Sparse Optical Flow for Autonomous Navigation in AI2-THOR

Introduction to Optical Flow

Optical flow refers to the pattern of apparent motion of objects or surfaces in a scene, as observed through a moving camera or moving objects. It is typically represented as a 2D vector field, where each vector indicates the displacement of a point or pixel from one frame to the next [1]. In essence, optical flow provides **temporal information** about how different parts of an image move between consecutive frames, enabling the perception of scene dynamics from a monocular camera [2].

Dense vs. Sparse Optical Flow: There are two broad categories of optical flow estimation: **dense** and **sparse**. Dense optical flow computes a motion vector for every pixel in the image (or a regular grid of pixels), producing a complete flow field. Techniques like Farnebäck's algorithm or modern deep learning methods yield dense flow but at a higher computational cost. **Sparse optical flow**, on the other hand, tracks only specific feature points (e.g. corners or textured patches) across frames using algorithms such as Lucas–Kanade. Sparse methods are generally faster since they focus on informative points and ignore homogeneous regions. They are well-suited for real-time applications where only the prominent motions are needed (for example, tracking obstacles or ego-motion), while dense methods provide rich detail useful for tasks like video interpolation or motion segmentation.

Illustration of dense vs. sparse optical flow on a simple scene. A white rectangle moves to the right between two frames. Red arrows (sampled on a grid) show dense flow vectors (Farnebäck method) capturing motion across the whole image, while green arrows show sparse flow vectors (Lucas–Kanade) tracked only at Shi-Tomasi corner features. The sparse method captures motion at the rectangle's corners, whereas the dense field captures motion in a broader area including object edges and background.

In navigation contexts (like autonomous driving or mobile robots), sparse optical flow is often preferred due to its efficiency and focus on pertinent environmental features [3]. By tracking a set of corner points or visual landmarks, an agent can infer its own movement and detect moving or looming obstacles without processing every pixel. Dense flow can provide a detailed motion *map*, but processing it in real-time can be challenging and may be unnecessary if we only need the overall motion cues and obstacle positions.

Theoretical Background

In this section, we cover the foundations of sparse optical flow as applied in our navigation system: the Lucas–Kanade tracking method, the Focus of Expansion (FOE) for understanding camera motion, and the concept of visual potential fields for navigation. Key equations will be introduced to clarify these concepts.

Lucas–Kanade Optical Flow Method

The **Lucas–Kanade (LK)** method is a classic approach for estimating optical flow for a sparse set of feature points. The core assumption in LK is that over a small neighborhood (window) around a feature, the motion (flow) is essentially constant [4]. This means that all pixels in that local patch undergo the same displacement. Under this assumption, the basic optical flow constraint equation can be written for each pixel in the patch:

$$I_x(x,y)\,u + I_y(x,y)\,v + I_t(x,y) = 0,$$

where I_x, I_y are the spatial image gradients, I_t is the temporal intensity difference between two consecutive frames, and (u,v) is the optical flow vector (the unknown) at that patch [5]. This linear equation represents the fact that the intensity of the pixel at location (x,y) in the first frame matches the intensity at $(x+u,\ y+v)$ in the second frame (brightness constancy), to first order.

For a single pixel, this equation is underdetermined (two unknowns u, v with one equation). Lucas–Kanade resolves this by considering a window of n pixels around the feature, leading to a system of n equations. Let these equations be written in matrix form as $A\mathbf{v} = \mathbf{b}$, where $\mathbf{v} = [u,\ v]^T$ is the flow vector, and A and $\mathbf{b}$ are formed from the gradients and temporal differences of all pixels in the window [6] [7]. Specifically,

- A is an $n \times 2$ matrix containing the spatial gradients $\begin{bmatrix} I_x(q_1) & I_y(q_1) \\ I_x(q_2) & I_y(q_2) \\ \vdots & \vdots \\ I_x(q_n) & I_y(q_n) \end{bmatrix}$ for pixels $q_1, \ldots, q_n$ in the window.
- $\mathbf{b}$ is an $n \times 1$ vector of the negative temporal gradients $-I_t(q_i)$.

Lucas–Kanade finds the least-squares solution for $\mathbf{v}$ by solving the normal equations $A^T A\,\mathbf{v} = A^T\mathbf{b}$. The resulting estimate for the flow is:

$$\mathbf{v} = (A^T A)^{-1} A^T \mathbf{b},$$

which can be expanded into the familiar form:

$$\begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} \sum_i I_{x_i}^2 & \sum_i I_{x_i} I_{y_i} \\ \sum_i I_{x_i} I_{y_i} & \sum_i I_{y_i}^2 \end{bmatrix}^{-1} \begin{bmatrix} -\sum_i I_{x_i} I_{t_i} \\ -\sum_i I_{y_i} I_{t_i} \end{bmatrix},$$

where sums are over all pixels i in the feature's neighborhood [8]. This equation computes the average optic flow (u,v) that best satisfies the constraint for all pixels in the patch, by minimizing the squared error.

In practice, the steps for sparse optical flow using Lucas–Kanade are: 1. **Feature Detection:** Identify good features to track (e.g., corners using Shi–Tomasi or Harris detector). Corners are ideal because they have well-defined gradients in two directions, making $A^T A$ invertible (high eigenvalues) [9]. 2. **Local Flow Computation:** For each feature, compute image gradients I_x, I_y and temporal difference I_t over the surrounding window in the two frames, then solve the above equations for u, v. 3. **Iterative & Pyramidal**

Refinement: For larger motions, an image pyramid is used. The flow is first estimated on coarse (scaled-down) images and then refined at higher resolutions, which helps capture large displacements by breaking them into smaller motions [10].

Below is a snippet demonstrating how one might use OpenCV's implementation of Lucas–Kanade in Python to track features between frames:

```
import cv2
# Assume prev_frame, curr_frame are subsequent RGB images from the simulation
prev_gray = cv2.cvtColor(prev_frame, cv2.COLOR_BGR2GRAY)
curr_gray = cv2.cvtColor(curr_frame, cv2.COLOR_BGR2GRAY)
# Detect corners to track in the previous frame
prev_points = cv2.goodFeaturesToTrack(prev_gray, maxCorners=200,
qualityLevel=0.01, minDistance=5)
# Calculate sparse optical flow (Lucas-Kanade) for those points
next_points, status, err = cv2.calcOpticalFlowPyrLK(prev_gray, curr_gray,
prev_points, None)
# Filter out points for which tracking failed
good_new = next_points[status==1]
good_old = prev_points[status==1]
```

In our implementation, we similarly use Shi–Tomasi to get up to a few hundred feature points in each frame, then track them via Lucas–Kanade to obtain their motions (as in the code above). The result is a set of sparse flow vectors $\{(x_i, y_i, u_i, v_i)\}$ for the tracked features, which forms the basis for higher-level navigation computations.

Focus of Expansion (FOE) Estimation

When the camera itself is moving (e.g., an ego-vehicle or robot moving forward), the optical flow induced by the static environment exhibits a characteristic pattern: all flow vectors **appear to radiate from a single point** in the image. This point is known as the **Focus of Expansion (FOE)**. The FOE corresponds to the direction of the camera's motion projected into the image plane – essentially the vanishing point of the motion [11]. If the camera is moving straight forward, the FOE is typically near the center of the image (assuming it's looking in the direction of travel). If the agent turns or the heading deviates, the FOE shifts accordingly (e.g., FOE will shift to the left side of the image if the camera is moving toward the left).

Mathematically, the FOE can be thought of as the **intersection point of all optical flow vectors** (for purely translational camera motion) [11]. In ideal noise-free conditions, one could find the FOE by extending each flow vector in the direction **opposite** to motion (i.e., reverse the arrows) and finding the common intersection. In practice, due to noise and the fact that only sparse flow vectors are available, a robust approach is used: solve a least-squares problem to find the point (x_{FOE}, y_{FOE}) that best fits the direction of all flow vectors [12]. One such formulation is:

- For each tracked feature with flow (u_i, v_i) at image position (x_i, y_i), we can impose that the vector from (x_{FOE}, y_{FOE}) to (x_i, y_i) is parallel to the flow (u_i, v_i). This leads to equations of the form

$(y_i - y_{FOE}) = m_i\,(x_i - x_{FOE})$ for some slope $m_i = v_i/u_i$. Solving these simultaneously yields the FOE coordinates.

- Equivalently, one can set up a linear system $A\mathbf{p} = \mathbf{b}$ where $\mathbf{p} = [x_{FOE},\, y_{FOE}]^T$. A convenient form comes from the relation that the optical flow direction (u_i, v_i) should point away from FOE: $v_i(x_i - x_{FOE}) - u_i(y_i - y_{FOE}) = 0$. This can be rearranged to $v_i x_{FOE} - u_i y_{FOE} = v_i x_i - u_i y_i$. Stacking similar equations for all points yields $A\mathbf{p} = \mathbf{b}$, which we solve by least squares [13] .

In our implementation, we compute FOE by constructing matrices as described in the literature [13] . The end result is an estimate (x_{FOE}, y_{FOE}) on the image plane. If the FOE lies at the center of the image, it suggests the agent's motion is directed straight ahead; if it's off-center, the motion is directed towards that off-center point (or the road is curving, in a driving scenario).

Illustration of a Focus of Expansion (FOE) in an optical flow field. Here a forward-moving agent produces a radial flow pattern where all flow vectors (red arrows) diverge from the center point (marked with a blue "×"). This FOE indicates the direction of motion (straight ahead in this example). If the agent were turning, the FOE would shift toward the direction of travel. [11]

The FOE is a crucial intermediate output: it provides a cue to the agent's heading direction relative to the scene. In navigation, we will use the FOE to decide if the agent is centered or if it needs to adjust its steering. For instance, if the FOE drifts to the left side of the image, it implies the agent is headed toward the left part of the scene – the navigation logic might then rotate the agent rightward to correct its course (and vice versa). Essentially, maintaining the FOE near the image center is analogous to keeping the agent moving straight toward an open area or down a corridor.

Another useful derivative of the FOE is the **Time-to-Contact (TTC)** or depth estimation. Flows closer to the FOE tend to be smaller, and flows far from FOE larger, for a given velocity. A simple approximation for the time to contact (the time before collision with an object) for a feature at (x_i, y_i) is given by $\mathrm{TTC}_i \approx \frac{\sqrt{(x_i - x_{FOE})^2 + (y_i - y_{FOE})^2}}{\sqrt{u_i^2 + v_i^2}}$ [14] . Intuitively, a feature that is moving faster (large $\sqrt{u^2 + v^2}$) and/or is located further from the FOE (larger radial distance) corresponds to a closer object and hence a smaller TTC (potential collision sooner). We leverage this concept in obstacle detection.

Visual Potential Fields for Navigation

To navigate autonomously, our agent uses a concept inspired by **Artificial Potential Fields (APF)** from robotics. The idea is to treat the camera's view as a kind of potential field map where the goal exerts an *attractive potential* (pulling the agent forward) and obstacles exert *repulsive potentials* (pushing the agent away). The agent can then "follow the gradient" of this virtual potential field to decide on motion commands. In our case, we derive a **Visual Potential Field** on the image plane using optical flow information [15] .

Attractive Potential (Goal): If a specific goal or direction is defined, we assign it a potential U_{att} that decreases with distance to the goal. A common choice is $U_{att} = \frac{1}{2}\,\alpha\, d^2(x, y)$, where $d(x, y)$ is the Euclidean distance in the image (or world) to the goal and α is a scaling constant [16] [17] . This creates a "gravity well" centered on the goal. The negative gradient of this potential gives a force (or desired velocity) pointing toward the goal: $\mathbf{F}_{att} = -\nabla U_{att}$ which is essentially $\alpha\, d(x, y)$ in magnitude pointing along the

direction of the goal [17]. If the goal is straight ahead in the image, this attractive force points straight forward.

Repulsive Potential (Obstacles): Obstacles are regions of high potential that the agent should be repelled from. We construct the obstacle potential field using the optical flow disturbances caused by obstacles. The key observation is that when an obstacle is directly in the agent's path, the optical flow vectors around that obstacle differ significantly from the general flow (e.g., features on the obstacle move faster or in different directions than the distant background). We generate a binary **obstacle map** $O(x, y)$ by segmenting out these flow **disturbances**. Specifically, we use an **Otsu threshold** on a measure of flow magnitude or TTC to distinguish likely obstacle points from background [18]. Points with unusually large flow (or small TTC) are marked as obstacle pixels in $O(x, y)$ (the intuition being that close obstacles move faster in the image).

Once we have an obstacle mask $O(x, y)$ from the current optical flow field, we refine it by convolving with a Gaussian filter and then taking a spatial gradient [19] [20]. The result is a smooth gradient field $g(x, y) = \nabla(G * O)$ which points from obstacle surfaces outward (this can be thought of as pointing from the obstacle region toward free space). This gradient is used to define a **repulsive force**. One formulation, adapted from prior work [21] [22], is:

$$\mathbf{F}_{rep} = \gamma \frac{1}{|R|} \left(\int_{(x,y)\in R} g(x, y)\, dxdy \Big/ \sum_{(x,y)\in R} \mathrm{TTC}(x, y) \right),$$

where R is a region-of-interest (e.g., the whole image or a portion of it) and γ is a gain factor [22]. In simpler terms, this equation says: sum up the obstacle gradient vectors in the region (making a vector pointing away from obstacles) and weight it by the inverse of TTC (so that closer obstacles yield a stronger repulsive push). The $1/|R|$ normalizes for the area of the region considered.

The agent's **total potential field** is then the sum of the attractive and repulsive potentials (and any others, such as road-lane potentials if applicable). Similarly, the **resultant force** or navigation vector $\mathbf{F}_{total}$ is the combination of the goal attraction and obstacle repulsion. In formula form, one might write:

$$\mathbf{F}_{total} = \mathbf{F}_{att} + \mathbf{F}_{rep} + \mathbf{F}_{road} + \cdots,$$

where $\mathbf{F}_{road}$ could be an optional term to keep the agent centered in a corridor or lane (not used in our indoor scenario, but used in autonomous driving to handle road boundaries [23] [24]).

Conceptual potential field in a 2D plane with a goal and an obstacle. The contour map shows potential values (hot colors = high potential, cool colors = low). A goal at the right exerts an attractive potential (global minimum at the blue star), and an obstacle at the center creates a repulsive potential hill (local maximum at the red circle). An agent navigating this field would be pulled toward the goal while being pushed away from the obstacle, following the negative gradient of the field (indicated by the slope of the contours).

In summary, by converting optical flow cues into a visual potential field, we obtain a framework to make navigation decisions: move in the direction that decreases the potential (toward open space and the goal) and away from high potential areas (obstacles). This approach is reactive and does not require an explicit map; it relies purely on instantaneous visual motion cues, which is advantageous for unknown environments [15].

Obstacle Detection via Optical Flow Disturbances

A critical capability for navigation is **obstacle detection** – identifying when an object is in the agent's path so that it can be avoided. In our system, obstacle detection is accomplished purely through the analysis of optical flow; no depth sensors or object recognizers are used. The premise is that obstacles (especially close ones) induce *measurable disturbances* in the flow field.

When the agent is moving and viewing a mostly static scene (e.g., down an empty hallway), the optical flow vectors follow a smooth pattern (approximately radial from the FOE). A looming obstacle (e.g., a chair in the hallway) will break this pattern. Features on the obstacle, being much closer, will move faster across the image (higher magnitude flow) and their presence may shift the FOE or distort the flow in that region. We leverage these effects as follows:

- **Flow Magnitude Thresholding:** We compute the magnitude $\sqrt{u_i^2 + v_i^2}$ for each sparse flow vector. Points with abnormally large flow relative to the median or mean flow are flagged. This is based on the idea that, for a forward moving agent, distant background points have smaller image motion, whereas closer objects have larger optical flow (assuming equal time steps). We automatically determine a cutoff using Otsu's method, which finds an intensity threshold that separates the flow magnitude histogram into two classes (background vs. foreground motion) [18]. The output is a binary mask $O(x, y)$ marking potential obstacle regions in the image.
- **Focus on Region in Front:** We may also limit our attention to the central area of the image (since objects directly ahead are of most immediate concern). In an indoor scenario, for example, flows near the image center (just below the FOE) might correspond to straight ahead, whereas flows on extreme periphery might be walls gliding by. Our implementation can define a region of interest R (for instance, a rectangular area in the middle of the image) and apply thresholding within that region to find obstacles.
- **Refining the Obstacle Map:** The initial obstacle mask from thresholding can be noisy. We apply a Gaussian smoothing $G * O$ to blur the mask slightly, then compute the spatial gradient $g(x, y) = \nabla(G * O)$ [20]. This yields a vector field pointing outward from obstacle regions. Think of this like finding the "edges" of the obstacle in the mask – these edges represent where free space meets obstacle, and the gradient points from obstacle to free space. This gradient field is extremely useful for navigation since it directly indicates which way is away from the obstacle (to be used in the repulsive force calculation as described earlier).

By monitoring the **optical flow vector field for disturbances**, the system can robustly pick out obstacles. For instance, if the agent approaches a table, features on the table will quickly move in the camera view, creating a cluster of high flow vectors that stands out from the floor or walls (which are farther and move slowly). Our approach does not explicitly calculate real-world distances – it infers *relative proximity* from optical flow, which is often sufficient to trigger avoidance behaviors.

One limitation to note is that if the entire scene uniformly has high optical flow (e.g., the agent accelerates or turns quickly), thresholding might not separate obstacle vs. background effectively. Similarly, low-texture obstacles might not provide enough features to track (the classic "aperture problem" or lack of Shi-Tomasi corners). In our experiments in AI2-THOR, however, most objects (furniture, appliances, etc.) have texture or edges that yield trackable points, and our agent moves at moderate, steady speeds so that optical flow remains analyzable.

AI2-THOR Integration

Having described the vision algorithms, we now explain how they tie into an **autonomous agent in the AI2-THOR simulator**. AI2-THOR is an interactive 3D environment with indoor scenes (rooms, kitchens, etc.) where an agent can move and receive visual observations (RGB frames) at each step. We implemented a navigation loop in AI2-THOR that uses sparse optical flow to decide the agent's actions in real-time.

Frame Capture: We initialize an AI2-THOR `Controller` to load a scene (e.g., "FloorPlan1", a kitchen) and set the camera resolution (e.g., 640×480). The agent is initially placed at some location in the scene. At each time step, the following occurs:

```
event = controller.step(action="MoveAhead")  # or "RotateRight", etc.
frame = event.frame  # RGB image (NumPy array) from the agent's camera
```

Here, `controller.step` executes an action in the environment. We primarily use **movement actions** like `MoveAhead` (move forward by a fixed small step) and slight rotations (`RotateLeft` or `RotateRight` by a few degrees) to adjust orientation. The simulator returns an `event` which contains, among other things, the `frame` (a raw RGB image) from the agent's camera after the action.

We maintain a loop where we always have the previous frame `prev_frame` and can thus compute optical flow between `prev_frame` and the current `frame`. The frames are converted to grayscale and passed into our **sparse optical flow pipeline** (Shi-Tomasi feature detection + Lucas-Kanade tracking) as described earlier. This yields a set of flow vectors for the current step.

Computing FOE and Potential Forces: With the flow vectors in hand, we calculate the FOE for the current frame using the least-squares method. The FOE gives us a sense of where the agent is heading in the image. We then update our obstacle map $O(x, y)$ by thresholding the new flow magnitudes (and possibly incorporating a memory or inertia to smooth out detection over successive frames). From the obstacle map, we derive the obstacle repulsive force F_{rep} , and we define an attractive direction F_{att} toward the goal. In AI2-THOR experiments, often the "goal" can be simply to move forward without collision or to reach the far end of the room. For simplicity, we might assume the goal is straight ahead in the camera's view (e.g., the agent intends to go forward indefinitely or to a distant point in front). Thus, F_{att} can be a fixed vector pointing down the image center (encouraging forward movement). More sophisticated setups could designate a particular object or location as a goal, and if we know its image coordinates, we could compute F_{att} accordingly.

Now we combine the cues: - If the FOE is perfectly centered, the agent is moving straight; if the FOE is off-center, it indicates a turn is needed. We interpret FOE offset as an indication of a **course correction**. For example, if x_{FOE} is significantly to the right of the image center, it suggests the agent's motion is veering right (or the corridor bends right); depending on context, we might turn slightly right (to align with the path) or left (to re-center the FOE). In a road-following scenario, one would steer *toward* the FOE to follow the road curvature [25]. In an indoor scenario, keeping FOE centered generally keeps the agent aligned in the middle of a hallway. - The obstacle force F_{rep} tells us which way to steer to avoid obstacles. If an obstacle is detected slightly to the left of center, the repulsive force will have a rightward component, nudging the agent to turn right. Conversely, an obstacle on the right yields a leftward repulsive push. The

magnitude of this force can be scaled by the confidence or size of the obstacle region (and by TTC – nearer obstacles provoke stronger responses).

Action Decision: We translate the combined force $F_{total} = F_{att} + F_{rep}$ (and implicit FOE information) into a discrete action. One simple approach is: - Compute the desired heading change: $\Delta\theta = k \cdot (x_{FOE} - x_{\text{center}}) + c \cdot F_{rep,x}$, where x_{center} is the image center's x-coordinate, $F_{rep,x}$ is the horizontal component of repulsive force, and k, c are tuning gains. This $\Delta\theta$ might be positive (turn right) or negative (turn left). - If $\Delta\theta$ is above a certain threshold, execute a small `RotateRight` action. If below a negative threshold, do a `RotateLeft`. If it's near zero, no rotation is needed. - Always move forward (unless an imminent collision is predicted, in which case maybe stop or step back – in our case, we assume forward is safe as long as obstacles are managed by turning).

Thus the agent continuously **moves ahead** and occasionally **yaws** left or right to dodge obstacles or follow the "flow" of the free space. The AI2-THOR API allows specifying continuous rotation angles, but we found it sufficient to use incremental 5°–15° rotations for course corrections each time step, making the agent's path smooth.

To summarize the integration algorithm in pseudocode:

```
controller.reset(scene)
prev_frame = None

for t in range(max_steps):
    if prev_frame is None:
        # first step, just capture initial frame
        event = controller.step(action="Pass")  # no movement, just get frame
        prev_frame = cv2.cvtColor(event.frame, cv2.COLOR_RGB2GRAY)
        continue

    # Move forward by default
    event = controller.step(action="MoveAhead")
    curr_frame = cv2.cvtColor(event.frame, cv2.COLOR_RGB2GRAY)

    # Sparse optical flow from prev_frame to curr_frame
    prev_pts = cv2.goodFeaturesToTrack(prev_frame, ...)
    next_pts, status, _ = cv2.calcOpticalFlowPyrLK(prev_frame, curr_frame,
prev_pts, None)
    flow_vectors = next_pts - prev_pts  # (u,v) for each point

    # Compute FOE from flow_vectors
    x_foe, y_foe = compute_FOE(flow_vectors, prev_pts)

    # Obstacle segmentation
    obstacle_mask = segment_obstacles(flow_vectors, threshold_method="otsu")
    # Compute repulsive force from obstacle_mask
    F_rep = compute_repulsive_force(obstacle_mask)
```

```
    # Decide rotation based on FOE and F_rep
    turn = 0
    if x_foe is not None:
        turn += k * (x_foe - image_center_x)
    turn += c * F_rep_x
    if turn > turn_threshold:
        controller.step(action="RotateRight", degrees=5)
    elif turn < -turn_threshold:
        controller.step(action="RotateLeft", degrees=5)
    # else, no turn

    prev_frame = curr_frame
```

In the above logic, `action="Pass"` is a no-op used to get the initial frame (AI2-THOR has an action to do nothing for one frame, which we use to initialize `prev_frame`). Then we enter a loop of moving and sensing. We continuously update `prev_frame` to the last seen frame to compute flow at the next iteration.

The actual implementation in our project follows this structure. The parameters k, c, \text{turn_threshold} were tuned heuristically (small values to yield gentle corrections). The function `compute_FOE` implements the least-squares solution described earlier, and `segment_obstacles` performs the Otsu threshold on flow magnitudes. `compute_repulsive_force` essentially finds the average gradient direction from the obstacle mask (pointing away from obstacle) and scales it by the mask density and flow magnitudes (as a proxy for TTC).

Results and Observations

We conducted trial runs of the above navigation strategy in AI2-THOR scenes. As a qualitative assessment, the agent was able to navigate forward and avoid simple obstacles using only the monocular camera input and optical flow analysis. Here we summarize some observations from the experiments:

- **Path Behavior:** In the absence of obstacles, the agent moves forward in a relatively straight line. The FOE remains near the image center, and the agent makes only minimal heading adjustments. This corresponds to a stable situation where the visual potential field has a smooth downhill slope straight ahead (goal ahead, no repellers). The agent's trajectory in this case is essentially a straight line, which is expected since the environment is unobstructed.

- **Obstacle Avoidance:** When an obstacle (e.g., a chair or a table) lies in the forward path, the optical flow vectors around that object spike in magnitude. The obstacle segmentation reliably detects the object when the agent gets closer – typically the flow-based detection kicks in when the obstacle begins to occupy a significant portion of the forward view (at that point, time-to-contact drops and flow magnitudes rise). Upon detection, the repulsive force F_{rep} causes the agent to turn. For instance, in one scenario a chair was slightly to the left; as the agent approached, the sparse flow on the chair legs stood out. The agent responded by rotating a few degrees to the right and then

continued forward, successfully bypassing the chair. Once past the obstacle, the disturbance in flow ceased and the FOE returned to center, allowing the agent to straighten out again.

- **FOE Guidance:** We logged the FOE position each frame. Without obstacles, x_{FOE} hovered around the center (~320 px for a 640 px width image) with small oscillations, indicating the agent maintained a centered course. When the agent executed an avoidance maneuver, we noted x_{FOE} would temporarily shift – for example, during a right-turn to avoid a left-side obstacle, the FOE shifted rightwards (as the agent's heading changed). Our controller would then sometimes slightly over-correct, causing a bit of a zig-zag (a known phenomenon in potential field navigation where the agent may oscillate if gains are not tuned). Smoothing the response or incorporating a slight penalty for oscillation could mitigate this, but even with minor oscillation the agent still avoided collisions. This behavior is qualitatively similar to what was reported in prior work [26] [27] – when obstacles are introduced, the path can become more **jerky** because the agent is constantly reacting to the changing flow/potential field.

- **Limitations Encountered:** In cluttered scenes with multiple obstacles, the simple strategy of combining forces showed some limitations. There were cases where two obstacles on either side created a tunnel effect; the optical flow indicated high motion on both sides, and the agent momentarily slowed (since repulsive forces canceled out laterally) before finding a gap. Since our current method doesn't explicitly plan a path but reacts greedily to the flow, it can get momentarily confused in complex scenarios. However, given that AI2-THOR environments are relatively structured (rooms, hallways), the agent did not get stuck in our tests – it always eventually found a way through by following whichever side had slightly less optical flow "pressure".

We did not quantitatively measure success rates or distances, but qualitatively, the sparse optical flow approach demonstrated **collision-free navigation** in the test scenes we tried. It's particularly noteworthy that all this was achieved **without** any mapping or prior knowledge of the environment – the agent's decisions were purely made on instantaneous visual cues, validating the potential of optical-flow-based reactive controllers.

(If available, one could include a plot of the agent's trajectory or a sequence of images showing the agent avoiding an object. In our case, logs were textual, so we describe the behavior in words.)

Conclusion and Future Work

In this report, we presented a detailed approach to autonomous navigation using **sparse optical flow** within the AI2-THOR simulation environment. The agent employs a vision-only, reactive control scheme inspired by the concept of visual potential fields. Key components of the system include Lucas–Kanade sparse tracking for efficient motion estimation, FOE calculation for heading direction, and optical-flow-based segmentation for obstacle detection. These elements come together to influence the agent's steering decisions in real time, enabling it to move toward a goal while avoiding collisions **without any explicit map or object recognition**.

Strengths of the Approach: This optical flow method is computationally light-weight (tracking a few hundred features per frame) and can be run in real-time. It mimics how living creatures use flow (e.g., how insects avoid obstacles by sensing optic flow divergence). The integration in AI2-THOR demonstrated that

such a biologically inspired, minimalist approach can handle basic navigation tasks. It's also inherently general – since it relies on motion, it can potentially work in any environment (indoor or outdoor) as long as there is texture to track and the agent's camera motion produces observable flow. Another advantage is that by using FOE and TTC, the agent gains some implicit understanding of depth and time-to-collision from a single camera [28] [29], which is valuable for planning evasive maneuvers.

Challenges and Limitations: As with any reactive method, there are scenarios that can be problematic. The optical flow can become unreliable in cases of rapid motion (motion blur) or when the environment lacks features (e.g., a blank wall yields no features for LK tracking). Strong lighting changes or reflections can also throw off the flow calculation. In our indoor tests, reflective surfaces like shiny floors occasionally created spurious corners and flows. Another known issue with potential field methods is the possibility of local minima – the agent might oscillate or get "stuck" in a niche if attractive and repulsive forces balance out incorrectly. While we did not observe a hard stuck case, the oscillatory behavior was noticed and would need smoothing. Moreover, the method currently doesn't have a memory or global planning; it won't remember where obstacles were beyond the current frame's view, which could lead to inefficient paths.

Future Work: There are several avenues to enhance this system: - **Dynamic Goal Estimation:** Instead of assuming a fixed goal direction, the agent could continuously infer a desirable direction from high-level objectives. For example, if the agent is supposed to reach a particular object, a higher-level vision module could detect that object and provide an approximate direction (which would then become the attractive potential in our framework). Merging such semantic information with optical flow could allow goal-directed navigation in complex scenes. - **3D or Multi-View Optical Flow:** Using stereo cameras or depth sensors could provide true 3D optical flow (often called **scene flow**). This would give actual distance measurements, making the obstacle avoidance more robust by distinguishing between a small nearby obstacle and a large faraway structure (which might produce similar flow magnitudes in monocular view). Even without adding new sensors, one could estimate **depth from optical flow** by incorporating more sophisticated Structure-from-Motion techniques, though that veers toward SLAM. A simpler improvement could be to use the agent's egomotion (if known) to subtract out expected background flow, highlighting moving obstacles (applicable if obstacles themselves can move, which in AI2-THOR they generally do not, but in real life they might). - **Learning-Based Refinement:** The current controller uses hand-tuned gains for combining FOE and repulsive forces. A learning-based approach (reinforcement learning or even a simple supervised mapping from optical flow features to actions) could potentially tune and adapt this better. The flow-based navigation could serve as a strong prior or initial policy that an RL agent can refine, combining the best of classical and learning-based methods. - **Integration of Rotation/Translation Estimation:** The FOE gives heading (translation direction). If we also estimate the **rotation** of the camera (from flow or gyroscope), we could compensate for rotational flow components to isolate purely translational flow for FOE and obstacle detection. In other words, removing ego-rotation helps focus on looming obstacles. AI2-THOR can provide egomotion data (since we command the moves), so we could subtract the optical flow caused by rotation between frames. This could improve obstacle mapping accuracy.

In conclusion, sparse optical flow proved to be a viable source of guidance for an autonomous agent in simulation. It underscores the idea that even in an era of deep learning and heavy sensor suites, **simple vision cues** like optical flow retain a lot of useful information for navigation [30] [31]. By tapping into these cues with a well-founded theoretical approach, we achieve a level of competency in navigation that is explainable, interpretable, and efficient. Moving forward, combining this approach with other modalities (depth, semantics) and improving the robustness will inch us closer to real-world deployment of vision-only navigation systems.

1 5 8 9 10 OpenCV: Optical Flow
https://docs.opencv.org/3.4/d4/dee/tutorial_optical_flow.html

2 3 ppms.cit.cmu.edu
https://ppms.cit.cmu.edu/media/project_files/OpticalFlow_IV2020_final.pdf

4 6 7 Lucas–Kanade method - Wikipedia
https://en.wikipedia.org/wiki/Lucas%E2%80%93Kanade_method

11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 OpticalFlow_IV2020_final.pdf
file://file-XmkAomLyUGWvJUkunjycS8