Appendix B. GSpulse Algorithm

Overview:

This appendix describes the GSPulse feedforward design tool in detail. GSPulse was introduced in section 4, and is a time-dependent equilibrium solver capable of optimizing for sequences of equilibria while including dynamic effects.

To recall, the objective of GSPulse is to minimize a quadratic cost function on the actuator effort and shaping errors:

$$\boxed{J = \sum_{k=1}^{N} ||u_{k-1}||^2_{W_u} + ||\Delta u_{k-1}||^2_{W_\Delta u} + ||\Delta^2 u_{k-1}||^2_{W^2_\Delta u} + ||e_k||^2_{W_e} + ||\Delta e_k||^2_{W_\Delta e} + ||\Delta^2 e_k||^2_{W^2_\Delta e}} \tag{B.1}$$

While subject to the Grad-Shafranov equilibrium force balance condition, axisymmetric conductor circuit dynamics, and Ip dynamics via the Ejima equation:

$$\boxed{\begin{aligned} \Delta^* \psi &= -\mu_0 R J_\phi \\ J_\phi &:= J^{pla}_\phi + J^{vac}_\phi \\ J^{pla}_\phi &= RP'(\psi) + \frac{FF'(\psi)}{\mu_0 R} \end{aligned}} \tag{B.2}$$

$$\boxed{u_i = R_i I_i + M_{ij}\dot{I}_j + \dot{\Phi}^{pla}_i} \tag{B.3}$$

$$\boxed{V_p = -\dot{\psi}_{bry} = R_p I_p + \frac{1}{I_p}\frac{\mathrm{d}}{\mathrm{d}t}\left(\frac{1}{2} L_I I_p^2\right)} \tag{B.4}$$

where,

u_k = actuator input (voltage) at time step k

$\Delta u_k := u_{k+1} - u_k$ = actuator input first-order-differences

$\Delta^2 u_k := u_{k+2} - 2u_{k+1} + u_k$ = actuator input second-order-differences

$W_{(\cdot)}$ = cost function weighting matrix

e_k = output errors such as isoflux shaping and field errors at time step k

$\Delta e_k := e_{k+1} - e_k$ = output error first-order differences

$\Delta^2 e_k := u_{k+2} - 2e_{k+1} + e_k$ = output error second-order differences

$\Delta^*(\cdot) = r\dfrac{\partial}{\partial r}\left(\dfrac{1}{r}\dfrac{\partial(\cdot)}{\partial r}\right) + \dfrac{\partial^2(\cdot)}{\partial z^2}$ = Grad-Shafranov operator

μ_0 = vacuum permeability constant

r = radial coordinate

J_ϕ = toroidal current density distribution

P = pressure

ψ = poloidal flux per unit radian

$F = rB_t$ = radius times toroidal magnetic field.

i, j = indices for conducting elements

R_i = resistance in conductor i

I_i = current in conductor i

M_{ij} = mutual inductance between conductors i and j

$\dot{\Phi}_i^{pla}$ = plasma coupling term, discussed in text

V_p = plasma loop voltage

ψ_{bry} = flux at plasma boundary

R_p = total plasma resistance

I_p = total plasma current

t = time

L_I = plasma internal inductance (unnormalized)

The core idea of the GSPulse algorithm is to use a Picard iteration scheme for the Grad-Shafranov plasma flux distribution coupled with a time-dependent, global, dynamic optimization of the vacuum conductors. This is analogous to the Picard iteration scheme employed in many static equilibrium solvers that alternates between updating the conductor currents and the Grad-Shafranov equilibrium. The difference is that with GSPulse we solve for more than 1 set of currents at a time, and also include the dynamics constraint within the optimization. Figure B.9 gives a cartoon of the convergence strategy used in GSPulse, where we alternate between the global MPC-like optimization and plasma Picard updates of the flux distribution.

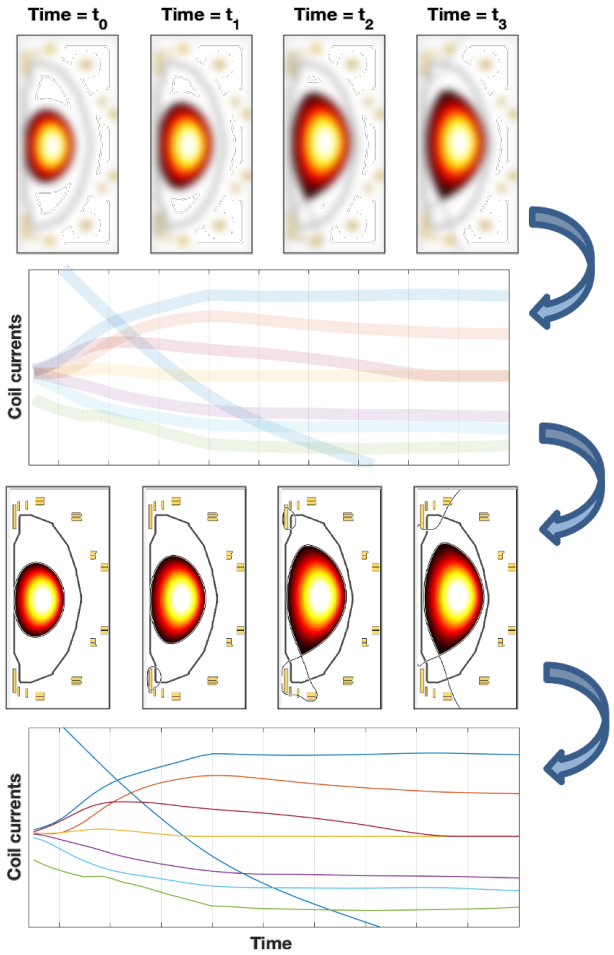

Figure B.9: Schematic of the algorithm process. The algorithm begins with a rough, un-converged estimate of the entire sequence of equilibria. Then, an optimization step plans the current evolution, before refining the estimate of the Grad-Shafranov flux distribution. The algorithm alternates between these two steps. A traditional method for feedforward design would be to solve for a single equilbirum timeslice and achieve complete convergence, then linearize the model, step forward in time, and solve for subsequent equilibrium. The GSPD method here allows for easier time-dependent penalties, such as having smooth trajectories.

GSPulse is written in MATLAB and Python and available open-source. Using GSPulse requires the user to define a standard set of inputs that are then passed to the solver. These include inputs that are typical to any equilibrium solver such as the machine geometry and mutual inductance tables, as well as parameters that are specific to the dynamic problem, such as weights on the dynamic evolution.

The required user inputs are: **A)** Specific settings for the optimization such as the time resolution and algorithmic choices. **B)** The machine geometry including coil resistances and inductances and limiter definition. Note that GSPulse can import geometry files from both the TokSys (tok_data_struct object) and MEQ (L object) code suites. **C)** The conductor initial condition, which is the starting coil currents and vessel currents at $t = 0$. The initial condition can also be specified as free parameters for the optimizer to solve for. **D)** The target shapes and weights. The shapes are defined in terms of isoflux control points on the target plasma boundary, and the GUI (figure ?) The weights are used to tune the outputs – for example they can be used to tune how important achieving a particular strike point is, or to penalize non-smooth trajectories for the currents. **E)** Core plasma properties that are used in the Grad-Shafranov equation. Multiple combinations of options are available but a typical

set would be the P' and FF' profile shapes, total plasma current evolution $I_p(t)$, and stored thermal energy evolution $W_{th}(t)$.

After parsing the user inputs, the GSPulse algorithm performs the following actions:

1. **Initialization:** Estimate the initial plasma current distribution at each time step.
2. **Compute target boundary flux:** Integrate the Ejima equation to find the target boundary flux.
3. **Optimize conductor evolution:** Set up and solve a QP for the conductor evolution.
4. **Grad-Shafranov Picard iteration:** Update the plasma flux distribution
5. **Convergence:** Repeat from Step 2 until convergence metrics are satisfied.

We will now describe each of the steps 1-4 in detail.

Step 1: Initialization

The purpose of this step is to obtain a rough estimate of the plasma current distribution. We use the target shape boundary and find the geometric centroid. Then each point on the grid is written in terms of a scaled distance x with x=0 at the centroid and x=1 at the boundary. The current is estimated with parabolic distribution:

$$J_\phi = \hat{J}(1 - x^a)^b \tag{B.5}$$

where a and b are constants and $\hat{J}$ is a constant scaled to match the target Ip.

Step 2: Calculate target boundary flux

The plasma internal inductance L_I can be measured from the current distribution. With initial condition $\psi_{bry}(t = 0)$ obtained from the starting equilibrium, the Ejima eq. (B.4) can be integrated to find the $\psi_{bry}(t)$ that provides the surface voltage to drive the target Ip.

Step 3: Optimize conductor evolution

This step updates the coil currents and applied flux, and is done by casting the update into the form of a quadratic optimization problem similar to model predictive control.

The first step is casting the circuit dynamics equation into state space form. The circuit equation is:

$$v_i = R_i I_i + M_{ij}\dot{I}_j + \dot{\Phi}_i^{pla} \tag{B.6}$$

We also have that the plasma-induced flux at any specific conductor is related through the grid mutuals and the plasma current distribution.

$$\dot{\Phi}_i^{pla} = M_{ig}\dot{I}_g \tag{B.7}$$

where M_{ig} is the mutual inductance between conductor i and each grid location g, and I_g is the plasma current in grid cell g. (Note the connection to the Grad-Shafranov equation: I_g is the plasma current distribution corresponding to the current density J_ϕ).

The conductor evolution is written for both coil and vessel elements. Combining eqs. (B.6) and (B.7) and being explicit about coils or vessel elements, we arrive at:

$$\begin{bmatrix} V_c \\ 0 \end{bmatrix} = R_{cv} \begin{bmatrix} I_c \\ I_v \end{bmatrix} + M_{cv,cv} \begin{bmatrix} \dot{I}_c \\ \dot{I}_v \end{bmatrix} + M_{cv,g} \dot{I}_g \tag{B.8}$$

Re-arranging this matrix equation gives us:

$$\dot{x} = Ax + Bu + w \tag{B.9}$$

where we have used the following substitutions

$$\begin{aligned} x &= [I_c^T \;\; I_v^T]^T \\ u &= V_c \\ A &= -M_{cv,cv}^{-1} R_{cv} \\ B &= M_{cv,cv}^{-1} \begin{bmatrix} I \\ 0 \end{bmatrix} \\ w &= -M_{cv,cv}^{-1} M_{cv,g} \dot{I}_g \end{aligned} \tag{B.10}$$

Note that w represents the influence of the plasma current on the conductors and can be calculated directly given the plasma current distribution at each time. Except for w which depends on the Grad-Shafranov solutions and is updated each iteration, the remaining dynamic terms are constant for a given machine (i.e. A and B do not change across GS iterations). This dynamics model is converted to discrete-time using the zero order hold method. For conciseness, we abuse the notation by using the same labels, making note that A, B, and w from hereon refer to their discrete time versions.

$$x_{k+1} = Ax_k + Bu_k + w_k \tag{B.11}$$

The shaping targets include parameters such as the flux at each of the control points, flux at target boundary location, and field at the target x-points. For any output y that is a linear function f of the grid flux distribution ψ_g (which is true for flux and field measurements) then the output can be represented:

$$\begin{aligned} y &= f(\psi_g) \\ &= f(\psi_g^{app}) + f(\psi_g^{pla}) \\ &= f(M_{g,cv} I_{cv}) + f(\psi_g^{pla}) \\ &:= f(M_{g,cv} I_{cv}) + y^{pla} \end{aligned} \tag{B.12}$$

In other words, the output is separated into a part that depends on the coil and vessel currents and part that depends on the plasma flux distribution. Moreover, for flux and field measurements the

function $f(\cdot)$ is a linear mapping (e.g. $f(x) = Cx$ where C is the greens functions for flux or field) so that we can write

$$y = CI_{cv} + y^{pla} \tag{B.13}$$

or using the state-space notation

$$y_k = Cx_k + y_k^{pla} \tag{B.14}$$

The dynamics eq. (B.11) and output eq. (B.14) form the basis of the prediction model for the optimization. In both equations we have intentionally separated the linear and nonlinear portions. The nonlinear terms w_k and y_k^{pla} are measured from the plasma current distribution and updated each iteration, while other terms can be computed once and re-used across iterations. Our cost function for the optimizer is:

$$J = \sum_{k=1}^{N} ||u_{k-1}||^2_{W_u} + ||\Delta u_{k-1}||^2_{W_\Delta u} + ||\Delta^2 u_{k-1}||^2_{W^2_\Delta u} + ||e_k||^2_{W_e} + ||\Delta e_k||^2_{W_\Delta e} + ||\Delta^2 e_k||^2_{W^2_\Delta e} \tag{B.15}$$

Where u_k is the power supply voltage and $e_k = r_k - y_k$ is the shape error between the reference target and actual, and the $W_{(x)}$ are user-defined weights on the magnitude, derivative, and second derivative of the voltages and errors.

The next few steps map the cost function into a standard quadratic program which can be solved by many available software packages such as MatLab's **quadprog**. We will transform the above cost function into the standard quadprog form

$$J = \hat{p}^T H \hat{p} + 2 f^T \hat{p} \tag{B.16}$$

To begin, we make the following definitions:

$$\begin{aligned}
\hat{u} &:= [u_0^T \; u_1^T \; ... \; u_{N-1}^T]^T \\
\hat{\Delta u} &:= [(u_1 - u_0)^T \; (u_2 - u_1)^T \; ... \; (u_{N-1} - u_{N-2})^T]^T \\
\hat{\Delta^2 u} &:= [(u_2 - 2u_1 + u_0)^T \; (u_3 - 2u_2 + u_1)^T \; ... \; (u_{N-1} - 2u_{N-2} + u_{N-3})^T]^T \\
\hat{e} &:= [e_1^T \; e_2^T \; ... \; e_N^T]^T \\
\hat{\Delta} e &:= [(e_2 - e_1)^T \; (e_3 - e_2)^T \; ... \; (e_N - e_{N-1})^T]^T \\
\hat{\Delta}^2 e &:= [(e_3 - 2e_2 + e_1)^T \; (e_4 - 2e_3 + e_2)^T \; ... \; (e_N - 2e_{N-1} + e_{N-2})^T]^T \\
\hat{w} &:= [w_0^T \; w_1^T \; ... \; w_{N-1}^T]^T \\
\hat{x} &:= [x_1^T \; x_2^T \; ... \; x_N^T]^T \\
\hat{y} &:= [y_1^T \; y_2^T \; ... \; y_N^T]^T \\
\hat{r} &:= [r_1^T \; r_2^T \; ... \; r_N^T]^T \\
\hat{C} &:= \mathbf{blkdiag}(\underbrace{C, C \; ... \; C}_{\times N}) \\
\hat{W}_{(x)} &:= \mathbf{blkdiag}(\underbrace{W_{(x)}, W_{(x)} \; ... \; W_{(x)}}_{\times N}), \qquad x \in u, \Delta u, \Delta^2 u, e, \Delta e, \Delta^2 e
\end{aligned} \tag{B.17}$$

Note that we can write matrix mappings between the $\hat{u}$ and $\Delta\hat{u}$, since these are just linear combinations of each other.

$$\hat{\Delta u} = \underbrace{\begin{bmatrix} -I & I & & \\ & \ddots & & \\ & & -I & I \end{bmatrix}}_{:=S_\Delta u} \hat{u} \tag{B.18}$$

Similarly for the 2nd derivatives, we have that:

$$\hat{\Delta^2 u} = \underbrace{\begin{bmatrix} I & -2I & I & & \\ & & \ddots & & \\ & & I & -2I & I \end{bmatrix}}_{:=S_\Delta^2 u} \hat{u} \tag{B.19}$$

And for equivalently-defined transition matrices for the errors, we have that:

$$\begin{aligned} \hat{\Delta e} &= S_{\Delta e}\hat{e} \\ \hat{\Delta^2 e} &= S_{\Delta^2 e}\hat{e} \end{aligned} \tag{B.20}$$

Thus the first 3 terms and last 3 terms in the cost function eq. (B.1) can be combined to rewrite the cost function as a matrix equation:

$$\begin{aligned} J &= \hat{u}^T H_u \hat{u} + \hat{e}^T H_e \hat{e} \\ H_u &:= \hat{W}_u + S_{\Delta u}^T \hat{W}_{\Delta u} S_{\Delta u} + S_{\Delta^2 u}^T \hat{W}_{\Delta^2 u} S_{\Delta^2 u} \\ H_e &:= \hat{W}_e + S_{\Delta e}^T \hat{W}_{\Delta e} S_{\Delta e} + S_{\Delta^2 e}^T \hat{W}_{\Delta^2 e} S_{\Delta^2 e} \end{aligned} \tag{B.21}$$

Now, lets turn our attention to the dynamics. Using a technique from Model Predictive Control, we can form a prediction model of the dynamic evolution. From the state-space dynamics equation we have that:

$$\begin{aligned} x_1 &= x_1 \\ x_2 &= Ax_1 + Bu_1 + w_1 \\ x_3 &= Ax_2 + Bu_2 + w_2 \\ &= A(x_1 + Bu_1 + w_1) + Bu_2 + w_2 \\ &= A^2 x_1 + \begin{bmatrix} AB & B \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} + \begin{bmatrix} A & I \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \end{bmatrix} \\ x_4 &= ... \end{aligned} \tag{B.22}$$

Extending this to all times gives:

$$\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_N \end{bmatrix} = \underbrace{\begin{bmatrix} I \\ A \\ \vdots \\ A^N \end{bmatrix}}_{E} x_1 + \underbrace{\begin{bmatrix} 0 & & & & \\ 0 & B & & & \\ 0 & AB & B & & \\ \vdots & \vdots & \ddots & & \\ 0 & A^{N-1}B & A^{N-2}B & \dots & B \end{bmatrix}}_{F} \begin{bmatrix} u_0 \\ u_1 \\ \vdots \\ u_{N-1} \end{bmatrix} + \underbrace{\begin{bmatrix} 0 & & & & \\ 0 & I & & & \\ 0 & A & I & & \\ \vdots & \vdots & \ddots & & \\ 0 & A^{N-1} & A^{N-2} & \dots & I \end{bmatrix}}_{F_w} \begin{bmatrix} w_0 \\ w_1 \\ \vdots \\ w_{N-1} \end{bmatrix} \tag{B.23}$$

Or equivalently,

$$\hat{x} = Ex_1 + F\hat{u} + F_w\hat{w} \tag{B.24}$$

Then using eq. (B.14) the predicted errors are:

$$\begin{aligned} \hat{e} &= \hat{r} - \hat{y} \\ &= \hat{r} - \hat{y}^{pla} - \hat{C}\hat{x} \\ &= \hat{r} - \hat{y}^{pla} - \hat{C}(Ex_1 + F\hat{u} + F_w\hat{w}) \\ &:= M\hat{p} + d \end{aligned} \tag{B.25}$$

where to obtain the last line we have defined new variables,

$$\begin{aligned} M &:= -\hat{C}\begin{bmatrix} E & F \end{bmatrix} \\ \hat{p} &:= \begin{bmatrix} x_1 \\ \hat{u} \end{bmatrix} \\ d &:= \hat{r} - \hat{y}^{pla} - \hat{C}F_w\hat{w} \end{aligned} \tag{B.26}$$

Note that $\hat{p}$ is the solution vector for the quadratic program, that it is, it is the same $\hat{p}$ as in eq. (B.16). $\hat{p}$ consists of the initial starting currents x_1 and the voltages at all times. The elements of x_1 can be specified either as constrained parameters, for the use case where an initial condition is specified, or they can be specified as free parameters for the optimizer to solve for.

The relation between the voltages and the primary optimization variables $\hat{p}$ is:

$$\hat{u} = \underbrace{\begin{bmatrix} 0 & \\ & I \end{bmatrix}}_{:=T_{up}} \hat{p} \tag{B.27}$$

Then, plugging in eqs. (B.25) and (B.27) into eq. (B.21), we arrive finally at the standard form of the quadratic program:

$$
\begin{aligned}
J &= \hat{u}^T H_u \hat{u} + \hat{e}^T H_e \hat{e} \\
&= \hat{p}^T T_{up}^T H_u T_{up} \hat{p} + (M\hat{p} + d)^T H_e (M\hat{p} + d) \\
&= \hat{p}^T H \hat{p} + 2 f^T \hat{p} \\
H &:= T_{up}^T H_u T_{up} + M^T H_e M \\
f &:= M^T H_e^T d
\end{aligned}
\tag{B.28}
$$

Solving the quadratic program eq. (B.28) can be done with standard QP solver packages such as **quadprog** in MATLAB. The solution to the QP is the initial starting currents as well as the trajectory of voltages. The voltage trajectory can be mapped back to the trajectory of currents and errors, via eq. (B.24) and eq. (B.25).

Constraints on the voltages, currents, and outputs can be added to the QP problem via the mappings presented in this derivation. In the standard QP form, constraints enter in the form of $A\hat{p} < b$, which is a linear mapping of the primary optimization variables. We will now describe how to put various types of constraints in thhis form. For constraints on the voltage ($\hat{u}$), which would include restrictions such as power supply voltage limits or slewrate limits, we use eq. (B.27) to write:

$$
\begin{aligned}
& A_u \hat{u} < b_u \\
\implies & A_u T_{up} \hat{p} < b_u
\end{aligned}
\tag{B.29}
$$

Similarly for constraints on the coil currents ($\hat{x}$) we use eq. (B.24) to write:

$$
\begin{aligned}
& A_x \hat{x} < b_x \\
\implies & A_x (E x_1 + F\hat{u} + F_w \hat{w}) < b_x \\
\implies & A_x ([E\ F]\hat{p} + F_w \hat{w}) < b_x \\
\implies & A_x [E\ F]\hat{p} < b_x - A_x F_w \hat{w}
\end{aligned}
\tag{B.30}
$$

And for constraints on the output errors ($\hat{e}$) we use eq. (B.25) to write:

$$
\begin{aligned}
& A_e \hat{e} < b_e \\
\implies & A_e (M\hat{p} + d) < b_e \\
\implies & A_e M \hat{p} < b_e - A_e d
\end{aligned}
\tag{B.31}
$$

Lastly, we conclude with a few observations about this QP.

- Note that, if only 1 time step is being solved for ($N = 1$), then the dynamical prediction model shrinks to identity and the QP only solves for the x_1 set of currents. In other words, it behaves exactly as a standard equilibrium solver updating a single set of currents at each iteration. This means that, in addition to being a dynamic equilibrium trajectory solver, GSPulse can also function as a standard static equilibrium solver, since the static problem is a sub-problem of the dynamic one.
- Assuming that the state-space output matrix $\hat{C}$ is linear, then H stays constant across iterations and only d (eq. (B.26)) needs to be updated. This means that many of the matrix objects can be pre-computed only one time and re-used, saving on computational time, since a large fraction

of the compute is absorbed by forming and multiplying the large matrices. Therefore it is very advantageous to ensure that the output matrix $\hat{C}$ is linear and does not need to be updated, which again is the primary advantage of specifying outputs in terms of flux units instead of spatial units (see discussion of section 3). For outputs that have units of B-field, flux, or current, then $\hat{C}$ is linear.

Step 4: Grad-Shafranov Picard iteration

In this step, the goal is to perform a Picard update the total flux distribution according to the Grad-Shafranov equation in order to converge the solution numerically. The full Picard iteration process is given as follows.

In the QP optimizer step, we computed the trajectory currents in the external conductors. (Actually, the QP solver obtained the initial currents and trajectory of voltages, but these are mapped to the conductor currents via the prediction model eq. (B.24).) This allows us to update the vacuum flux distribution via:

$$\psi_{vac}^{k+1} = M_{g,cv} I_{cv}^{k+1}, \tag{B.32}$$

where k is the iteration index. Next, we do a partial update of the total flux distribution, taking the partially-updated flux as a sum of the previous plasma flux contribution and the updated vacuum flux.

$$\psi^{k+1/2} := \psi_{pla}^{k} + \psi_{vac}^{k+1} \tag{B.33}$$

The remaining steps are to update the P' and FF' profiles and total flux of the Grad-Shafranov equation:

$$\begin{aligned} J_{\phi,pla}^{k+1} &= RP'(\psi^{k+1/2}) + \frac{FF'(\psi^{k+1/2})}{\mu_0 R} \\ \Delta^* \psi^{k+1} &= -\mu_0 R \left(J_{\phi,pla}^{k+1} + J_{\phi,vac}^{k+1} \right) \end{aligned} \tag{B.34}$$

The above relations describe the Picard update process which is a common method for solving Grad-Shafranov equilibria. We will now describe a few of the details specific to the GSPulse solver. The present implementation of GSPulse allows for two different methods of constraining the P' and FF' profiles.

The first method is that GSPulse has been integrated with the FBT code developed by EPFL that is part of the MEQ toolbox, which is free to access and use with an academic license. FBT supports a number of flexible and fast methods for constraining the P' and FF' profiles, including various basis function types for the profile shapes, and multiple options to specify combinations of core plasma properties such as the total plasma current I_p, total stored thermal energy W_{th}, plasma beta β_p, or q-profile value on-axis q_A. Examples of using GSPulse with FBT for the plasma picard solver are available in the GSPulse repo.

The second method is a built-in GSPulse option that does not require installation of the FBT/MEQ codes, but has slightly less flexibility in specification of the core plasma properties. With this method, the user must specify input basis function shapes for the P' and FF' profiles, as well as input waveforms

for the plasma current I_p and thermal energy W_{th}. At each iteration, GSPulse scales the coefficients multiplying the basis function profiles in order to match the target I_p and W_{th}.

To describe the built-in method with more detail:

After using the optimizer to obtain the updated conductor currents, we compute $\psi_{k+1/2}$ from eq. (B.33) which is the partially-updated flux on the grid. Then we use touch-point finder, x-point finder, and o-point finder algorithms on this flux distribution. Out of the identified potential touch-points, x-points, and o-points, we select the magnetic axis and boundary-defining point of the flux distribution (where the boundary-defining point is either a touch-point or an x-point). The algorithm to identify which points are the magnetic axis and boundary follows the logic presented in [1] section 2.3, which is an efficient method for identifying the boundary. The output of all these steps is to identify the flux at the magnetic axis ψ_{mag} and the flux at the boundary ψ_{bry} as well as to identify the physical boundary of the plasma.

Since the user has provided input profile shapes on a normalized ψ basis, our job is to appropriately scale the coefficients in order to match the target I_p and W_{th}. Let the input basis shapes be $P'_b(\psi_N)$ and $FF'_b(\psi_N)$ where ψ_N is the normalized flux:

$$\psi_N := \frac{\psi - \psi_{bry}}{\psi_{mag} - \psi_{bry}} \tag{B.35}$$

The total profiles are then:

$$\begin{aligned} P'(\psi) &= c_p P'_b(\psi_N) \;/\; (\psi_{mag} - \psi_{bry}) \\ FF'(\psi) &= c_f FF'_b(\psi_N) \;/\; (\psi_{mag} - \psi_{bry}) \end{aligned} \tag{B.36}$$

The scaling coefficients c_p and c_f are terms that we will solve for, and the last term arises from de-normalizing the ψ basis, since for any parameter x:

$$\begin{aligned} \frac{dx}{d\psi} &= \frac{dx}{d\psi_N}\frac{d\psi}{d\psi_N} \\ &= \frac{dx}{d\psi_N}\frac{1}{(\psi_{mag} - \psi_{bry})} \end{aligned} \tag{B.37}$$

The first constraint is on the total plasma current,

$$I_p = \int_{\Omega_p} J_{\phi,pla} dA \tag{B.38}$$

Substituting in the Grad-Shafranov eq. (B.2), we obtain that

$$I_p = \frac{1}{\psi_{mag} - \psi_{bry}} \left[c_p \int_{\Omega_p} R P'_b dA + c_f \int_{\Omega_p} \frac{FF'_b}{\mu_0 R} dA \right] \tag{B.39}$$

This equation depends linearly on the profile coefficients, while the integrals are computed numerically at each iteration. The second constraint is on the stored thermal energy W_{th}, where we have that:

$$\begin{aligned} W_{th} &= \int_{\Omega_p} \frac{3}{2} P(\psi_N) dV \\ &= \int_{\Omega_p} 3\pi R P(\psi_N) dA \end{aligned}, \tag{B.40}$$

where the pressure is found from integrating the P' basis function from the edge of the plasma inwards (assuming that the pressure is zero at the edge).

$$P(\psi_N) = c_p \int_{\hat{\psi}_N=1}^{\hat{\psi}_N=\psi_N} P'_b d\hat{\psi}_N \tag{B.41}$$

The linear system of 2 equations (eqs. (B.39) and (B.41)) and 2 unknowns can be solved to obtain the profile coefficients. This gives us the necessary total P' and FF' profiles, which are used to obtain the total flux distribution according to eq. (B.34).

At this stage we have now updated the plasma current and total flux distributions, and the solver can proceed to the next iteration.

Appendix B: Additional algorithm details

Appendix B.1. ***Spline basis compression***

The computational complexity of obtaining an optimal solution to the QP (eq. (B.28)) can vary dramatically depending on the number of equilibria that are solved for and the complexity of the evolution. Solving this type of optimization problem can have poor numerical scaling, since in general the number of optimization variables that is solved for is $n = (N_{equlibria}+2)*N_{coils}$, and the theoretical time complexity for solving linear-constrained quadratic programs is $O(n^3)$. In practice the run-time performance also depends on the complexity of the equilibrium evolution; observationally, highly dynamic events such as strike point sweeping take more time to solve. Therefore, it is advantageous to employ methods that reduce the computational complexity of the QP perhaps at the expense of sacrificing some of the optimality of the solution.

One technique for data compression is to use a spline basis for the primary optimization variables. If we have a spline basis of the form,

$$\hat{p} = S_m \hat{c} \tag{B.42}$$

where $\hat{p}$ is the original primary variables, S_m is a splining matrix, and $\hat{c}$ are the spline basis coefficients. Then, instead of solving for the original variables we optimize for the spline coefficients. The quadratic program is then transformed as:

$$\begin{aligned} \min\ J &= \hat{p}^T H \hat{c} + 2 f^T \hat{p} \\ &= \hat{c}^T S_m^T H S_m \hat{c} + 2 f^t S_m \hat{c} \\ \text{subj to:} & \\ & A\hat{p} \leq b \\ \implies & A S_m \hat{c} \leq b \end{aligned} \tag{B.43}$$

After solving for the spline coefficients, the original variables are constructed from the spline basis above eq. (B.42).

Appendix B.2. ***Optimization with multiple time intervals***

Another important technique for reducing the time complexity of the QP-solve is to solve for the evolution in multiple stages. For example, instead of solving for 200 equilibria at a time, we may solve for 50 equilibria at a time across 4 overlapping stages reducing the complexity from $O(200^3)$ to $4 \times O(50^3)$. This reduces the optimality of the solve since the first stage is not looking far into the future. However, in many cases this optimality reduction is not significant, i.e. details of the rampdown trajectory do not significantly affect the details of the rampup trajectory.

GSPulse enables this multiple-interval approach by overlapping the the equilibria from one stage to the next. The 1^{st} and 2^{nd} equilibria are automatically enforced to be the $(N-1)^{th}$ and N^{th} equilibria from the previous stage. These equality constraints enforce continuity of the trajectory, and also give smooth trajectories provided that the user has weighting penalties on the second derivative (non-smooth trajectories).

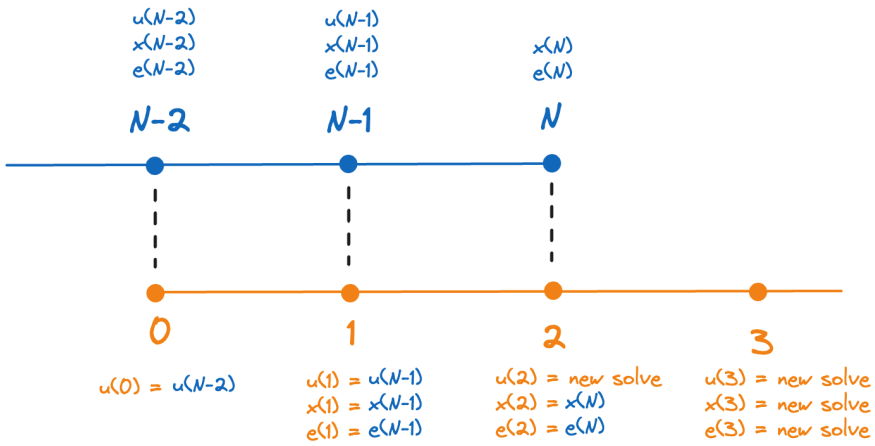

Figure B.10: Overlapping strategy when using GSPulse to optimize across multiple intervals. The previous stage is shown in blue and subsequent stage in orange. The stages are overlapped by 2 equilibria (2 time points each for the voltages u, currents x, and shaping errors e).

The multiple-interval strategy is shown in fig. B.10. As shown in the figure, the subsequent stage needs to enforce equality constraints on u_0, u_1, x_1, e_1, x_2 and e_2.

- The terms x_1, u_0 and u_1 make up the first three terms of the optimization variables $\hat{p}$ (see eq. (B.26)). Therefore, these terms can be enforced by equality constraint on the QP solver.
- The equality constraint on x_2 is enforced automatically and implicitly through the other terms, since $x_2 = Ax_1 + Bu_1 + w_1$ and all terms on the RHS are specified.
- The remaining terms are e_1 and e_2. Our prediction model for the errors was eq. (B.25) and repeated here:

$$\hat{e} := \begin{bmatrix} e_1 \\ e_2 \\ e_3 \\ \vdots \\ e_N \end{bmatrix} = M\hat{p} + d \tag{B.44}$$

We achieve equality constraint by replacing the rows of M and elements of d corresponding to e_1 and e_2:

$$\begin{bmatrix} e_1 \\ e_2 \\ e_3 \\ \vdots \\ e_N \end{bmatrix} = \underbrace{\left[\begin{array}{c} 0 \\ 0 \\ \hline M_{((2\times n_e+1):N,:)} \end{array}\right]}_{\tilde{M}} \hat{p} + \underbrace{\left[\begin{array}{c} e_1 \\ e_2 \\ \hline d_{(2\times n_e+1):N} \end{array}\right]}_{\tilde{d}} \tag{B.45}$$

Then, eq. (B.45) is used in place of eq. (B.44) when constructing the QP, for any stage after the first.

References

[1] J.-M. Moret, B. Duval, H. Le, S. Coda, F. Felici, H. Reimerdes, Tokamak equilibrium reconstruction code liuqe and its real time implementation, Fusion Engineering and Design 91 (2015) 1–15. URL: http://dx.doi.org/10.1016/j.fusengdes.2014.09.019. doi:10.1016/j.fusengdes.2014.09.019.