Approximation of the view factor for thermal simulations of satellites using binary classifiers

Hussein Albazzal, Florent Noisette, Jules Vanaret, Wanqing Wang

SEME : PROJET DOREA, APPROXIMATION OF THE VIEW FACTOR FOR THERMAL SIMULATIONS OF SATELLITES USING BINARY CLASSIFIERS

HUSSEIN ALBAZZAL[1], FLORENT NOISETTE[2], JULES VANARET[3], WANQING WANG[4]

1: houssein.bazzal@hotmail.com
2: florent.noisette@math.u-bordeaux.fr
3: jules.vanaret@univ-amu.fr
4: wanqing.wang@polytechnique.edu

1. Introduction

1.1. **Definition of the View/Form Factor.** The goal of DOREA is to compute heat transfer in satellites. In order to do that, one of the main tool is to compute the view factors. Given two infinitesimal surfaces dS_1 and dS_2, with normal vectors $\overrightarrow{n_1}$ and $\overrightarrow{n_2}$, denoting by $\overrightarrow{r}$ the vector from dS_1 to dS_2, the *infinitesimal view factor* $dF_{1\to 2}$ from dS_1 to dS_2 is

$$dF_{1\to 2} := \frac{\cos(\theta_1)\cos(\theta_2)}{\pi r^2} dS_2, \tag{1}$$

where θ_1 is the angle between $\overrightarrow{r}$ and $\overrightarrow{n_1}$ and θ_2 is the angle between $\overrightarrow{r}$ and $\overrightarrow{n_2}$.

Now the *view factor* between two general surfaces is computed by integrating this formula over S_2 and taking the mean value over S_1 :

$$F_{1\to 2} := \frac{1}{\pi |S_1|} \int_{S_1} \int_{S_2} \frac{\cos(\theta_1)\cos(\theta_2)}{r^2} ds_1 ds_2. \tag{2}$$

This formulation is called the Double Integral formulation of the view factor. There exists other formulations for it. For example, if we denote by $\mathcal{C}_1 := \partial S_1$ and $\mathcal{C}_2 := \partial S_2$ the boundaries of S_1 and S_2 respectively, using Stokes theorem, we get that

$$F_{1\to 2} = \frac{1}{\pi |S_1|} \oint_{\mathcal{C}_1} \oint_{\mathcal{C}_2} \ln(r)\, d\overrightarrow{\tau_2} d\overrightarrow{\tau_1}. \tag{3}$$

In the particular case of the view factor between triangles, or more generally between any polygons, we can deduce from (3) a closed formula for the View Factor (see [2] and [1] respectively for more details). By this we mean that instead of having to compute an integral, we only have to compute a sum with index the vertices of the polygons in question.

However, in the case of interest, the view between two faces of a satellite can be obstructed by another part of said satellite, called *obstacle* in what follows. With that in mind, we need to take into account the potential obstacles and their geometry to compute view factors.

Date: November 27, 2023.

1.2. **Method employed by DOREA.** In his PhD thesis [3], Vincent Vadez tackled the problem of finding efficient ways to compute View Factors in the case of partial occlusions. Among the several optimization strategies he proposed, we will describe in this paragraph the ones that are used currently on real use cases. In Section 2, we detail possible improvements to these methods.

The core idea of their approach is to perform a splitting by recursion of the receiving triangle. For a given emitting triangle S_1 and a given receiving triangle S_2, they first check if the receiving triangle is fully visible (resp. fully overshadowed) from the emitting triangle. If it is, then the closed formula discussed in [2] (resp. zero) returns the desired result of view factor. If it is not, then they split the receiving triangle into sub-triangles, and repeat this process until each sub-triangle satisfies the return condition (fully visible or fully overshadowed), see Figure 1 or until the number of iteration exceeds the maximum depth of recursion.

If the maximum depth of recursion is exceeded, then they use an approximation of the view factor instead of the exact formula for the sub-triangle which do not respect the return condition. The method to compute this approximate value of the view factor for small triangles relies on the hypothesis that those triangles are small enough. Under this hypothesis, the infinitesimal view factor is almost a constant (i.e θ_1 and θ_2 from equation (1) do not vary much across the triangle). Therefore the view factor can be approximated by its value in the case in which there is no obstacle, multiplied by the percentage of rays that reach the receiving triangle. In practice, rays are emitted from and to points that are sampled on the emitting and receiving triangle by using an efficient uniform sampling technique called Bounded Centroidal Voronoi (BCV) sampling. This guarantees that points are optimally chosen (under no additional prior information on the positions of obstacles).

In order to have a good polygon splitting for penumbra, Vadez (see 1.3) suggested using advanced computational tools to determine to the boundary of the shadow during recursion. The idea behind that is to try to stop the recursion quicker, while still having enough information to get a precise approximation of the view factor.

1.3. **Linear Support Vector Machines.** Even in the case in which the shadow boundary is close to a straight line, a wide penumbra region considerably complicates the accurate computation of the view factor. Vadez [3] considered using a Linear Support Vector Machine classifier (linSVM) to learn the boundary and provide an optimal single cut candidate to split the receiving triangle.

LinSVM is a *supervised classification algorithm*, meaning that given a set of points and a set of labels associated to those points $((x_1, y_1), \ldots, (x_n, y_n)) \in (X \times Y)^n$, they learn a decision function $f : x \in X \mapsto y \in Y$ through a training procedure. This decision function predicts/approximates the real label y^* associated to any x in X (that is, the training ends up minimizing $|y^*(x) - f(x)|$). To do so, the algorithm learns the optimal hyperplane separating the given classes in the feature space X. In the case of binary classification, the decision function traditionally maps one class to the value -1, and the other to 1.

In the implementation by Vadez, several rays are sent between points sampled on the emitter and the receiver, and the information of whether the rays intersect an obstacle or not

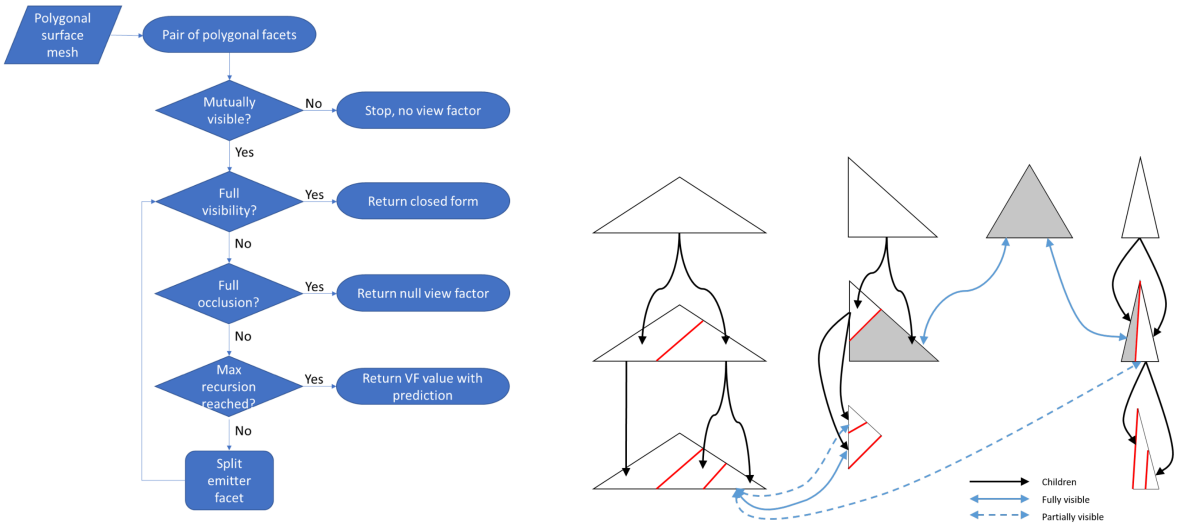


FIGURE 1. **(a)** Flow chart of the method used in Vadez [3](Figure 3.3) for the computation of the view factors. Most notably, the receiving triangle is recursively split near the penumbra region until the infinitesimal view factor can be approximated as constant over the surface of the receiving triangle. **(b)** Typical use cases of the splitting procedure when partial occlusion occurs.

is stored. This information is then used as ground-truth to train a linSVM classifier, with the points on the receiver as features. The linSVM provides the optimal hyperplane (here a line) approximating the boundary between shadow and light. Vadez provided an implementation in this simple case, and concluded that the method was promising, but potentially slow.

In our case, the penumbra area can be seen as a mix of both classes (light and dark), and thus the two classes are strictly speaking not linearly separable. The method can still be applied at the cost of specifying the value of a parameter C that balances a trade-off between robustness to outliers and accuracy. We argue that it should be possible to optimize this parameter once and for all by defining a training set composed of many pairs of randomly oriented triangles with known linear shadow boundary. An unbiased optimal value of the parameter can be obtained using a classical optimization procedure like cross-validation (see Appendix A).

In order to integrate this method in the code for real use cases, it needs to be able to generalize to arbitrary boundaries (as opposed to linear boundaries considered until now). A strategy to do so would be to split the initial receiving triangle into smaller sub-triangles until the boundary can be considered locally linear on the sub-triangles close to the penumbra. The sub-triangles can be obtained recursively by training a separate instance of a linSVM classifier at each recursion step to check whether or not the boundary is well approximated locally (this would manifest as a small proportion of outliers in the classification), at which point the recursion branch is stopped. However, this procedure is expected to be particularly slow because it requires to train a separate SVM classifier for each sub-triangle of the splitting hierarchy.

In Section 2, we propose two non-linear methods that can learn any shadow boundaries up to an arbitrary level of precision, while potentially minimizing the extent of the recursion procedure.

2. Our approach/hints of solutions

In paragraphs 2.1 and 2.2, we describe two algorithms, called respectively Kernel SVM classifier and Decision Tree classifier.

In the strategies that we propose, both classification approaches aim at estimating the shadow boundary up to an arbitrary level of precision. This could allow us to speed up the computation of the view factor by providing an optimal splitting, or even to circumvent entirely the need to use the triangle splitting recursion.

2.1. **Kernel SVM classifier.** Kernel SVM (kSVM) is an extension of linear SVM [5] that allows for the classification of non-linearly separable data by mapping it into a higher-dimensional space through a transformation function that defines a kernel function

$$k : (x_i, x_j) \in X^2 \mapsto k(x_i, x_j) \in \mathbb{R}.$$

Performing a dot product between two vectors in the transformed space is equivalent to applying the kernel on said vectors. Basically, Kernel SVM acts like linear SVM, but on the transformed space: it finds a linear hyperplane that separates the data in the transformed space.

The choice among the possible kernel functions provides robustness and flexibility to this method. Most notably the Gaussian kernel, also called *Radial Basis Functions* (RBF),

$$k_{RBF}(x_i, x_j) := e^{-\frac{|x_i - x_j|^2}{2\sigma^2}}, \tag{4}$$

is a popular choice due to the interpretability of its parametrization: the parameter σ can be seen as a "typical length scale" at which points close to the boundaries are sensitive to their neighbors. For example, the case $\sigma \to +\infty$ corresponds to linear SVM (each point is equally sensible to any other point). However, kSVMs are more computationally intensive to train than their linear counterpart, and they do not provide a closed form equation for the boundary.

Remark: an efficient implementation of kernel SVM is available in the C++ library libSVM.

2.2. **Decision Tree classifier.** *Decision Tree* is another non-linear supervised classifier algorithm. It builds its decision function by recursively splitting the dataset based on the feature that provides the best separation according to a chosen criterion (e.g. minimizing Gini impurity for probability of missclassification, or maximize entropy for information gain). The process continues until a stopping criterion is met (e.g., maximum depth, minimum samples per leaf). The final nodes are called leaf nodes and represent the predicted class. To make predictions, a new data point traverses the tree from the root to a leaf, following the decision criteria at each node. The majority class in the leaf is assigned as the predicted class.

The parameters to tune in order to use this method are the tree's *maximum allowed depth* and the minimum number of samples that can be considered a leaf. Similarly to Paragraph

2.1, we propose to optimize those parameters once and for all on a dataset composed of randomly oriented triangles with known shadow boundary using cross-validation.

Remark: an efficient implementation of Decision Tree classifiers is available in the C++ library ROOT-TMVA.

2.3. **Optimization strategies.**

2.3.1. *Optimization using kernel SVM.* To address the problem of the computational cost, our strategy consists in training the kSVM only once, at the level of the initial emitting and receiving triangles. Training points and labels are obtained by sending a relatively small amount of rays on the receiving triangle and computing whether or not they reach the receiver. This minimizes training time and circumvents entirely the use of the splitting recursion. Once the training is done, because we lack a closed form for the boundary, we propose to sample a large amounts of points on the receiving triangle using the Bounded Centroidal Voronoi sampling scheme, and to classify these points using the trained kSVM, which is computationally cheap even for a large amount of points. Afterwards, either the point-based quadrature formula or a coarser approximation using the visibility ratio (ratio of points in the light over the total number of points sampled) can be used to compute the view factor.

This strategy can in fact also be used with the Decision Tree classifier, especially since it is particularly faster to infer with (see the Results section below). We provide a pseudo-code implementation of the method using kernel SVM below, and a schematics of the idea (Fig. 2):

Inputs:
Emitter and receiver triangles T_i and T_j
Typical "length" scale σ
Regularization parameter C
VF approximation method: "quadrature" or "visibility ratio"
if Fully visible **then**
  VF given by closed form
else if Full occlusion **then**
  VF = 0
else
  Sample n points $X^i = \{x_1^i, ..., x_n^i\}$ on T_i using the BCV sampler
  Sample n points $X^j = \{x_1^j, ..., x_n^j\}$ on T_j using the BCV sampler
  for all Pair $\left(x_p^i, x_q^j\right)_{p,q}$ **do**
    Store if the ray $x_p^i \to x_q^j$ intersects with obstacles:
      $y_{p,q} := -1$ if intersection, else 1
  end for
  Collect labels $Y = \{y_{1,1}, ..., y_{n,n}\}$
  Train classifier using points X^j and labels Y:
    $f = SVM(X^j, Y, \sigma, C)$
  Sample N points $\mathcal{X}^j = \{\chi_1^j, ..., \chi_N^j\}$ on T_j using the BCV sampler
  Determine for each point in $\mathcal{X}^j$ if it lies in the shadow or not using the classifier:
    $y_k = f(\chi_k^j), \chi_k^j \in \mathcal{X}^j$

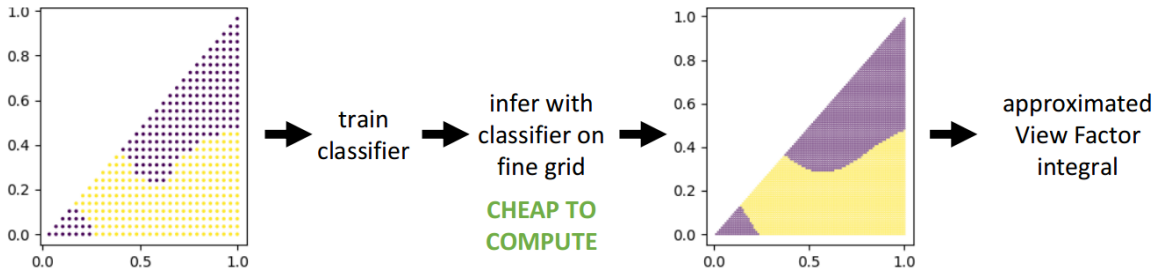


FIGURE 2. First strategy relying on training on few rays, and inferring on a fine grid to compute the View Factor integral.

Compute subset of point in the light $\bar{\mathcal{X}}^j := \{\chi_k^j \in \mathcal{X}^j, f(\chi_k^j) = 1\}$
if quadrature method **then**
  Compute VF using the point-based quadrature method on subset $\bar{\mathcal{X}}^j$
else if view ratio method **then**
  Compute view ratio $r := \frac{\#(\bar{\mathcal{X}}^j)}{N}$
  Compute closed form VF and multiply by view ratio
end if
end if

2.3.2. *Optimization using the Decision Tree classifier.* The splitting procedure can be seen as choosing an optimal linear constraint over the features (e.g "$2x + 3y \leq 5$", which defines a 2D half-plane), see more in [6, 7]. As a result, the learnt boundary is naturally polygonal (or at least several polygonal regions are enclosed). We thus propose to train a Decision Tree classifier only once in the same spirit as in Paragraph 1.3. That is training points and labels are obtained by sending a relatively small amount of rays on the receiving triangle and computing whether or not they reach the receiver. Once the Decision Tree is learnt, the splitting constraints can be used to extract the position of the vertices of the boundary (as demonstrated in the Python scripts provided). This allows a direct use of Schröder's View Factor formula for polygons [1], without the need for any additional computations. To extract the boundary polygon, one must mix the hierarchical constraints learnt by the Decision Tree (which can produce infinite regions with the constraint that the boundary is confined to the inside of the triangle. This could be solved as a Linear Programming problem using the Simplex method. A potential algorithm would imply starting with the vertices of the triangle and iteratively progressing through the tree and adding constraints one by one to rebuild the vertices.

We provide a pseudo-code implementation of the method below, and a schematics of the idea (Fig. 3):

Inputs:
Emitter and receiver triangles T_i and T_j
Maximum allowed depth max_depth
Minimum number of nodes that can be considered a leaf min_samples_leaf
if Fully visible **then**
  VF given by closed form

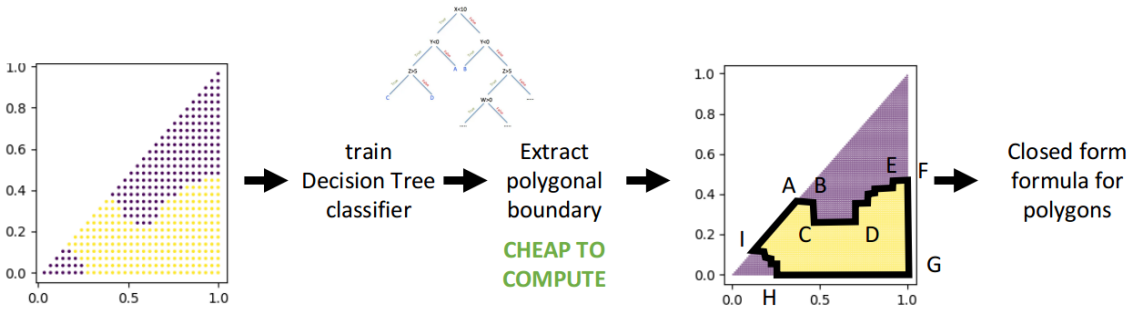


FIGURE 3. Second strategy relying on training a Decision Tree on few rays, and extracting the polygon of constraints.

else if Full occlusion **then**
  VF $= 0$
else
  Sample n points $X^i = \{x_1^i, ..., x_n^i\}$ on T_i using the BCV sampler
  Sample n points $X^j = \{x_1^j, ..., x_n^j\}$ on T_j using the BCV sampler
  for all Pair $\left(x_p^i, x_q^j\right)_{p,q}$ **do**
    Store if the ray $x_p^i \to x_q^j$ intersects with obstacles, $y_{p,q} := -1$ if intersection, else 1
  end for
  Collect labels $Y = \{y_{1,1}, ..., y_{n,n}\}$
  Train classifier using points X^j and labels Y:
    $f = DecisionTree(X^j, Y, \text{max\_depth}, \text{min\_samples\_leaf})$
  Extract splitting constraints and compute polygonal boundary
  Compute VF using Schröder's formula for polygons
end if

2.4. **Quantitative analysis of the proposed strategies.** We now discuss the advantages of the algorithm detailed in Paragraphs 2.1 and 2.2.

We wish to compare the theoretical time needed with the initial algorithm and with the one we propose now. To do that we will compare the number of call to the costly functions (AABB full occlusion, AABB full light, sending a ray and testing the occlusion for this ray). To address the problem of assessing general performances, each metric computed is averaged over a large amount of randomly generated shadows, as seen on see Fig. 5. To build a shadow, we sample a grid of points on a triangle, and assign each point a random value between 0 and 1. Then, this image is blurred using a Gaussian filter, and finally it is thresholded (see implementation details in the Python scripts).

First, we plot the computation time (training and inference) of the different classifiers as a function of the number of rays (in the case of training time) or grid points (in the case of inference) averaged for $N_s = 100$ random arbitrary shadow. We observe (Fig. 7) a noticeably smaller training time with the Decision Tree, whereas both SVM methods are comparable on that matter. Interestingly, the inference time with Decision Tree is minuscule compared to the time for the two SVM methods, making it a good candidate for both our first and second

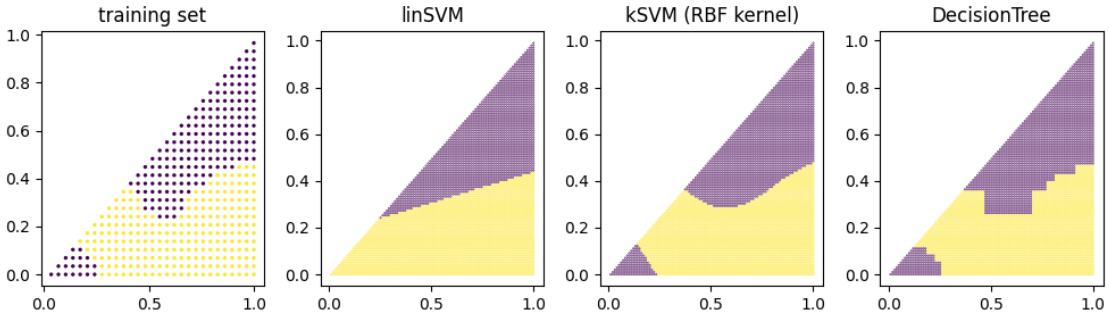


FIGURE 4. Left, training set used to train the classifiers. The other plots represent the classification result of three classifiers (left linear SVM, middle kernel SVM, and right Decision Tree) over a finer grid. All classifiers are used with their default parameters as defined in the Python library Scikit-learn. The Decision Tree has parameter "max_depth" set to 5 to deliberately show a rough polygonal estimation.

optimization strategies.

As for the time related to sending the probing rays used in the training set, in the code in use today, the recursion depth is typically set to 5. This means that the initial triangles are both divided into up to $4^5 = 1024$ triangles (if the penumbra occupies up to 10% of the surface of the receiver, then we have at least 100 triangles). Then the prediction formula is obtained for each of this triangle, this formula requires to send between 10 and a 100 ray per triangle (depending of the wished precision). In total, one can reasonably expect 1000 to 100000 rays (depending on the complexity of the shadow and the wanted precision) to be sent in order to compute the view factor. Moreover, this method also requires to do the full visibility and full occlusion tests, which can be quite costly as well.

In comparison, for the algorithm without recursion using decision algorithm, a shadow reconstruction accuracy of 90% can be reached with a few hundreds of rays (see Fig 6). We also obtain the percentage of rays being blocked for free using the decision function. This means that we can skip sending the rays for the prediction step. We also don't need to do full occlusion and full enlightenment test at each point of the recursion. However, there is the additional cost of having to train and use the decision algorithm. Moreover for the case of kernel SVM, if one wants to obtain the classification probabilities (which allow the estimation of the continuous gradient of luminosity in the penumbra), the training and inference time is longer.

Interestingly, for kernel SVM it is also possible to access the learnt probability that a given point belongs to each of the two class. This usually manifests as a smooth gradient between the area of full occlusion and the area of full luminosity. We argue that this could be related to the gradient of luminosity due to the penumbra phenomenon (Fig. 8). By computing the integral of this probability distribution profile over the surface of the triangle, one can obtain an accurate estimation of the view ratio fully taking the penumbra phenomenon into account.

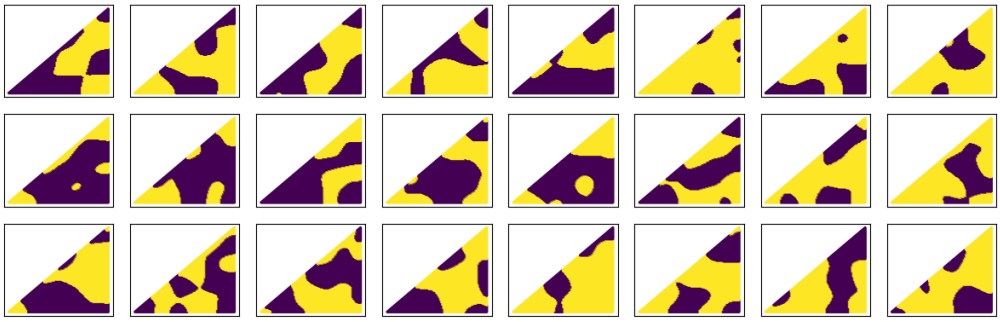


FIGURE 5. Some examples of random shadows simulated to average performances.

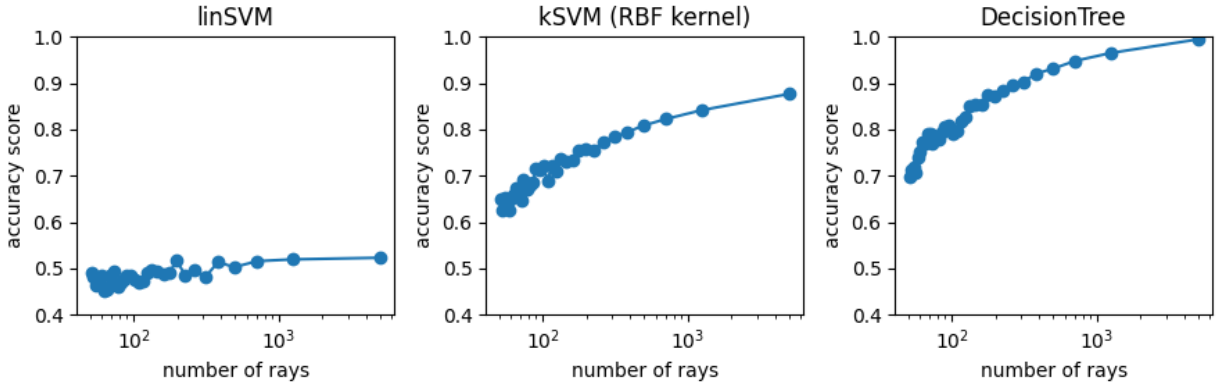
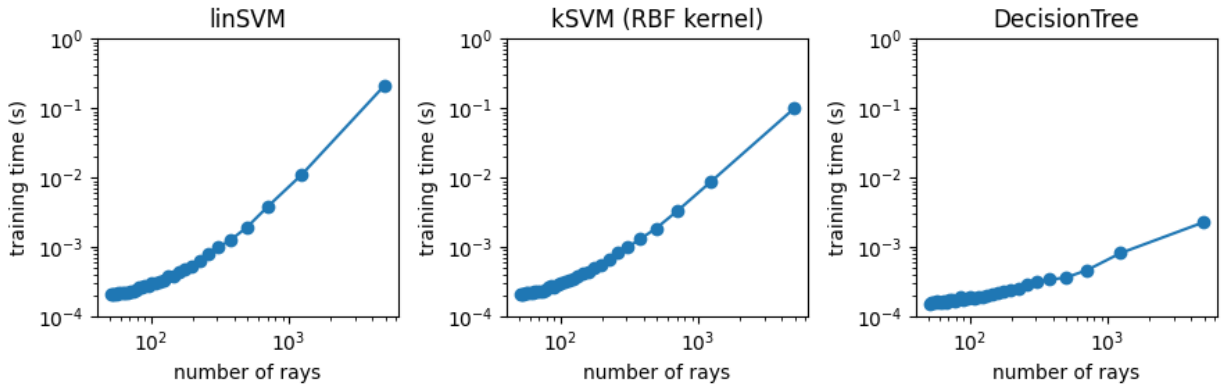


FIGURE 6. Comparison of the average classification accuracy (quality of approximation of the boundary) of the three classifiers over $N_s = 100$ random shadow profiles for different number of rays used to make the training set. Default parameters are used for the classifiers, and the Decision Tree has parameter "max_depth" set to 10.

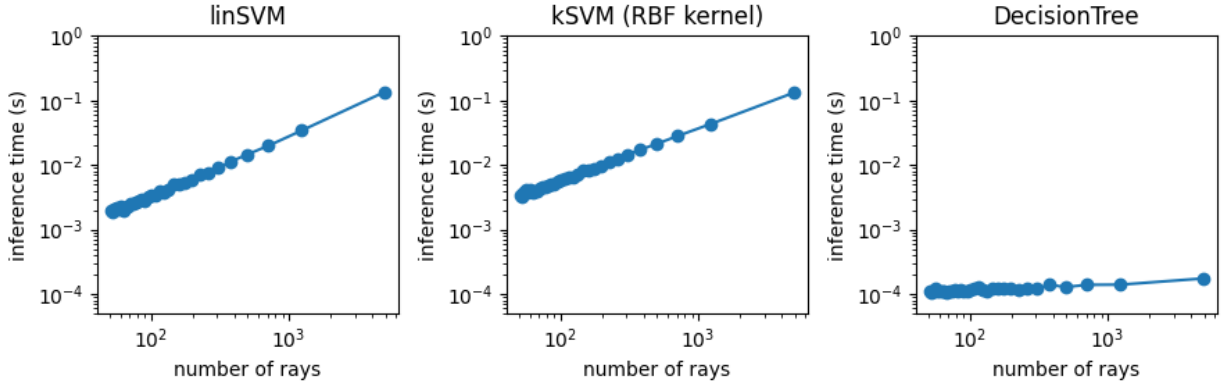
3. Concluding remarks

In this report, we tested classical non-linear machine learning classifiers in order to learn a separation between occluded and visible parts of triangular mesh elements from satellites. We proposed two strategies to compute the view factors with the potential to circumvent the use of a costly recursive splitting scheme:

- Notably, by probing the potential occlusion of a few rays (few in comparison to the number of rays needed in the solution implemented currently) on the receiving triangle, a simple machine learning classifier like kernel SVM or Decision Tree can be trained and then used to infer a binary profile of the shadow boundary. Our experiments suggest that training a Decision Tree classifier is usually much faster than training a kernel or linear SVM, and that it is potentially three orders of magnitude faster to infer with. It is also possible to get a continuous profile of the luminosity gradient adapted to model the penumbra region (at the cost of a slightly increased inference cost). Because inference is usually much faster than training for these classifiers, we propose to train



(A) Training time as a function of number of rays in the training set.



(B) Inference time as a function of number of rays in the inference set.

FIGURE 7. Comparison of the average computation time (training and inference) of the three classifiers over $N_s = 100$ random shadow profiles for different number of rays used to make the training/inference set. Default parameters are used for the classifiers, and the Decision Tree has parameter "max_depth" set to 10.

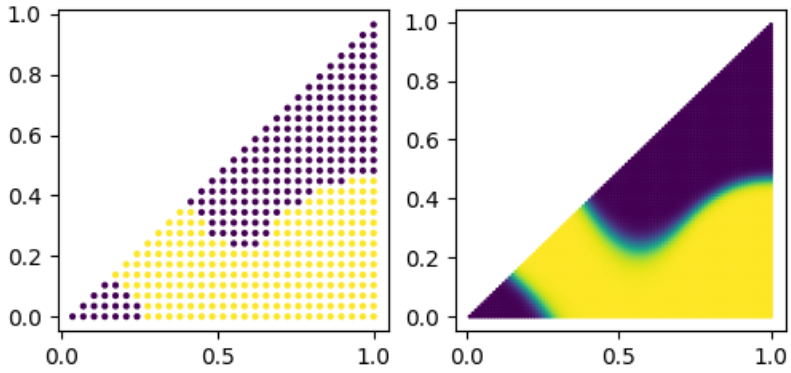


FIGURE 8. Visualization of the probability of belonging to each class, obtained with kernel SVM. Close to the boundary, *i.e* in the penumbra region, the probability describes a smooth gradient.

on the results a few probing rays and to infer the profile of the boundary on a much finer grid.
- Another strategy is using the fact that the learnt decision boundaries of Decision Trees are naturally polygonal. This fact allows the direct computation of the View Factor using Schröder's formula, provided that one can extract the vertices of the boundary polygon from the set of splitting constraints learnt by the Decision Tree. If such a solution is found, the training speed and low memory footprint of Decision Tree classifier make it a perfect candidate solution to be benchmarked in a real case application (both with real meshes/shadows, and in a C++ implementation).

Due to a lack of additional available time, we did not provide a proper benchmark on the full C++ code. However, we provided Python scripts demonstrating the classification of arbitrary shadows generated randomly, which gave promising results and should serve as proofs of concept.

One of the key finding in this work is that sending a relatively small number of probing rays (for the training of the classifiers), namely a few hundreds, should be sufficient to reach a high approximation precision.

Appendix A. Cross-validation for classifiers

Cross-validation is a classical method used to optimize one or several parameters of a machine-learning algorithm when a training dataset is available. The process involves dividing the training dataset into k folds, training the model on $k-1$ folds, and validating it on the remaining fold in an iterative fashion. To optimize a specific parameter, a grid of possible values is defined. The model is trained and validated for each value in the grid, and the performance is evaluated across all folds. The optimal parameter value is selected based on the best average performance. Finally, the model is trained on the entire dataset using the chosen parameter and assessed on a separate test set to ensure generalization. The approach is illustrated on Fig. 9. This systematic approach helps in preventing bias in the way the performance of the optimal parameter is evaluated.

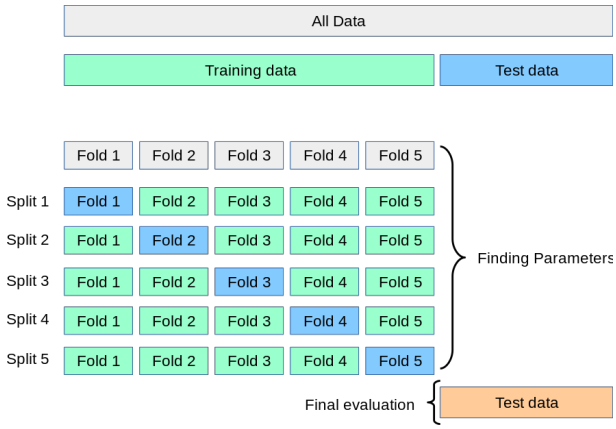


Figure 9. Schematics of the cross validation scheme. The training data is split multiple times so that it can be used for the unbiased optimal estimation of a parameter of the classifier.

Appendix B. Intuition of the role of the main parameters of the classifiers

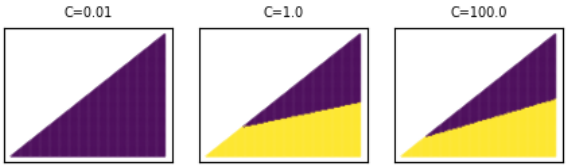


Figure 10. Qualitative comparison of the classification results obtained with linear SVM when varying the regularization parameter C. The training set is the same as in Fig. 4.

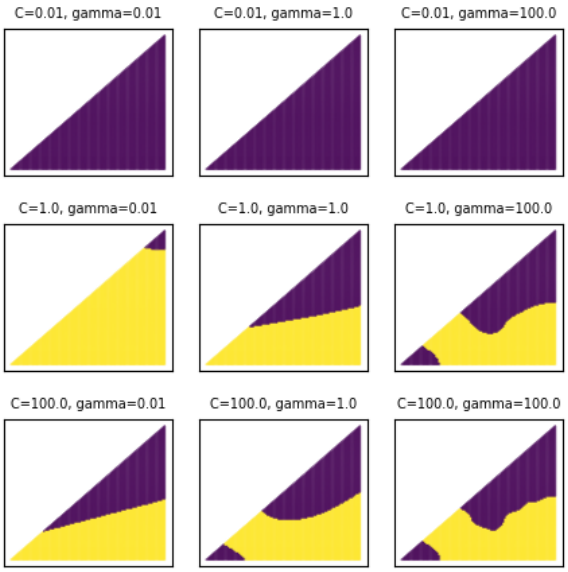


FIGURE 11. Qualitative comparison of the classification results obtained with kernel SVM (RBF kernel) when varying the regularization parameter C and the scale parameter γ (γ is inversely related to σ, the typical length scale). The training set is the same as in Fig. 4.

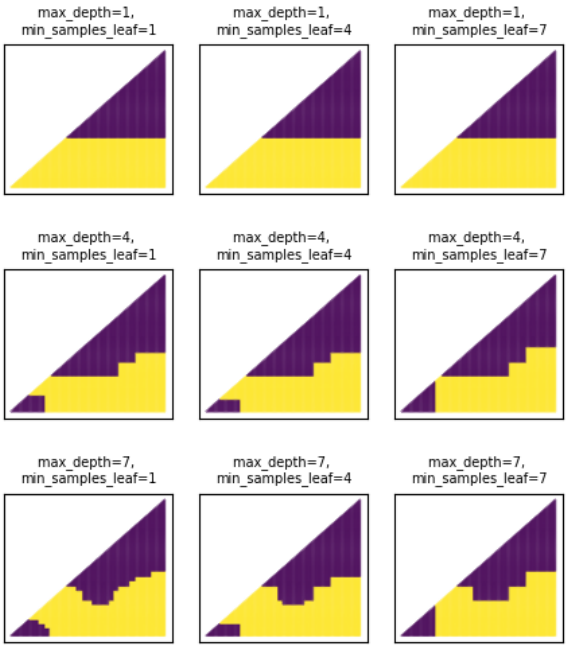


FIGURE 12. Qualitative comparison of the classification results obtained with the Decision Tree classifier when varying the maximum depth of the tree and the minimum number of samples in a leaf. The training set is the same as in Fig. 4.

References

[1] P.Schröder, P.Hanrahan, (1993) On the Form Factor between Two Polygon. Department of Computer Science Princeton University.
[2] G.Ritoux, (1982) Evaluation numérique des facteurs de forme. Revue de Physique Appliquée, 17 (8), pp.503-515.
[3] V.Vadez, (2022) Geometric simplification for radiative thermal simulation of satellites https://theses.hal.science/tel-03824886v2
[4] G.N.Walton, (2002). Calculation of Obstructed View Factors by Adaptive Integration. NISTIR 6925
[5] B.E.Boser, I.M.Guyon; V.N.Vapnik, (1992). A Training Algorithm Optimal Margin Classifiers. Proceedings of the fifth annual workshop on Computational learning theory
[6] L. Breiman,(2001). Random Forests. Machine Learning
[7] S.R. Safavian , D. Landgrebe, (1991). A survey of decision tree classifier methodology. IEEE Transactions on Systems, Man, and Cybernetics