Technical Explanation of a V2V Narrowband Channel Ray-Tracing Simulation

Cédric Sipakam

July 13, 2025

1 Introduction and Global Physical Model

This document provides an exhaustive analysis of a MATLAB V2V channel simulation. Its core principle is to model the communication channel as a sum of discrete rays (Multipath Components), including a direct Line-of-Sight (LOS) path and multiple reflected paths from environmental objects (walls). This deterministic approach, known as ray-tracing, offers a site-specific and physically grounded alternative to more abstract statistical channel models.

1.1 Channel Transfer Function and Power

The total narrowband channel is the coherent sum of the complex gains (α_n) of all N valid paths found between the transmitter (TX) and receiver (RX):

$$H_{NB} = \sum_{n=1}^{N} \alpha_n \tag{1}$$

This coherent sum accounts for the phase of each ray, leading to constructive and destructive interference, which manifests as small-scale fading.

Instantaneous received power, which includes this fading effect, is given by:

$$P_{RX} = P_{TX}|H_{NB}|^2 = P_{TX}\left|\sum_{n=1}^{N} \alpha_n\right|^2 \tag{2}$$

Local average power, which smooths out the rapid fading by summing the powers of the individual rays incoherently, is given by:

$$\langle P_{RX} \rangle = P_{TX} \sum_{n=1}^{N} |\alpha_n|^2 \tag{3}$$

1.2 Complex Gain of a Single Path (α_n)

The complex gain of an individual ray is the central physical calculation, combining path loss, phase shift, and reflection losses. It is calculated in the function `calculateAlpha_n.m`.

$$\alpha_n = \underbrace{j\frac{\lambda Z_0}{4\pi^2 R_a d_n}}_{\text{Amplitude}} \cdot \underbrace{\left(\prod_{k=1}^{K_n} \Gamma_k\right)}_{\text{Reflection Loss}} \cdot \underbrace{e^{-j\beta d_n}}_{\text{Phase Shift}} \tag{4}$$

where d_n is the total path length, $\beta = 2\pi/\lambda$ is the wavenumber, Γ_k is the Fresnel reflection coefficient for the k-th bounce out of K_n total bounces for that path, and other terms are physical constants. The leading j term introduces a constant 90-degree phase shift, which is a convention tied to the specific antenna and channel model being used.

2 Algorithmic Deep Dive: Function by Function

We will analyze the functions in a logical, top-down order that follows the simulation's call stack, starting from the main execution script and descending into the core geometric and physics engines.

2.1 `main.m`: The Orchestrator

Purpose: As the main script, `main.m` is the top-level entry point for the entire simulation. It is responsible for defining all simulation parameters, managing the sequence of analyses (e.g., single point, vs. distance, 2D heatmap), calling the core ray-tracing engine, and passing the results to visualization functions.

Execution Flow:

1. **Setup:** It begins by defining all physical and geometric parameters. This includes physical constants (frequency `fc`, speed of light `c`), channel properties (transmit power `P_TX`), antenna parameters (`R_a`), and the environment itself (the coordinates and relative permittivity `eps_r` of each wall in the `walls` struct). The maximum reflection order `K` is also set here.

2. **Analysis Loop (e.g., Heatmap Generation):** The script's primary computational load resides in its analysis loops. For the 2D heatmap, it establishes a grid of receiver points ($\vec{r}_{RX}$) and iterates through each one.

   - For each grid point, it calls the main ray-tracing engine: `[alphas, ~] = runRayTracing(walls, K, tx_pos, rx_pos, params)`. This single call encapsulates the entire complex process of discovering all valid LOS and reflected paths.
   - If the engine returns one or more valid paths (i.e., `~isempty(alphas)`):
     - It calculates the Instantaneous Power by first coherently summing the complex gains to get the total channel response, $H_{NB} = \sum \alpha_n$, and then computing $P_{RX} = P_{TX}|H_{NB}|^2$.
     - It calculates the Local Average Power by incoherently summing the power contributions of each path: $\langle P_{RX} \rangle = P_{TX} \sum |\alpha_n|^2$.
   - The resulting power values (in dBm) are stored in 2D matrices corresponding to the grid.

3. **Visualization:** Finally, it passes the result matrices to various plotting functions to generate figures.

```

            % --- Instantaneous Power P_RX ---
            h_nb = sum(alphas);
            P_RX_insts = params.P_TX * abs(h_nb)^2;
            P_RX_inst_dBm(j, i) = 10 * log10(P_RX_insts * 1000);

            % --- Local Average Power <P_RX> ---
            power_of_each_path = params.P_TX * abs(alphas).^2;
            P_RX_local_avgs = sum(power_of_each_path);
```

Listing 1: Core power calculation loop inside `main.m`

Line-by-Line Mathematical and Algorithmic Mapping:

- **Line 229:** This is the primary external function call from the main loop. It invokes the **entire** ray-tracing process for a single TX-RX pair, delegating the complex discovery and validation logic.
- **Line 231:** Implements the coherent sum for the channel transfer function, $H_{NB} = \sum_{n=1}^{N} \alpha_n$.
- **Line 232:** Implements the instantaneous received power formula: $P_{RX} = P_{TX}|H_{NB}|^2$.
- **Line 235:** Implements the incoherent power sum for local average power by first calculating the power of each path individually (`params.P_TX * abs(alphas).^2`) and then summing them: $\langle P_{RX} \rangle = \sum P_{RX,n} = P_{TX} \sum |\alpha_n|^2$.

2.2 `runRayTracing.m`: The Main Engine

Purpose: To coordinate the entire process of finding all LOS and reflected rays for a single TX-RX pair. It acts as a manager, first handling the special case of the LOS path and then initiating the recursive search for reflected paths.

Core Algorithm:

1. **LOS Path First:** The function explicitly checks for an unobstructed Line-of-Sight path. This is a critical first step.
   - It loops through every wall defined in the environment.
   - In each iteration, it calls `findSegmentIntersection` to determine if the direct line segment from $\vec{r}_{TX}$ to $\vec{r}_{RX}$ intersects the current wall segment.
   - If *any* intersection is found, a flag `is_los_obstructed` is set to true, and the loop terminates immediately.
   - If the loop completes without finding any intersections, the LOS path is deemed valid. Its properties (distance, gain) are calculated, and the resulting ray data is stored.
2. **Reflected Paths:** After the LOS check, the function initiates a search for reflected paths.
   - It loops through reflection orders ‘order = 1‘ to the maximum specified, ‘K‘.
   - For each ‘order‘, it starts the recursive search by making the top-level call to `findReflectedRaysRecursive`. This call begins the exploration of all possible reflection sequences of that specific order.
3. **Consolidate and Return:** It combines the valid LOS ray (if found) and all valid reflected rays (returned from the recursive calls) into a single cell array. It also extracts just the complex gains (‘all_alphas‘) into a separate vector for convenience.

```
%
% OUTPUTS:
%   all_alphas           - 1xN complex vector of channel gains for the N found rays.
%   all_rays_data        - 1xN cell array of structs, each with detailed ray info.

    all_rays_data = {};

    % --- LINE-OF-SIGHT (LOS) PATH CALCULATION ---
    % Check if the direct path between TX and RX is obstructed by any wall.
    is_los_obstructed = false;
    for i = 1:length(walls)
```

```
12          intersection_point = findSegmentIntersection(tx_pos, RX_pos, walls(i).
    coordinates(1,:), walls(i).coordinates(2,:));
13          if ~isempty(intersection_point)
14              is_los_obstructed = true;
15              break; % If one wall obstructs, no need to check others.
16          end
17      end
18
19      % If not obstructed, calculate the properties of the LOS ray.
20      if ~is_los_obstructed
21          los_ray.coordinates = [tx_pos; RX_pos];
22          los_ray.type = 'LOS';
23          los_ray.distance_total = norm(RX_pos - tx_pos);
24          los_ray.gamma_prod = 1; % Reflection coefficient product is 1 for LOS.
25          los_ray.alpha_n = calculateAlpha_n(los_ray, params);
26          all_rays_data{end+1} = los_ray;
27      end
```

Listing 2: LOS check and recursion initiation in `runRayTracing.m`

Line-by-Line Mathematical and Algorithmic Mapping:

- **Lines 16-20:** This loop implements Step 1 of the algorithm. It is a direct obstruction check. The call to `findSegmentIntersection` on line 17 is the core of this check.
- **Lines 25-29:** Implements the final part of Step 1. If the LOS path was not obstructed, its physical data is packaged into a struct, including the call to `calculateAlpha_n` to compute its gain.
- **Lines 32-40:** Implements Step 2. The `for` loop iterates through the desired reflection orders. The call on line 36, `findReflectedRaysRecursive(...)`, is the entry point to the entire reflection discovery process for a given order. All rays found by the recursive search are then appended to the main results list on line 39.

2.3 `findReflectedRaysRecursive.m`: The Discovery Engine

Purpose: To implement the Method of Images using recursion. This function builds and explores a search tree of potential reflection sequences to discover all possible ray paths for a given reflection order. Each node in the tree represents an image source.

Core Algorithm (A Recursive Depth-First Search):

- **Base Case (Termination):** The recursion terminates when `num_reflections_remaining` is 0. This signifies that a full path hypothesis has been formed (i.e., a specific sequence of N walls has been chosen).
  - At this point, the sequence is purely hypothetical. The function calls `validateRayPath` to perform the crucial check of whether this sequence corresponds to a physically possible, unobstructed path in the finite geometry.
  - If validation is successful (`validateRayPath` returns a non-empty result), it then calls `calculatePhysicalProperties` to compute the ray's actual length, reflection losses, and final complex gain.
  - The fully formed and validated ray data is then returned up the call stack.
- **Recursive Step:** If more reflections are needed (`num_reflections_remaining > 0`):

- It iterates through all walls, considering each as a potential candidate for the *next* reflection surface.
- *Pruning Step:* An important optimization is to skip the wall if it was the one just reflected from (`if i == last_wall_index`). This prevents physically redundant paths (e.g., A → Wall 1 → A).
- *Image Method:* For each potential reflecting wall, it creates a new virtual source (an image) by reflecting the current source across the wall's infinite line. This is done by calling the geometric utility `findSymmetricAcrossLine`.
- *Descend the Tree:* It then makes a recursive call to itself: `findReflectedRaysRecursive(...)`. This call uses the newly created image as the source, decrements the number of remaining reflections, and passes along the updated history of walls and image positions.
- *Collect Results:* It collects any valid rays found by the recursive call and appends them to its own list of results, which will be passed up to its own caller.

```
%   valid_rays_found    - A cell array of completed, valid ray data structs
%                         found from this recursive branch.

    valid_rays_found = {};
      % Initialize a cell array to store results from this recursive path.

    % --- BASE CASE: No more reflections needed ---
    % When the desired number of reflections is reached, the full path from the
    % final image source to the receiver can be traced and validated.
    if num_reflections_remaining == 0
        % The full reflection sequence is now hypothesized; validate its geometric path.
        ray_coordinates = validateRayPath(RX_pos, walls, processed_wall_indices,
    all_tx_images_pos);

        % A non-empty result from validateRayPath means the path is unobstructed
        % and physically possible.
        if ~isempty(ray_coordinates)
            ray_data = calculatePhysicalProperties(ray_coordinates,
    processed_wall_indices, walls, params);  % If the path is valid, compute its
    physical characteristics (e.g., length, gain).
            valid_rays_found = {ray_data};
                    % Package the complete, valid ray data to be returned up the call
    stack.
        end
        return;
                    % Terminate this branch of recursion and return the findings.
    end

    % --- RECURSIVE STEP: Find the next reflection ---
    last_wall_index = 0;
      % Will hold the index of the wall used in the previous reflection step.
    if ~isempty(processed_wall_indices)
        last_wall_index = processed_wall_indices(end);
           % Get the index of the most recent wall to avoid immediate re-reflection.
    end

    % Iterate through all possible walls to serve as the next reflecting surface.
    for i = 1:length(walls)
        % This check prevents the ray from immediately reflecting back and forth off
        % the same wall, which is a physically redundant path.
```

```
32          if i == last_wall_index
33              continue;
34          end
35
36          % This is the core of the Method of Images: create a new virtual source
37          % by reflecting the current source across the plane of the wall.
```

Listing 3: Base case and recursive step in `findReflectedRaysRecursive.m`

Line-by-Line Mathematical and Algorithmic Mapping:

- **Lines 17-26:** This is the **Base Case**. The recursion has reached its maximum depth. Line 20 calls the `validateRayPath` function to see if the hypothesis is real. If so, line 24 calls `calculatePhysicalProperties` to compute the physics.
- **Lines 31-53:** This is the **Recursive Step**.
- **Line 43:** This line performs the core geometric operation of the Method of Images by calling `findSymmetricAcrossLine` to create a new virtual source for the next level of recursion.
- **Lines 46-49:** This is the recursive call itself. It descends one level deeper into the search tree to find the next reflection, passing the new state (new source, decremented counter, updated history).
- **Line 53:** This line aggregates results. As the recursion unwinds, this line collects all valid rays found from all successful branches of the search tree originating from this node.

2.4 `validateRayPath.m`: The Verification Engine

Purpose: To determine if a ray path, hypothesized by the Method of Images, is geometrically and physically valid. The Method of Images works with infinite lines, but in reality, walls are finite and other walls can cause obstructions. This function verifies the path against these real-world constraints.

Core Algorithm: Backward Ray-Tracing The logic traces the hypothesized path backward, from the receiver to the transmitter. This is more efficient as it immediately prunes invalid paths.

1. **Initialize:** The starting target point is the receiver: $\vec{p}_{target} = \vec{r}_{RX}$. A matrix is allocated to store the path's vertices.
2. **Backward Loop:** Loop from the last reflection ($i = K$) down to the first ($i = 1$):
   (a) **Find Reflection Point:** The current ray segment arriving at $\vec{p}_{target}$ must appear to come from the image source $\vec{r}_{TX}^{(i+1)}$. The function finds the intersection of the line segment $[\vec{r}_{TX}^{(i+1)}, \vec{p}_{target}]$ with the reflecting wall i. This is done by calling `findSegmentIntersection`.
   (b) **Validation 1 (On-Segment Check):** The call to `findSegmentIntersection` itself performs the first validation. If it returns empty, it means the ray hits the wall's infinite line but misses the finite segment. The path is invalid, and the function aborts.
   (c) **Validation 2 (Obstruction Check):** If the reflection point $\vec{r}_{R_i}$ is valid, the function must check if the segment $[\vec{r}_{R_i}, \vec{p}_{target}]$ is blocked by any *other* wall. It loops through all walls and calls `findSegmentIntersection` for each. If an intersection is found with any wall that is not part of the valid path, the path is obstructed and invalid. The function aborts.
   (d) **Update:** If the segment is clear, the new target for the next iteration is the current reflection point: $\vec{p}_{target} = \vec{r}_{R_i}$.

3. **Final Segment Validation:** After the loop completes, one final check is needed for the first segment of the path, from the true transmitter to the first reflection point, $\vec{r}_{TX} \rightarrow \vec{r}_{R_1}$. This segment is also checked for obstructions against all other walls.

4. **Return:** If and only if all segments pass all checks, the function returns the full list of validated coordinates: $[\vec{r}_{TX}; \vec{r}_{R_1}; ...; \vec{r}_{R_K}; \vec{r}_{RX}]$. Otherwise, it returns an empty array.

```
% Pre-allocate matrix for all ray vertices: TX, N reflection points, RX.
potential_ray_coordinates = zeros(K + 2, 2);
  % Allocate memory for the path vertices: TX, N reflection points, and RX.
original_tx_pos = all_tx_images_pos{1};
  % The true transmitter.
potential_ray_coordinates(1,:) = original_tx_pos;
  % The path must start at the true transmitter (TX).
potential_ray_coordinates(end,:) = RX_pos;
  % The path must terminate at the receiver (RX).
is_path_valid = true;
  % Assume the path is valid until a check fails.
current_target_point = RX_pos;
  % Begin the backward trace from the receiver's location.

% --- Trace backwards from RX to TX and check for obstructions ---
for i = K:-1:1
  % Trace path segments backwards, from the final reflection toward the first.
    image_source_for_segment = all_tx_images_pos{i+1};
  % The ray segment appears to originate from the corresponding image source.
    reflecting_wall_index = processed_wall_indices(i);
  % Identify the wall associated with this step of the reflection sequence.
    reflecting_wall_coordinates = walls(reflecting_wall_index).coordinates;
  % Get the start and end coordinates of this reflecting wall.

    % Find the reflection point on the current reflecting wall.
    % This is the intersection of the line from the current target point
    % to the image source with the reflecting wall segment.
    reflection_point = findSegmentIntersection(...
   % Calculate the exact point of reflection on the wall.
        image_source_for_segment, current_target_point, ...
   % A line is cast from the image source to the target of the previous segment.
        reflecting_wall_coordinates(1,:), reflecting_wall_coordinates(2,:));
   % The intersection with the finite wall segment is the reflection point.
    if isempty(reflection_point)
   % If the line of sight to the image misses the physical wall segment.
        is_path_valid = false;
   % the geometric path is impossible.
        break;
   % Stop processing, as the entire ray path is invalid.
    end
    potential_ray_coordinates(i+1,:) = reflection_point;
   % Store this valid reflection point as a vertex in the ray path.

    % Check for obstructions on the current ray segment (reflection_point to
current_target_point).
    for j = 1:length(walls)
   % Iterate through all walls in the environment to check for blockages.
        % Skip checking the current reflecting wall and the next reflecting wall
        % (if applicable) as they are part of the intended path.
        is_this_segments_wall = (j == reflecting_wall_index);
  % The ray is allowed to hit the wall it's reflecting from.
```

```
32              is_next_segments_wall = (i < K && j == processed_wall_indices(i+1));
    % And the wall of the next reflection (the previous in this backward trace).
33              if ~is_this_segments_wall && ~is_next_segments_wall
    % For all other walls...
```

Listing 4: The backward-tracing loop in `validateRayPath.m`

Line-by-Line Mathematical and Algorithmic Mapping:

- **Line 30:** This begins the main backward-tracing loop (Step 2 of the algorithm).
- **Lines 36-39:** This implements Step 2a. The call to `findSegmentIntersection` determines the point of reflection by tracing from the target back toward the relevant image source.
- **Line 40:** This implements Step 2b. A simple `isempty` check determines if the reflection point lies on the finite wall. If not, the path is invalid.
- **Lines 45-58:** This implements Step 2c, the critical obstruction check. The inner loop iterates through all potential blocking walls. The `if` conditions on lines 48 and 50 ensure that the segment is not incorrectly flagged as obstructed by the very walls it is supposed to reflect from. The call to `findSegmentIntersection` on line 51 performs the actual check.
- **Line 62:** This implements Step 2d, updating the target for the next step in the backward trace. This point becomes the end-point for the next segment to be validated.

2.5 `calculatePhysicalProperties.m`: The Physics Engine

Purpose: To take a geometrically validated path (represented by its vertices) and compute its physical characteristics: total distance, cumulative reflection loss, and final complex gain.

Core Algorithm:

1. **Initialize:** Total distance d_{tot} is set to 0. The cumulative reflection coefficient product Γ_{prod} is initialized to 1 (since a LOS path has no reflection loss).
2. **Loop Through Segments:** The function iterates through each segment of the validated path (e.g., TX→R1, R1→R2, ..., R(K)→RX).
   (a) It calculates the length of the current segment and adds it to d_{tot}.
   (b) If the segment ends in a reflection (i.e., it is not the final segment to the RX):
   - It calculates the angle of incidence, θ_i, with respect to the wall normal. This is found using the dot product between the incident ray vector and the wall's normal vector.
   - It calculates the Fresnel reflection coefficient, Γ_k, for that bounce, using the formula for perpendicular polarization. This calculation uses θ_i and the wall's stored relative permittivity, ϵ_r.
   - It multiplies this new coefficient into the cumulative product: $\Gamma_{prod} = \Gamma_{prod} \cdot \Gamma_k$.
3. **Finalize:** After processing all segments, it has the final d_{tot} and Γ_{prod}. It packages these into a struct and makes the final call to `calculateAlpha_n` to compute the overall complex gain for this specific path.

Key Mathematics:

$$\cos(\theta_i) = \frac{|\vec{v}_{inc} \cdot \vec{n}|}{\|\vec{v}_{inc}\|\|\vec{n}\|}$$

$$\Gamma_k(\text{perp. pol.}) = \frac{\cos(\theta_i) - \sqrt{\epsilon_r - \sin^2(\theta_i)}}{\cos(\theta_i) + \sqrt{\epsilon_r - \sin^2(\theta_i)}}$$

```
    is_reflection_segment = (n <= K);                                                    % True for all segments except the final one that lands at the receiver.
    if is_reflection_segment
        incident_vector = end_segment - start_segment;                                   % The vector representing the ray's direction as it strikes the wall.
        reflecting_wall = walls(hit_walls_indices(n));

        wall_vector = reflecting_wall.coordinates(2,:) - reflecting_wall.coordinates(1,:);
        normal_vector = [wall_vector(2), -wall_vector(1)];
            % A vector perpendicular to the wall's surface, essential for angle calculations.

        % Use the dot product to find the cosine of the angle between the
        % incident ray and the line normal to the wall's surface.
        cos_theta_n = abs(dot(incident_vector, normal_vector)) / (norm(incident_vector) * norm(normal_vector));
        sin_theta_n = 1 - cos_theta_n^2;
        epsilon_r = reflecting_wall.eps_r;                                               % The wall's relative permittivity, a material property governing reflection.
```

Listing 5: Reflection coefficient calculation in `calculatePhysicalProperties.m`

Line-by-Line Mathematical and Algorithmic Mapping:

- **Line 33:** Implements the formula for $\cos(\theta_i)$ using the normalized dot product of the incident vector and the wall normal.
- **Line 34:** Calculates $\sin^2(\theta_i)$ using the identity $\sin^2(\theta) + \cos^2(\theta) = 1$.
- **Lines 36-37:** These lines directly implement the numerator and denominator of the Fresnel reflection coefficient formula for perpendicular polarization.
- **Line 39:** This line implements the update step for the cumulative reflection product, $\Gamma_{prod} = \Gamma_{prod} \cdot \Gamma_k$, accumulating the loss from each bounce.
- **Line 50:** This line makes the final call to `calculateAlpha_n`, passing it the fully computed distance and cumulative reflection coefficient to get the final result.

2.6 Geometric Utilities (`findSymmetric...` & `findSegment...`)

These are low-level functions that implement fundamental, self-contained geometric operations. They are the building blocks upon which the higher-level algorithms are built.

2.6.1 `findSymmetricAcrossLine.m`

Purpose: Reflects a point across a line, the core operation of the Method of Images. **Mathematical Implementation:** It reflects point $\vec{p}$ across a line defined by a point $\vec{l}_0$ and a normal vector $\vec{n}$. It directly

calculates the reflected point $\vec{p}'$ using the vector projection formula:

$$\vec{p}' = \vec{p} - 2 \cdot \text{proj}_{\vec{n}}(\vec{p} - \vec{l}_0) = \vec{p} - 2\frac{(\vec{p} - \vec{l}_0) \cdot \vec{n}}{\|\vec{n}\|^2}\vec{n} \tag{5}$$

The code implements this by finding a vector on the line, calculating its perpendicular normal vector, and then applying the formula.

2.6.2 `findSegmentIntersection.m`

Purpose: Finds the intersection point, if any, of two *finite* line segments. **Mathematical Implementation:** It solves the system of parametric vector equations for the intersection point $\vec{I}$:

$$\vec{I} = \vec{p}_1 + t(\vec{p}_2 - \vec{p}_1)$$
$$\vec{I} = \vec{p}_3 + u(\vec{p}_4 - \vec{p}_3)$$

This vector equality represents a system of two linear equations in two variables, the parameters t and u. The code solves for t and u using Cramer's rule. The crucial final step is the validation check: an intersection point is valid if and only if it lies on both segments, which mathematically means $t \in [0, 1]$ and $u \in [0, 1]$. If this condition is not met, the infinite lines may intersect, but the segments do not, so an empty array is returned.