List of Supporting Documents

A. Derivation of the Alternating Direction Method of Multipliers and IMPULSE

B. Vectorized SAR oracle

- Supporting Information Figure S1
- Supporting Information Figure S2

C. FAI Update

- Supporting Information Figure S3

D. VOP compression

- Supporting Information Figure S4
- Supporting Information Figure S5

E. Tuning the IMPULSE algorithm

- Supporting Information Figure S6
- Supporting Information Video S1 (rho_small.avi)
- Supporting Information Video S2 (rho_best.avi)
- Supporting Information Video S3 (rho_large.avi)

F. Flip Angle Maps

- Supporting Information Figure S7

G. High-Level Overview

- Supporting Information Figure S8

A. Derivation of the Alternating Direction Method of Multipliers and IMPULSE

We want to solve a problem of the form

Problem 1:

$$\text{MINIMIZE } g(y) + h(z)$$
$$\text{SUBJECT TO } y = z$$

where $g(y)$ and $h(z)$ are possibly nonconvex functions.

This is equivalent to solving

Problem 2:

$$\text{MAXIMIZE } v(w) = \min_{y,z}(g(y) + h(z) + w^T(y - z))$$

The reasoning for this equivalency is that for a given w, $v(w)$ will give a lower bound on the optimal value for Problem 1. The maximum of this lower bound over all values of w will give the solution of Problem 1. The constraint $y = z$ is implicitly satisfied by solving Problem 2 since if $y \neq z$ then it will always be possible to find some w such that $v(w)$ is infinity. Problem 2 is called the dual problem of Problem 1 (which is called the primal problem) and solving this dual problem is a standard approach for equality constrained optimization. y and z are called primal variables and w is called the dual variable or Lagrange multiplier (hence the name method of multipliers). To solve Problem 2, a gradient ascent approach can be used with the basic idea of computing the gradient (or technically subgradient if nondifferentiable) of the dual cost function with respect to the dual variable w and iteratively updating w with a step in the positive gradient direction. The gradient (at iteration k) can be computed in a two step manner:

1. $\left(y^{(k+1)}, z^{(k+1)}\right) = \text{argmin}_{y,z}(g(y) + h(z) + {w^{(k)}}^T(y - z))$
2. $\nabla v\left(w^{(k)}\right) = y^{(k+1)} - z^{(k+1)}$

Then the update of w is

$$w^{(k+1)} = w^{(k)} + \alpha(y^{(k+1)} - z^{(k+1)})$$

where α is the step size.

Since the expression $g(y) + h(z) + {w^{(k)}}^T(y - z)$ is completely separable in y and z, $\left(y^{(k+1)}, z^{(k+1)}\right) = \text{argmin}_{y,z}(g(y) + h(z) + {w^{(k)}}^T(y - z))$ can be easily decomposed into $y^{(k+1)} = \text{argmin}_y(g(y) + {w^{(k)}}^T y)$ and $z^{(k+1)} = \text{argmin}_z(h(z) + {w^{(k)}}^T z)$. Unfortunately the algorithm as stated has poor convergence properties. This can be improved by adding a penalty or regularization term to the objective of problem 1:

$$\text{MINIMIZE } g(y) + h(z) + \frac{\rho}{2}\|y - z\|_2^2$$
$$\text{SUBJECT TO } y = z$$

This is equivalent to the previous problem since when the constraint is met, the penalty is 0. The algorithm is now:

1. $\left(y^{(k+1)}, z^{(k+1)}\right) = \text{argmin}_{y,z}(g(y) + h(z) + {w^{(k)}}^T(y - z) + \frac{\rho}{2}\|y - z\|_2^2)$

2. $\nabla v\left(w^{(k)}\right) = y^{(k+1)} - z^{(k+1)}$

The optimal step size can be shown to be ρ so the update of w is

$$w^{(k+1)} = w^{(k)} + \rho(y^{(k+1)} - z^{(k+1)})$$

Unfortunately, because of the penalty term, the first step is no longer separable with y and z. Therefore, the joint minimization in step 1 cannot be accomplished exactly in a sequential way. Nonetheless, by performing the updates sequentially (or in alternating directions) and then iterating convergence can be shown to occur (even though at each iteration the exact gradient is not found so the direction of ascent is not the steepest). The ρ parameter can be tuned to optimize the convergence rate. So the algorithm is

$$y^{(k+1)} = \text{argmin}_y \left(g(y) + h\left(z^{(k)}\right) + {w^{(k)}}^T\left(y - z^{(k)}\right) + \frac{\rho}{2}\left\|y - z^{(k)}\right\|_2^2\right)$$

$$= \text{argmin}_y \left(g(y) + {w^{(k)}}^T y + \frac{\rho}{2}\|y\|_2^2 - \rho y^T z^{(k)}\right)$$

$$= \text{argmin}_y \left(g(y) + \frac{\rho}{2}\|y\|_2^2 + \rho y^T\left(\frac{1}{\rho}w^{(k)} - z^{(k)}\right)\right)$$

$$= \text{argmin}_y \left(g(y) + \frac{\rho}{2}\left\|y + (\frac{1}{\rho}w^{(k)} - z^{(k)})\right\|_2^2\right)$$

$$z^{(k+1)} = \text{argmin}_z \left(g\left(y^{(k+1)}\right) + h(z) + {w^{(k)}}^T\left(y^{(k+1)} - z\right) + \frac{\rho}{2}\left\|y^{(k+1)} - z\right\|_2^2\right)$$

$$= \text{argmin}_z \left(h(z) + {u^{(k)}}^T z + \frac{\rho}{2}\|z\|_2^2 - \rho z^T y^{(k+1)}\right)$$

$$= \text{argmin}_z \left(h(z) + \frac{\rho}{2}\|z\|_2^2 + \rho z^T\left(\frac{1}{\rho}w^{(k)} - y^{(k+1)}\right)\right)$$

$$= \text{argmin}_z \left(h(z) + \frac{\rho}{2}\left\|z + (\frac{1}{\rho}w^{(k)} - y^{(k+1)})\right\|_2^2\right)$$

$$w^{(k+1)} = w^{(k)} + \rho(y^{(k+1)} - z^{(k+1)})$$

For simplicity define $u = \frac{1}{\rho}w$ so the complete ADMM algorithm for each iteration is:

1. $y^{(k+1)} = \text{argmin}_y \left(g(y) + \frac{\rho}{2}\left\|y + u^{(k)} - z^{(k)}\right\|_2^2\right)$
2. $z^{(k+1)} = \text{argmin}_z \left(h(z) + \frac{\rho}{2}\left\|z + u^{(k)} - y^{(k+1)})\right\|_2^2\right)$
3. $u^{(k+1)} = u^{(k)} + y^{(k+1)} - z^{(k+1)}$

This ADMM algorithm can be applied to the minSAR problem by first expressing it as an unconstrained minimization:

$$\text{MINIMIZE: COST}\left(\underline{\boldsymbol{B}}\right) + \sum_{s=1}^{N_S} I_{(s)}(\underline{\boldsymbol{b}}_{(s)})$$

Where $I_{(s)}$ is the indicator function for the set $\mathcal{F}_{(s)}$: $I_{(s)}(\underline{\boldsymbol{b}}_{(s)}) = \begin{cases} 0, & \underline{\boldsymbol{b}}_{(s)} \in \mathcal{F}_{(s)} \\ \infty, & \underline{\boldsymbol{b}}_{(s)} \notin \mathcal{F}_{(s)} \end{cases}$.

(This is a standard notational trick to express a constrained minimization as an unconstrained minimization: if $\boldsymbol{b}_{(s)} \notin \mathcal{F}_{(s)}$, the value will be ∞ which is equivalent to a constraint being violated.) Now applying ADMM to minSAR problem is straightforward by defining $g(\underline{\boldsymbol{Y}})$= $\mathrm{COST}(\underline{\boldsymbol{Y}})$ and $h(\underline{\boldsymbol{Z}}) = \sum_{s=1}^{N_S} I_{(s)}(\underline{\boldsymbol{z}}_{(s)})$. Thus the algorithm is:

$$\text{GIVEN: } k = 0;\ \underline{\boldsymbol{Y}}^{(0)}, \underline{\boldsymbol{U}}^{(0)}, \underline{\boldsymbol{Z}}^{(0)}$$

$$\text{REPEAT:}$$

$$\underline{\boldsymbol{Y}}^{(k+1)} = \mathrm{argmin}_{\underline{\boldsymbol{Y}}}(\mathrm{COST}(\underline{\boldsymbol{Y}}) + \frac{\rho}{2}\left\|\underline{\boldsymbol{Y}} - \boldsymbol{Z}^{(k)} + \underline{\boldsymbol{U}}^{(k)}\right\|_2^2))$$

$$\underline{\boldsymbol{Z}}^{(k+1)} = \mathrm{argmin}_{\underline{\boldsymbol{Z}}}(\sum_{s=1}^{N_S} I_{(s)}(\underline{\boldsymbol{z}}_{(s)}) + \frac{\rho}{2}\|\underline{\boldsymbol{Y}}^{(k+1)} - \underline{\boldsymbol{Z}} + \underline{\boldsymbol{U}}^{(k)}\|_2^2$$

$$\underline{\boldsymbol{U}}^{(k+1)} = \underline{\boldsymbol{U}}^{(k)} + \underline{\boldsymbol{Y}}^{(k+1)} - \underline{\boldsymbol{Z}}^{(k+1)}$$

$$\text{UNTIL: } \underline{\boldsymbol{Z}}^{(k+1)} = \underline{\boldsymbol{Y}}^{(k+1)}$$

The z-update can be further decomposed into N_S independent (and parallelizable) operations $\underline{\boldsymbol{z}}_{(s)}^{(k+1)} = \mathrm{argmin}_{\underline{\boldsymbol{z}}_{(s)}}(I_{(s)}(\underline{\boldsymbol{z}}_{(s)}) + \frac{\rho}{2}\left\|\underline{\boldsymbol{y}}_{(s)}^{(k+1)} - \underline{\boldsymbol{z}}_{(s)} + \underline{\boldsymbol{u}}_{(s)}^{(k+1)}\right\|_2^2)$ for $s = 1, \dots, N_S$

This operation minimizes the deviation of the variable $\underline{\boldsymbol{z}}$ from $\underline{\boldsymbol{y}}_{(s)}^{(k+1)} + \underline{\boldsymbol{u}}_{(s)}^{(k+1)}$ while requiring that $\boldsymbol{z}$ be a feasible pulse ($\underline{\boldsymbol{z}} \in \mathcal{F}_{(s)}$). In fact, the ρ parameter becomes irrelevant and this step is equivalent to the Euclidean projection of the point $\underline{\boldsymbol{y}}_{(s)}^{(k+1)} + \underline{\boldsymbol{u}}_{(s)}^{(k+1)}$ onto the set $\boldsymbol{\mathcal{F}}_{(s)}$:

$\underline{\boldsymbol{z}}_{(s)}^{(k+1)} = \mathrm{proj}_{\mathcal{F}_{(s)}}(\underline{\boldsymbol{y}}_{(s)}^{(k+1)} + \underline{\boldsymbol{u}}_{(s)}^{(k+1)})$ for $s = 1, \dots, N_S$

So the complete ADMM algorithm for the minSAR problem is:

INITIALIZE: $k = 0;\ \underline{\boldsymbol{U}}^{(0)}, \underline{\boldsymbol{Z}}^{(0)}$

REPEAT:

$\underline{\boldsymbol{Y}}^{(k+1)} = \mathrm{argmin}_{\underline{\boldsymbol{Y}}}(\mathrm{COST}(\underline{\boldsymbol{Y}}) + \frac{\rho}{2}\|\underline{\boldsymbol{Y}} - \boldsymbol{Z}^{(k)} + \underline{\boldsymbol{U}}^{(k)}\|_2^2$

$(PARALLEL)$ FOR $s = 1, \dots, N_S$

$\quad \underline{\boldsymbol{z}}_{(s)}^{(k+1)} = \mathrm{proj}_{\mathcal{F}_{(s)}}(\underline{\boldsymbol{y}}_{(s)}^{(k+1)} + \underline{\boldsymbol{u}}_{(s)}^{(k+1)})$

END FOR

$\underline{\boldsymbol{U}}^{(k+1)} = \underline{\boldsymbol{U}}^{(k)} + \underline{\boldsymbol{Y}}^{(k+1)} - \underline{\boldsymbol{Z}}^{(k+1)}$

UNTIL: $\underline{\boldsymbol{Z}}^{(k+1)} = \underline{\boldsymbol{Y}}^{(k+1)}$

B. Vectorized SAR oracle

We wish to construct an "oracle" that is queried with a sequence of channel weightings $\underline{\boldsymbol{B}} \in \mathbb{C}^{N_C x N_K N_S}$ and returns the value of $\mathrm{COST}(\underline{\boldsymbol{B}}) \in \mathbb{R}$, where COST has the form $\mathrm{COST}(\boldsymbol{B}) = \max_{r=1,\ldots,N_R}\left(\sum_{i=1}^{N_S N_K} \underline{\boldsymbol{B}}_i^H \boldsymbol{R}_r \underline{\boldsymbol{B}}_i\right)$, and its gradient (or more correctly its subgradient since COST is nondifferentiable) $\underline{\boldsymbol{G}}(\underline{\boldsymbol{B}}) \in \mathbb{C}^{N_C x N_K N_S}$ where $\underline{\boldsymbol{G}}$ satisfies $\mathrm{COST}(\underline{\boldsymbol{B}} + \Delta\underline{\boldsymbol{B}}) \geq \mathrm{COST}(\underline{\boldsymbol{B}}) + \sum_{i=1}^{N_K N_S} \underline{\boldsymbol{G}}_i{}^T \Delta\underline{\boldsymbol{B}}_i$ for any $\Delta\underline{\boldsymbol{B}} \in \mathbb{C}^{N_C x N_K N_S}$ since COST is convex. The oracle makes use of vectorized computation that allows for simple GPU acceleration. While this new technique is especially important for IMPULSE as the oracle must be queried numerous times throughout the course of the optimization, it is fundamentally more efficient than naive methods relying on "for loops", for computing time- and spatially-averaged local SAR or temperature for an entire pulse sequence with arbitrarily many voxels and RF pulses.

The general procedure to compute COST and $\underline{\boldsymbol{G}}$ for a particular $\underline{\boldsymbol{B}}$ (shown in detail in Figure B) is:

1. Generate a vector $\boldsymbol{p} \in \mathbb{R}^{N_R}$ where the $\boldsymbol{r}$th element of $\boldsymbol{p}$ is $\sum_{i=1}^{N_S N_K} \underline{\boldsymbol{B}}_i^H \boldsymbol{R}_r \underline{\boldsymbol{B}}_i$
2. Find the value and index of the largest element of $\boldsymbol{p}$: $[\mathbf{COST},\ \boldsymbol{r}_{max}] = \mathbf{max}(\boldsymbol{p})$
3. Compute a subgradient analytically: $\underline{\boldsymbol{G}} = 2\boldsymbol{R}_{r_{max}} \underline{\boldsymbol{B}}$

The most costly step is the first one and we describe a vectorized method to accomplish it efficiently that is orders of magnitude more efficient than using a "for loop". Not only is the vectorized method fundamentally more efficient in terms of total number of floating point operations needed, but because it expresses the computation as a matrix vector multiplication, there is an additional efficiency gain when using mathematical software that has built-in optimizations for matrix multiplication such as MATLAB. An added benefit is that because of MATLAB's built-in abstraction for using GPUs to multiply matrices (*gpuArray*), GPU acceleration can be achieved with no extra programming effort.

As an overview of the procedure, it is possible to exploit the symmetry of $\boldsymbol{R}_r$ to evaluate $\underline{\boldsymbol{B}}_i^H \boldsymbol{R}_r \underline{\boldsymbol{B}}_i$ in a minimal number of floating point operations as the inner product of two *real* valued vectors $R_{r,vec}$ and $B_{i,vec}$. Then $\sum_{i=1}^{N_S N_K} \underline{\boldsymbol{B}}_i^H \boldsymbol{R}_r \underline{\boldsymbol{B}}_i$ can be computed efficiently as the inner product of $R_{r,vec}$ and $\bar{b} = \sum_{i=1}^{N_S N_K} B_{i,vec}$. The entire vector p can be constructed as a single matrix multiplication $\bar{R}^T \bar{b}$ where $\bar{R}$ is a matrix with the rth column equal to $R_{r,vec}$.

The expressions for $R_{r,vec}$ and $B_{i,vec}$ are found by expanding $\underline{\boldsymbol{B}}_i^H \boldsymbol{R}_r \underline{\boldsymbol{B}}_i$. For simplicity, we will drop the subscripts and use the notation $\boldsymbol{b}^H \boldsymbol{R} \boldsymbol{b}$ in the derivation where single subscripts indicate vector elements and double subscripts indicate matrix elements.

$$\boldsymbol{b}^H \boldsymbol{R} \boldsymbol{b} = \sum_{i=1}^{N_C}\sum_{j=1}^{N_C} \boldsymbol{R}_{ij} \boldsymbol{b}_i^* \boldsymbol{b}_j = \sum_{i=1}^{N_C} \boldsymbol{R}_{ii} |\boldsymbol{b}_i|^2 + \sum_{i=1}^{N_C}\sum_{j=1}^{i} (\boldsymbol{R}_{ij} \boldsymbol{b}_i^* \boldsymbol{b}_j + \boldsymbol{R}_{ij}^* \boldsymbol{b}_i \boldsymbol{b}_j^*) = \sum_{i=1}^{N_C} \boldsymbol{R}_{ii} |\boldsymbol{b}_i|^2 + 2\sum_{i=1}^{N_C}\sum_{j=1}^{i} \mathrm{Re}\{\boldsymbol{R}_{ij} \boldsymbol{b}_i^* \boldsymbol{b}_j\}$$

$$= \sum_{i=1}^{N_C} \boldsymbol{R}_{ii} |\boldsymbol{b}_i|^2 + 2\sum_{i=1}^{N_c}\sum_{j=i+1}^{N_c} (\mathrm{Re}\{\boldsymbol{R}_{ij}\}\mathrm{Re}\{\boldsymbol{b}_i^* \boldsymbol{b}_j\} - \mathrm{Im}\{\boldsymbol{R}_{ij}\}\mathrm{Im}\{\boldsymbol{b}_i^* \boldsymbol{b}_j\})$$

It can be seen that the above expression can be written as the inner product of two real valued vectors $\hat{R}, \hat{b} \in \mathbb{R}^{N_c{}^2}$

$$\hat{R} = \begin{bmatrix} \boldsymbol{R}_{1,1} \\ \vdots \\ \boldsymbol{R}_{N_c,N_c} \\ 2\mathrm{Re}\{\boldsymbol{R}_{12}\} \\ -2\mathrm{Im}\{\boldsymbol{R}_{12}\} \\ \vdots \\ 2\mathrm{Re}\{\boldsymbol{R}_{1,N_c}\} \\ -2\mathrm{Im}\{\boldsymbol{R}_{1,N_c}\} \\ \vdots \\ \vdots \\ 2\mathrm{Re}\{\boldsymbol{R}_{N_c-1,N_c}\} \\ -2\mathrm{Im}\{\boldsymbol{R}_{N_c-1,N_c}\} \end{bmatrix} = \begin{bmatrix} 1 \\ \vdots \\ 1 \\ 2 \\ -2 \\ \vdots \\ 2 \\ -2 \\ \vdots \\ \vdots \\ 2 \\ -2 \end{bmatrix} \circ \begin{bmatrix} \boldsymbol{R}_{1,1} \\ \vdots \\ \boldsymbol{R}_{N_c,N_c} \\ \mathrm{Re}\{\boldsymbol{R}_{12}\} \\ \mathrm{Im}\{\boldsymbol{R}_{12}\} \\ \vdots \\ \mathrm{Re}\{\boldsymbol{R}_{1,N_c}\} \\ \mathrm{Im}\{\boldsymbol{R}_{1,N_c}\} \\ \vdots \\ \vdots \\ \mathrm{Re}\{\boldsymbol{R}_{N_c-1,N_c}\} \\ \mathrm{Im}\{\boldsymbol{R}_{N_c-1,N_c}\} \end{bmatrix} \quad \text{and} \quad \hat{b} = \begin{bmatrix} |\boldsymbol{b}_1|^2 \\ \vdots \\ |\boldsymbol{b}_{N_c}|^2 \\ \mathrm{Re}\{\boldsymbol{b}_1^*\boldsymbol{b}_2\} \\ \mathrm{Im}\{\boldsymbol{b}_1^*\boldsymbol{b}_2\} \\ \vdots \\ \mathrm{Re}\{\boldsymbol{b}_1^*\boldsymbol{b}_{N_c}\} \\ \mathrm{Im}\{\boldsymbol{b}_1^*\boldsymbol{b}_{N_c}\} \\ \vdots \\ \mathrm{Re}\{\boldsymbol{b}_{N_c-1}^*\boldsymbol{b}_{N_c}\} \\ \mathrm{Im}\{\boldsymbol{b}_{N_c-1}^*\boldsymbol{b}_{N_c}\} \end{bmatrix} = \begin{bmatrix} \mathrm{Re}\{\boldsymbol{b}_1\}^2 \\ \vdots \\ \mathrm{Re}\{\boldsymbol{b}_{N_c}\}^2 \\ \mathrm{Re}\{\boldsymbol{b}_1\}\mathrm{Re}\{\boldsymbol{b}_2\} \\ -\mathrm{Im}\{\boldsymbol{b}_1\}\mathrm{Re}\{\boldsymbol{b}_2\} \\ \vdots \\ \mathrm{Re}\{\boldsymbol{b}_1\}\mathrm{Re}\{\boldsymbol{b}_{N_c}\} \\ -\mathrm{Im}\{\boldsymbol{b}_1\}\mathrm{Re}\{\boldsymbol{b}_{N_c}\} \\ \vdots \\ \mathrm{Re}\{\boldsymbol{b}_{N_c-1}\}\mathrm{Re}\{\boldsymbol{b}_{N_c}\} \\ -\mathrm{Im}\{\boldsymbol{b}_{N_c-1}\}\mathrm{Re}\{\boldsymbol{b}_{N_c}\} \end{bmatrix} +$$

$$\begin{bmatrix} \mathrm{Im}\{\boldsymbol{b}_1\}^2 \\ \vdots \\ \mathrm{Im}\{\boldsymbol{b}_{N_c}\}^2 \\ \mathrm{Im}\{\boldsymbol{b}_1\}\mathrm{Im}\{\boldsymbol{b}_2\} \\ \mathrm{Re}\{\boldsymbol{b}_1\}\mathrm{Im}\{\boldsymbol{b}_2\} \\ \vdots \\ \mathrm{Im}\{\boldsymbol{b}_1\}\mathrm{Im}\{\boldsymbol{b}_{N_c}\} \\ \mathrm{Re}\{\boldsymbol{b}_1\}\mathrm{Im}\{\boldsymbol{b}_{N_c}\} \\ \vdots \\ \mathrm{Im}\{\boldsymbol{b}_{N_c-1}\}\mathrm{Im}\{\boldsymbol{b}_{N_c}\} \\ \mathrm{Re}\{\boldsymbol{b}_{N_c-1}\}\mathrm{Im}\{\boldsymbol{b}_{N_c}\} \end{bmatrix} = \begin{bmatrix} 1 \\ \vdots \\ 1 \\ 1 \\ -1 \\ \vdots \\ 1 \\ -1 \\ \vdots \\ \vdots \\ 1 \\ -1 \end{bmatrix} \circ \begin{bmatrix} \mathrm{Re}\{\boldsymbol{b}_1\} \\ \vdots \\ \mathrm{Re}\{\boldsymbol{b}_{N_c}\} \\ \mathrm{Re}\{\boldsymbol{b}_1\} \\ \mathrm{Im}\{\boldsymbol{b}_1\} \\ \vdots \\ \mathrm{Re}\{\boldsymbol{b}_1\} \\ \mathrm{Im}\{\boldsymbol{b}_1\} \\ \vdots \\ \vdots \\ \mathrm{Re}\{\boldsymbol{b}_{N_c-1}\} \\ \mathrm{Im}\{\boldsymbol{b}_{N_c-1}\} \end{bmatrix} \circ \begin{bmatrix} \mathrm{Re}\{\boldsymbol{b}_1\} \\ \vdots \\ \mathrm{Re}\{\boldsymbol{b}_{N_c}\} \\ \mathrm{Re}\{\boldsymbol{b}_2\} \\ -\mathrm{Re}\{\boldsymbol{b}_2\} \\ \vdots \\ \mathrm{Re}\{\boldsymbol{b}_{N_c}\} \\ -\mathrm{Re}\{\boldsymbol{b}_{N_c}\} \\ \vdots \\ \vdots \\ \mathrm{Re}\{\boldsymbol{b}_{N_c}\} \\ -\mathrm{Re}\{\boldsymbol{b}_{N_c}\} \end{bmatrix} + \begin{bmatrix} \mathrm{Im}\{\boldsymbol{b}_1\} \\ \vdots \\ \mathrm{Im}\{\boldsymbol{b}_{N_c}\} \\ \mathrm{Im}\{\boldsymbol{b}_1\} \\ \mathrm{Re}\{\boldsymbol{b}_1\} \\ \vdots \\ \mathrm{Im}\{\boldsymbol{b}_1\} \\ \mathrm{Re}\{\boldsymbol{b}_1\} \\ \vdots \\ \vdots \\ \mathrm{Im}\{\boldsymbol{b}_{N_c-1}\} \\ \mathrm{Re}\{\boldsymbol{b}_{N_c-1}\} \end{bmatrix} \circ \begin{bmatrix} \mathrm{Im}\{\boldsymbol{b}_1\} \\ \vdots \\ \mathrm{Im}\{\boldsymbol{b}_{N_c}\} \\ \mathrm{Im}\{\boldsymbol{b}_2\} \\ \mathrm{Im}\{\boldsymbol{b}_2\} \\ \vdots \\ \mathrm{Im}\{\boldsymbol{b}_{N_c}\} \\ \mathrm{Im}\{\boldsymbol{b}_{N_c}\} \\ \vdots \\ \vdots \\ \mathrm{Im}\{\boldsymbol{b}_{N_c}\} \\ \mathrm{Im}\{\boldsymbol{b}_{N_c}\} \end{bmatrix}$$

In this way it is seen that $\boldsymbol{b}^H\boldsymbol{R}\boldsymbol{b} = \hat{R}^T\hat{b}$ where $\hat{R} = \mathrm{sgn}_R \circ \boldsymbol{R}[\mathrm{idx}_R]$ and $\hat{b} = \mathrm{sgn}_b \circ b_{[1]} \circ b_{[2]} + b_{[3]} \circ b_{[4]} = \mathrm{sgn}_b \circ \boldsymbol{b}[\mathrm{idx}_{b,1}] \circ \boldsymbol{b}[\mathrm{idx}_{b,2}] + \boldsymbol{b}[\mathrm{idx}_{b,3}] \circ \boldsymbol{b}[\mathrm{idx}_{b,4}]$; $\circ$ is the elementwise product; sgn_R is a ${N_C}^2$ element vector consisting of 1, -2 and 2; sgn_b is a ${N_C}^2$ element vector consisting of 1, and -1; $\boldsymbol{R}_{[]} = \boldsymbol{R}[\mathrm{idx}_R]$, $b_{[1]} = \boldsymbol{b}[\mathrm{idx}_{b,1}]$, $b_{[2]} = \boldsymbol{b}[\mathrm{idx}_{[2]}]$, $b_{[3]} = \boldsymbol{b}[\mathrm{idx}_{[3]}]$, and $b_{[4]} = \boldsymbol{b}[\mathrm{idx}_{[4]}]$ are ${N_C}^2$ element real-valued vectors formed by accessing the real or imaginary components of $\boldsymbol{R}$ or $\boldsymbol{b}$ at the locations specified by idx_R, $\mathrm{idx}_{b,1}$, $\mathrm{idx}_{b,2}$, $\mathrm{idx}_{b,3}$, and $\mathrm{idx}_{b,4}$.

Returning to the original notation it can be seen that $\hat{R}$ corresponds to $R_{r,vec}$ and $\hat{b}$ corresponds to $B_{i,vec}$. Therefore, it can be seen that the matrix $\bar{R} \in \mathbb{R}^{{N_C}^2 \mathrm{x} N_R}$ can be constructed with each column computed as above for a particular $\boldsymbol{R}_r$. Each $B_{i,vec}$ is also computed as above for a particular $\underline{\boldsymbol{B}}_i$. It is possible to precompute $\bar{R}$, sgn, $\mathrm{idx}_{[1]}$, $\mathrm{idx}_{[2]}$, $\mathrm{idx}_{[3]}$, and $\mathrm{idx}_{[4]}$ offline. Evaluating p for a particular $\underline{\boldsymbol{B}}$ (step 1 in Supporting Information Figure S1), requires the following subcomputations:

A. Generate $B_{[1]}, B_{[2]}, B_{[3]}, B_{[4]} \in \mathbb{R}^{{N_C}^2 \mathrm{x} N_S N_K}$ from $\underline{\boldsymbol{B}} \in \mathbb{C}^{N_C \mathrm{x} N_S N_K}$.

   This involves constructing each column of $B_{[1]}, B_{[2]}, B_{[3]}, B_{[4]}$ from each column of $\underline{\boldsymbol{B}}$ by the procedure described above through reindexing at precomputed indices. This procedure requires no floating point computations and negligible time.

B. Compute $\bar{B} = \mathrm{SGN} \circ B_{[1]} \circ B_{[2]} + B_{[3]} \circ B_{[4]}$.

This requires three elementwise matrix products and one matrix addition. The matrix product with SGN (a matrix with $N_S N_K$ columns each of which is equal to sgn) takes negligible time since it is not a floating point computation – SGN consists of only -1 and 1. Thus this step has a total of 3 floating point matrix computations (2 multiplications, 1 addition). The number of floating point operations (FLOPs) for each is equal to the number of elements in the matrix or ${N_C}^2 N_S N_K$ and thus the total number of FLOPs is $3{N_C}^2 N_S N_K$. However, because each of the ${N_C}^2 N_S N_K$ FLOPs can be completely parallelized across elements of the matrix with appropriate multithreading, the number of matrix computations (=3) is a better measure of the complexity of this step than the total number of FLOPs.

C. Compute $\bar{b} = \bar{B}\mathbf{1}$.

This requires 1 matrix operation (summing across columns of $\bar{B}$) and a total of ${N_C}^2 N_S N_K$ FLOPs. With appropriate multithreading, the computational complexity is $O(\log(N_S N_K))$ (since multithreaded summing of a list of numbers has computation complexity $O(\log(length\ of\ list))$ and complete parallelization of computation across rows of $\bar{B}$ is assumed).

D. Compute matrix multiplication $p = \bar{R}^T \bar{b}$.

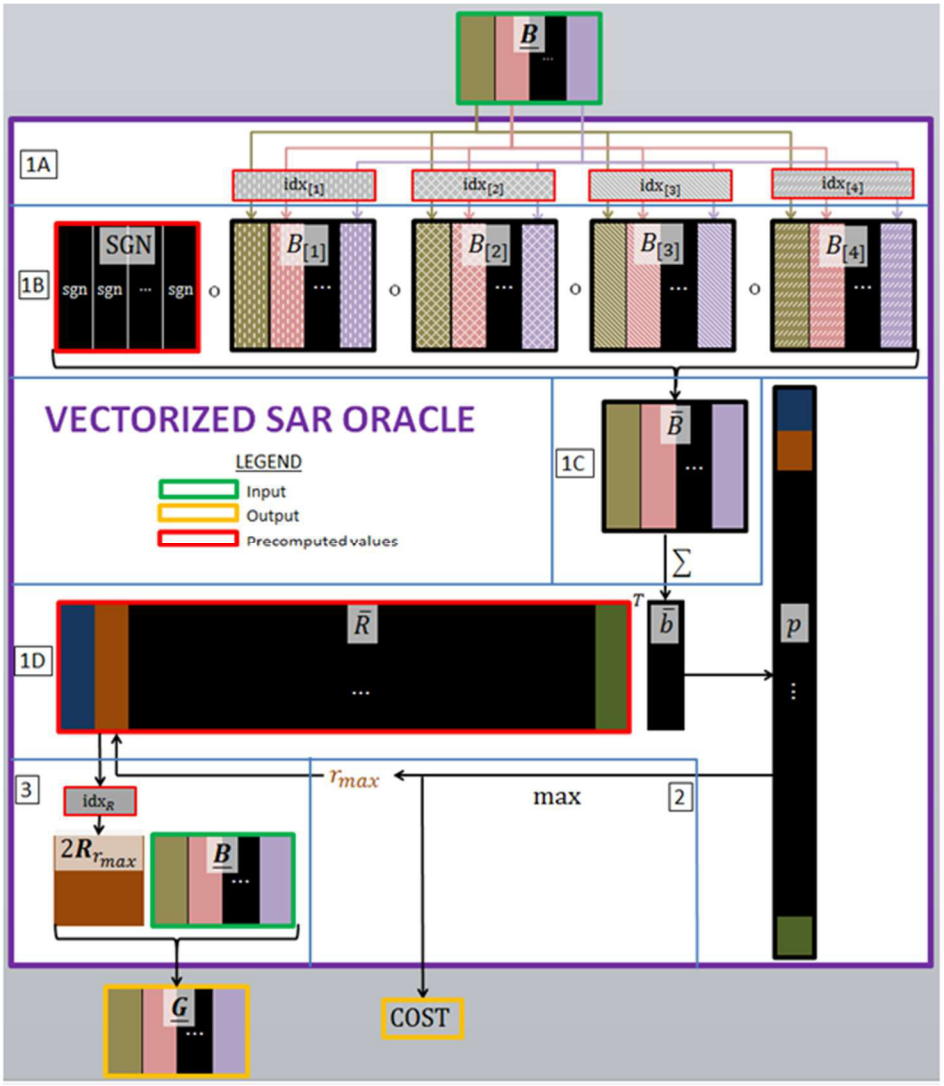
This requires 1 matrix multiplication and a total $N_R {N_C}^4$ total FLOPs. The complexity of matrix multiplication with multithreading is difficult to describe but with the most efficient GPU implementations, duration of matrix multiplication can be made to be approximately constant until the size of the matrices exceeds the memory capacity of the GPU

Once the vector p has been computed, step 2 involves finding the index (r_{max}) and value (COST) of the largest element of p. Step 3 requires selecting the r_{max}th column of $\bar{R}$ and reconstructing the matrix $\boldsymbol{R}_{r_{max}}$ through a reverse vectorization operation (which takes negligible time since it only involves reindexing at the predetermined locations, idx_R). Then the subgradient matrix can be found as $\underline{\boldsymbol{G}} = 2\boldsymbol{R}_{r_{max}}\underline{\boldsymbol{B}}$. Both steps 2 and 3 take negligible time compared to step 1. Supporting Information Figure S1 encapsulates this entire procedure graphically.

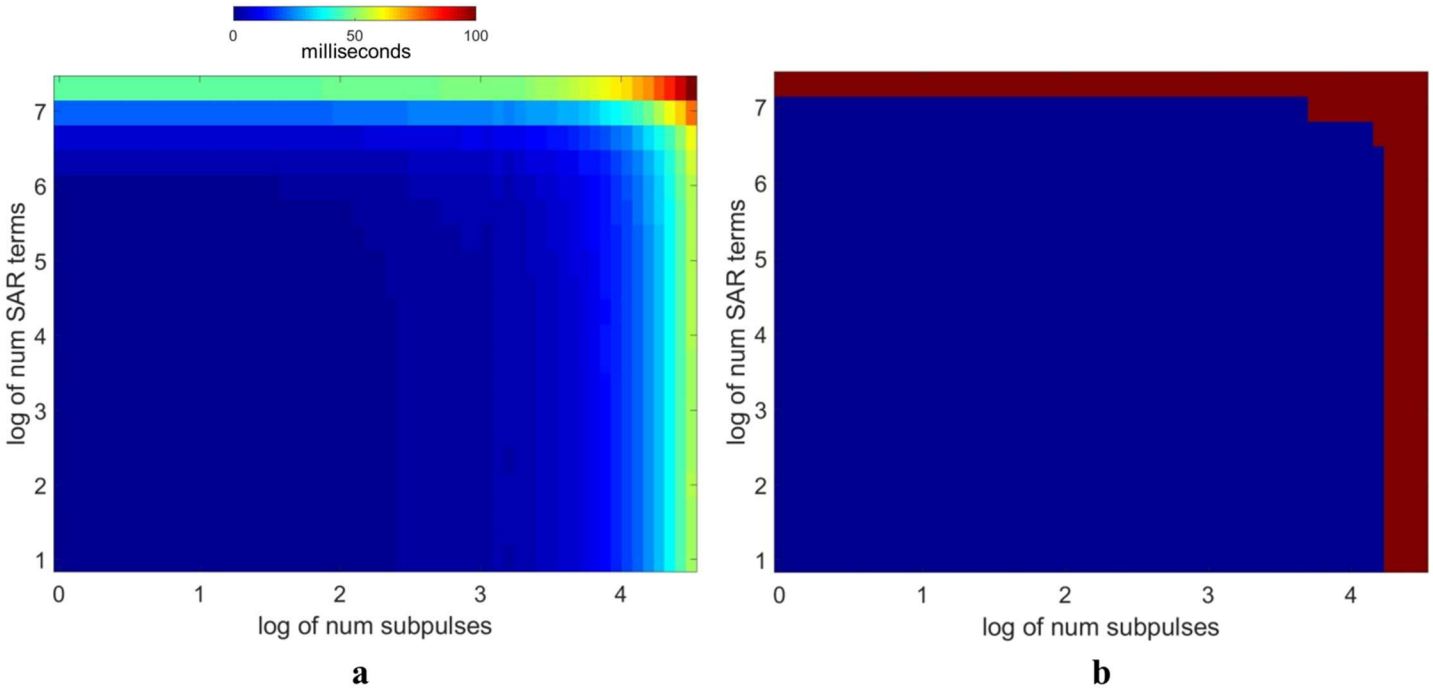
Based on the analysis above, assuming a very large number of threads (ie with a GPU) the computation time for the oracle is not dependent on the number of floating point operations and takes constant time. The requirement for this is that $\bar{R}$, SGN, $\text{idx}_{[1]}$, $\text{idx}_{[2]}$, $\text{idx}_{[3]}$, $\text{idx}_{[4]}$, and idx_R have been stored onto the GPU prior to the start of optimization. In real time, each input $\underline{\boldsymbol{B}}$ is transferred to the GPU, steps 1 through 3 are performed on the GPU, then $\underline{\boldsymbol{G}}$ and COST are transferred back to the CPU. Alternatively, if a GPU is not available, the computations can all be performed on the CPU and significant speedup compared to use of a "for" loop will still be observed. However, memory transfer from CPU to GPU can take significant amount of time and thus it is desirable to perform as much computation on the CPU (at the expense of incomplete parallelization) in order to reduce the amount of data that must be transferred to the GPU in real-time. It is necessary to devise a strategy that manages this tradeoff by deciding when to transfer data and begin performing computations on the GPU (the input $\underline{B}$ is found on the CPU as an output of the previous ADMM iteration). One option is to transfer $\underline{B}$ to the GPU and perform Steps 1 through 4 on the GPU; another option is to perform steps 1 through 3 on the CPU and then transfer b_{full} to the GPU for step 4 (in both cases R_{full} is assumed to already be stored on the GPU). The advantage of the first option is that steps 2 and 3 are better parallelized on a GPU and will therefore take less time than in the second option. However, the advantage of the second option is that transferring the entire large matrix $\underline{B}$ is undesirable

and time-consuming compared to transferring the much smaller vector b_{full}. Additionally, GPUs have the disadvantage that the reindexing operation in step 1 which takes negligible time on a CPU actually takes a significant amount of time on a GPU. The performance of each option will vary depending on the exact specifications of the processors used for computation.

In summary, by completely vectorizing the computation of SAR and its subgradient it is possible to exploit built-in optimizations from matrix multiplications in most numerical computing packages. Describing the complexity of the vectorized computation exactly is difficult because of the intricacies of the caching and single input multiple data (SIMD) optimizations made by the numerical computing package but empirically favorable scaling is observed with both number of SAR terms and number of pulses. Supporting Information Figure S2 shows this scaling behavior using a NVIDIA GeForce GTX 1080 Ti GPU.

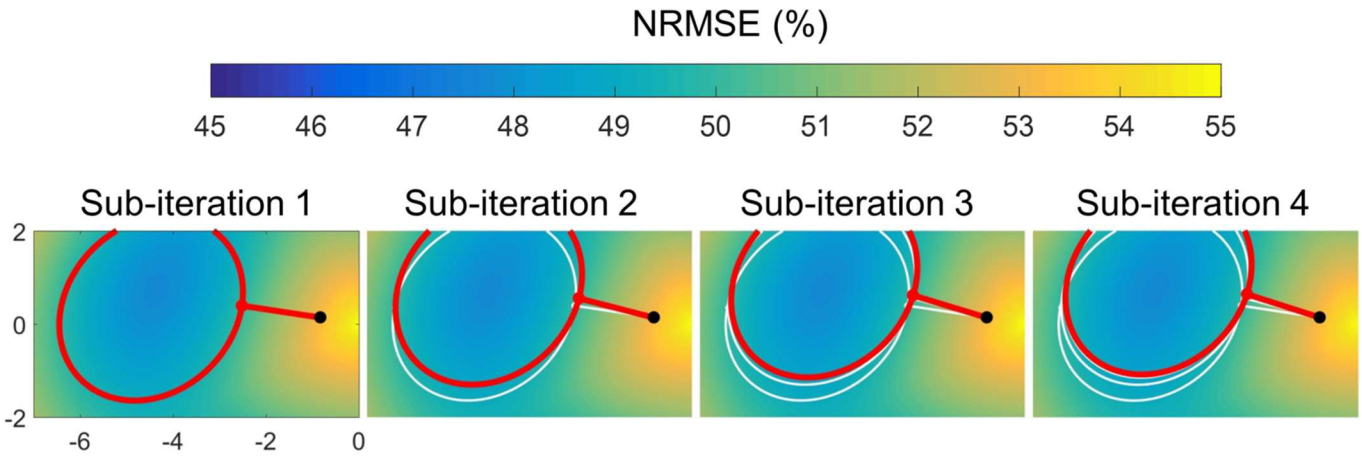


Supporting Information Figure S1: Diagram of operations in vectorized SAR oracle explained in Supporting Document B.



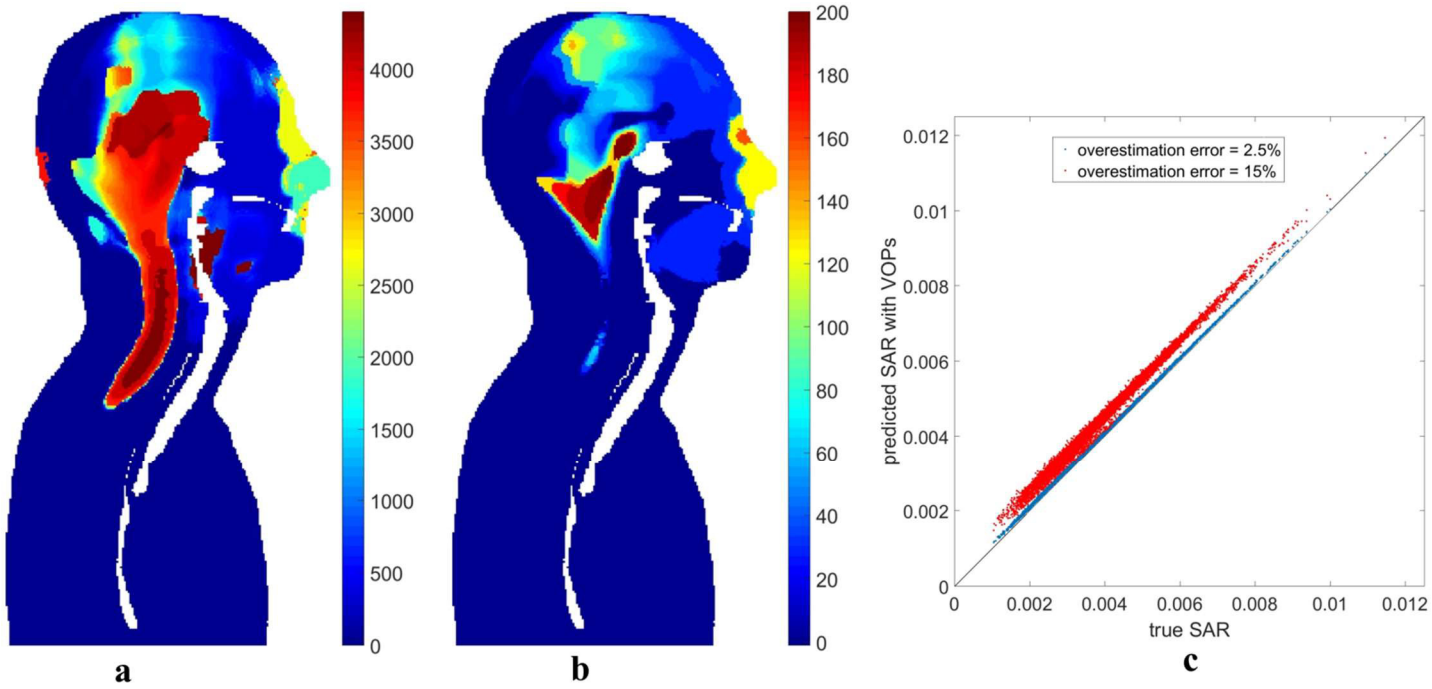
Supporting Information Figure S2: (a) Computation time using Nvidia GeForce GTX 1080 Ti for evaluation of the objective function and its gradient for different number of SAR terms and subpulses using the described vectorized oracle. The total number of SAR terms is equal to the total number of voxels in the SAR model plus the number of global SAR and per channel power terms. The total number of subpulses is equal to the total number of spokes per pulse multiplied by the total number of pulses (slices). (b) The range of pulses that can be designed with IMPULSE in approximately constant time is shown in blue based on <30 ms evaluation time in (a) since the duration of the SAR update will likely be significantly less than that of that of the FAI update when evaluation time is less than 30 ms. Accordingly, a pulse design including up to 10^3 to 10^4 subpulses and up to 10^6 to 10^7SAR terms should be possible with IMPULSE. This corresponds, for example, to over 500 slices with 2 spokes per slice and a single high resolution (< 2 mm^3 voxel size) head model or a cluster of 25-50 lower resolution (> 3 mm^3 voxel size) models.

C. FAI Update

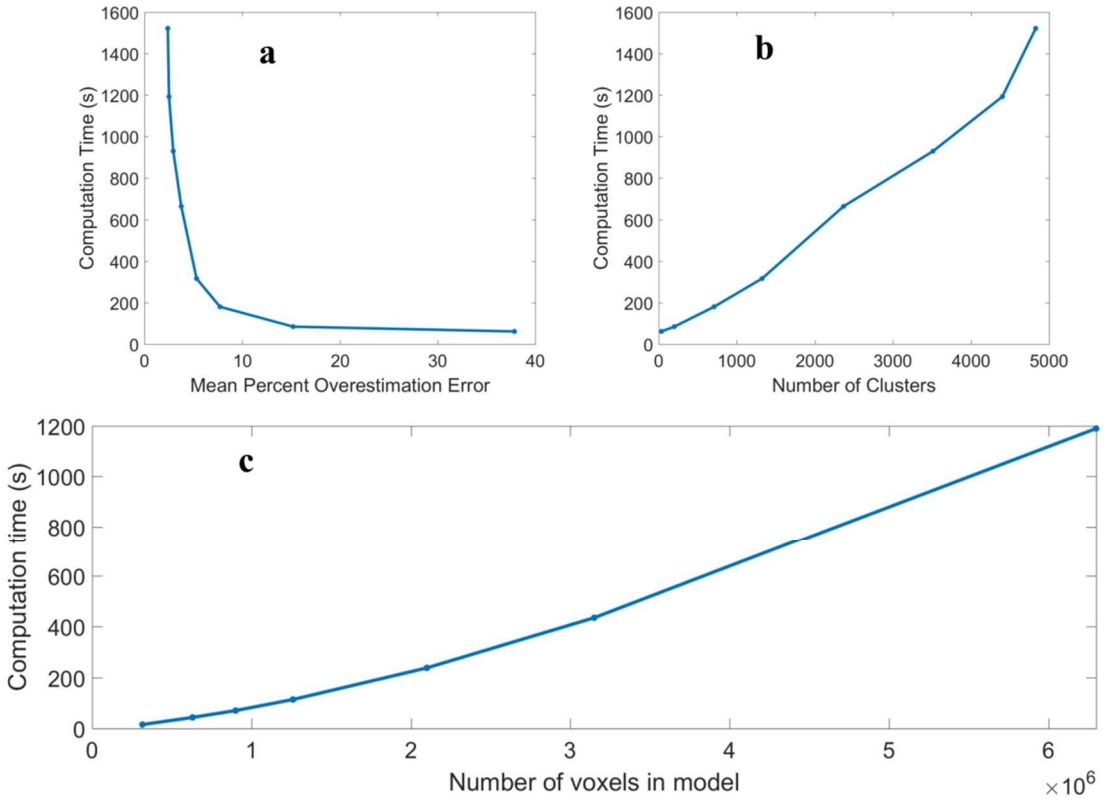


Supporting Information Figure S3: Visualization of the sub-iterations of FAI update for NRMSE tolerance of 50% for 2 channels. The background image is the NRMSE for each 2-channel weight. For a fixed set of spokes locations and target phases, the "feasible" weights that achieve the NRMSE tolerance form an ellipse (ellipse for current sub-iteration shown in red). The output of the SAR-update plus the residual ($\underline{\boldsymbol{y}} + \underline{\boldsymbol{u}}$) which violates the NRMSE tolerance is shown as a black dot. The goal is to project onto the feasible set (ie. find a feasible point with minimum distance from the black dot). At each sub-iteration the black dot is projected onto ellipse defined by the current spokes locations and target phase. Then the spokes locations and target phase are updated to minimize the NRMSE with a fixed value of the channel weights leading to a new ellipse in the next sub-iteration (new ellipse shown in red, prior ellipses shown in white). The projection onto the new ellipse will have a smaller distance to the black dot than the prior ellipse. Applying this interleaved procedure of projecting and updating spokes locations and target phases gives a fast approximate solution to the FAI update despite it being nonconvex. This procedure is performed in parallel for all slices.

D. Virtual Observation Points



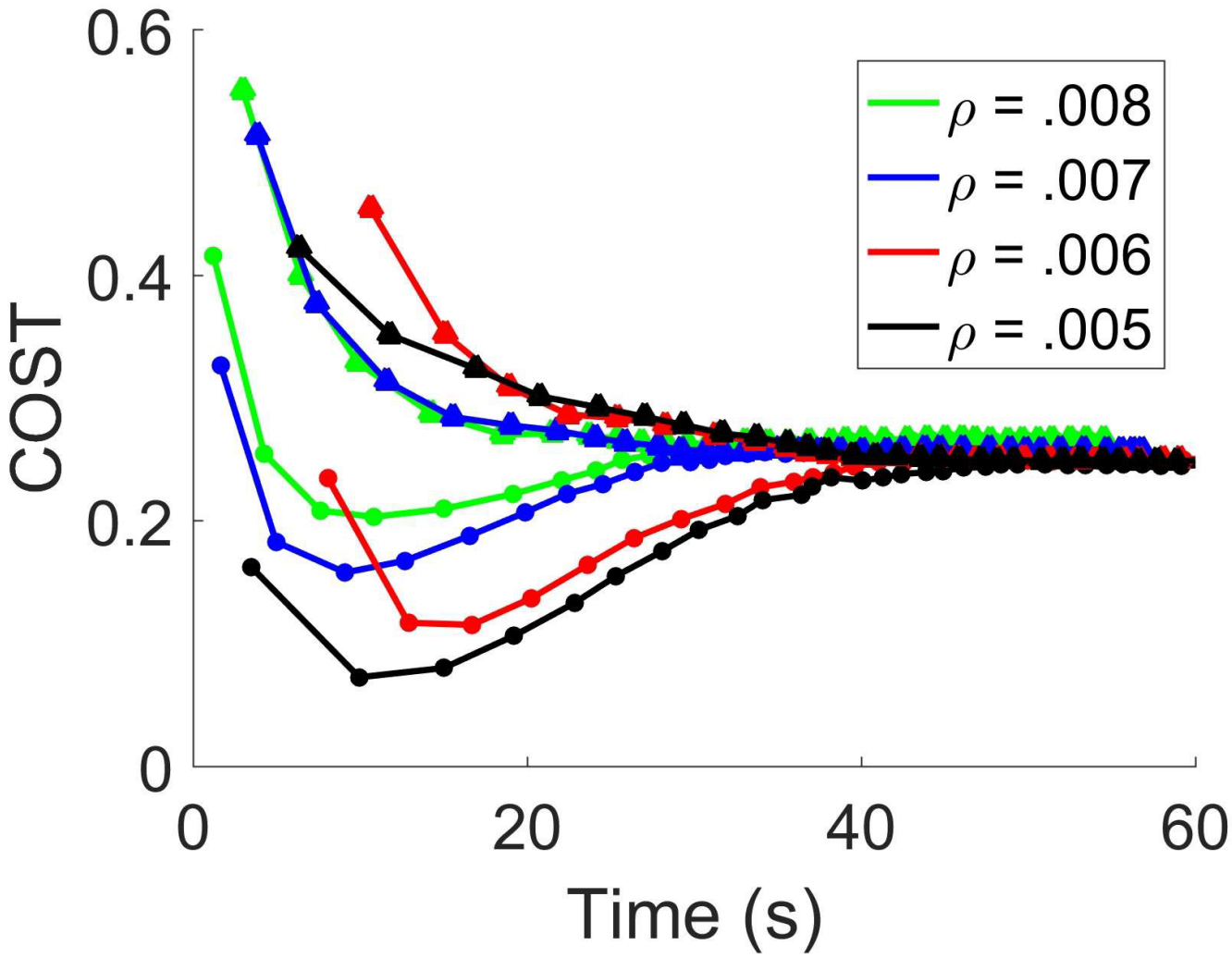
Supporting Information Figure S4 **(a)** 4401 clusters in the VOP model with 2.5% overestimation error **(b)** 200 clusters in the VOP model with 15% overestimation error **(c)** plot of true SAR vs predicted SAR with VOPs for the two VOP models for 10000 random shim weights. Original SAR model had 6371811 voxels.



Supporting Information Figure S5: **(a)** Plot of computation time for the VOP compression with 6371811 voxels for different values of overestimation error computed over 10000 random shim weights **(b)** Corresponding plot of computation time vs number of clusters for each value of the overestimation error **(c)** Plot of computation time for different numbers of starting voxels in the model with 2.5% overestimation error. Computation time increases superlinearly with number of voxels. The compression was run using C++/Eigen/OpenMP with 48 threads. The computation time significantly exceeds duration of IMPULSE for overestimation errors less than about 15% and number of voxels more than about 1 million.

E. Tuning the IMPULSE Algorithm

The choice of the ρ hyperparameter for controlling the regularization in the SAR update will affect the rate of convergence and final value of SAR found after optimization; thus it must be chosen in a way that achieves a SAR value as close as possible to the true optimum SAR while still converging in a time frame that is suitable for on-scanner design. IMPULSE for 120 slices and 6371811 voxel Duke model was run with a range of ρ values. We chose the value of ρ for which the SAR was minimum after 45s of running IMPULSE; the duration of 45s was chosen because it is long enough to achieve convergence for a wide range of ρ while still being significantly shorter than the computation time needed for a generic optimization algorithms and being practical for on-scanner pulse design. To verify that IMPULSE solves the minSAR problem accurately, an identical problem was solved using a generic solver in MATLAB (*fmincon* using the sequential quadratic programming (SQP) algorithm taken as the ground truth) and the results were compared. As seen in Supporting Information Figure S6, for values of ρ between 0.005 and 0.008 convergence is achieved within 40-60 seconds. A value of $\rho = 0.006$ was chosen as having lowest SAR after 45 seconds. This minimum SAR value is within 4% of the value of SAR after full convergence of IMPULSE, and within 7% of the optimum SAR found using the SQP algorithm run to convergence.



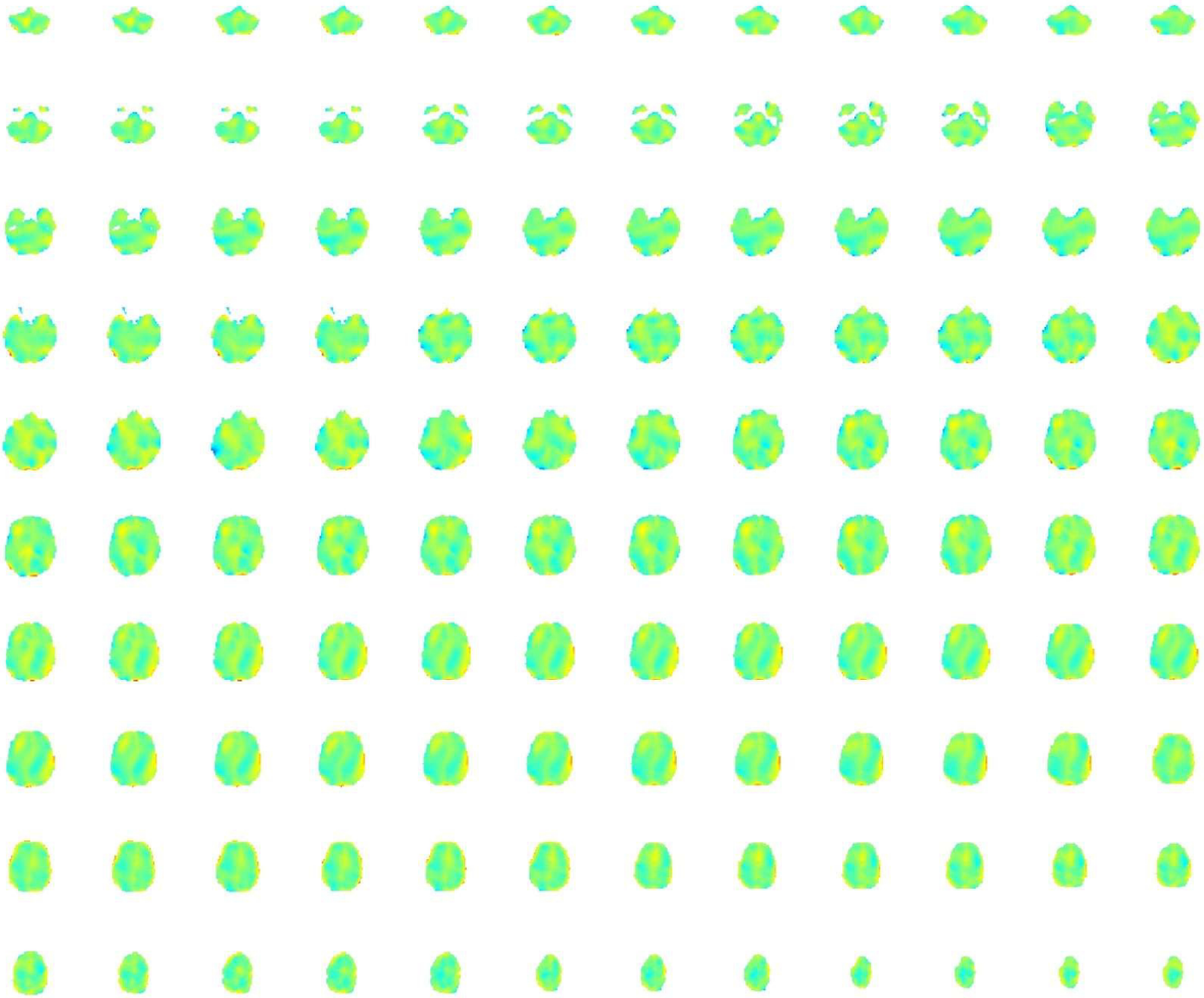
Supporting Information Figure S6: Effect of choice of ρ parameter of IMPULSE algorithm. Triangles indicate output of FAI update and dots indicate output of the SAR update. Value of $\rho = 0.006$ seems to achieve good convergence in under 45 seconds Larger values of ρ converge faster but to a suboptimal value while smaller values take longer to converge.

The effect of the ρ parameter is further understood through the animations in movie files rho_small.avi, rho_best.avi, rho_large.avi. With a value of ρ that is too large (rho_large.avi), the residual $\underline{\boldsymbol{y}} - \underline{\boldsymbol{z}}$ (white vector) converges quickly but $\underline{\boldsymbol{z}}$ (red dot) takes longer to converge (the optimum is reached in about 8 iterations). With a value of ρ that is too small (rho_small.avi), $\underline{\boldsymbol{z}}$ initially converges quickly but the residual takes longer to converge (the optimum is reached in about 10 iterations). With the optimal value of ρ (rho_best.avi), $\underline{\boldsymbol{z}}$ and the residual converge at approximately the same rate (the optimum is reached in about 5 iterations).

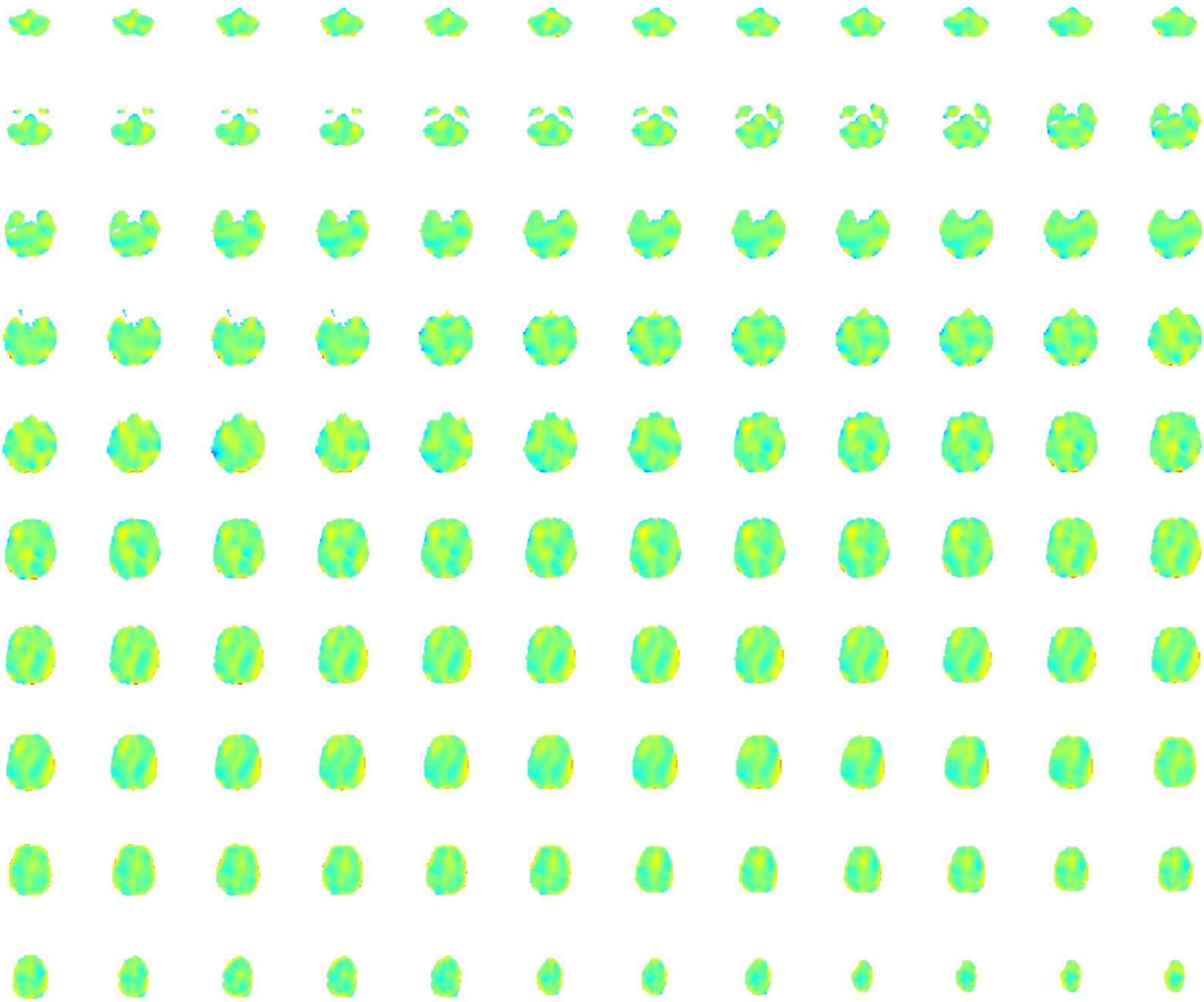
F. Flip Angle Maps

Supporting Information Figure S7: Flip angle maps of all 120 slices for the three algorithms in Figure 7 (same colorscale).

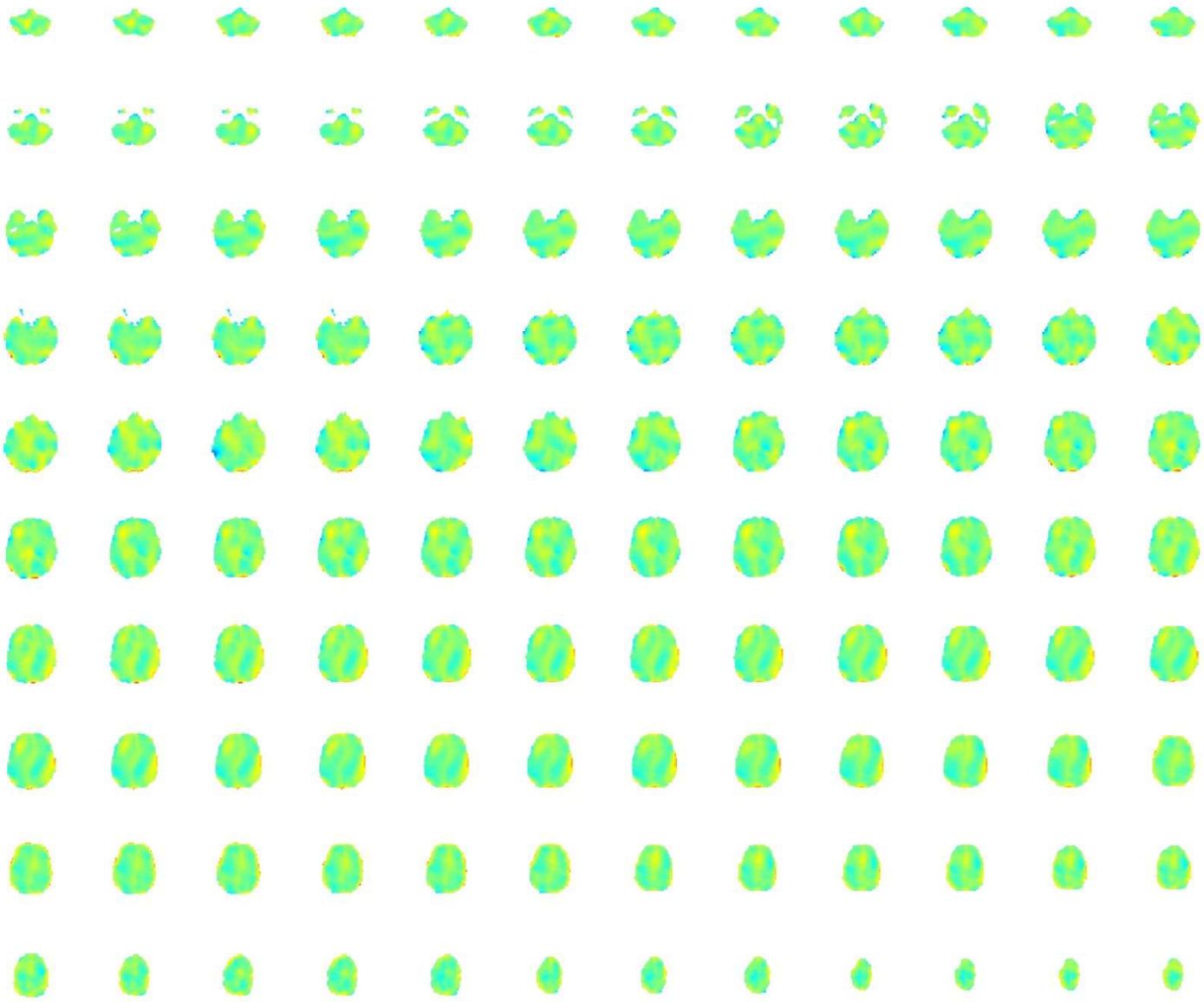
(a) Convex



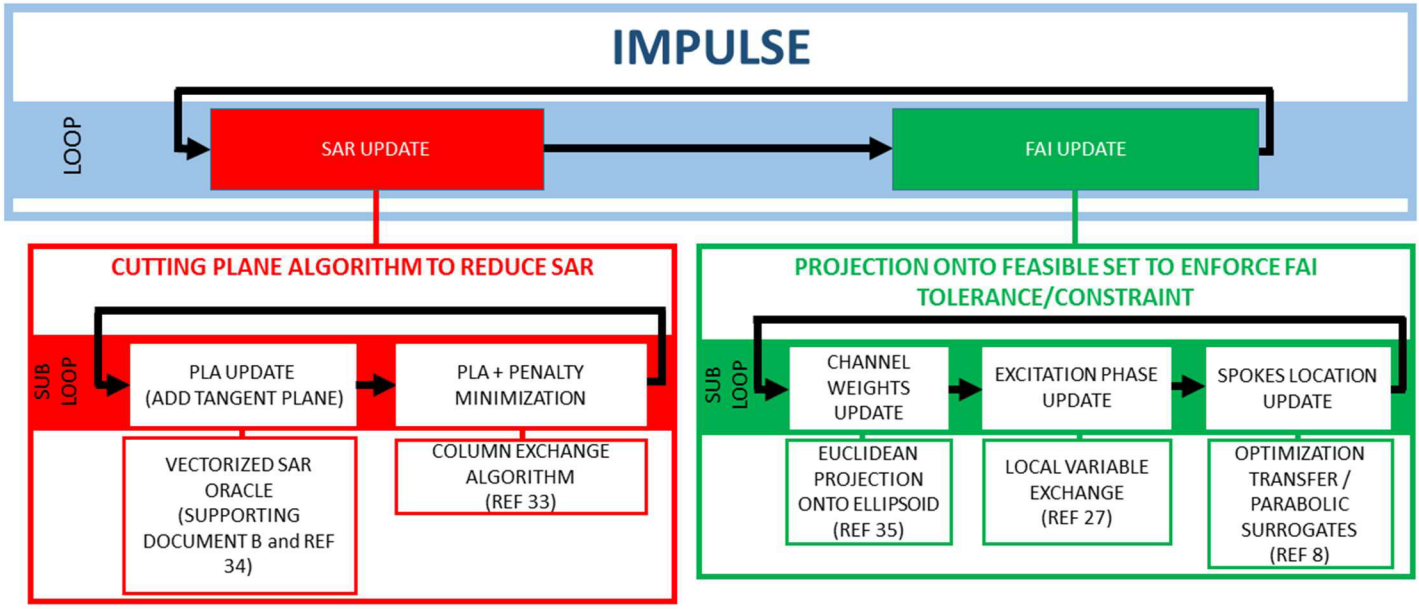
(b) SQP



(c) IMPULSE



G. High-Level Overview



Supporting Information Figure S8: High-level overview of IMPULSE algorithm decomposed into the individual sub-algorithms and associated references for details on implementation.