

Improving the Scalability of a Continuous Approach for Shadow Flicker Prediction

Erik Schwarz

RWTH Aachen University

August 25, 2025

Motivation

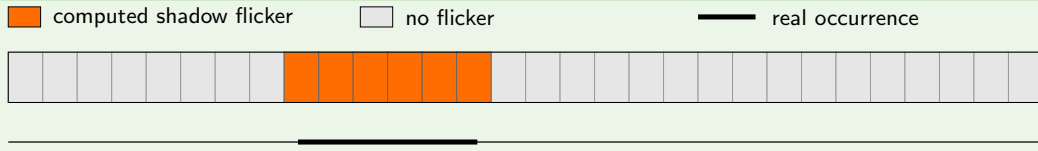
- ▶ Europe expands wind energy (285 GW today, +187 GW expected by 2030).
- ▶ EU target: 42.5% of the energy should come from green sources.
- ▶ 425 GW of wind capacity needed to fulfill the goal
- ▶ Turbines up to 15 MW, growing in size and number.
- ▶ Planning challenge: maximize energy while reducing impact on people and nature.
- ▶ Germany regulates limits: 30 min/day, 30 h/year.

Current Industry Standard

Methodology

- ▶ Compute Sun's position for certain time steps
- ▶ Determine whether shadow flicker occurs at that time step
- ▶ Introduces discretization error
- ▶ Tradeoff: precision vs. runtime

Discrete Error



Introduction of the Original Approach

A Continuous Model for Shadow Flicker Computation

Continuous Approach

- ▶ Goal: compute resolution-independent results
- ▶ Idea: model shadow flicker as a continuous function
- ▶ Advantages: avoids discretization and lower computational effort
- ▶ Solve the function using numerical models
- ▶ The original approach is structured into four parts:
 - ▶ Sun Position Algorithm
 - ▶ Ray Tracing Algorithm
 - ▶ Distance Function
 - ▶ SF Computation

Sun Position Algorithm

- ▶ Goal: compute position of the sun at a given time step t
- ▶ Use of SPA by Meeus, calculates the position with uncertainties as low as 0.003°
- ▶ Requires input parameters to function:
 - ▶ Latitude and longitude (degrees)
 - ▶ Timezone (hours)
 - ▶ Elevation (meters)
 - ▶ Pressure (millibars)
 - ▶ Temperature ($^\circ\text{C}$)
 - ▶ Atmospheric refraction (degrees)
 - ▶ Delta_ut1 (seconds)
 - ▶ Delta_t (seconds)
 - ▶ Year
- ▶ Note that delta_ut1 accounts for fluctuations in Earth's rotation, and delta_t deals with the difference between terrestrial time (TT) and universal time (UT1).

Sun Position Algorithm

- ▶ Position of the Sun is described using two angles
- ▶ Elevation angle $\theta(t)$
 - ▶ $\theta(t) \in [0^\circ, 90^\circ]$
- ▶ Azimuth angle $\psi(t)$
 - ▶ $\psi(t) \in [0^\circ, 360^\circ]$

Ray Tracing Algorithm

- Idea: trace a line from the receiver to the sun and see whether it intersects with the plane

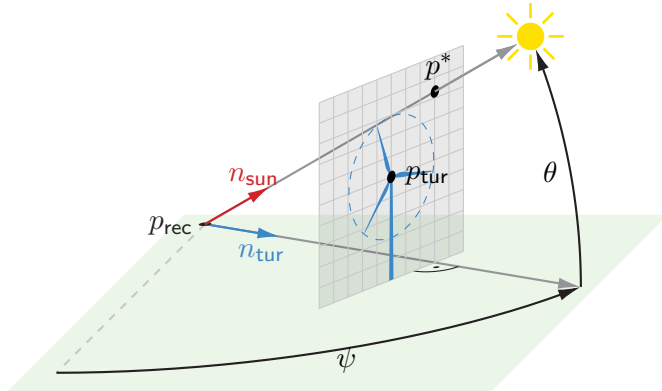


Figure: Illustration of the ray-tracing algorithm

Ray Tracing - special case

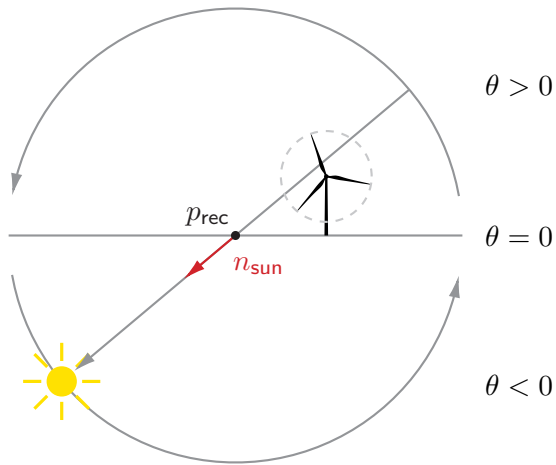


Figure: Special case where $dist_{p_{rec}}(\psi, \theta) < r_{tur}$ and $\theta(t) < 0$

Distance Function

define $\text{dist}_{p_{\text{rec}}}(\psi(t), \theta(t))$ such that it is robust against the described special case

$$\text{dist}_{p_{\text{rec}}}^+(t) = \begin{cases} \text{dist}_{p_{\text{rec}}}(\psi(t), \theta(t)) - r_{\text{tur}} & \theta(t) \geq 0 \\ \text{dist}_{p_{\text{rec}}}(\psi(t), 0) - r_{\text{tur}} & \text{otherwise} \end{cases} \quad (1)$$

- ▶ Shadow Flicker occurs if and only if there exists a $t \in d$ such that $\text{dist}_{p_{\text{rec}}}^*(t) < 0$
- ▶ $\text{dist}_{p_{\text{rec}}}(\psi(t), \theta(t))$ is continuous

Shadow Flicker Computation

We characterize shadow flicker by an indicator function $\mathbb{1}_{\text{SF}}(t)$, which evaluates to 1 when the conditions for shadow flicker are satisfied at time t , and 0 otherwise.

$$\mathbb{1}_{\text{SF}}(t) = \begin{cases} 1 & 0^\circ < \theta(t) < 90^\circ \wedge \text{dist}_{\text{prec}}^+(t) \leq 0 \\ 0 & \text{otherwise} \end{cases} \quad (2)$$

(Cases where $\theta \in \{-90^\circ, 90^\circ\}$ are excluded)

We calculate the shadow flicker duration T for a day $d = [t_{\text{start}}, t_{\text{end}}]$ using

$$T = \int_{t_{\text{start}}}^{t_{\text{end}}} \mathbb{1}_{\text{SF}}(t) \, dt \quad (3)$$

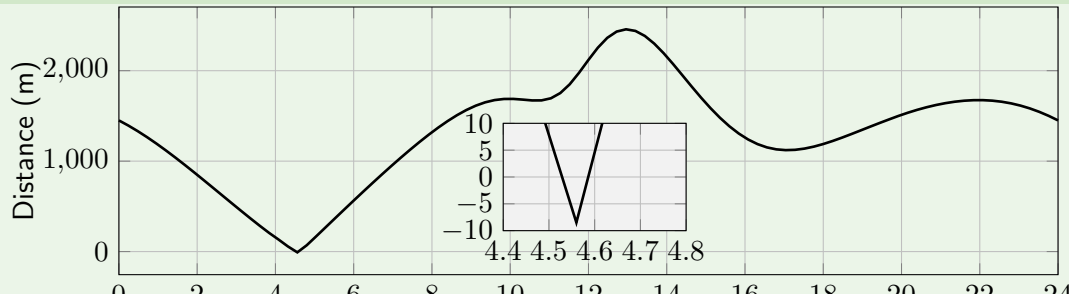
Note that this equation cannot be solved analytically using standard integrable models.

Minimization Subroutine

for a day d we compute the minimum in the following way:

1. Minimize over the whole day (use 12 as an initial guess for Brent's method) to find t_{min}
2. Minimize again in case we did not find the right minimum:
 $t_{min} < 12 \rightarrow$ search in the evening
 $t_{min} > 12 \rightarrow$ search in the morning

Example trajectory of the shadow finder



Root Finding Subroutine

Assume a t_{min} with $dist_{prec}^+(t_{min}) < 0$ is found using the minimization subroutine

- ▶ as we abstracted the rotor radius, roots are start and endpoint of the shadow flicker occurrence
- ▶ goal: compute roots of the function
- ▶ idea: consider absolute value of $dist_{prec}^+(t)$:
 - ▶ start and endpoint of the shadow flicker occurrence are now minima
 - ▶ compute $start = brent[t_{min} - 2, t_{min} - 0.1, t_{min}]$ for the start of the interval
 - ▶ compute $end = brent[t_{min}, t_{min} + 0.1, t_{min} + 2]$ for the end of the interval
 - ▶ return $end - start$

Assumptions

Meteorological Shadow Flicker

- ▶ Sun always shines
- ▶ No weather effects are considered: turbine always operates, sun is always visible
- ▶ Turbine always faces the sun

By this we overestimate the actual shadow flicker occurrence.

In addition we also assume that shadow flicker occurs once per day and that we reliably find the minimum of our continuous function.

Rare Double Flicker Events Exist

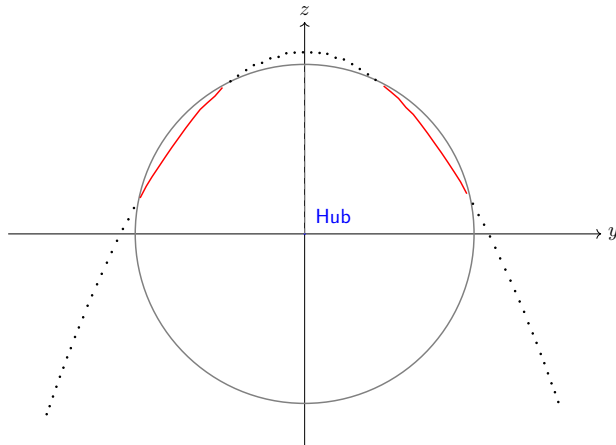


Figure: Simplified showcase of the projected solar path

Performance Improvements

to the Original Approach

Case Study

Across the thesis we employ a single case study

- ▶ Wind turbine placed at Aachen ($50.7753^{\circ}N, 6.0839^{\circ}E$)
- ▶ Height of $100m$ and rotor radius of $80m$
- ▶ Turbine is placed in the center of a $3000m$ by $3000m$ area
- ▶ Receivers are placed with a step size of $50m$
- ▶ Ground height is $0m$
- ▶ Year of interest is 2025

We are going to use this case study to validate that each implemented feature is working as intended if we produce the same result as the original approach

Technical Improvements to the Original Approach

time interval

- ▶ Implementation of the original approach is based on calculating the duration for a day
- ▶ Goal: compute and store exact `time_interval`
- ▶ Instead of returning the duration, we return the `time_interval shadow`
Compute the duration using `shadow.start` and `shadow.end`

Technical Improvements to the Original Approach (Cont.)

More Robust Interval Estimation

Assuming that we want to use the Root finding subroutine.

Root-finding happens in a static interval such as

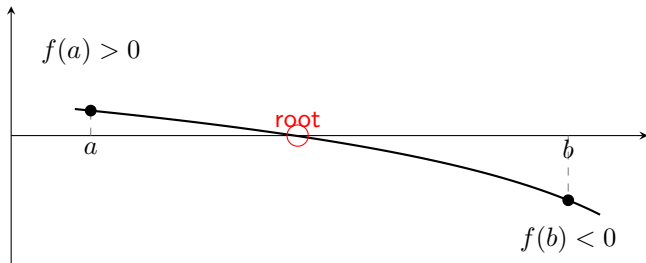
`start = brent[ $t_{\min} - 2$ ,  $t_{\min} - 0.1$ ,  $t_{\min}$ ]`

In rare cases (close to the wind turbine) this duration might not be enough.

Solution: verify that sign change happens between interval borders.

In case the sign change does not happen we extend the interval by *30min* till the sign change is detected.

Brent's Method:



Requirements

1. executable function $f(x)$
2. continuous function $f(x)$
3. bracketed interval $[a, b]$ such that $f(a) \cdot f(b) < 0$

Root finding

Current method: Minimize twice over the absolute value function

Replace Root finding subroutine by a more robust and efficient one

We choose to continue using Brent's method but use a correct implementation for root finding.

We also increased the precision from 0.05 to 10^{-7}

Reusing the Prior Day's Guess

Sun's path changes gradually from day to day.

Idea: store the computed minimum of the last day $t_{last_day_min}$.

In case $dist_{prec}^+(t_{last_day_min}) < 0$ we can skip the whole minimization subroutine and directly compute the roots.

```
1  typedef struct {  
2      bool had_sf_yesterday;    // 1 → shadow flicker yesterd  
3      double last_day_guess;    // time of minimum from previo  
4      double last_day_guess_value; // function value at last_da  
5  } sf_initial_guess;
```

Listing 1: Structure for initial guesses

Reusing the Midpoint of Prior Day's Interval

Instead of using the prior day's guess, we could instead use the midpoint of the shadow flicker interval which occurred yesterday.

$$t_{midpoint} = \frac{t_{start} + t_{end}}{2}$$

Note that we can only use this feature if shadow flicker occurs on the prior's day.

```
1  typedef struct {  
2      bool had_sf_yesterday;  
3      double last_day_guess;  
4      double last_day_guess_value;  
5      double midpoint_guess;  
6      double midpoint_guess_value;  
7  } sf_initial_guess;
```

Listing 2: Extended structure for initial guesses

Early Termination of Minimization Subroutine

To compute the root's of a function, it is sufficient to provide a bracketed interval $[a, b]$ where $f(a) \cdot f(b) < 0$.

We can stop minimizing once we find a value such that $dist_{p_{rec}}^+(t) < 0$ as we then can compute the roots already.

```
1      /* This is the one function evaluation per iteration. */
2      fu = (*f)(u, params);
3      if (fu < 0.0) {
4          *xmin = u;
5          return fu;
6      }
```

Listing 3: Check for negative function value in Brent iteration

Reducing Computational Effort

Goal: find a heuristic that minimizes this number of receivers that need to be evaluated.

Propagation

We assume that the receivers are placed in a grid-like structure (Case Study: $61 \cdot 61$ receivers)

- ▶ Initialization
 - ▶ Start at the turbine center and evaluate shadow flicker along the x - and y -axes.
 - ▶ If a grid point has nonzero duration, use it as the start point.
 - ▶ If no grid point experiences shadow flicker, terminate the search (no flicker that day).
- ▶ Propagation
 - ▶ Begin with a grid point with nonzero duration.
 - ▶ Use a queue: enqueue every neighbor with nonzero shadow flicker.
 - ▶ Avoid duplicate work: track grid points already enqueued or computed.

Reducing Computational Effort (Cont.)

Propagation: Optimizations

- ▶ Avoid daily re-initialization:
 - ▶ Store the index of the grid point with the longest flicker duration.
 - ▶ Use it as the initial guess for the next day.
- ▶ Integrate optimizations from the original approach:
 - ▶ Duration is now computed for all receivers day by day.
 - ▶ Allocate an instance of the `sf_initial_guess` struct for each receiver.
 - ▶ Increases runtime memory consumption compared to the original approach.

Average Runtime per Method

We implemented each feature independently and evaluated each config for the respective location 3 times and computed the average runtime.

Method	Aachen	Arizona	Tokyo
Original	174.56	188.07	191.57
Accurate Root Finding	170.45	184.94	190.76
Last Day Guess	158.46	165.68	176.15
Midpoint Guess	165.79	180.27	185.32
Early Term Min.	171.66	184.19	190.85
Propagation	16.98	12.21	12.05
All Features (no Prop)	153.27	164.55	170.38

Number of Function Evaluations

We computed the following metrics for the original implementation and the features that includes all features

Metric	Original Approach	Optimized Approach
Brent minimizer calls	2,669,836	2,582,880
Function evals (minimizer)	31,874,089	30,421,318
Brent root-finder calls	136,678	136,678
Function evals (root)	1,901,957	1,113,131
Last Day Guess succeeded	–	63,746
Midpoint Guess succeeded	–	2,082

Table: Comparison of performance metrics

Memory Usage Optimization

to the Original Approach

Memory Usage Optimization to the Implementation

We compute the following results using the original approach:

Field	Type	Size
maximum_daily	double	8 bytes
annual	double	8 bytes
intervals[366]	time_interval ($366 \times (8 + 8)$)	5856 bytes
interval_count	int	4 bytes

Table: Results are stored in a struct called *sf_results*

Memory Usage Optimization to the Implementation (Cont.)

It is inefficient to allocate the whole array for every receiver.

Case study: 18.37 shadow flicker occurrences per receiver.

Only 5% of the allocated memory for the daily intervals is actually used to store nonzero values.

Sparse Representation

As we have a majority of zero durations, we store only the nonzero durations and the index.

0, 0, 0, 0, 4, 0, 0, 0, 0, 0, 5

would translate to

(4, 4), (10, 5)

Linked List

An example for a dynamic data structure is a Linked List.

We create an entry for a day only if the receiver is exposed to shadow flicker. To traverse the list we store the head and to append another element we store the tail. The list is only traversed to compute the annual duration or to write the output.

Since we do not need random access, the list is efficient in our case.

We reduce the memory by 90% using a Linked List

- ▶ annual: double (8 bytes)
- ▶ interval count: int (4 bytes)
- ▶ maximum daily: double (8 bytes)
- ▶ $19 \times (\text{pointer} + \text{interval}): 19 \times (8 + 16) = 456$ bytes
- ▶ Head and tail pointers: $2 \times 8 = 16$ bytes
- ▶ Total: 480 bytes per receiver / 1.8 megabytes (for the whole case study)

Multiple Turbine Calculation

based on the Original Approach

Multiple Turbine Calculation

- ▶ For one receiver r and n turbines $t_1, \dots, t_n$:
 - ▶ Compute shadow flicker intervals $I_1, \dots, I_n$ (start–end times).
 - ▶ Intervals from different turbines may overlap.
- ▶ To compute the total shadow flicker exposure:
 1. Sort all intervals by start time ($\mathcal{O}(n \log n)$).
 2. Merge overlapping intervals ($\mathcal{O}(n)$).
- ▶ Result: merged, pairwise-disjoint intervals $\tilde{I}_1, \dots, \tilde{I}_m$.

$$D(d, r) = \sum_{j=1}^m \left(\tilde{t}_{\text{end}}^{(j)} - \tilde{t}_{\text{start}}^{(j)} \right)$$

Total shadow flicker duration for receiver r on day d

Coordinate Transformation

- ▶ Geodetic (GPS): latitude ϕ , longitude λ
- ▶ ECEF: convert to Earth-Centered, Earth-Fixed (X, Y, Z): origin at Earth's center of mass, axes in meters
- ▶ ENU (East–North–Up): choose reference point, express all points relative to it
- ▶ For shadow flicker: use only East–North coordinates: custom 2D meter-based map

Precomputation

Wind Farm Layout Optimization

- ▶ Real-world projects: WiFO, TOPFARM, WindPRO
- ▶ Shadow flicker exposure must respect daily and annual limits
- ▶ Exceeding limits means that turbines shut down temporarily
- ▶ Results in suboptimal productivity if not considered
- ▶ Complexity grows with:
 - ▶ Number of turbines
 - ▶ Landscape constraints
 - ▶ Variables: turbine height, rotor diameter, etc.
- ▶ Optimal layouts require evaluating many iterations

Precomputing Shadow Maps

- ▶ Efficient computation: precompute shadow maps for a turbine type
- ▶ Assumption: moving the turbine up to $20km$ away from the Position which we choose for the SPA
- ▶ Shadow maps computed once, then reused via lookup
- ▶ Problem statement:
 - ▶ Given SPA parameters, turbine config, shadow-map config
 - ▶ Precompute maps for receiver height layers
 - ▶ For each receiver: use nearest grid point
 - ▶ Interpolate if height does not match a precomputed layer

Methodology

- ▶ Program split into two parts, each with a config file
- ▶ Part 1: Precompute shadow maps
 - ▶ Central point chosen for SPA
 - ▶ Turbine height, rotor radius, map dimensions
 - ▶ Parameters: $x_{\min}$, $x_{\max}$, $y_{\min}$, $y_{\max}$, step size, number of height layers, min/max height
 - ▶ Result: array of daily intervals + annual durations
- ▶ Part 2: Evaluate layouts
 - ▶ Config file with turbine and receiver geographic positions
 - ▶ Calculate relative positions
 - ▶ Map receivers to nearest grid points
 - ▶ Interpolate across heights if needed

Interpolation

- ▶ Receiver heights not always known exactly
- ▶ Spline interpolation fits smooth piecewise polynomials through data points with continuous derivatives.
- ▶ Special case: interpolating intervals
 - ▶ Interpolate $t_{\text{start}}, t_{\text{end}}$ individually
 - ▶ Provides daily intervals, not just durations
 - ▶ Must avoid mixing non-zero with zero intervals

Conclusion

- ▶ First we applied various optimization strategies to reduce the computational effort
 - ▶ Replaced root-finding algorithm
 - ▶ Stopping minimization once a negative function value is found
 - ▶ Predicting shadow flicker intervals based on data of the previous day
 - ▶ Propagation search
- ▶ Reduced Memory Consumption
- ▶ Support the calculation between multiple turbines
- ▶ Precomputation using shadow maps

Comparison

