Algorithm 1 Simulated annealing beam orientation optimization

Input: $iters$, s_f, s_a^{max}, s_a^{min}, η, T, θ_{max}, θ_{min}, n_b, W, d, $beams[1...n_b]$.
struct
{
    θ; % scalar
    ϕ; % scalar
    $fluenceMap$; % 2D array
    $dose$; % 3D array
} $beam$;

$doseLoss$; % 1D array of size $iters$
$smoothnessLoss$; % 1D array of size $iters * n_b$
$perturbLoss$; % 1D array of size $iters * n_b * 2$

% initial dose calculation
for $b = 1, 2, \ldots, n_b$ **do**
    $beams[b].fluenceMapToDose()$;
end for

for $i = 1, 2, \ldots, iters$ **do**
    FLUENCE_MAP_UPDATE($beams, n_b, W, d, \eta, s_f, doseLoss,$
$smoothnessLoss$)

    % update T
    if $i > 1$ **then**
        $absDiff = \sum_{b=1}^{n_b} abs(perturbLoss[2 * ((i-2) * n_b + b)] -$
            $perturbLoss[2 * ((i-2) * n_b + b) - 1]);$
        $T = absDiff * \log(2)/n_b;$
    end if

    % update s_a using linear decay
    $s_a = (s_a^{max} * (iters - i) + s_a^{min} * i)/iters;$

    % do perturbation
    PERTURBATION($beams, n_b, W, d, T, s_a, \theta_{max}, \theta_{min}$)
end for

Algorithm 2 Fluence map update

procedure FLUENCE_MAP_UPDATE($beams, n_b, W, d, \eta, s_f, doseLoss, smoothnessLoss$)

  % $beam[b].dose$ is up-to-date for $b = 1, 2, ...n_b$

  % calculate $totalDose$, $doseLoss$, and $doseGrad$

  $totalDose = \sum_{b=1}^{n_b} beams[b].dose;$

  $doseLoss[i], doseGrad = calcDoseGrad(totalDose, W, d);$

  for $b = 1, 2, \ldots, n_b$ **do**

    $fluenceGrad = beams[b].calcFluenceGrad(doseGrad);$

    $smoothnessLoss[(i-1) * n_b + b], smoothnessGrad =$

      $beams[b].calcSmoothnessGrad();$

    $totalGrad = normalize(fluenceGrad + \eta * smoothnessGrad);$

    $beams[b].fluenceMap = beams[b].fluenceMap - s_f * totalGrad;$

    $beams[b].fluenceMapToDose();$ % update $beams[b].dose$

  end for

end procedure

Algorithm 3 Beam angle perturbation

procedure PERTURBATION($beams, n_b, W, d, T, s_a,\ \theta_{max},\ \theta_{min}$)
  for $b = 1, 2, \ldots, n_b$ **do**
    $totalDose = \sum_{bb=1}^{n_b} beams[bb].dose$;
    $doseLoss_0, \_ = calcDoseGrad(totalDose, W, d)$;
    $\theta_0 = beams[b].\theta$;
    $\phi_0 = beams[b].\phi$;

    % update beam angles;
    $beams[b].\theta = beams[b].\theta + random01() * s_a$;
    $beams[b].\phi = beams[b].\phi + random01() * s_a$;
    $beams[b].\theta = clamp(beams[b].\theta, \theta_{max}, \theta_{min})$; % to avoid collision

    $beams[b].fluenceMapToDose()$; % dose update
    $totalDose = \sum_{bb=1}^{n_b} beams[bb].dose$;
    $doseLoss_1, \_ = calcDoseGrad(totalDose, W, d)$;

    if $doseLoss_1 < doseLoss_0$ **then**
      $probability = 1$;
    else
      $probability = \exp((doseLoss_0 - doseLoss_1)/T)$;
    end if

    if $random01() > probability$ **then**
      % do not take the new angles, restore old angles and dose
      $beams[b].\theta = \theta_0$;
      $beams[b].\phi = \phi_0$;
      $beams[b].fluenceMapToDose()$;
    end if
    % else, take the new angles, do nothing
  end for
end procedure