0} \tag{1.22}$$

The inductance can be split into $L = L_i + L_e$. It can be shown that

$$\begin{aligned} L_i &= \frac{1}{2}\mu_0 R_0 \ell_i \\ L_e &= \mu_0 R_0 (\ln \frac{8R_0}{a} - 2) \end{aligned} \tag{1.23}$$

where ℓ_i is a parameter that can be calculated for various geometries. F_θ is then

$$F_\theta = F_\theta^{\text{in}} + F_\theta^{\text{out}} = \frac{1}{2}\mu_0 I_P^2 \left[\ln \frac{8R_0}{a} - 1 + \frac{\ell}{2} \right] \tag{1.24}$$

1.3.3.3 $\frac{1}{R}$ force

This force is due to the toroidal field being proportional to $\frac{1}{R}$ in toroidal geometry. The toroidal field coils produce the magnetic field

$$B_\phi^{\text{out}}(r) = \frac{\mu_0 I_c}{2\pi R} \tag{1.25}$$

By assuming that the plasma current flows on a thin layer on the plasma surface the magnetic field produced by the plasma current inside the plasma column partially cancels the field applied by the coils:

$$B_\phi^{\text{in}}(r) = \frac{\mu_0 (I_c - I_P)}{2\pi r} < B_\phi^{\text{out}}(r) \tag{1.26}$$

The magnetic field difference on the plasma surface $(B_{\text{out},1}^2 - B_{\text{in},1}^2)S_{\text{in}} > (B_{\text{out},2}^2 - B_{\text{in},2}^2)S_{\text{out}}$ produces a net force F_ϕ that, assuming a large aspect ratio $\frac{R_0}{a} >> 1$, can be evaluated as

$$F_\phi = 2(\pi a)^2 \left[\frac{B_\phi^2(a)}{2\mu_0} - \frac{< B_\phi^2 >}{2\mu_0} \right] \tag{1.27}$$

1.3.3.4 Vertical field

The overall outward toroidal force is then

$$F_T = F_p + F_\theta + F_\phi = \frac{1}{2}\mu_0 I_P^2 \left[\ln\frac{8R_0}{a} - \frac{3}{2} + \frac{\ell}{2} + \beta_\theta\right] \tag{1.28}$$

let Λ be $\Lambda = \frac{\ell}{2} + \beta_\theta - 1$:

$$F_T = \frac{1}{2}\mu_0 I_P^2 \left[\ln\frac{8R_0}{a} + \Lambda - \frac{1}{2}\right] \tag{1.29}$$

The compensating force F_V is

$$\mathbf{F_T} + \mathbf{F_V} = 0 \tag{1.30}$$

and the horizontal force produced by a vertical field B_V can be evaluated as:

$$\mathbf{F_V} = -I_P B_V 2\pi R_0 \mathbf{R} \tag{1.31}$$

From eq. 1.30 and eq. 1.31 the required vertical field can be calculated as

$$B_{V,\mathrm{eq}} = \frac{\mu_0 I_P}{4\pi R_0} \left[\ln\frac{8R_B}{a} + \Lambda - \frac{1}{2}\right] \tag{1.32}$$

The required vertical field is generated with the *field shaping coils* (see section 1.4.2.1).

1.3.4 Vertical forces

The cross section of a plasma with large aspect ratio tends to be circular. However, to maximize the plasma volume inside the vessel vertically elongated plasmas have to be used. Plasma elongation is generated by appling additional forces to the upper and lower plasma surfaces with zero total force:

$$\Delta\mathbf{F}_U + \Delta\mathbf{F}_L = 0 \tag{1.33}$$

Forces are applied to the plasma using a quadrupole field. The quadrupole field produces a radial field B_R, and with this configuration any slight shift of the plasma centroid from the quadrupole field centre produces a difference between $\Delta\mathbf{F}_U$ and $\Delta\mathbf{F}_L$, and the plasma tends to move upwards or downwards.

The vertical displacement is measured and the poiloidal field system (see section 1.4.2.1) produces the horizontal field required to compensate (see section 3.2.2.2).

1.3.5 Ellipticity and triangularity

Ellipticity is defined as the ratio between the major and minor axis of an ellipse.

Let $r(\theta)$ be the plasma radius, defined as the distance between the plasma boundary and the vacuum vessel centre at the angle θ. The plasma radius function can also be seen as the sum of its harmonics, and in such case the amplitudes of the 2nd harmonics in $\cos\theta$ and $\sin\theta$ express the ellipticity of the plasma shape.

Triangularity is defined as:

$$\delta = (R_{\max} - R_0)/a \tag{1.34}$$

Triangularity is expressed as the amplitude of the 3rd harmonic in $\cos\theta$.

1.4 Magnetic system

In a plasma system measures of relevant plasma parameters are taken over time. Some of those measures are directly used in feedback loops, while other feedback signals are elaborated from the measures that are directly available (see section 3.1.1).

The discharge is driven by comparing the feedback signals (both measured and calculated) with preset feedforward references, and evaluating the input for the actuators that is required to compensate the difference (the *system response*).

The magnetic system of a fusion machine generates the toroidal, poloidal and vertical components of the magnetic field.

1.4.1 Toroidal field coils

The *toroidal field coils* (TF) are structured as a series of coils that poloidally wind up the vessel as shown in fig. 1.8, and generate the toroidal component of the magnetic field B_ϕ. Ions and electrons gyrate around the field lines of B_ϕ.

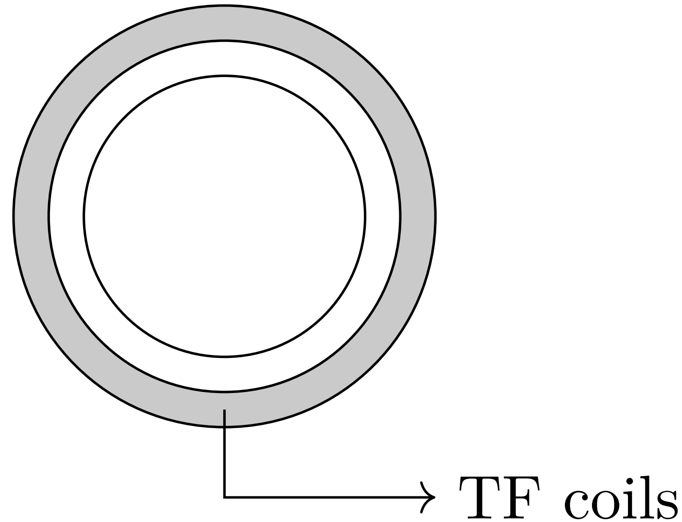

Figure 1.8: Outline of toroidal field coils (draft).

1.4.2 Poloidal field system

The *poloidal field system* is a set of windings producing the poloidal component of the magnetic field. They include:

- the field shaping coils
- the magnetizing winding

1.4.2.1 Field shaping coils

The *field shaping coils*, FS are one of the poloidal field windings and are structured as a series of coils that toroidally wind up the vessel as shown in fig. 1.9, and generate the poloidal component B_θ of the magnetic field for plasma positioning and shaping. They are composed of:

1. radial equilibrium coils: these coils generate the vertical field B_V (see section 1.3.3)
2. vertical equilibrium coils: these coils generate the horizontal field B_H (see section 1.3.4)
3. shape coils: these coils generate the field that gives the plasma the required shape (see sections 1.3.5 and 1.3.5).

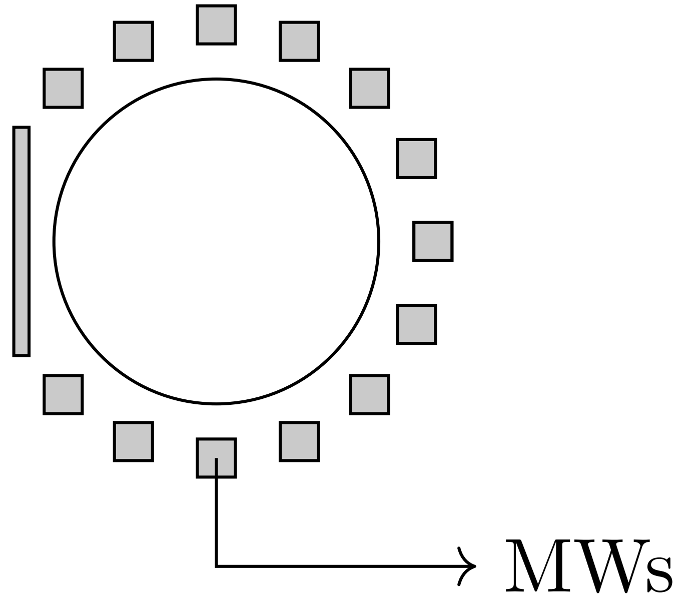

Figure 1.9: Outline of poloidal field coils (draft).

1.4.2.2 Magnetizing winding

The magnetizing winding (MW) induce a current I_P in the plasma by generating a flux variation $\Delta\Phi$. The current in turns generates the poloidal field B_θ required by the $\nabla_\perp B$ and $\mathbf{E} \times \mathbf{B}$ drifts.

To do so, the MW is structured as a series of coils placed in the center of the toroidal vessel, in concentric position with respect to the z-axis and the plasma column (fig. 1.10).

The field produced by the MW is not used for equilibrium, thus the field lines should not go through the plasma. This can be achieved in two ways:

1. the coils wind up on a ferromagnetic core. Field lines flow inside the core and do not go thorugh the plasma (fig. 1.10a)

2. the coils wind up in air. Additional coils (concentric to the z-axis) are required so that the field lines do not close in the plasma (fig. 1.10b)

Toroidal and poloidal field coils are orthogonal and do not interact with each other. The MW on the other hand is coupled with the other poloidal field sources: the field shaping coils (FS) and the plasma (as shown in fig. 1.11) [7].

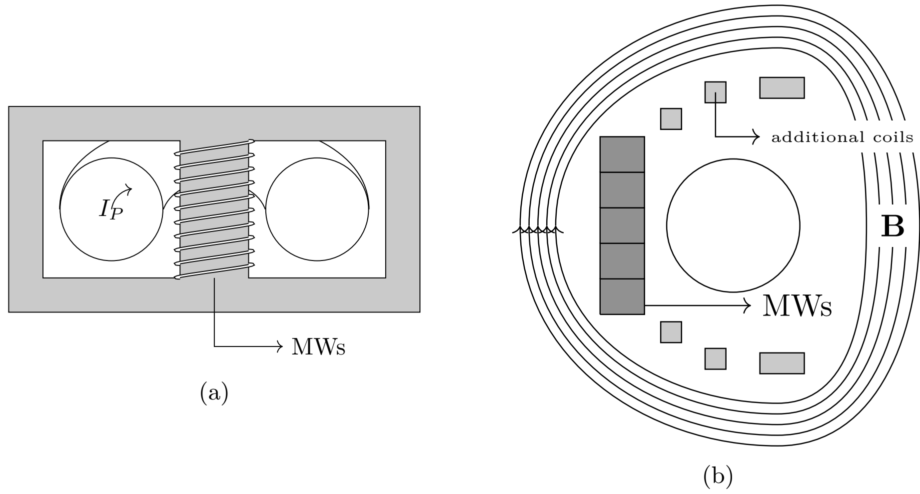

Figure 1.10: Outline of the magnetizing windings (MWs). (a) MWs with ferromagnetic core, where flux flows inside the core (b) MWs with additional coils.

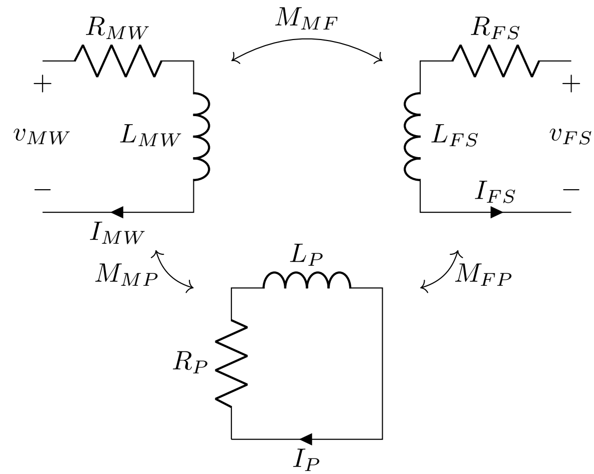

Figure 1.11: Interaction between poloidal field sources.

The plasma circuit is then characterized by the following:

$$V_P = M_{MP}\frac{\mathrm{d}I_{MW}}{\mathrm{d}t} + M_{FP}\frac{\mathrm{d}I_{FS}}{\mathrm{d}t} + L_P\frac{\mathrm{d}I_P}{\mathrm{d}t} + R_P I_P \tag{1.35}$$

To have the required I_P flowing in the plasma the magnetizing windings must produce a flux varaition $\Delta\Phi$ that can be calculated integrating eq. 1.35.

Chapter 2

Interface components

2.1 The need for interfaces

Following the steadily growing number of fusion applications and algorithms, the variety of tools used to implement them has also increased. For example, using graphical programming tools is becoming more and more common, and so is using visual modeling. On the other hand, high level languages such as Python are of increasingly importance in plasma equilibrium and shape models.

Interoperability between systems can be achieved in various ways. For example, a graphically programmed equilibrium model could be translated to C so as to run it in a C-based framework. However, this kind of approach may introduce subtle differences with respect to the source. To achieve interoperability in the best way possible the framework should then be able to natively run as many external models as possible, regardless of the programming language they are written with.

To do so, the framework should

1. provide common communication interfaces between the framework and the model that needs to be run
2. be able to link different models coming from different sources

Task no. 2 is a feature of the selected framework. Here task no. 1 will be discussed and pertinent framework modules (*components*) will be developed.

The most common situations are the following:

1. a model is turned into C code or into a library

2. the model programming tools make available an interpreter that the framework should use to directly run the source model (this is typical for high level programming languages)

In the former case, model inputs and outputs should be available to the framework as pointers, while in the latter an additional step is required.

By using a generic interface component per each source of external models the process is also more reliable, since checks can be carried out for the converter (or the interpreter) and for the framework interface instead of for each new model that needs to be interfaced.

In this section three *wrappers* will be built, that is, three framework components that can be instantiated by the framework and can in turn instantiate and run a model created

1. in a graphical programming environment
2. with an interpreted language
3. with mathematical functions

To run such models the wrapper component should:

- load the model
- compare the input and output layout of the model with the one configured for the framework, in order to make sure that the framework and the loaded model can talk to each other
- transcribe the framework inputs and outputs to the model
- transcribe the model outputs to the framework

How this is done for each of the three cases above is shown in the following sections. Note that

2.2 Talking with graphical models

A graphical model is the description of a system with its inputs and outputs in terms of graphical elements connected by lines. Each element represents a primitive operation within the system, while lines represent the flow of inputs and outputs from one element to another. Graphical models offer a convenient

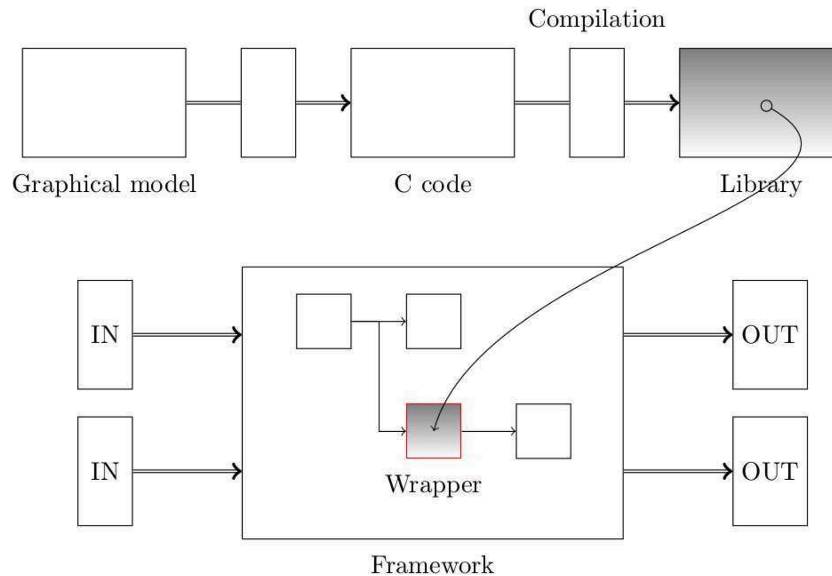

Figure 2.1: Steps required to turn a graphical model into a library and use it into the framework. The component built in this chapter is the wrapper. The framework runs each component once per cycle.

way to represent fusion models because of its simplicity and readability. Use of graphical modelling is thus steadily increasing and even a whole equilibrium and shape model can be represented with it (see chapter 3). Hence the need of a wrapper to include graphical models into the framework.

A graphical model can then be turned into a library[1] as shown in fig. 2.1.

2.2.1 A framework component to run the library

Here a framework component (*model wrapping component* or *component*) is built that is able to load and run a graphical model in the form of a library.

2.2.2 Component configuration

The component should be configurable to match the layout of the model inputs and outputs. The component is thus given the following configuration options:

`library_name` the file name of the generated library

[1] A library is a programme that can be run by another programme.

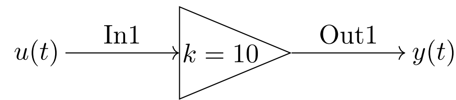

Figure 2.2: An example of a simple multiplication model.

`model_name` the name of the model. This name is used internally to refer to the model

`inputs` and `outputs` a list of the inputs and outputs from and to the framework. Names should be the same as those used in the model

The component is also given some optional options:

`constants` a list of attribute-value pairs of the model internal parameters whose value should be updated before running the model

`constant_source` the name of the object providing the values for model constants, should they not be declared in the previous `constants` list

The component can be configured with

1. a list of attribute-value pairs, whose name must match those listed above
2. the `inputs` and `outputs` options showing the layouts of inputs and outputs
3. the `constants` option for model constant values

An example configuration for a model such as that in fig. 2.2 is given below in an attribute-value pair that shall be read by the component:

```
{
library = "model.so";
model_name = "model";
inputs = {
  in = {
    type = "double";
    elements = "1";
    };
  };
outputs = {
  out = {
    type = "double";
```

```
    elements = "1";
    };
  };
constants = {
    k = "5";
  };
};
```

Note that, with respect to the $k = 10$ gain constant of fig. 2.2, the k value here is set to 5. This means that, during the model initialisation (see section 2.2.4.1) the value of the parameter will be changed to 5. This lets the same model to be used not just for cases when the required k is 10, but also all other cases, provided that the wrapping component `constants` section specifies a different value for the k constant.

2.2.3 Component classes

During initialisation, the component retrieves from the model the layout of model inputs, outputs and constants. These parameters are put in three instances of the class `model_signals_and_constants`, shown in example 2.1.

```
class model_signals_and_constants
{
  char* name;
  unsigned int type;
  void *model_ptr;
  void *module_ptr;
  unsigned int n_elems;
  unsigned long int size;
  bool transpose;

  bool transpose_and_transcribe(void *destination, void *
    source);
}

class model_inputs
{
  bool transcribe();
}

class model_outputs
```

```
20 {
21   bool transcribe();
22 }
23
24 class model_constants
25 {
26   bool transcribe();
27 }
28
29 bool model_inputs::transcribe()
30 {
31   bool retval = (model_ptr != NULL);
32   retval &= (module_ptr != NULL);
33
34   if (!transpose)
35   {
36     write(model_ptr, module_ptr, size);
37   }
38   else
39   {
40     transpose_and_transcribe(model_ptr, module_ptr);
41   }
42
43   return retval;
44 }
```

Listing 2.1: The `model_signals_and_constants` class, with type and dimensionality of model inputs, outputs and constants.

This class contains 6 elements and 2 methods. The elements contain the signal (or constant) type and dimensionality, as well as its total size (that is, the size of its type times the number of its elements). Element `model_ptr` is set to where the model reads inputs and constants from or writes outputs to, while `module_ptr` is what the framework will use for its version of the same signal (or constant).

The method `transcribe` transcribes, as the name suggests, the signal (or constant) value from the component to the model (for inputs and constants) and from the model to the model (for outputs). Since the method has a different usages, each subclass has its own implementation of it. The method writes the signal as it is or uses `transpose_and_trascribe`, depending on the current require-

ments for the signal (or constant) being transcribed. `model_inputs::transcribe` is shown in example 2.1. The method `transpose_and_trascribe` is used by the `transcribe` method when the signal (or constant) requires transposition. However, this method is expected to significantly slow down the `step` function.

2.2.3.1 Structured inputs and outputs

In some applications, inputs and outputs are expected to be structured , that is, a nested structure of signals with logical or physical relation. The model wrapping component uses linearisation to deal with structured inputs and outputs: instead of a structure, a one-dimensional array is created, each element of which represents one of the structured signal elements. For example, a structure like the following:

```
- structure_1
  |- input_1
  |- input_2
- structure_2
  |- input_3
  |- sub_structure
     |- input_4
     |- input_5
```

is described in the component configuration as follows:

```
{
inputs = {
  structure_1 = {
    input_1 = {
      type = "double";
      elements = "1";
    };
    input_2 = {
      type = "float";
      elements = "2";
    };
  };
  structure_2 = {
    input_3 = {
      type = "int";
```

```
      elements = "1";
    };
    sub_structure = {
      input_4 = {
        type = "double";
        elements = "1";
      };
      input_5 = {
        type = "float";
        elements = "8";
      };
    };
  };
};
outputs = {
  out = {
    type = "double";
    elements = "1";
  };
};
constants = {
  k = "5";
};
};
```

but gets linearised in the `model_inputs` array as:

```
model_inputs[0]: structure_1.input_1
model_inputs[1]: structure_1.input_2
model_inputs[2]: structure_2.input_3
model_inputs[3]: structure_2.sub_structure.input_4
model_inputs[4]: structure_2.sub_structure.input_5
```

The array can then be included in a loop to use all the structured signal elements.

2.2.3.2 Enumerations

An enumeration is a list of numbers in which each number is associated to a label. Since lists are a discrete object, only integer numbers can be associated to labels. An example of enumeration is shown below:

```
{
1: "On";
2: "Off";
}
```

The model can then use the flag `On` to refer to the value `1`.

The model wrapping component maps enumeration inputs and outputs to their corresponding integer value, and `On` must be referred to as `1`.

2.2.4 Component functions

The model wrapping component has two main functions:

`initialisation` **function** , which is run only once and performs all the preparatory actions such as model loading and instantiation

`step` **function** , which is run continually while the framework is running. This function effectively runs the model library, provides it with inputs and receives its outputs

Inputs for the model come from other framework components, and outputs are sent to other framework components as well.

2.2.4.1 `initialisation` function

The component `initialisation` function performs the following steps:

1. read the component configuration options
2. load the model library using the parameters retrieved from the configuration options
3. use the library handles to get model number of inputs, outputs and constants
4. create input, output and constant arrays and use library handles to retrieve their size
5. make sure that inputs and outputs configured in the component options correspond to the model inputs and outputs in type, size and dimensionality

6. update model constants using the values declared in the component configuration options

The model wrapping component is initialised by reading the configuration options. An example of the configuration syntax is shown in section 2.2.2

Once the component options have been read from the configuration the model library can be loaded. The library should have two main functions (`inst_func` for initialisation and `step_func` for running) whose handles must be retrieved to run the model. These handles are put into pointers so that they can be recalled later.

```
void* (*inst_func)(void);
bool (step_func)(void);

inst_func = (void*(*)(void))(library->get_function("inst_func"));
step_func = (bool(*)(void))(library->get_function("step_func"));
```

The library can now be instantiated by calling the `inst_func`:

```
model_struct = (*inst_func)();
```

The layout of `model_struct` is constant and known to the model wrapping component. The configuration of the model (that is, its number of inputs, outputs and constants) is thus retrieved.

These values are checked against what was set in the component configuration, with an error being issued if any incongruity is found. If none is found, these values are used to allocate three related arrays of objects `model_inputs`, `model_outputs` and `model_constants` in which the layout of inputs, outputs and constants used in the model is put.

Constants are then retrieved using the structure provided by the model (`const_struct`) and a recursive function `get_constants`:

```
const_struct = model_struct->model_constants;
for (unsigned int i = 0; i < n_constants; i++)
{
    get_constants(i, const_struct);
}
```

The function recalls itself when encountering a structured constant to descend the tree.

In a very similar fashion, the `model_inputs` and `model_outputs` arrays are filled. Firstly, the relevant structure is retrieved from the model library, then the `get_signals` recursive function is called in a loop going from the first to the last signal. This is done for both inputs and outputs, and results are put in the related array of structures.

```
input_struct = model_struct->model_inputs;
output_struct = model_struct->model_outputs;
for (unsigned int i = 0; i < n_inputs; i++)
{
    get_signals(i, input_struct, model_inputs);
}
for (unsigned int i = 0; i < n_outputs; i++)
{
    get_signals(i, output_struct, model_outputs);
}
```

Since inputs and outputs must be declared with their type and dimensionality in the component initial settings, the corresponging parameters that have been retrieved with `get_signals` and `get_constants` must be looked over and compared to what was declared in the component initial settings, thus making sure that the `transcribe` method will work properly.

While the model is running (see section 2.2.4.2) its inputs are transcribed from the framework to the model. Conversely, outputs are transcribed from model to the framework.

In general, a single model can be reused in various ways provided that its constants can be changed. While this can be achieved by modifying the model and recompile the library anew, this step is not necessary and constant values can be modified on the go.

Firstly, each model constant is matched with a constant in the component configuration section `constants`. If a match exists, the value `model_constants[i]->module_ptr` is updated. Then, using the parameter `model_constants[i]->model_ptr` and `model_constants[i]->size` the value of the constant can be updated by the wrapping component using the class method `model_constants::transcribe`:

```
bool model_inputs::transcribe()
{
    bool retval = (model_addr != NULL);
    retval &= (module_addr != NULL);

```

```
6       if (!transpose)
7       {
8           write(model_addr, module_addr, size);
9       }
10      else
11      {
12          transpose_and_transcribe(model_addr, module_addr);
13      }
14
15      return retval;
16  }
```

2.2.4.2 `step` function

The `step` function performs the following steps:

1. update model signal values using the framework signal values
2. run the model `step_func` function
3. update the framework signal values using the model signal values

`step` uses the `transcribe` method illustrated in example 2.1.

```
1  bool step()
2  {
3    bool retval = (mf_struct != NULL);
4
5    for (unsigned int idx = 0; (idx < n_inputs) && retval; idx
       ++)
6    {
7      retval = model_inputs[idx]->transcribe();
8    }
9
10   if (retval)
11     retval = (*step_function)();
12
13   for (unsigned int idx = 0; (idx < n_outputs) && retval;
       idx++)
14   {
15     retval = model_outputs[idx]->transcribe();
16   }
```

```
17
18   return retval;
19 }
```

Listing 2.2: component step function

2.2.5 Cycle time

In this section the time required to run a whole cycle of a model is measured by running both the graphical programmed models and the libraries created from them.

The duration of a whole cycle is measured by adding the current time to an array τ. At the end of the desired number of cycles, the duration of each cycle can be calculated as the difference between each pair of consecutive values in the array:

$$\Delta t_{\mathrm{mdl}} = \tau(t+1) - \tau(t)$$

The framework, on the other hand, offers a time calculation function allowing to measure the duration of each cycle and output it as just another signal of the framework.

These measurements have been carried out for a simple model and for the whole RFP equilibrium and shape model. Comparisons are shown in the following sections. Note that for larger models (as in section 2.2.5.2) the average time is more similar among the examples shown.

2.2.5.1 A simple model

The model in fig. 2.2 has been turned into a library and used in the framework.

The model completes 1000 cycles, each cycle time is measured and an average value is calculated.

A comparison between cycle times with the graphical model and with the framework-run library is shown in fig. 2.3 and in table 2.1. The model takes, on average, 199.3 µs to complete a whole cycle. The same model is approximately 40 times as fast when run by the framework, taking on average 4.330 µs per cycle.

Max	Min	Average
27.0 µs	4.0 µs	4.330 µs

Table 2.1: Cycle times

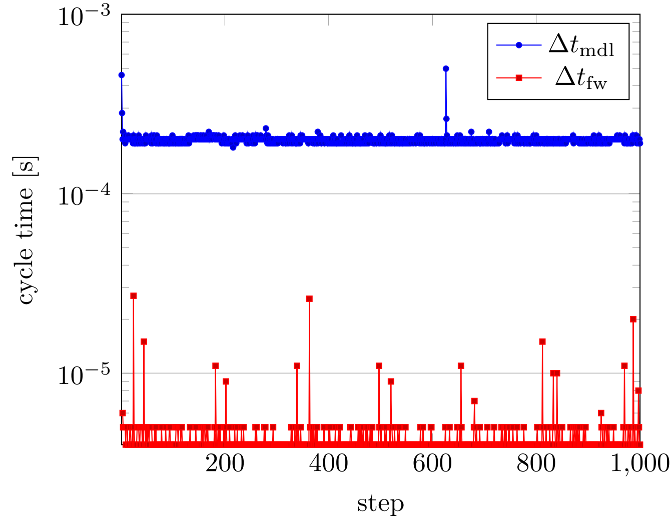

Figure 2.3: Time required to run a cycle of the model in fig. 2.2 (Δt_{mdl}) and the same model as a library loaded in the framework (Δt_{fw}), logarithmic scale. Time is measured for 1000 consecutive cycles.

2.2.5.2 Equilibrium and shape model

The equilibrium and shape model described in chapter 3 has been turned into a library and used in the framework.

The model completes a whole discharge, adding up 3000 steps.

A comparison between cycle times with the graphical model and with the framework-run library is shown in fig. 2.4 and in table 2.2. The model takes, on average, 397.27 µs to complete a whole cycle as a graphical model. The same model is approximately 10 times as fast when run by the framework, taking on average 42.160 µs per cycle.

Max	Min	Average
68.9 µs	37.0 µs	42.160 µs

Table 2.2: Cycle times

Results shown that 97% of the lines are covered and outputs from all the

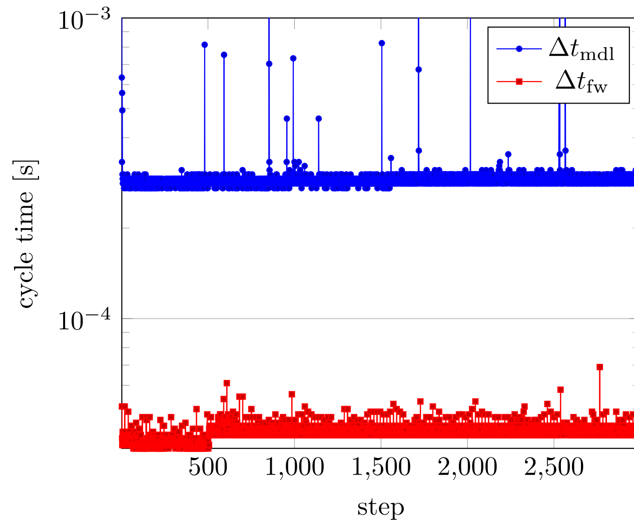

Figure 2.4: Time required to run a cycle of the equilibrium and shape model (see chapter 3) (Δt_{mdl}) and the same model as a library loaded in the framework (Δt_{fw}), logarithmic scale. Time is measured for 3000 consecutive cycles.

required functions is as expected.

2.3 Talking with modules

Python is often used as a modeling tool in physics. Being able to quickly include Python functions into the framework without the need to translate it into C is convenient since:

1. a programme can be inserted into the framework straightforwardly, possibly with the sole addition of some boilerplate lines
2. there is no need of translating the code into C, which is often a source of errors and may result in the programme initially not working as expected

2.3.1 A framework component to run an interpreted programme

Here a framework component (*module wrapping component* or *component*) is built that will be able to load and run a (properly formatted) Python function.

2.3.2 Writing the interpreted programme

In order to be run by the component, the interpreted programme must include:

1. a `p_init` function to be run once when the programme is first loaded

2. a `p_step` function to be run continually

The `p_` suffix is used not to confuse these functions with the component methods `initialise`. The programme should also include the number, type and dimensionality of the `p_step` function inputs and outputs in a structure that the framework component will read. All the constants, whose value should be assignable from the framework, should be in a list as well. Format of said lists is shown in example 2.3.

```
inputs = [
  ('in1', int, (1,1)),
  ('in2', float, (1,1))
]

outputs = [
  ('out', float, (1,1))
]

constants = [
  ('a', int, (1,1), [1]),
  ('b', float, (1,2), [1.1, 2.2])
  ('c', int, (2,2), [[1,2],[3,4]]),
]
```

Listing 2.3: The interpreted programme should declare these three structures.

An additional module to be included (`p_component` module, see section 2.3.2.1) provides functions to append elements to these lists and makes the constants in the `constants` list available to the module functions as elements of a structure `const`. For example, constant `a` of example 2.3 can be referred to as `const.['a']`, like in the example below. Similarly, inputs and outputs are referred as `ins.['in1']` and `outs.['out1']`.

```
outs.['out1'] = ins.['in1']*const['a'][1]
```

The framework works with scalars, vectors and matrices, thus parameters up to $N \times M$ can be used.

2.3.2.1 `p_component` module

The function to be wrapped should include a `p_component` module. This module shall provide a set of functions the framework component will need in order to figure out the layout of the `p_step` inputs, outputs and constants.

The module includes:

1. the `inputs`, `outputs` and `constants` list
2. a set of functions to read the lists and see if all required fields are present and their values are of the required type (e.g., all `value` fields must be numeric)
3. a set of functions to return the entries of the lists to the wapping module
4. a set of functions to create the lists programmatically

These functions are shown below. Note that:

1. exceptions are not included for brevity
2. no additional module is used for simplicity, although for example floating point numbers and arrays would be easier to use with an appropriate module

To add an element the `p_component` module provides the functions `add_input`, `add_output` and `add_constant`:

```
add_input(["in1", int, (1,1)])
add_output(["out", float, (1,1)])
add_constant(["a", int, (1,2), [1,2]])
```

These functions take the following inputs:

1. name of the element (input, output or constant)
2. type of the element
3. dimensionality of the element (up to 2×2)
4. only for constants: the constant default value (scalar or array)

The `add` functions are shown in example 2.4:

```python
def add_input(arg_in):

  ok = 0

  if arg_ok(arg_in) == 1:
    inputs.append(tuple(arg_in))
    ok = 1

  return ok

def add_output(arg_in):

  ok = 0

  if arg_ok(arg_in) == 1:
    outputs.append(tuple(arg_in))
    ok = 1

  return ok
```

Listing 2.4: Adding elements

These functions use the `arg_ok` and `const_ok` functions to make sure that the format of the input elements correspond (in type, size and dimensionality) to what the module expects. For example, the `arg_ok` function must make sure that:

1. the input argument is a list (or compatible)
2. the input argument has exactly 3 elements
3. said elements are, in order, a `str`, a `type` and a `list` element
4. the `list` element is a 2×2 list (for parameters up to matrices)
5. the `list` elements are of type `int`

The `const_ok` function make sure that the value of a constant is:

1. either numeric or a list
2. if it's a list, it can contain either a number (scalar), more than one number (vector) or two more lists (matrix)

The module provides the following interfaces to the framework:

1. `get` functions for the number of inputs, outputs and constnats

2. `get` functions for the type and dimensionality of inputs, outputs and constants

3. `get` functions for input, output and constant values

4. `set` function for constant values

The module then initialises itself by creating the structures that can be used in the programme to retrieve inputs, outputs and constant values.

A programme that the component is then able to load and run has the following outline.

Firstly, the `p_module` module must be loaded. Then, the `add_inputs` and `add_outptus` functions of the `p_module` are to be used to add the required inputs and outputs to the programme. These functions take a list as input, which in turn has the signal name, type and dimensionality as elements:

```
p_module.add_input(["in", float, [1,1]])
p_module.add_output(["out", float, [2,2]])
```

Constants are added in a similar way but a default value of the constant is also required.

```
p_module.add_constant(["a", float, [1,1], 10])
```

Then, an `init` function is required. This function may contain optional preparatory calculations, but is required to call the `p_module.p_init` function which makes available the `ins`, `outs` and `const` structures to the `step` function.

The `step` functions uses said structures to calculate outputs from inputs and constants.

Suppose a programme is needed that takes two inputs and carries out the following calculation:

$$y(z) = ku_1(z) + u_2(z)$$

The corresponding module will look like the following:

```
import p_module as p

p.add_input(["in1", float, [1,1]])
```

```
4  p.add_input(["in2", float, [1,1]])
5
6  p.add_output(["out", float, [1,1]])
7
8  p.add_constant(("k", int, [1,1]), 10)
9
10 def init():
11   p.p_init()
12
13
14 def step():
15   p.outs["out"]=p.const["k"]*p.ins["in1"]+p.ins["in2"]
```

Listing 2.5: A `p_module` implementation

Line by line, this is what the module does:

- line 1: load the `p_module`
- lines 3-4: add the required inputs, in this case 2 scalar `float` inputs
- line 6: add the required output
- line 8: add the gain constant with a default value of 10
- lines 10-11: initialisation function
- lines 14-16: step function

2.3.3 Component configuration

The component should be configurable to match the layout of the function in terms of inputs, outputs and constants. The component is thus given the following configuration options:

`file_name` the name of the file containing the function that should be used

`function_name` the name of the function that the present module is required to run

`inputs` **and** `outputs` a list of inputs and outputs from and to the framework. Names should be the same as those used in the function input parameters and return values

The module also has some optional options:

`constants` a attribute-value list of the model internal parameters whose value should be updated before running the model

`output_check` can be 1 or 0. If 1, the type of the output value is checked (see below)

The component can be configured with

1. a list of attribute-value pairs, whose name must match those listed above
2. the `inputs` and `outputs` options showing the layouts of inputs and outputs
3. the `constants` option for model constant values

An example configuration for a multiplication function (such as that of example 2.5) is given in example 2.6.

```
{
  file_name = "sum.py";
  function_name = "sum";
  inputs = {
    in1 = {
      type = "double";
      elements = "1";
    };
    in2 = {
      type = "double";
      elements = "1";
    };
  };
  outputs = {
    out = {
      type = "double";
      elements = "1";
    };
  };
  constants = {
    k = "5";
  };
```

```
};
```

Listing 2.6: An example configuration of the module to include a multiplication function in the framework.

Note that with respect to the $k = 10$ constant of example 2.5, the k value here is set to 5. If properly configured the module during initialisation (see section 2.3.5.1) is capable of changing the value of the parameter from 10 to 5. This lets the same module to be used not just for cases when the required k is 10, but also all other cases, provided that the wrapping module `constants` section specifies a different value for the k constant.

2.3.4 Component classes

This module retrieves from the function type and dimensionality of model inputs, outputs and constants. These parameters are put in three instances of the class `function_signals_and_constants` shown in example 2.7.

```
class module_signals_and_constants
{
  char* name;
  unsigned int type;
  void *func_ptr;
  void *module_ptr;
  unsigned int n_elems;
  unsigned long int size;
}

class module_outputs : public module_signals_and_constants
{
  bool transcribe();
}

class module_constants : public module_signals_and_constants
{
  bool transcribe();
}

void module_inputs::transcribe()
{
  bool retval = (func_ptr != NULL);
```

```
24   retval &= (module_ptr != NULL);
25
26   if (retval)
27   {
28     write(func_ptr, module_ptr, size);
29   }
30
31   return retval;
32 }
```

Listing 2.7: The `function_signals_and_constants` class, with type and dimensionality of model inputs, outputs and constants.

This class contains 6 elements and 2 methods. The elements contain the signal (or constant) name, type and dimensionality, as well as its total size (that is, the size of its type times the number of elements). Element `model_ptr` is set to where the model reads inputs and constants from or writes outputs to, while `module_ptr` is what the framework will use for its version of the same signal (or constant). The method `transcribe` transcribes, as the name suggests, the signal (or constant) value from the module to the function (for inputs and constants) and from the function to the model (for outputs). Since the method has different usages, each subclass has its own implementation of it. The method is basically a loop over all signal elements and is called after the function is run to update the signal values in the framework with the values of the function outputs.

2.3.5 Component functions

The module wrapping component has two main methods:

`init` method which is run only once and performs all the preparatory actions such as function loading

`step` method which is run continually while the framework is running. This method effectively runs the function providing it with inputs and receiving its outputs

Inputs for the function come from other framework components, and outputs are sent to other framework components as well.

2.3.5.1 `initialisation` function

The component `initialisation` function performs the following steps:

1. read the component configuration options (as described in section 2.3.3)
2. load the interpreted function
3. use the module interfaces (section 2.3.2.1) to get function number of inputs, outputs and constants
4. create input, output and constant arrays and use module interfaces to retrieve their size
5. make sure that inputs and outputs configured in the component options correspond to the module inputs and outputs in type, size and dimensionality
6. update module constants using the values declared in the component configuration options
7. call the module `p_init` function

The module is initialised by reading the configuration options.

Once the module options have been read from the configuration the interpreter is loaded and using it both the `p_module` and the programme are loaded as well.

While loading, the programme module runs all the `add_input`, `add_output` and `add_constant` functions, thus building the list of inputs, outputs and constants.

Then, a handle to the module `p_init` function is retrieved and used to call the function.

The interfaces of the module can now be used to retrieve the module configuration, so the module number of inputs, outputs and constants are retrieved.

These values are used to allocate three arrays of objects `model_inputs`, `model_outputs` and `model_constants`.

Size is calculated as the product of type size and number of elements. These values are compared with those set in the component configuration options.

While the function is running (see section 2.3.5.2) its inputs are transcribed from the framework to the module. Conversely, outputs are transcribed from the module to the framework.

In general, a single module can be reused in various ways provided that the values of its constants can be changed. While this can be achieved by modifying the module itself, the module wrapping component has an interface to do this.

Firstly, each module constant is matched with a constant in the component configuration section `constants`. If a match exists, the value `model_constants[i]->component_ptr` is updated. Then, using the parameter `model_constants[i]->model_ptr` and `model_constants[i]->size` the value of the constant can be updated by the wrapping component using the class method `model_constants::transcribe`.

```
class module_signals_and_constants
{
  char* name;
  unsigned int type;
  void *func_ptr;
  void *module_ptr;
  unsigned int n_elems;
  unsigned long int size;
}

class module_outputs : public module_signals_and_constants
{
  bool transcribe();
}

class module_constants : public module_signals_and_constants
{
  bool transcribe();
}

void module_inputs::transcribe()
{
  bool retval = (func_ptr != NULL);
  retval &= (module_ptr != NULL);

  if (retval)
  {
    write(func_ptr, module_ptr, size);
  }

  return retval;
```

```
32 }
```

Listing 2.8: Function wrapping module classes firstline

2.3.5.2 `step` function

This function updates inputs from the framework, run the module `p_step` function and updates the outputs to the framework performing the following steps:

1. update module inputs using the framework signal values
2. run the module `p_step` function
3. update the framework signal values using the model outputs

```
bool step()
{
  bool retval = (step_function != NULL);

  for (unsigned int idx = 0; (idx < n_inputs) && retval; idx
    ++)
  {
    retval = model_inpus[idx]->transcribe();
  }

  if (retval)
    retval = p_call(step_function);

  for (unsigned int idx = 0; (idx < n_outputs) && retval;
    idx++)
  {
    retval = model_outpus[idx]->transcribe();
  }

  return retval;
}
```

Listing 2.9: Step function

2.3.6 Cycle time

In this section the time required to run a whole cycle of a module is measured by running both the module itself and the module run by the wrapping component.

The duration of a cycle of the model itself is measured by adding, for each cycle, the current epoch time to an array τ. At the end of the desired number of cycles, the duration of each cycle can be calculated as the difference between each pair of consecutive values in the array:

$$\Delta t_{\text{mdl}} = \tau(t+1) - \tau(t)$$

The framework, on the other hand, offers a time calculation function allowing to measure the duration of each cycle and output it as just another signal of the framework.

These measurements have been carried out for a simple sum and multiplication module. Comparisons are shown in the following sections.

2.3.6.1 Sum and multiplication

The example module in example 2.5 can be used in an external programme or in the framework.

The module completes 1000 cycles, each cycle time is measured and an average value is calculated.

A comparison between cycle times with the module and with the framework-run module is shown in fig. 2.5. The function takes, on average, 114.9 µs to complete a whole cycle as a module. The same function is approximately 8 times as fast when run by the framework, taking on average 14.792 µs per cycle.

2.4 Talking with mathematics

Models are most of the times formulated in terms of mathematical functions or systems of functions. Before they can be used in a framework, such models often need to be turned into programmin language, hence the need of a wrapper component to include mathematical functions into the framework.

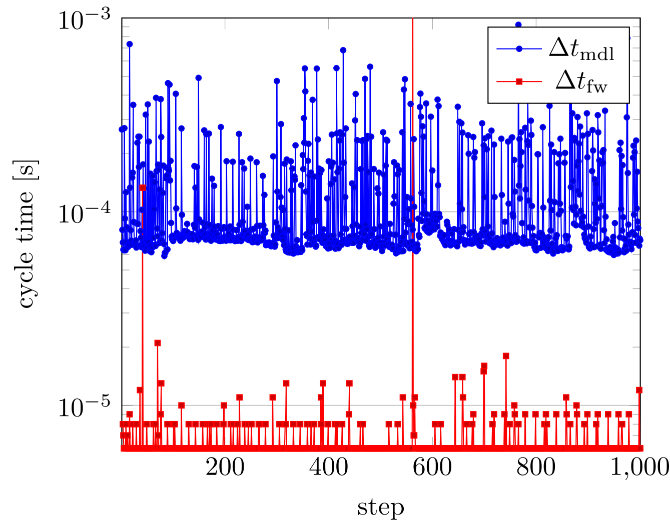

Figure 2.5: Time required to run a cycle of a sum and multiplication function as a module (Δt_{mdl}) and as a interpreted module in the framework (Δt_{fw}), logarithmic scale. Time is measured for 1000 consecutive cycles.

2.4.1 A framework component to run mathematical functions

In this section a framework component (*math function component* or *component*) is built that will be able to load and run a math functions or systems of math functions.

A framework library or an additional library can be used to calculate the result of mathematical function in postfix notation. The math function component is then required to:

1. read a math function in infix notation
2. translate the infix notation into postfix notation
3. match the variables on the right-hand side of each function to the component inputs, and those on the left-hand side to the component outputs
4. read the component inputs and use the framework library to calculate the value of the outputs

2.4.2 Component configuration

The math function component should be configurable with just one parameter, the `functions` parameter. The `functions` parameter is a string of single mathematical functions separated by a semicolon.

The mathematematical function for multiplying an input by a constant k is

$$\begin{aligned} k &= 5 \\ y &= k \cdot x \end{aligned} \tag{2.1}$$

The component can be configured to run a function such as that of eq. 2.1:

```
{
functions = "k = 5; y = k*x;"
inputs = {
  x = {
    type = "double";
    elements = "1";
    };
  };
outputs = {
  y = {
    type = "double";
    elements = "1";
    };
  };
};
```

The value of the constants is given into the `functions` string as one of the functions ($k = 5$ in this case).

The component will match each left-hand side variable of the functions with the wrapper outputs and each right-hand side variable of the functions with the wrapper inputs.

2.4.3 Component functions

The math function component has two main functions:

`initialisation` function which is run only once and performs all the preparatory actions such as `functions` string loading

`step` **function** which is run continually while the framework is running. This function effectively runs the math functions, provides it with inputs and receives its outputs

Inputs for the model come from other framework components, and outputs are sent to other framework components as well.

2.4.3.1 `initialisation` function

The component `initialisation` function performs the following steps:

1. read the configuration options
2. turn the infix expression to postfix expression
3. initialise the framework library that will calculate the result of the input `functions`

Once the module options have been read from the configuration the function is turned into postfix notation by a `postfix` library that has been written specifically for this component. The postfix function is then used to instantiate the `postfix_calc` object provided by the postfix calculation library by passing the postfix functions to it.

```
postfix_func = new postfix();

ret = (postfix_func != NULL);
if (ret)
{
  postfix_func->set_function(functions);
  postfix_func->to_infix();

  output_calc = new postfix_calc(postfix_func->get_infix());
}

ret = output_calc->read_variables();
```

After calling `output_calc->read_variables()` the `postfix_calc` object methods shall be used to set the type and location of each variable in `functions`. The location is the same the framework uses, so that there is no need to transcribe the values of the component inputs to the `postfix_calc` object input values.

```
for (unsigned int i = 0; i < n_inputs && ret; i++)
{
  ret = output_calc->set_input_variable_type(inputs[i]->name
    , inputs[i]->type);
  ret = output_calc->set_input_variable_addr(inputs[i]->name
    , inputs[i]->addr);
}
for (unsigned int i = 0; i < n_outputs && ret; i++)
{
  ret = output_calc->set_output_variable_type(outputs[i]->
    name, inputs[i]->type);
  ret = output_calc->set_output_variable_addr(outputs[i]->
    name, inputs[i]->addr);
}

ret = output_calc->init();
```

If any of the inputs set in the component configuration are not among the `functions` string variables the component stops. If all variables are matched with the corresponding component signal the `postfix_calc` object is initialised. After that, the `postfix_calc->step()` method can be used to calculate the `functions` results, which are directly available at the component output signal locations.

2.4.3.2 `step` function

The math function component `step` function is only calling the `postfix_calc->calculate()` method:

```
ret = output_calc->calculate();
```

2.4.4 Cycle time

In this section the time required to run a whole cycle of a math function component is measured. The framework offers a time calculation function allowing to measure the duration of each cycle and output it as just another signal of the framework. The duration of 1000 cycles is measured and the average duration is calculated for each function in tab. 2.3. Fig. 2.6 to fig. 2.10 shows the duration of each cycle while running the corresponding function.

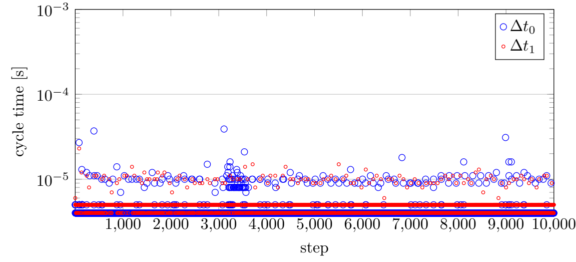

Figure 2.6: Multiplication step time in math function component while running function 1 of tab. 2.3 (Δt_1) compared to the framework cycle time without the math function component (Δt_0), logarithmic scale. Time is measured for 1000 consecutive cycles.

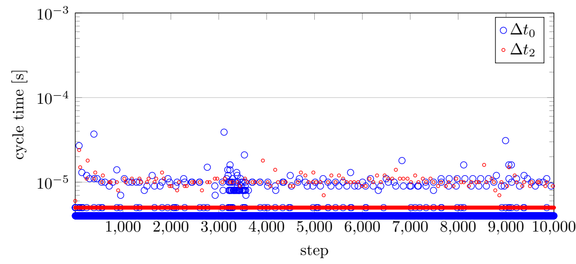

Figure 2.7: Multiplication step time in math function component while running function 2 of tab. 2.3 (Δt_1) compared to the framework cycle time without the math function component (Δt_0), logarithmic scale. Time is measured for 1000 consecutive cycles.

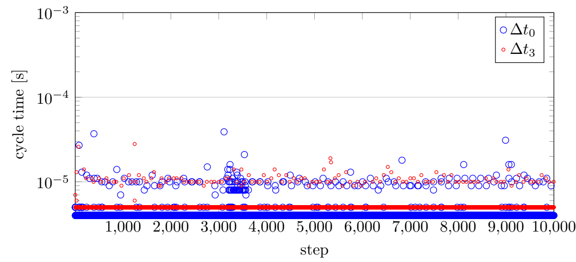

Figure 2.8: Multiplication step time in math function component while running function 3 of tab. 2.3 (Δt_1) compared to the framework cycle time without the math function component (Δt_0), logarithmic scale. Time is measured for 1000 consecutive cycles.

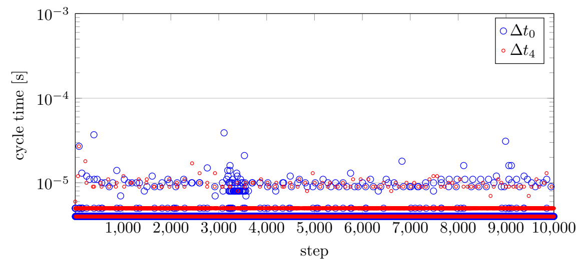

Figure 2.9: Multiplication step time in math function component while running function 4 of tab. 2.3 (Δt_1) compared to the framework cycle time without the math function component (Δt_0), logarithmic scale. Time is measured for 1000 consecutive cycles.

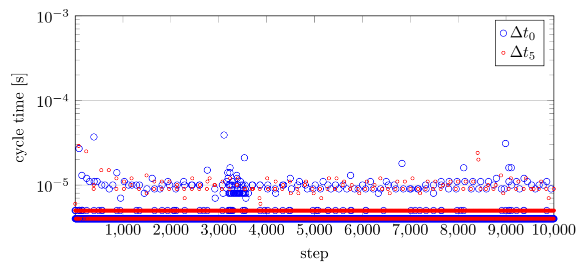

Figure 2.10: Multiplication step time in math function component while running function 5 of tab. 2.3 (Δt_1) compared to the framework cycle time without the math function component (Δt_0), logarithmic scale. Time is measured for 1000 consecutive cycles.

Functions	Average cycle time	Max cycle time
$y = 5x_1$	4.793×10^{-6} s	2.3×10^{-5} s
$y = \frac{x_1^2}{\cos x_1}$	5.070×10^{-6} s	2.4×10^{-5} s
$\begin{cases} a = 2.0 \\ b = 5 \\ y = \\ \quad (x_1 + a)+ \\ \quad (b + x_1 - \frac{a}{b}x_1)(\frac{x_1}{a} - \frac{x_1}{b})+ \\ \quad (x_1 + a)^2 \end{cases}$	5.079×10^{-6} s	2.8×10^{-5} s
$y = \frac{1}{x_1^2 + x_2^2}$	4.449×10^{-6} s	2.7×10^{-5} s
$\begin{cases} a = 2.0 \\ b = 5 \\ y = \frac{1}{1 - \cos ax_1} + \cos \frac{x_2^2}{b} \end{cases}$	4.791×10^{-6} s	2.9×10^{-5} s

Table 2.3: Math function cycle times. The average cycle time with no function evaluation is $\Delta t_0 = 4.120 \times 10^{-6}$ s.

On average, the additional time added to the framework cycle due to the math function component is 0.717×10^{-6} µs.

Results shown that 93% of the lines are covered and outputs from all the required functions is as expected.

Chapter 3

Review of equilibrium and shape system

In this chapter the current equilibrium and shape system [8] [4] is reviewed and updated to adjust to new requirements for future machines and frameworks (see section 2.2).

The equilibrium and shape model is responsible for three main tasks:

1. regulating plasma current during the *flat-top phase* of the discharge
2. maintaining the plasma in the desired position
3. maintaining the plasma in the desired shape

This is accomplished by producing reference values for the *field shaping winding* and for the *magnetizing winding* (for the latter, only during the flat-top phase). As previously stated, field shaping windings produce the magnetic field required to maintain the plasma in the desired position with the desired shape, while magnetizing winding regulates the plasma current by varying the poloidal flux enclosing the plasma and consequently the applied loop voltage.

As such, the model will be represented here as graphical elements linked by the inputs and outputs they exchange, and will be referenced as *the model* in the rest of the chapter. It should be considered however that the model uses C code for actual implementation (see section 3.3).

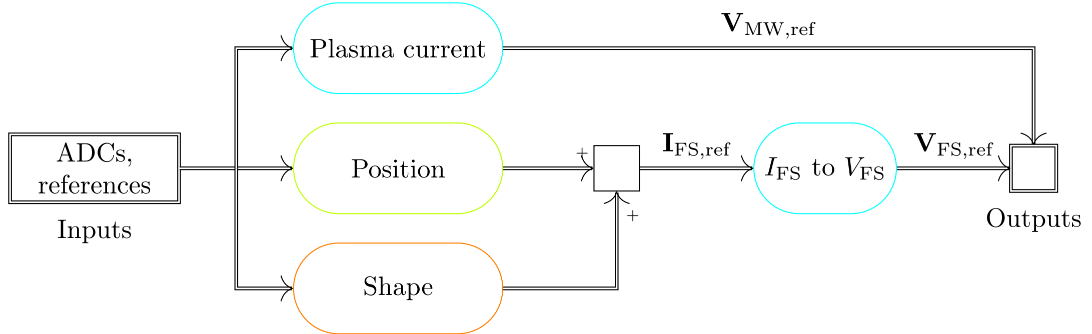

Figure 3.1: General outline of shape and equilibrium model. Inputs and outpus of the model are shown in sections 3.1.1 and 3.1.2. The 3 main elements of the model (plasma current, plasma position and plasma shape) are shown in section 3.2.

3.1 General outline of the model

A general outline of the model is shown in fig. 3.1:

The model has 3 main elements

1. the *plasma current element*, which generates the voltage reference for the magnetizing winding required to follow a preset plasma current value

2. the *plasma position element*, which generates the voltage reference for the field shaping winding to keep the plasma in the required horizontal and vertical position in the vessel

3. the *plasma shape element*, which generates the references required to shape the plasma. These references can be either for the field shaping coils or the saddle coils (see section 3.2.2.3)

These elements together produce the outputs of the system: the voltage reference for the magnetizing winding V_{MW} and the voltage reference for the field shaping coils V_{FS}.

The model has a time step of 2×10^{-4} s, going from -0.6 s to 1 s. Discharge begins at 0 s.

3.1.1 Model inputs

Model inputs and outputs are listed in tab. 3.2.

Inputs can be divided into:

measured inputs coming directly from diagnostics

calculated inputs that is plasma parameters that are not directly available as a diagnostic output but can be modeled and calculated from measured inputs and known variables

reference waveforms that is, signals that are preset and will normally represent a reference to be followed during the discharge

Reference inputs includes the following:

MW power supply voltage V_{MW}, the preset reference for the MW power supply that is to be followed in an ideal case

horizontal shift reference ΔH_{ref} the reference value for the horizonal shift

plasma current I_P^{ref} the reference value for the plasma current

Each calculated input is computed from measured quantities as follows:

3.1.1.1 Horizontal and vertical shifts Δ_H, Δ_V

Horizontal and vertical shifts (Δ_H and Δ_V) are calculated from a set of arrays of magnetic pickup coils measuring the poloidal magnetic field. Each pickup coil of an array is placed on the outer side of the shell at 8 poloidal angles $\theta_{1,\ldots,8}$. Flux measures are used to reconstruct the plasma shape and derive the plasma shape function $r(\theta)$ by reconstructing the flux distribution and finding the last closed flux surface [4].

$r(\theta)$ is then used to calculate the shift of the plasma centroid (x_p, z_p) as shown below:

$$\begin{aligned} \Delta_V &= \frac{1}{\pi}\int r(\theta)\sin\theta d\theta \\ \Delta_H &= \frac{1}{\pi}\int r(\theta)\cos\theta d\theta \end{aligned} \tag{3.1}$$

Δ_H and Δ_V are calculated as the distances of the plasma centroid from the centre of the vacuum vessel.

	Description		Dim.	Unit
Measured inputs				
1	Toroidal loop voltages	V_ϕ	1	V
2	Toroidal loop voltage differences	V_θ	1	V
3	Poloidal fluxes	Φ_ϕ	1	Wb
4	Poloidal flux differences		1	Wb
5	Toroidal current derivatives		1	A/s
6	Poloidal loop voltages		1	V
8	Toroidal currents	I_ϕ	1	A
9	Toroidal fluxes		1	Wb
11	Poloidal fields		1	T
12	Toroidal currents		1	A
13	Magnetizing currents		1	A
14	Field shaping currents		1	A
Calculated inputs				
15	Average toroidal loop voltage	$\bar{V}_\phi$	0	V
16	Average poloidal loop voltage	$\bar{V}_\theta$	0	V
17	Average toroidal flux	$\bar{\Phi}_\phi$	0	Wb
18	Average toroidal flux at the wall		0	Wb
19	Average toroidal field		0	T
20	Average vertical field	$\bar{B}_V$	0	T
21	Average toroidal current derivative	$\frac{\mathrm{d}I_T}{\mathrm{d}t}$	0	A/s
22	Average toroidal current		0	A
23	Average toroidal field at the wall	$\bar{B}_{Tw}$	0	T
24	Horizontal shift	ΔH	0	m
25	Vertical shift	ΔV	0	m
26	Horizontal shift (shell)	ΔH_s	0	m
27	Vertical shift (shell)	ΔV_s	0	m
28	Plasma current	I_P	0	A
29	Vessel current	I_{vessel}	0	A
30	Poloidal fields		1	T
31	Poloidal fluxes		1	Wb
32	Poloidal fluxes at the shell		1	Wb
33	Poloidal fluxes at the coils		1	Wb
35	Second-order harmonic in cosine of the plasma radius	A_2	0	m
36	Third-order harmonic in cosine of the plasma radius	A_3	0	m
37	FS coils radial forces		1	N
38	FS coils vertical forces		1	N
39	FS coils flag		0	A
40	Plasma radius	R	1	m
41	Average plasma radius	$\bar{R}$	0	m
42	Lambda coefficient	Λ	0	
43	Radial position of current centroid	x_p	0	m
44	Vertical position of current centroid	z_p	0	m

Table 3.2: Inputs and outputs of the equilibrium and shape model

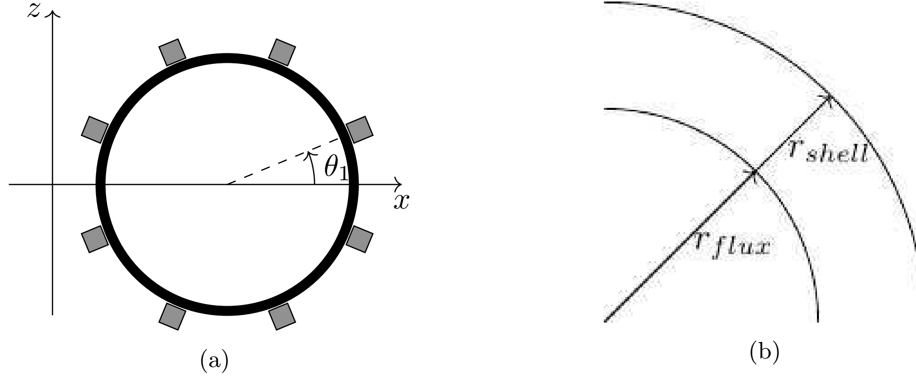

Figure 3.2: Calculation of shell horizontal shift. $\Delta r = r_{\text{flux}} - r_{\text{shell}}$

3.1.1.2 Average toroidal flux at the wall $\bar{B}_{t,w}$

The average toroidal flux at the wall is calculated as follows:

$$\bar{B}_{\phi,w} = \frac{\mu_0}{2\pi R_0}\left[N_\phi S_{\phi C} I_{\phi,coils} - \frac{V_\phi}{R_{\phi,\text{vessel}}}\right]$$

where:

N_ϕ number of turns per toroidal coil sector

S_C multiplying factor depending on the configuration of the toroidal sectors in each discharge [7]

$I_{\phi,coils}$ total current on toroidal coils

V_ϕ poloidal loop voltage

$R_{\phi,\textbf{vessel}}$ poloidal vessel resistance

3.1.1.3 Plasma current I_P

Plasma current is considered as the toroidal current $\bar{I}_\phi$ measured by the corresponding poloidal coils minus the current flowing in the vessel, which is also measured by the coil. Given the value of toroidal and poloidal inductance of the vessel, the only significant contribution is resistive. Thus:

$$I_P = \bar{I}_\phi - I_{\text{vessel}}$$

I_{vessel} is in turn:

$$I_{\text{vessel}} = \frac{V_\phi}{R_{\phi,\text{vessel}}}$$

where:

V_ϕ toroidal loop voltage

$R_{\phi,\textbf{vessel}}$ toroidal vessel resistance

3.1.1.4 FS coil forces flag

This flag is set to `0` if the forces acting on each FS coil exceed the amount that is considered acceptable for the FS coils themselves.

Both vertical and radial forces on each of the 8 FS coils are calculated and checked against a preset limit. Radial (resp. vertical) forces are calculated starting from a 8×9 matrix $\mathbf{f}_r$ (resp. $\mathbf{f}_V$), which is the cross product between the radial (resp. vertical) magnetic field and the i-th current versor. Therefore, the matrix contains the force produced on the i-th FS coil by the j-th magnetic field source per unit current (for each coil and for each source) and is calculated beforehand. Both the 8 FS coils and the toroidal current (the sum of plasma current and vessel current) are considered as a magnetic field sources, hence the 9th column of the matrix.

The force on a current-carrying conductor in a magnetic field is given by:

$$\mathbf{F} = I_{\text{wire}} \ell \times \frac{\mu_0 \mathbf{I}_{\text{source}}}{2\pi d} \tag{3.2}$$

In our case each magnetic source j produces a force on the i-th FS coil, thus:

$$\mathbf{f}_{I,ij} = I_{\text{FS},i} \ell_{\text{FS},i} \times \frac{\mu_0 \mathbf{I}_{\text{FS},j}}{2\pi d_{i,j}}$$

with $d_{i,j}$ distance between the coils. Considering the involved currents (on the source and on the coil) as unitary, eq. 3.2 becomes:

$$\mathbf{f}_{i,j} = \ell_{\text{FS},i} \times \frac{\mu_0 \mathbf{i}_{\text{source}}}{2\pi R_{\text{FS},i}} \tag{3.3}$$

The coefficients of $\mathbf{f}_r$ and $\mathbf{f}_V$ are respectively the radial and vertical components of eq. 3.3. The absolute values of radial and vertical forces on the i-th coil are then:

$$F_{r,i} = (\sum_{j=1}^{8} f_{r,ij} \cdot I_{\text{FS},j} + f_{r,i9} \cdot I_\phi) \cdot I_{\text{FS},i}$$

$$F_{V,i} = (\sum_{j=1}^{8} f_{V,ij} \cdot I_{\text{FS},j} + f_{V,i9} \cdot I_\phi) \cdot I_{\text{FS},i}$$

where:

$I_{\mathbf{FS},i}$ is the current on the i-th FS coil

I_ϕ is the total toroidal current ($I_P + I_{\text{vessel}}$)

3.1.1.5 Average vertical field $\bar{B}_V$

This signal is the average vertical magnetic field evaluated on the equatorial plane of the machine.

To calculate the average vertical field the poloidal magnetic flux measurements are taken into account and the magnetic field is derived by using eq. 3.4

$$\Phi = \mathbf{B} \cdot \mathbf{S} = BS \cos\theta \tag{3.4}$$

8 continuous flux loops are available. Firstly poloidal flux differences between loops 1 and 5 ($\Delta\Phi_{15}$ at $\theta_{1,5} = 22.5°, 202.5°$), and between loops 4 and 8 ($\Delta\Phi_{48}$ at $\theta_{4,8} = 157.5°, 337.5°$), are calculated. The flux differences are then divided by the surface of the annuli that the loops themselves project on the equatorial plane as calculated in eq. 3.5.

$$\begin{aligned} S_{15} &= 4\pi R_{\text{vessel},M} R_{\text{vessel},m} \cos \frac{22.5\pi}{180} \\ S_{48} &= 4\pi R_{\text{vessel},M} R_{\text{vessel},m} \cos \frac{\pi - 22.5\pi}{180} \end{aligned} \tag{3.5}$$

The average vertical field is then calculated as:

$$\bar{B}_V = \frac{\Delta\Phi_{15}/S_{15} + \Delta\Phi_{48}/S_{48}}{2} \tag{3.6}$$

3.1.1.6 q factor

The q *factor* is calculated as:

$$q = \frac{2\pi r_{tiles}^2 B_{\phi,w}}{\mu_0 R I_P}$$

where:

r_{tiles} is the tile surface radius

$B_{\phi,w}$ is the toroidal field at the wall (see section 3.1.1.2)

R is the vessel major radius

I_P is the plasma current (see section 3.1.1.3)

3.1.1.7 Toroidal current derivative $\frac{\mathrm{d}I_T}{\mathrm{d}t}$

This input is the total current flowing toroidally, that is, the sum of plasma current I_P and vessel current I_{vessel}. The signal is calculated as the average among the 4 poloidal voltages directly measured by 4 poloidal coils.

3.1.1.8 Toroidal loop voltage V_ϕ

The signal is calculated as the average among the 8 toroidal loop voltages, directly measured by the corresponding 8 toroidal coils.

3.1.1.9 Average toroidal flux Φ_ϕ

The signal is calculated as the average among the 10 poloidal loop voltages, directly measured by the corresponding 10 poloidal coils.

3.1.1.10 Gaps g

Gaps are used in the vertical equilibrium system as a secondary source of vertical shift measure when the plasma shape is reconstructed with the gaps themselves and not from geometrical parameters (ellipticity and triangularity).

Gaps are measured in correspondence of the flux loop pick-up coils, at the following angles:

$$\theta_{\text{FL}} = [22.5°67.5°112.5°157.5°202.5°247.5°292.5°337.5°]$$

3.1.1.11 Harmonics of the plasma radius function $r(\theta)$

The 2nd and 3rd harmonics of the plasma radius function $r(\theta)$ are used in the ellipticity and triangularity adjustment systems (sections 3.2.3.1 and 3.2.3.2).

The plasma radius function $r(\theta)$ is calculated from the flux pickup coils as shown in section 3.1.1.1.

3.1.2 Model outputs

The model outputs two sets of voltage references for the power supplies to follow:

MW reference a voltage reference for the magentizing winding power supply produced by the plasma current management element (see section 3.2.1)

FSW reference a vector of 8 voltage references for the field shaping windings power supply produced by the plasma position and shape management element (see sections 3.2.2 and 3.2.3)

3.2 Equilibrium and shape elements

In this section each element of the plasma equilibrium and shape model is shown. The equilibrium and shape model is composed of 3 main elements:

1. the plasma current element, which generates the voltage references required to drive the plasma current

2. the plasma position element, which generates the voltage references required to keep the plasma centroid in the desired position

3. the plasma shape element, which generates the voltage references required to adjust the plasma shape (ellipticity and triangularity)

3.2.1 Plasma current element

This element (see fig. 3.1) produces the voltage references for the *magnetizing windings* during the *flat-top phase* [8]. The element comprises three main subelements (feedthrough, single null (SGN) configuration and RFP configuration) and two auxiliary elements (resistive voltage drop compensation and plasma state calculation). The elements are linked as shown in figure 3.3.

The element receives as inputs the following quantities.

References:

MW power supply feedforward reference V_{MW}, the preset reference for the MW power supply that is to be followed in an ideal case. The plasma current element applies small corrections to this baseline quantity.

plasma current reference I_P^{ref}, the preset reference for the plasma current to be followed

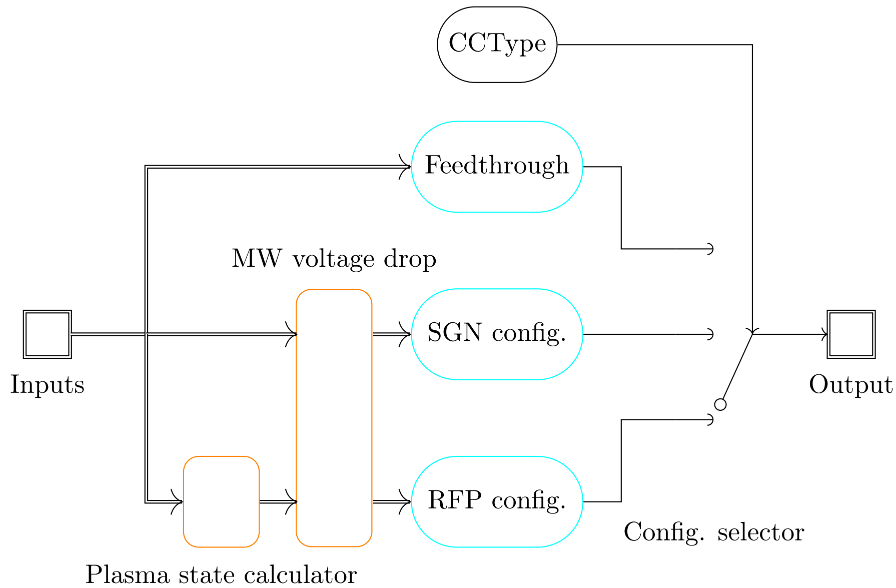

Figure 3.3: Outline of the plasma current management element. In this configuration the model is using the RFP current drive.

Current measures:

plasma current I_P, the actual plasma current value calculated as shown in section 3.1.1.3.

MW currents $\mathbf{I}_{\mathrm{MW}}$, a vector of the 4 currents flowing through each of the MW sectors.

FS coil currents $\mathbf{I}_{\mathrm{FS}}$, a vector of the 8 currents flowing through each of the FS coil sectors.

Voltage measures:

toroidal loop voltage V_ϕ, as shown in section 3.1.1.8

toroidal current derivative $\frac{\mathrm{d}I_T}{\mathrm{d}t}$, as shown in section 3.1.1.7

The `CCType` element has an input parameter that allows to choose among the three different types of current drives based upon the type of desired discharge.

3.2.1.1 Feedthrough

The feedthrough element does not modify the input MW reference. Selecting this element effectively turns off any feedback action applied to the plasma

current and only the feedforward value of $V_{\mathrm{MW,ref}}$ is used.

3.2.1.2 Single null configuration current management

This element actively performs a *feedback action on plasma current* during the *flat-top phase*. The element takes as inputs the plasma current feedforward reference $I_{P,\mathrm{ref}}$ and the MW voltage feedforward reference $V_{\mathrm{MW,ref}}$, and uses the present value of the plasma current to apply small corrections to the MW voltage feedforward value in order to make I_P follow $I_{P,\mathrm{ref}}$. A time signal t and the amplitude of a mode $B_{p,mn}$ are used to switch among the state machine states.

This element is a state machine with 4 possible states:

1. feedforward
2. fast ramp-up
3. operation
4. ramp-down

The controller starts in *feedforward* mode, where it behaves as described in section 3.2.1.1.

When the time signal becomes greater than a preset threshold the element switches to the *operation* state and starts following the current reference $I_{P,\mathrm{ref}}$. This is done with a purely proportional action by applying a gain $K_{p,\mathrm{CC}}$ to the current error as shown in eq. 3.7.

$$V_{\mathrm{MW,ref}} = K_{p,\mathrm{CC}}(I_{P,\mathrm{ref}} - I_P) \tag{3.7}$$

The state is switched to *fast ramp-up* when both time and the amplitude of the selected mode $B_{p,mn}$ are greater than some preset thresholds. In this state the $V_{\mathrm{MW,ref}}$ output is purely feedforward and set equal to the preset voltage. The current reaches the preset value with a ramp due to the mainly inductive nature of the plasma load. This state lasts until I_P reaches the plasma current marginal value that would start an $m = 2$, $n = 1$ unstable mode. At this point, the element switches back to the operation state.

Lastly, the element switches to *ramp-down* state if $t > T_{\mathrm{end}}$. Additionally, the element checks that I_P stays within limits set upon its absolute value and its derivative. A plasma current value outside the operational limits, or a too

steep variation of its value, would also result in the element switching to rampdown state. In this state current is progressively brought to $I_P = 0$ with a ramp whose duration $\Delta T_{\text{rampdown}}$ is preset as a discharge parameter, as shown in eq. 3.8 where T_{start} is the instant in which the rampdown action is started.

$$V_{\text{MWref,t}} = V_{\text{MWref,t-1}} - \frac{V_{\text{MWref,t-1}}(t - T_{\text{start}})}{\Delta T_{\text{rampdown}}} \tag{3.8}$$

3.2.1.3 RFP configuration current management

This element actively performs a *feedback action on plasma current* during the *flat-top phase*. As in section 3.2.1.2, the element takes as inputs the plasma current feedforward reference $I_{P,\text{ref}}$ and the MW voltage feedforward reference $V_{\text{MW,ref}}$, and uses the present value of the plasma current to apply small corrections to the MW voltage feedforward value in order to make I_P follow $I_{P,\text{ref}}$. A time signal t is used to switch among the state machine states. Plasma ohmic power P_Ω and the plasma resistance R_P are used to apply small corrections to the output reference value.

This element includes an implementation of a lead-lag compensator as described in eq. 3.9:

$$G(s) = k\frac{(1 + s\tau_z)}{(1 + s\tau_p)} \tag{3.9}$$

The compensator is represented as state-space equations:

$$\begin{aligned} x(k+1) &= Ax(k) + Bu(k) \\ y(k) &= Cx(k) + Du(k) \end{aligned} \tag{3.10}$$

To evaluate the coefficients of eq. 3.10 one must consider that a lead-lag compensator in discrete time can be represented as:

$$F(z) = k\frac{(T/2 + \tau_z)z + (T/2 - \tau_z)}{(T/2 + \tau_p)z + (T/2 - \tau_p)} = k\frac{b(1)z + b(0)}{a(1)z + a(0)} = kd + \frac{k \cdot c(0)}{a(1)z + a(0)}$$

where $d = \frac{b(1)}{a(1)}$ and $c(0) = b(0) - d \cdot a(0)$. The state-space equations are then discretised as:

$$
\begin{aligned}
X(z) &= \frac{k \cdot c(0) \cdot U(z)}{a(1)z + a(0)} \\
Y(z) &= X(z) + kdU(z)
\end{aligned}
\tag{3.11}
$$

The system is discrete, thus it can be assumed that $x(k+1) = -\frac{a(0)}{a(1)}x(k) + \frac{k \cdot c0}{a(1)}u(k)$, and by comparison with eq. 3.11 it can be inferred that

$$
\begin{aligned}
A &= -\frac{a(0)}{a(1)} \\
B &= \frac{k \cdot c(0)}{a(1)} \\
C &= 1 \\
D &= kd
\end{aligned}
\tag{3.12}
$$

This element is a state machine with 3 possible states:

1. feedforward
2. operation
3. ramp-down

The controller starts in *feedforward* mode, where it behaves as described in section 3.2.1.1.

The element switches to *operation* state and starts following the current reference $I_{P,\text{ref}}$ if:

- t becomes greater than a preset threshold
- I_P becomes greater than a preset threshold
- $\frac{\mathrm{d}I_P}{\mathrm{d}t}$ becomes less than a preset threshold. Differently from the single null configuration (see section 3.2.1.2) there is no *fast ramp-up* state, so a slowly increasing current must be dealt with by the operation state.

While in this state the element evaluates the error $\epsilon_I = I_{P,\text{Ref}} - I_P$. The lead-lag compensator uses ϵ_I to calculate the required feedback action as shown in eq. 3.13:

$$
\begin{aligned}
V_{\text{MW,fb}} &= Cx(t-1) + D\epsilon_I \\
x(t) &= Ax(t-1) + B\epsilon_I
\end{aligned}
\tag{3.13}
$$

The element also calculates a purely feedforward action to compensate plasma resistive and inductive voltage drop:

$$V_{\mathrm{MW,ff}} = R_P I_{P,\mathrm{Ref}} + \frac{(L_P + L_d)\Delta I_{P,\mathrm{Ref}}}{\Delta t} \tag{3.14}$$

where $\Delta I_{P,\mathrm{Ref}}$ is the difference between the current value of the current reference and the value at the previous step, L_P is the plasma inductance and L_d is the stray inductance. The complete reference is then calculated as the sum of the feedback and feedforward action:

$$V_{\mathrm{MW,Ref}} = V_{\mathrm{MW,ff}} + V_{\mathrm{MW,fb}}$$

The element remains in operation state unless one of the following conditions becomes true:

- $t > T_{\mathrm{end}}$ (end of discharge)
- $P_\Omega > P_{\Omega,\mathrm{max}}$ (overheating)
- $I_P < \frac{3}{5} I_{P,\mathrm{Ref}}$ (too distant from preset value)

In such cases the element switches to *ramp-down state*.

3.2.1.4 MW resistive voltage drop compensation

This element includes the compensation of the resistive voltage drop on the MW power supply connection cables in the MW voltage reference $V_{\mathrm{MW,Ref}}$.

During the flat-top phase MW coils and FS coils are connected in parallel with each other to balance the mutual inductance matrix[1], and in series with the transfer resistors TR. Each sector of the MWs is connected with two sectors of the FS coils to the same cable.

Ideally one would want to calculate the voltage drop on each sector. However, an average value is used since the MW power supply reference is unique in order to avoid circulation currents between the nodes connecting each sector. Assuming that each cable has the same resistance the average voltage drop on the connection cables can be calculated as follows:

$$\overline{\Delta V}_{\mathrm{cable}} = \frac{(\sum_i \mathbf{I}_{\mathrm{M}} + \sum_i \mathbf{I}_{\mathrm{FS}})\bar{R}_{\mathrm{cable}}}{4}$$

The voltage drop on the magnetizing windings only can be calculated as:

[1] Balancing is required to avoid circulation currents in the coil connections.

$$\overline{\Delta V}_{\text{Mag}} = \frac{\sum_i \mathbf{I}_{\text{M}} \cdot \bar{R}_{\text{Mag}}}{4}$$

The voltage drop on the transfer resistance R_{TR} is calculated with a voltage divider:

$$\overline{\Delta V}_{\text{TR}} = V_{\text{MW,Ref}} \frac{R_{\text{cable}}}{R_{\text{TR}} + R_{\text{cable}}}$$

The total voltage drop is then:

$$\overline{\Delta V}_{\text{MWs}} = \overline{\Delta V}_{\text{cable}} + \overline{\Delta V}_{\text{Mag}} + \overline{\Delta V}_{\text{TR}}$$

This value is added to $V_{\text{MW,Ref}}$.

3.2.1.5 Plasma state calculator

This element uses the toroidal loop voltage, the plasma current and the toroidal current derivative to evaluate the plasma dissipated power P_Ω and plasma resistance R_P.

P_Ω is used to set an operation threshold in order to avoid excessive power dissipation in the plasma and is calculated as:

$$P_\Omega = (V_{\text{loop}} - L_P \frac{\mathrm{d}I_T}{\mathrm{d}t}) I_P$$

where L_P is the plasma internal inductance. L_P can be calculated as the integral of the plasma magnetic energy over the whole plasma and is expected to vary during the discharge. For simplicity reasons it is here calculated beforehand and considered constant.

R_P is on the other hand calculated during the discharge as:

$$R_P = \frac{V_{\text{loop}} - L_P \frac{\mathrm{d}I_T}{\mathrm{d}t}}{I_P}$$

Note that only the RPF current management element uses these values as inputs.

3.2.2 Plasma position element

The purpose of this element is to keep the plasma centroid in the desired position with respect to the shell. This element is divided in two main sub-elements and one auxiliary sub-element:

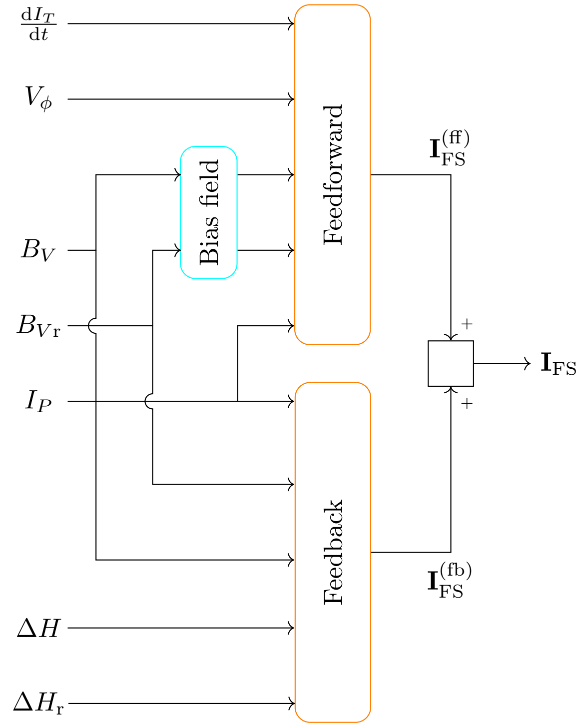

Figure 3.4: Plasma position element. The feedforward element is shown in fig. 3.5, while the feedback element is shown in fig. 3.8.

1. horizontal shift element
2. vertical shift element
3. bias vertical magnetic field element (auxiliary)

The elements are connected as shown in fig. 3.4

3.2.2.1 Horizontal shift

This element evaluates the vertical magnetic field B_V required to keep the plasma in the desired horizontal position and generates the corresponding voltage references for the FS coils.

Horizontal shift feedforward action The outline of this element is shown in fig. 3.5. This element takes as inputs:

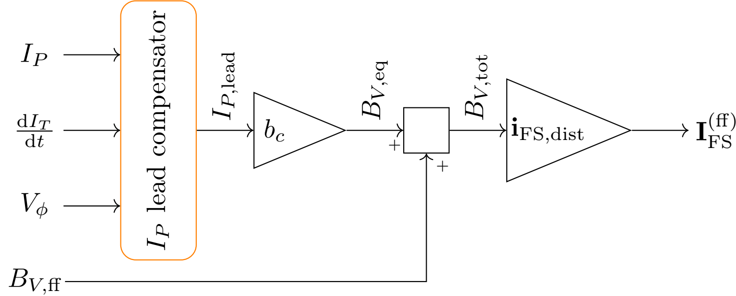

Figure 3.5: Plasma position feedforward element. The I_P compensator element is shown in fig. 3.6. Calculation of b_c is shown in eq. 3.21.

plasma current I_P, the actual plasma current value calculated as shown in section 3.1.1.3

toroidal current derivative $\frac{\mathrm{d}I_T}{\mathrm{d}t}$, as shown in section 3.1.1.7

feedforward vertical field reference $B_{V,\mathrm{ff}}$, a preset waveform

The element outputs a *contribution to the FS current references* that will be added to the contributions produced by the plasma shape element (section 3.2.3) and the feedback part of the position element (section 3.2.2.1).

The element is responsible for applying the equilibrium equation 3.15:

$$B_{V,\mathrm{eq}} = \frac{\mu_0 I_P}{4\pi R_0}\left[\ln\frac{8R_P}{a} + \Lambda - \frac{1}{2}\right] \tag{3.15}$$

thus evaluating the amplitude of vertical field required to compensate the horizontal forces induced by the plasma current (see section 1.3.3).

The value of I_P is preprocessed in a lead compensator to compensate the delay introduced by the field shaping system (power supplies, coils and cables), which is responsible for creating the vertical field required by this element. This delay has been extimated to be $\tau_1 = 7.6\mathrm{ms}$

In order to anticipate the response of the power supplies the following transfer function should be applied to the plasma current:

$$G(s)I_P(s) = \frac{1 + s\tau_1}{1 + s\tau_2} I_P(s) \tag{3.16}$$

where $\tau_2 = 4.5\mathrm{ms}$ is another time constant associated to the lead compensator pole in order to shift the pole away from the zero, its value having been optimized experimentally.

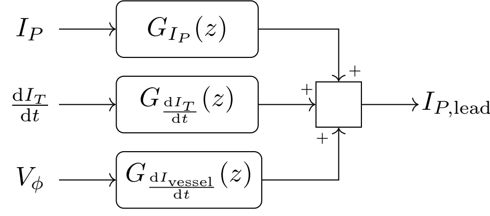

Figure 3.6: Plasma position lead compensator element.

Nevertheless, since the toroidal current derivative is available to the same element, eq. 3.16 can be separated into the two following equations:

$$\begin{aligned} G_{I_P}(s) &= \frac{1}{1+s\tau_2} \\ G_{\frac{\mathrm{d}I_T}{\mathrm{d}t}}(s) &= \frac{\tau_1}{1+s\tau_2} \end{aligned} \tag{3.17}$$

By applying the trapezoidal rule $s = \frac{2}{T}\frac{z-1}{z+1}$, eqs. 3.17 can be discretized:

$$\begin{aligned} G_{I_P}(z) &= \frac{z+1}{(1+\frac{2}{T}\tau_2)z + (1-\frac{2}{T}\tau_2)} \\ G_{\frac{\mathrm{d}I_T}{\mathrm{d}t}}(z) &= \frac{\tau_1(z+1)}{(1+\frac{2}{T})z + (1-\frac{2}{T}\tau_2)} \end{aligned} \tag{3.18}$$

The toroidal current derivative signal includes the contribution from the vessel current that should be subtracted before applying eq. 3.15. The vessel current quota $I_{\text{vessel}} = V_{\text{loop}}/R_{\text{vessel}}$ is associated to the transfer function:

$$G_{\frac{\mathrm{d}I_{\text{vessel}}}{\mathrm{d}t}}(s) = \frac{s\tau_1}{1+s\tau_2}\frac{1}{R_{\text{vessel}}} \tag{3.19}$$

which, once discretized, becomes:

$$G_{\frac{\mathrm{d}I_{\text{vessel}}}{\mathrm{d}t}}(z) = \frac{\frac{2}{T}\tau_1(z-1)}{(1+\frac{2}{T}\tau_2)z + (1-\frac{2}{T}\tau_2)}\frac{1}{R_{\text{vessel}}} \tag{3.20}$$

The combination of the above transfer functions in the lead compensator element is shown in fig. 3.6, and the resulting effect is shown in fig. 3.7.

The input of eq. 3.15 is considered to be $I_{P_{\text{lead}}}$ only, since all other parameters of the equation are constant (depending on the machine geometry) or can be assumed as constant without losing much precision. In this case R_0, R_P and a are considered constants and in RFP discharges it can be assumed that $\Lambda = -0.2$.

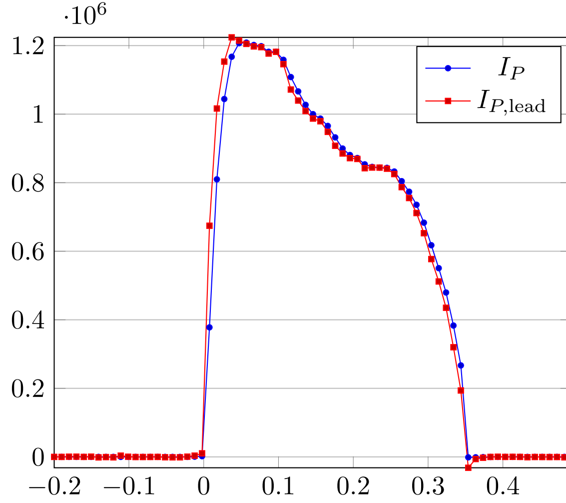

Figure 3.7: Effect of lead compensator on I_P. The derivative effect allows to compensate the delay introduced by the power supplies.

These considerations reduces eq. 3.15 to:

$$\underbrace{B_{V,\text{eq}}}_{\text{Output}} = \underbrace{I_{P,\text{lead}}}_{\text{Input}} \cdot \underbrace{\frac{\mu_0}{4\pi R_0}\left[\ln\frac{8R_P}{a} + \Lambda - \frac{1}{2}\right]}_{\text{constants}} = I_{P,\text{lead}} \cdot b_c \tag{3.21}$$

The constant part b_c represents the required value of vertical field per unit plasma current, measured in $\left[\frac{T}{A}\right]$.

The resulting magnetic vertical field $B_{V,\text{eq}}$ is then distributed along the FS coils using a vector product with experimentally determined coefficients:

$$\mathbf{I}_{\text{FS,eq}} = B_{V,\text{eq}}\mathbf{i}_{\text{FS,dist}}$$

Horizontal shift feedback action The outline of this element is shown in fig. 3.8. This element takes as inputs:

measured horizontal shift ΔH, the plasma horizontal shift with respect to the chamber centre 3.1.1.1

shell horizontal shift ΔH_s, the horizontal shift of the plasma centroid calculated as if the shell is a flux surface

horizontal shift reference ΔH_r, a preset waveform of the desired vertical shift during the discharge

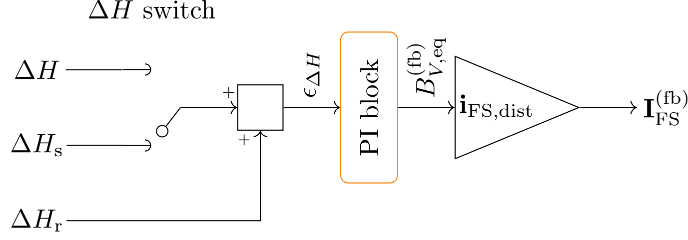

Figure 3.8: Plasma position feedback element. ΔH: horizontal shift of the plasma centroid. ΔH_s: horizontal shift of the plasma centroid calculated as if the shell is a flux surface. ΔH_r: horizontal shift reference to be followed.

The element outputs a *contribution to the FS current references* that will be added to the contributions produced by the plasma shape element (section 3.2.3) and the feedforward part of the position element (section 3.2.2.1).

Firstly, a two-way switch (ΔH switch) selects the desired horizontal shift among the two available to the element. The horizontal shift reference is subtracted to the selected horizontal shift in order to obtain the error on the horizontal shift $\epsilon_{\Delta H}$.

This error is used as input for a PI sub-element which calculates the response in terms of a contribution to the vertical field converting the horizontal displacement into a request of vertical field $B_{V,\mathrm{eq}}^{(\mathrm{fb})}$. The PI element (proportional-integral, no derivative part is used) coefficients are:

$$K_p = 4$$
$$K_i = 60$$

The integral part is implemented by means of a discrete-time integrator $\frac{T(z+1)}{2(z-1)}$ and is activated only when $t > 0.05\,\mathrm{s}$. This is due to the fact that in the early phase of the discharge the plasma is formed near the inner edge of the vessel and as such has a horizontal position error reversed with respect to the error during the discharge, when the plasma tends to expand towards the outer part of the vessel. Thus, the integral part in the early phase would increase the horizontal shift error that should then be compensated.

Eventually the required vertical field is distributed along the FS coils using a vector product with the calculated coefficients:

$$\mathbf{I}_{\mathrm{FS,eq}} = B_{V,\mathrm{eq}} \mathbf{i}_{\mathrm{FS,dist}}$$

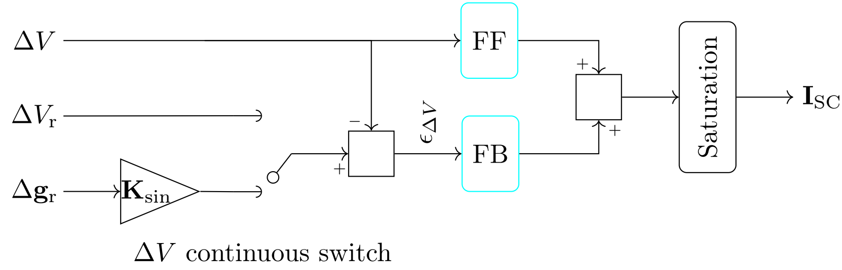

Figure 3.9: Vertical shift stabilization. `FF`: feedforward action on the vertical shift. `FB`: feedback action on vertical shift error.

3.2.2.2 Vertical shift

This element implements the active stabilization of the plasma vertical position and is thus activated during shaped-boundary plasma discharges since only elongated shapes need vertical stabilization, or when a vertically shifted circular plasma is required.

In order to perform vertical stabilization a horizontal component of the magnetic field is required (see section 1.3.4). The element thus outputs a reference for two rows of saddle coils that are mounted on the equatorial plane of the machine, one row on the outer plane and one on the inner plane. All coils of each row are virtually connected in series, and the two rows are then virtually connected in anti-series. With this configuration the direction of the produced horizontal field is concordant and only one current reference is sufficient to drive the whole coil array.

The element inputs are:

vertical shift ΔV, the vertical shift of the plasma centroid with respect to the desired position (see section 3.1.1.1)

vertical shift reference ΔV_{ref}, the vertical shift of the plasma centroid as shown in section 3.1.1.1

gap references $\Delta \mathbf{g}_{\mathrm{r}}$, the vector of the requested 8 radial gaps between the plasma and the first wall as described in section 3.1.1.10

The element outputs a current reference for the power supplies of the saddle coils $\mathbf{I}_{\mathrm{SC}}$.

When using a circular configuration the vertical shift reference is given directly. When using an single null configuration the reference is given consistently

with the 8 measures of the gaps between the plasma and the wall. Starting from these values an appropriate vertical shift reference is calculated. This allows the vertical shift to be calculated based on the desired plasma shape instead of passing it to the model as an unrelated input. Nevertheless, since the discharge begins as circular and only later evolves towards the single null configuration, the reference must be shifted accordingly from the direct vertical shift reference used during the early phase of the discharge and the gap-based vertical shift used during the second half of the discharge. A `continuous switch` is implemented that cross-fades from one reference to the other.

The gap-based vertical shift reference is calculated as the scalar product

$$\Delta V_{\mathrm{g}} = \Delta \mathbf{g}_{\mathrm{r}} \cdot \mathbf{K}_{\sin}$$

where $\mathbf{K}_{\sin}(i) = -\frac{1}{4}\sin\theta_{\mathrm{FL}}(i)$ (see section 3.1.1.10 for the values of θ_{FL}).

The error $\epsilon_{\Delta V}$ is calculated as the difference between the selected reference and the actual vertical shift. A PI element (feedback element `FB` in fig. 3.9) elaborates the adequate response to the input error. The proportional and integral coefficients are:

$$K_p = 10$$
$$K_i = 60$$

Again, the integration start is delayed in order to make the response more prompt during the early phase of the discharge.

In parallel to the PI element, a lead compensator (feedforward element `FF` in fig. 3.9) is applied to the selected vertical shift reference in order to compensate the delays introduced by the saddle coil system (power supplies, cables and coils). An experimentally determined constant coefficient is used to convert the vertical shift reference into the corresponding horizontal field and then into the current reference for the saddle coils.

The feedback and feedforward actions are added together. The signal is saturated at $\pm 50\,\mathrm{A}$ so as not to exceed the quota of the saddle coil current dedicated to vertical stabilization.

3.2.2.3 Bias vertical magnetic field feedback action

This PI element is an additional feedback component that manages the creation of a vertical bias field before the discharge to avoid delays due to the time

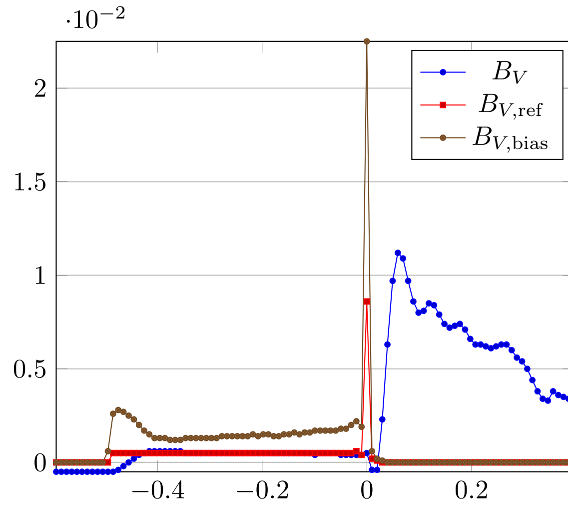

Figure 3.10: Effect of bias field produced by the PI element on the vertical magnetic field reference $B_{V,\mathrm{ref}}$. Pulse begins at $t = 0\,\mathrm{s}$

constant of the magnetic field penetration in the shell, or to compensate other magnetic fields from e.g. the magnetizing winding. Its coefficients are:

$$K_p = 2.25$$
$$K_i = 45$$

The effect on the magnetic field reference are shown in fig. 3.10

3.2.3 Plasma shape element

This element shapes the plasma by generating a 2nd order cosine distribution of references for the FS coils. Depending on the required type of discharge the inputs of this element can be

1. geometrical parameters (ellipticity and triangularity, see section 3.1.1.11)
2. gap measures used to reconstruct the plasma shape (see section 3.1.1.10)

3.2.3.1 Vertical ellipticity

This element implements the active adjustment of the plasma grade of vertical ellipticity. The element outline is shown in fig. 3.11. As stated in section 1.3.5,

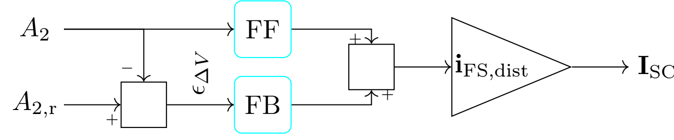

Figure 3.11: Ellipticity element. FF: feedforward action on A_2 (amplitude of 2nd harmonic of $r(\theta)$). FB: feedback action on A_2 (amplitude of 2nd harmonic of $r(\theta)$).

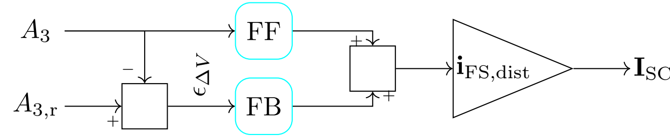

Figure 3.12: Triangularity element. FF: feedforward action on A_3 (amplitude of 3rd harmonic of $r(\theta)$). FB: feedback action on A_3 (amplitude of 3rd harmonic of $r(\theta)$).

vertical ellipticity can be expressed as the amplitude of the 2nd armonic in $\cos\theta$ of the plasma radius $r(\theta)$. As such, the element inputs are:

amplitude of the 2nd cosine harmonic of the plasma radius A_2, the feedback signal used to calculate the response

reference of 2nd cosine harmonic of the plasma radius $A_{2,\text{ref}}$

Calculation of said harmonics is carried out outside the element as shown in section 3.1.1.11. The element outputs a *contribution to the FS current references* that will be added to the contributions produced by the plasma shape element (section 3.2.3) and the plasma position element (section 3.2.2).

Similarly to the vertical shift element, this element consist of a lead compensator element applied to the entire $r(\theta)$ 2nd harmonic reference and a PI feedback element applied to the 2nd harmonic error $\epsilon_{\Delta A_2}$ evaluated as the difference between the 2nd harmonic reference and the calculated 2nd harmonic.

Both these responses are then added together.

The vertical field required to adjust the plasma ellipticity is produced by the FS coils through a $cos2\theta$ distribution, thus the response is distributed to the FS coils using the coefficient $\mathbf{i}_{\text{FS,dist}}$.

3.2.3.2 Triangularity

This element implements the active adjustment of the plasma grade of triangularity. The element outline is shown in fig. 3.12. As stated in section 1.3.5, triangularity can be expressed as the amplitude of the 3rd armonic in $\cos\theta$ of the plasma radius $r(\theta)$. As such, the element inputs are:

amplitude of the 3rd cosine harmonic of the plasma radius A_3, the feedback signal used to calculate the response

reference of 3rd cosine harmonic of the plasma radius $A_{3,\text{ref}}$

Calculation of said harmonics is carried out outside the element as shown in section 3.1.1.11. The element outputs a *contribution to the FS current references* that will be added to the contributions produced by the plasma shape element (section 3.2.3) and the plasma position element (section 3.2.2).

Similarly to the vertical shift element, this element consist of a lead compensator element applied to the entire $r(\theta)$ 3rd harmonic reference and a PI feedback element applied to the 3rd harmonic error $\epsilon_{\Delta A_3}$ evaluated as the difference between the 3rd harmonic reference and the calculated 3rd harmonic.

Both these responses are then added together.

The vertical field required to adjust the plasma triangularity is produced by the FS coils through a *cos3θ* distribution, thus the response is distributed to the FS coils using the coefficient $\mathbf{i}_{\text{FS,dist}}$.

3.2.4 Additional elements

The following elements introduce additional contributions such as activations, voltage drop compensations etc.

3.2.4.1 FS voltage drop compensation

This element converts the overall FS coil current reference produced by the elements in sections 3.2.1, 3.2.2 and 3.2.3 into voltage references for the FS coil power supplies. The element is also responsible for the following compensarions:

1. compensation of resistive voltage drop on the FS coils produced by the currents flowing in the coils
2. compensation of the inductive coupling among the FS coils, produced by current variations in the coils themselves

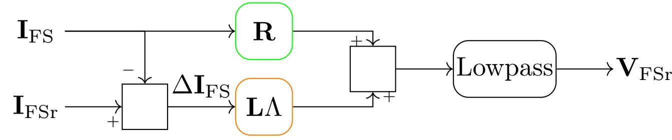

Figure 3.13: FS voltage drop compensation.

The element inputs are:

FS coil currents $\mathbf{I}_{\mathrm{FS}}$, measured on the coils

FS coil current references $\mathbf{I}_{\mathrm{FS,ref}}$, produced by the upstream elements

The element outputs the voltage references $\mathbf{V}_{\mathrm{FS,ref}}$ for the FS coil power supplies.

An outline of the element is shown in fig. 3.13.

Firstly the element evaluates the required current variation $\Delta\mathbf{I}_{\mathrm{FS}}$ with respect to the preset feedforward reference. A resistance matrix $\mathbf{R}$ is applied to the measured currents, thus introducing an additional voltage reference that compensates the resistive voltage drop on the total poloidal system.

To compensate the current errors due to inductive coupling between the coils the inductance matrix $\mathbf{L}\Lambda$ is applied to the current variation vector $\Delta\mathbf{I}_{\mathrm{FS}}$. Diagonalizing the state matrix $\mathbf{A}$ of the system the current variation rates can be obtained. By defining the diagonal matrix Λ whose elements are equal to the current error variation rates, the inductive coupling can be compensated by multiplying the current errors by the factor:

$$K_L = -\mathbf{L}\Lambda$$

where $\mathbf{L}$ is the system inductance matrix.

3.2.4.2 Ramp-down element

A ramp-down element activates a preprogrammed soft ramp-down of the coil currents in case any of the coil current limits is passed.

3.3 Model library

The equilibrium and shape model is shown using a graphical element diagramming. To be used during real discharges however, the model needs to be com-

piled into C code.

The result shall be a library that can be linked. The library shall offer some handles that can be used to load and run the model from any other application. These handles include:

`initialise` this symbol will allow to load the model and all the variables that will be used during the model run

`step` this function will make the model advance of one step, using its inputs to calculate and update the outputs

To talk to each other, the model and the programme that runs it share a structure initialized by the model itself containing all the relevant instructions, such as number and dimensions of inputs and outputs and number and dimensions of parameters. The structure layout must be known by the auxiliary programme by including an appropriate header.

3.4 Updates to the model

The model has been reorganized and extensively commentated to facilitate usage and upkeep.

3.4.1 Circular and single-null configurations

The old equilibrium and shape model was actually split into two main models. The former was used to obtain *circular configurations*, while the latter for *single-* and *double-null configurations*.

The main differences between the two models are:

1. in the horizontal shift feedback element (3.1.1.1): the single null version of the model has the option to calculate the horizontal shift reference from the gap references, just as the model does for the vertical shift
2. circular to single null transition: the single null model includes elements to handle the transition from a circular-shaped plasma to an single null-shaped plasma, due to the fact that every single null discharge ultimately starts as circular. In particular, elements that do this in the single null model are the horizontal shift feedforward element and the vertical shift element

3. coil connections: coils connections to power supplies are different between circular and single null discharges, and FS5 is usually not used. FS voltage drop compensation includes these modifications

This layout required the libraries to be swapped manually each time the configuration was changed. Furthermore, any modification and upgrade of the model had to be applied to both models.

3.4.1.1 Merging of circular and single null configurations

Conditional statements were therefore introduced in the model and used to upgrade the existing elements that operated differently in the circular and single null version of the model. A conditional element contains multiple versions of an element and is capable of switching on the most suitable one based on how the current discharge is defined (as a circular or as an single null one).

3.4.2 Structured inputs and outputs

The model has also been modified in order to generalize its interface with the auxiliary programme.

The inputs and outputs of the model are now structured. The auxiliary programme has been updated in a recursive fashion so as to be able to fully explore the structured components of the model signals. This resulted in a neater model which is also easier to interface with other components of the framework, since a single structured array now contains all the relevant inputs and outputs. On the other hand, no flexibility is lost since on the framework side each component of the structured inputs and outputs can be extracted and used singularly.

3.4.3 Model summary

During the review each element of the model has been revised and a thorough explanation has been added to all relevant elements. Furthermore, all element descriptions have been collected into an extensive model summary that shows every aspect of the model working principles.

Chapter 4

Final design of the equilibrium framework

In this chapter a complete plasma system is set up and its flexibility is tested by running test discharges in various magnetic configurations.

The plasma system uses

1. the *new components* of chapter 2
2. the *equilibrium and shape model* (chapter 3), run with the model wrapping component of section 2.2
3. a simple *signal pre elaboration* run with the mathematical function component (section 2.4)

Input signals are taken from previous discharges.

4.1 Plasma system outline

The outline of the plasma system is shown in fig. 4.1.

4.1.1 Components

The system encloses

1. a plasma equilibrium and shape model, run with the model wrapping component of section 2.2

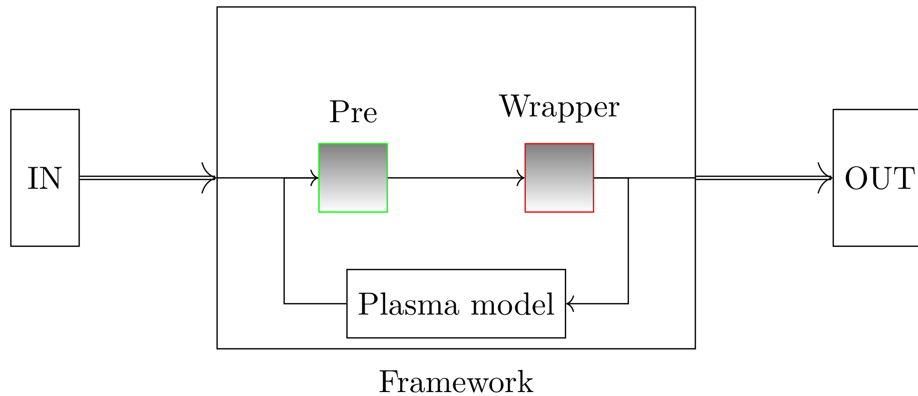

Figure 4.1: The figure shows a plasma real-time system using the components built in the previous chapters. `Pre` is the math function component carrying out the pre elaboration of signals. `Wrapper` is the model wrapping component running the plasma equilibrium and shape model.

2. a pre elaboration component, calculating some of the inputs required by the equilibrium and shape model

4.1.2 Inputs

The inputs for the model are taken from previous discharges. Some of the inputs require pre elaboration that was previously carried out by specific framework components. In this simple plasma system the math function component is used to carry out such calculations.

4.1.2.1 Pre elaboration of signals

The following signals are not available as measured quantities and must be calculated:

1. plasma current I_P
2. q factor

Based on the input calculation shown in section 3.1.1 the math function component is configured as follows to calculate the required input signals:

```
{
functions = "
  // I_P
```

```
4    R_vessel = 0.00115;
5    I_vessel = V_tor_loop/R_vessel;
6    I_P = I_phi - I_vessel;
7    // q
8    r_tiles = 0.46;
9    R = 1.99;
10   q = (6.28*r_tiles^2*B_phi)/(0.000001256*R*I_P);
11 "
12 inputs = {
13   I_phi = {
14     type = "double";
15     elements = "1";
16     };
17   V_tor_loop = {
18     type = "double";
19     elements = "1";
20     };
21   B_phi = {
22     type = "double";
23     elements = "1";
24     };
25   };
26 outputs = {
27   I_P = {
28     type = "double";
29     elements = "1";
30     };
31   };
32   q = {
33     type = "double";
34     elements = "1";
35     };
36   };
37 };
```

4.2 Outputs

In this section the output of the plasma system built in this work are compared with the outputs of a previous system. The PCS is run in the configuration

shown above.

The following outputs are considered:

1. magnetizing winding voltage reference V_{MW} for plasma current during the flat-top
2. field shaping coil voltage references V_{FS}. These coils generate the poloidal magnetic field required to drive the plasma horizontal shift, ellipticity and triangularity
3. equatorial saddle coil current reference I_{SC} for plasma vertical equilibrium

4.2.1 RFP circular discharge

Figures 4.2 to 4.4 show a RFP circular run with plasma current I_P up to 1 MA.

The voltage reference for the FS coils in an RFP run are shown in fig. 4.3.

4.2.2 Single null discharges

In this section single null configuration runs are shown.

4.2.2.1 Single null discharge

Figures 4.5 to 4.7 show an single null configuration run.

The voltage reference for the FS coils in a single-null run are shown in fig. 4.6.

4.2.2.2 Discharge ends at $t = 0.62\,\mathrm{s}$

Figures 4.8 to 4.10 show a single null run that ends at 0.62 s for having too much current on one of the field shaping coils.

4.2.2.3 β_θ variation

Figures 4.11 to 4.13 show a signle null run with β_θ (density) variation at $t = 0.6\,\mathrm{s}$ which is compensated by the plasma system.

4.2.2.4 MW voltage variation

Figures 4.14 to 4.16 show a single null run with MW voltage variation at $t = 0\,\mathrm{s}$.

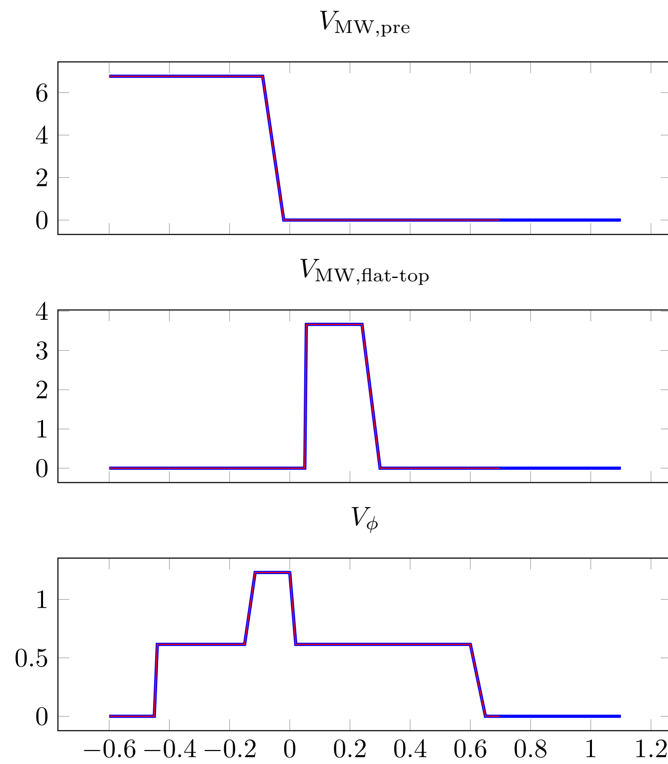

Figure 4.2: MW voltage references before and after $t = 0\,\mathrm{s}$ ([V]) and toroidal field coil reference V_ϕ ([V]) in an RFP discharge. The reference has been generated by the vertical shift element (see section 3.2.2.2). Blue: reference value. Red: the model run into the model wrapping component.

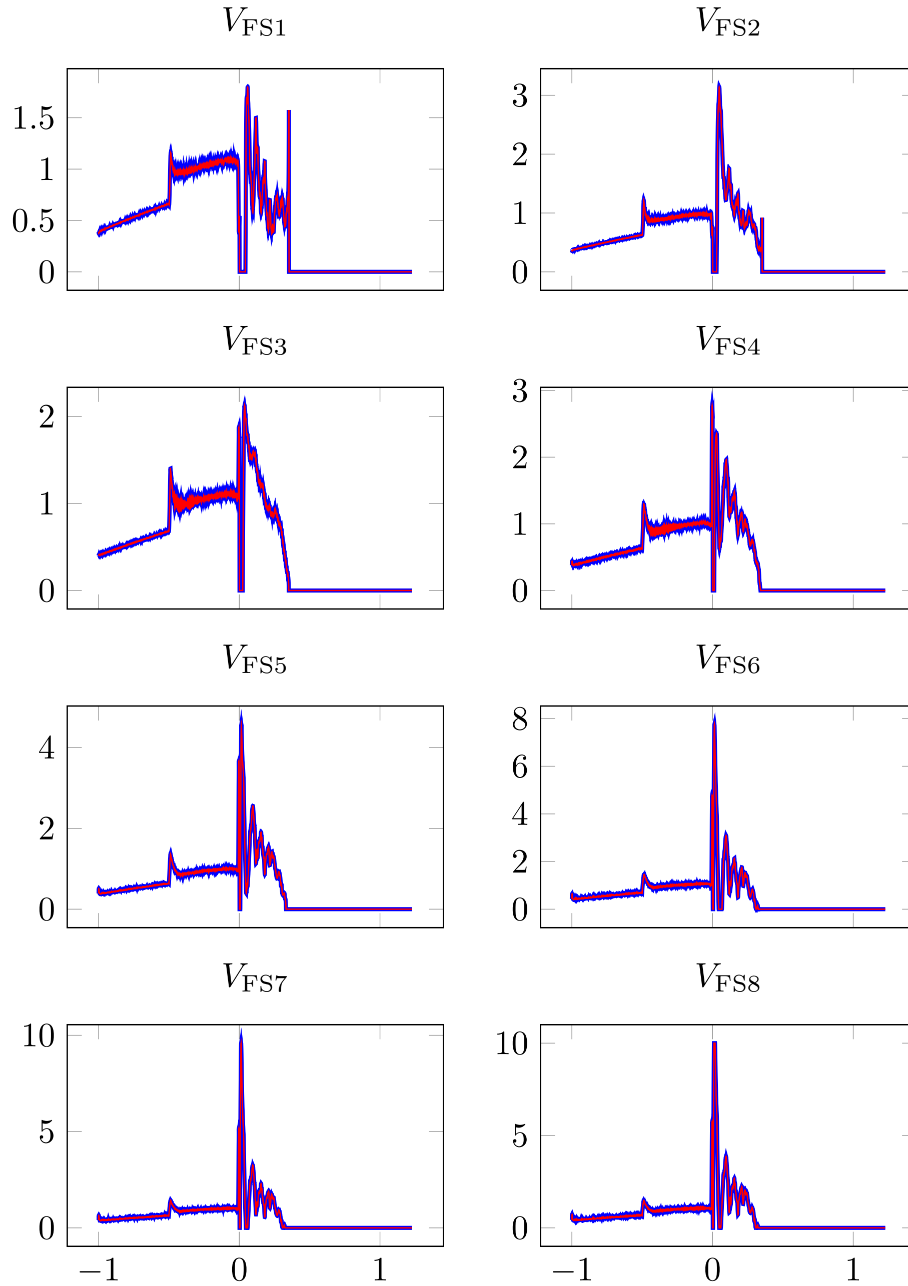

Figure 4.3: FS coil voltage reference [V] in an RFP discharge. Blue: reference value. Red: the model run into the model wrapping component.

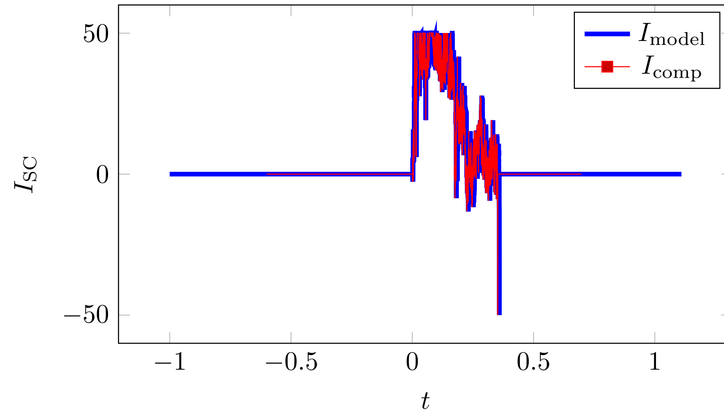

Figure 4.4: SC coil current reference [A] in an RFP discharge. The reference has been generated by the vertical shift element (see section 3.2.2.2). Blue: reference value. Red: the model run into the model wrapping component.

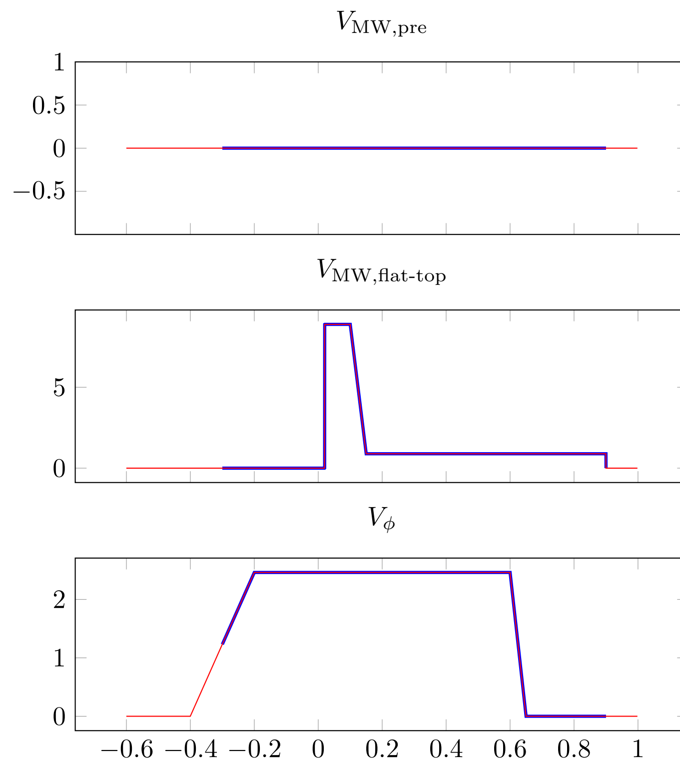

Figure 4.5: MW voltage references before and after $t = 0\,\mathrm{s}$ ([V]) and toroidal field coil reference V_ϕ ([V]) in a single null discharge. The reference has been generated by the vertical shift element (see section 3.2.2.2). Blue: reference value. Red: the model run into the model wrapping component.

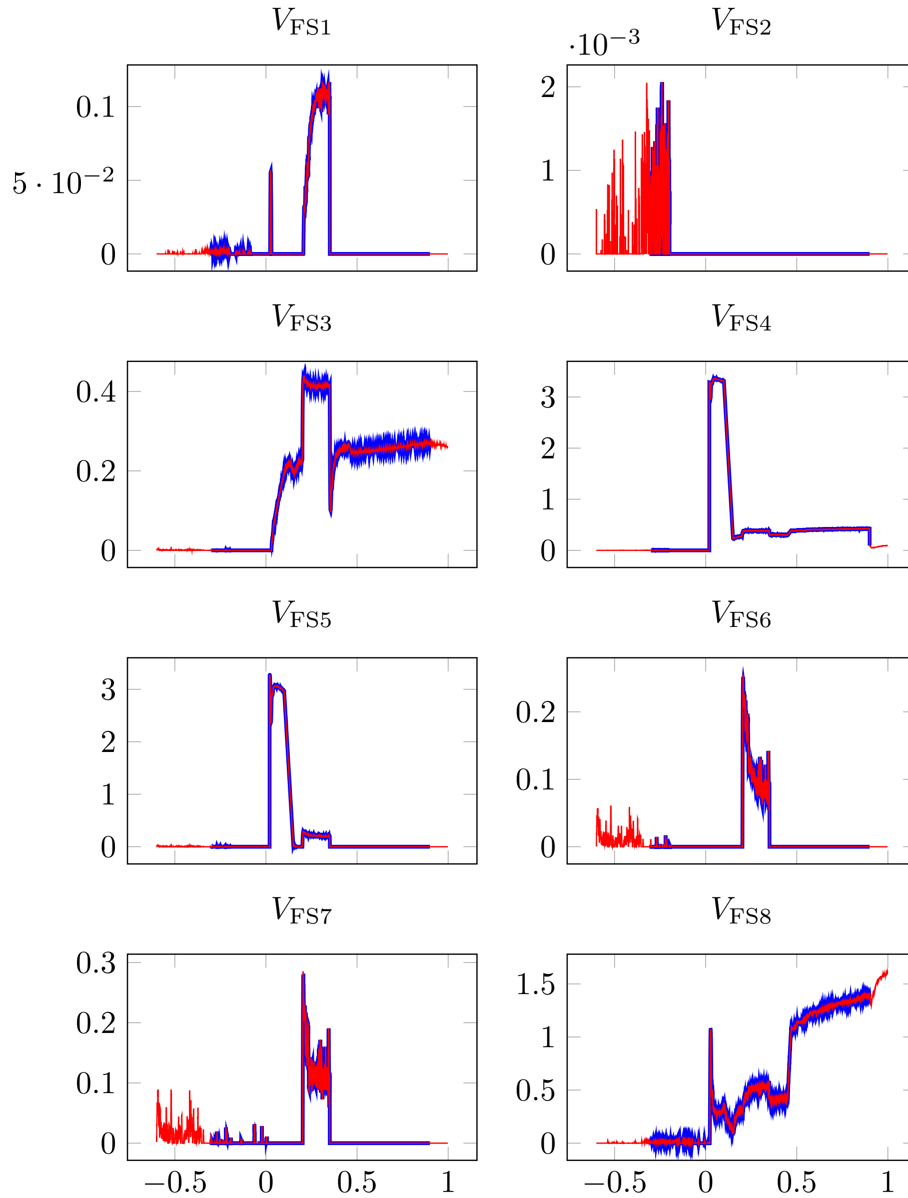

Figure 4.6: FS coil voltage reference [V] in a single null discharge. Blue: reference value. Red: the model run into the model wrapping component.

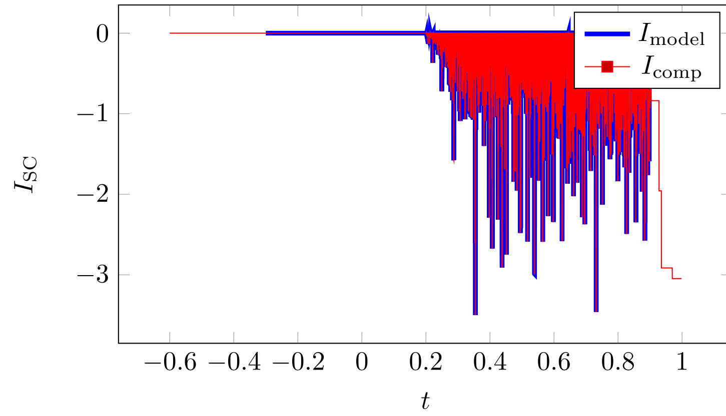

Figure 4.7: SC coil current reference [A] in a single null discharge. The reference has been generated by the vertical shift element (see section 3.2.2.2). Blue: reference value. Red: the model run into the model wrapping component.

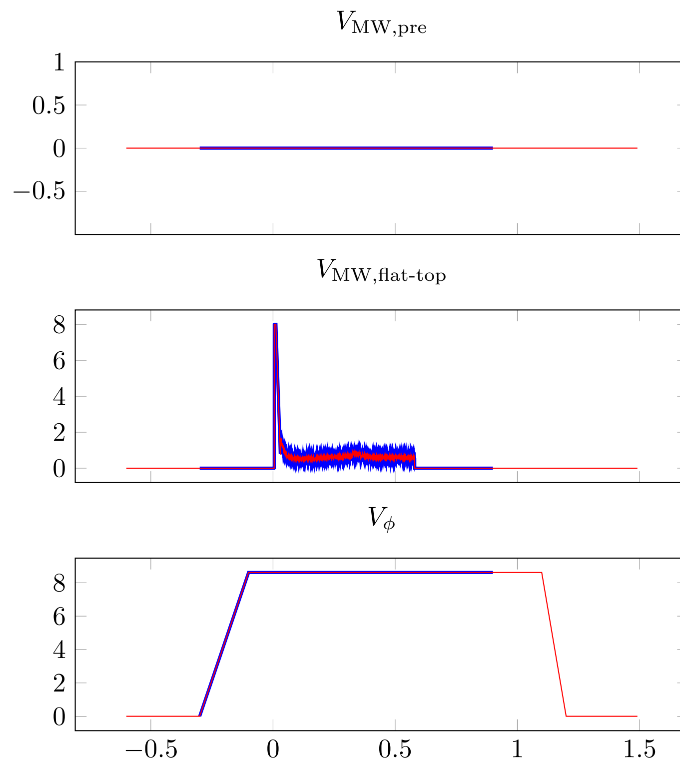

Figure 4.8: MW voltage references before and after $t = 0\,\mathrm{s}$ ([V]) and toroidal field coil reference V_ϕ ([V]) in a single null discharge that ends at $t = 0.62\,\mathrm{s}$. The reference has been generated by the vertical shift element (see section 3.2.2.2). Blue: reference value. Red: the model run into the model wrapping component.

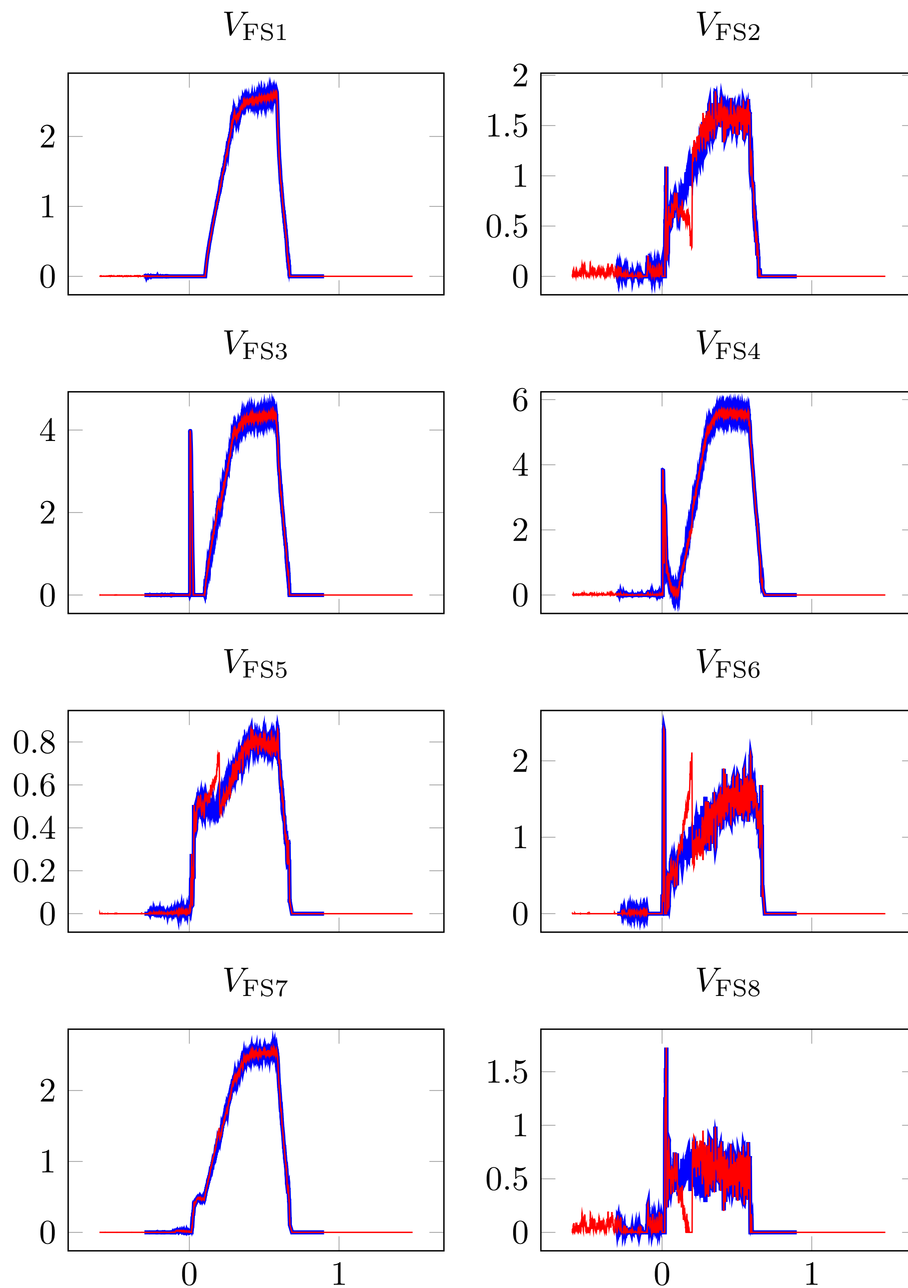

Figure 4.9: FS coil voltage reference [V] in a single null discharge that ends at $t = 0.62\,\text{s}$. Blue: reference value. Red: the model run into the model wrapping component.

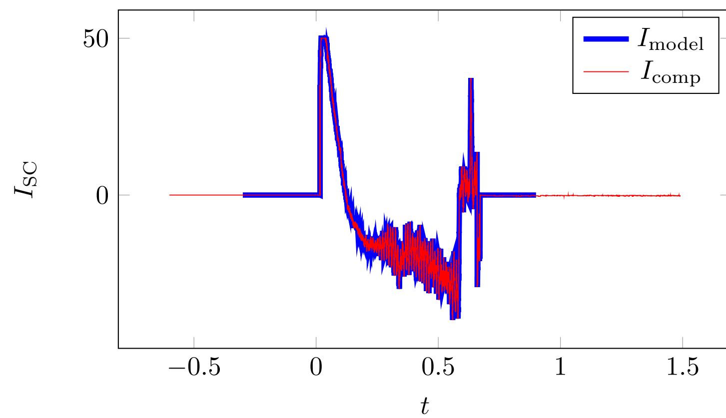

Figure 4.10: SC coil current reference [A] in a single null discharge that ends at $t = 0.62\,\mathrm{s}$. The reference has been generated by the vertical shift element (see section 3.2.2.2). Blue: reference value. Red: the model run into the model wrapping component.

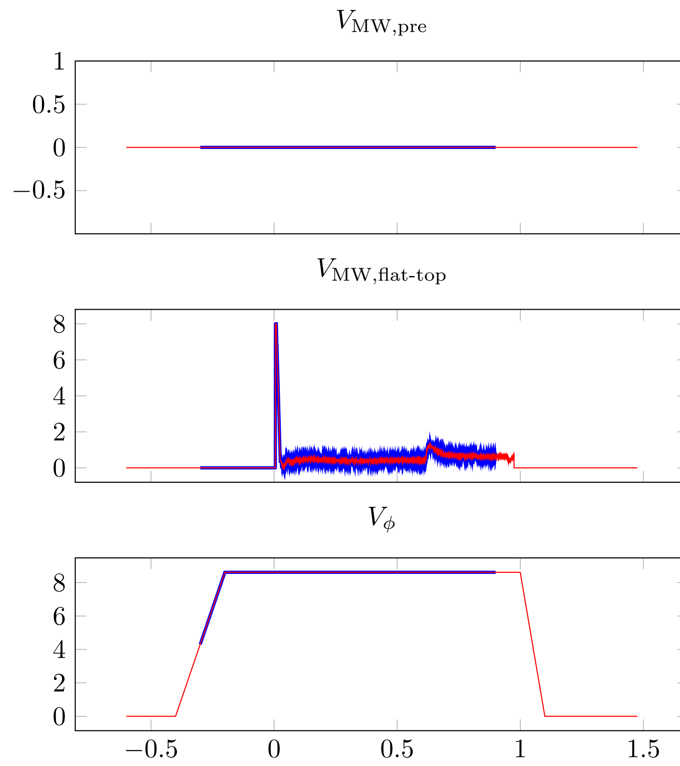

Figure 4.11: MW voltage references before and after $t = 0\,\mathrm{s}$ ([V]) and toroidal field coil reference V_ϕ ([V]) in a single null discharge with β_θ variation at $t = 0.6\,\mathrm{s}$. The reference has been generated by the vertical shift element (see section 3.2.2.2). Blue: reference value. Red: the model run into the model wrapping component.

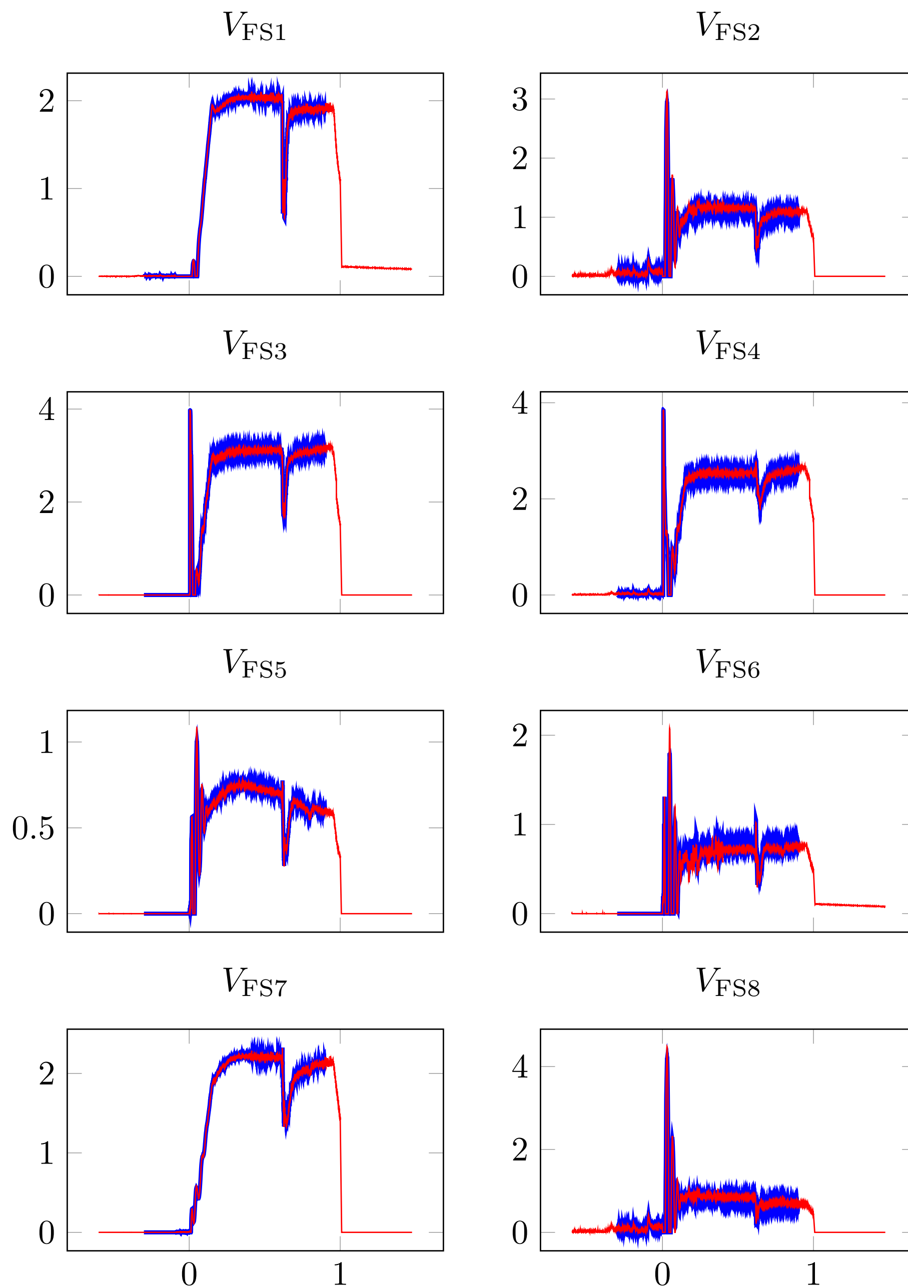

Figure 4.12: FS coil voltage reference [V] in a single null discharge with β_θ variation at $t = 0.6\,\mathrm{s}$. Blue: reference value. Red: the model run into the model wrapping component.

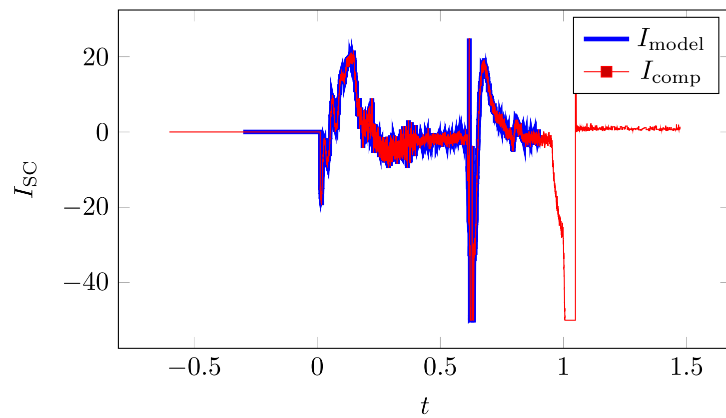

Figure 4.13: SC coil current reference [A] in a single null discharge with β_θ variation at $t = 0.6\,\text{s}$. The reference has been generated by the vertical shift element (see section 3.2.2.2). Blue: reference value. Red: the model run into the model wrapping component.

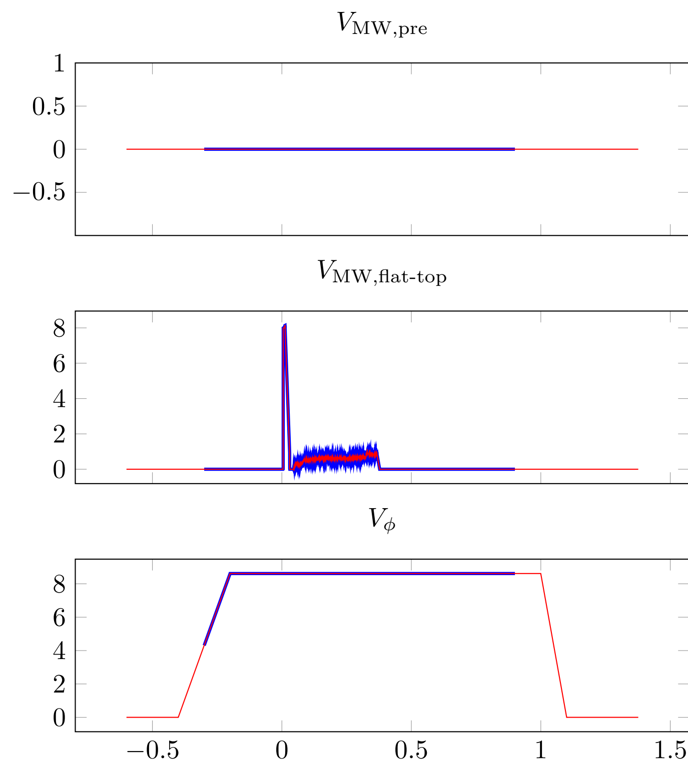

Figure 4.14: MW voltage references before and after $t = 0\,\mathrm{s}$ ([V]) and toroidal field coil reference V_{ϕ} ([V]) in a single null discharge with MW voltage variation at $t = 0\,\mathrm{s}$. The reference has been generated by the vertical shift element (see section 3.2.2.2). Blue: reference value. Red: the model run into the model wrapping component.

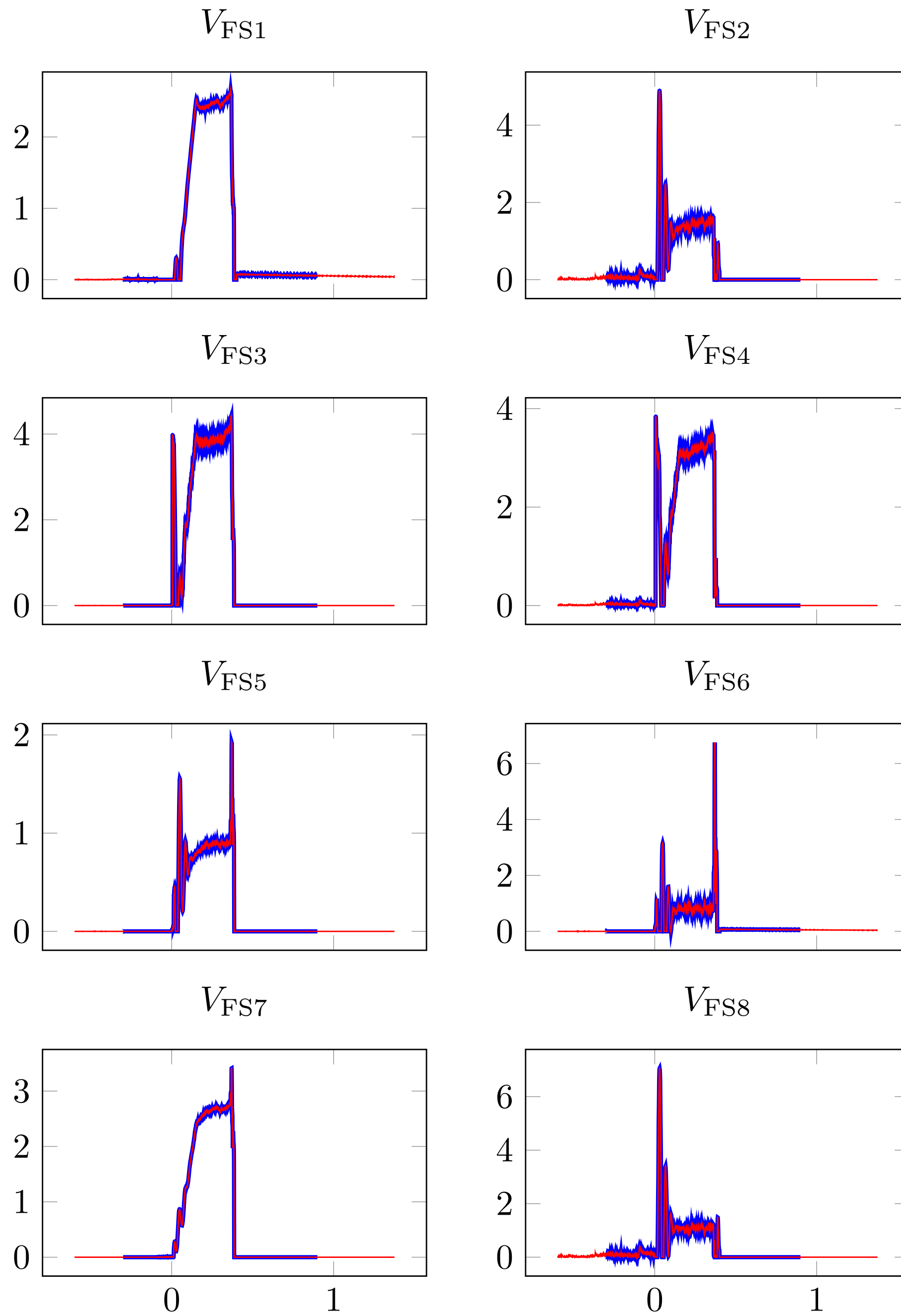

Figure 4.15: FS coil voltage reference [V] in a single null discharge with MW voltage variation at $t = 0\,\mathrm{s}$. Blue: reference value. Red: the model run into the model wrapping component.

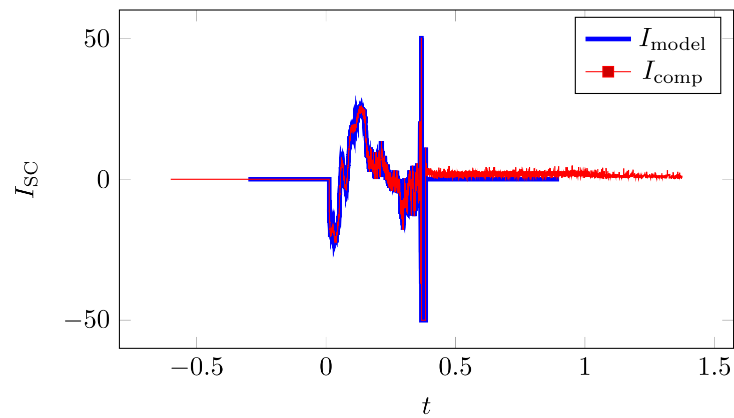

Figure 4.16: SC coil current reference [A] in a single null discharge with MW voltage variation at $t = 0\,\mathrm{s}$. The reference has been generated by the vertical shift element (see section 3.2.2.2). Blue: reference value. Red: the model run into the model wrapping component.

Chapter 5

Conclusions

As magnetic confinement fusion machines grow in number and dimensions, so does the related real-time system whose task is to elaborate outputs for the actuators based on inputs from diagnostics [2][1]. Outputs are calculated from inputs using a wide variety of models created in a great number of ways: tools offered to modellers include programming with compiled languages, programming with interpreted languages, graphical programming and mathematical models.

A fusion framework requires all models to be created with the same architecture the framework was built upon, or models with different architectures to be reworked so as to be run in the framework, which may introduce subtle differences with respect to the source model.

In this work, a flexible plasma system has been built. To do so, a fusion framework [2] [3] has been provided with the components required to interface with the most widely used modelling tools. Components interface with the framework signals and can be included seamlessly in the framework feedback loop.

A plasma equilibrium and shape model [4] has been reviewed and updated to adjust to new requirements for future machines and the graphical model wrapper has been used to include the equilibrium and shape model in the plasma framework.

A complete plasma system has been set up using the new components, which includes the equilibrium and shape model, a simple plasma current model run from a mathematical function. The plasma system flexibility is confirmed by running test discharges in various magnetic configurations, showing that such a plasma system is ready to be used in future fusion applications [5] [9].

Bibliography

[1] G. Manduchi et al. "From distributed to multicore architecture in the RFX-mod real time control system". In: *Fusion Engineering and Design* 89 (Mar. 2014), pp. 224–232.

[2] G. Manduchi et al. "The new feedback control system of RFX-mod based on the MARTe real-time framework". In: *2012 18th IEEE-NPSS Real Time Conference*. 2012, pp. 1–5.

[3] G. Manduchi et al. "MARTe2 and MDSplus integration for a comprehensive fast control and data acquisition system". In: *Fusion Engineering and Design* 161 (Dec. 2020), p. 111892.

[4] G. Marchiori et al. "Design and operation of the RFX-mod plasma shape control system". In: *Fusion Engineering and Design* 108 (Oct. 2016), pp. 81–91.

[5] G. Manduchi et al. "The upgrade of the control and data acquisition system of RFXMod2". In: *Fusion Engineering and Design* 167 (June 2021), p. 112329.

[6] G. Marchiori et al. "Upgraded electromagnetic measurement system for RFX-mod". In: *Fusion Engineering and Design* 123 (Mar. 2017).

[7] A. Stella et al. "The RFX magnet system". In: *Fusion Engineering and Design* (Apr. 1995), pp. 373–399.

[8] P. Bettini et al. "Adaptive plasma current control in RFX-mod". In: *Fusion Engineering and Design* 86 (Oct. 2011), pp. 1005–1008.

[9] G. Marchiori et al. "Advanced MHD mode active control at RFX-mod". In: *Fusion Engineering and Design* 84 (June 2009), pp. 1249–1252.