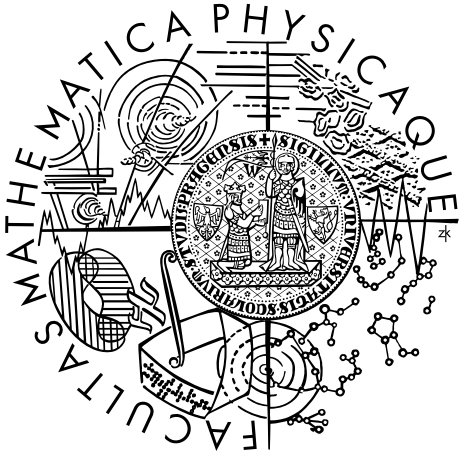
Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS



Michal Šebesta

Synthesis of digital landscape surface data

Department of Software and Computer Science Education

Supervisor of the master thesis: Mgr. Martin Kahoun

Study programme: Computer Science

Study branch: Software Systems

Prague 2015

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ..... signature of the author

Title: Synthesis of digital landscape surface data

Author: Michal Šebesta

Department: Department of Software and Computer Science Education

Supervisor: Mgr. Martin Kahoun, Department of Software and Computer Science Education

Abstract: A procedural generation of landscapes often meets a need for real spatial data at finer resolution that data available at the moment. We introduce a method that refines the spatial data at the coarse resolution into the finer resolution utilizing other data sources which are already at the better resolution. We construct weighted local linear statistical models from both the coarse and utility data and use the by-models-learned dependencies between the data sources to predict the needed data at better resolution. To achieve higher computational speed and evade utility data imperfection, we utilize truncated singular value decomposition which reduce a dimensionality of the data space we work with. The method is highly modifiable and its application shows plausible real-like results. Thanks to this, the method can be of practical use for simulation software development.

Keywords: land cover, refinement, GIS, elevation maps, MODIS

Contents

Introduction

Many various applications, such as flight simulators, games, battle simulators or visualization applications, make an effort to build huge landscape scenes which resemble the reality as close as possible.

At some point of the landscape generation, the ground needs to be colored and the scene needs to by populated by objects such as trees, shrubs, grass straws or others. Trained artists and proprietary tools are required for creating a believable landscape by hand. By growing a scene to huge sizes, the scene by-hand-creation can become extremely expensive from both time and financial point of view and can obstruct a rapid scenario production. Therefore automated methods are desired to build the landscape procedurally without the need for artists or special tools Smelik et al. [2014].

The exact placement of all objects in a countryside is seldom of real importance. They just affect a perception of a realistic look in larger areas. This leads to ecosystems models, which determine the placement of plants within the larger areas. Each ecosystem thus describes plant types with their frequency and placement, as well as ground colors within a region the ecosystem is bound with (Hammes [2001]).

It is customary to use these ecosystems to build the large landscapes procedurally, instead of placing all the objects by hand. The objects placing problem transforms itself into a problem of an ecosystem distribution, which is easier to solve. Basically is sufficient to provide a 2D map of a landscape surface where each item of the surface map binds the corresponding surface area with a specific ecosystem. The surface map can be provided by artists or from any data sources. With a knowledge about the ecosystems and their distributions, whole the huge landscape scene can be finally constructed (Smelik et al. [2014]).

There exist many and various data sources which cover the Earth and are available. The common and well-known data types are for example ortho-photomap and height-field. Other types are for instance surface materials data, water and vegetation maps etc. Each data source has different resolution and different meaning.

A building of huge landscapes is in the interest of many game development companies, companies working with simulation, etc. Some of them even work on covering the whole Earth (Google[1], BISim[2], Outerra[3]). This thesis is motivated by practical problems of such companies. The thesis creation was accompanied with a consultation with specialists from these companies (more precisely BISim).

One choice of data source for Earth scene construction using the ecosystems is MODIS Land Cover. It is a sources which classifies different land cover (surface) types therefore can be used for binding with proper ecosystems. We chose this data set because it is easily accessible, covers whole Earth and is relatively up-to-date. Other similar source differing surface types is US Land Cover. It may seem better because exists in better resolution (90 m/pixel) in comparison with MODIS Land Cover (500 m/pixel), but covers only United States of America.

[1] `http://earth.google.com`

[2] `http://www.bisimulations.com`

[3] `http://www.outerra.com`

When the surface is viewed from larger distance it can seem too coarse in some situations. For example in flight simulator. Moreover, the surface texture is typically used to texture an elevation map which is usually in much better resolution. Therefore the goal is to refine surface maps to better resolution (to correspond the elevation maps).

For the refinement, we utilize another data source – namely the SRTM NASA V3 elevation maps[4]. The resolution of these maps is 90 m/pixel. An usage of this data source is not the only option, other better data exist, but we chose this one because SRTM is available for the whole world and is easily accessible. We use these specific MODIS and SRTM sources to demonstrate a general problem with some real data.

The refinement results must cause more plausible results in comparison with an usage of the coarse surface maps. On the other hand, the results should resemble reality.

In this thesis we introduce a method based on weighted local linear regression (wLLR) for the refinement of surface maps into a finer resolution utilizing elevation maps at that resolution. Specifically, we demonstrate this method with MODIS Land Cover and SRTM data sets which have 500 m/pixel and 90 m/pixel resolutions, respectively.

The background of the problem

A requirement of procedural generation of landscapes came up at the end of the 20th century with an increase in rendering speeds which enabled possibility to add more and more objects into virtual scenes to improve realistic perception. Flight simulators, games, virtual battlefield applications or other visualization application laid stress on wide landscapes where huge amount of objects, such as plants, were present. Due to limited amount of RAM or due to a fact that artists work on large scenes was too expensive, a better solution to construct realistic looking landscape scenes had to be discovered (Hammes [2001], Wells [2005]).

Natural scenes do not have to be defined by exact positions of plants. A relative plants placement is important for the realistic scene perception. Such rules for relative placing can be defined by ecosystem which is a form of compression for natural environments. The procedural generation of landscapes revealed this compression advantages in comparison with methods which stored all present plants in the base scene representation.

Many procedural approaches for the landscape generation were discovered.

Hirtz et al. [1999] used a geospatial data to place vegetation according to predefined templates for a real-time landscape visualization. This approach was restricted to small areas of interest where a geospecific placement of vegetation objects was possible to be gained from hight-resolution remote sensing data and forestry maps.

Deussen et al. [1998] occupied themselves with procedural vegetation placing in scene using user defined ecosystems. Lane and Prusinkiewicz [2002] formalized and extended the methods for defining plant distribution (originally proposed by Deussen et al. [1998]) and also introduced systems to model groups of plants rather then single plant alone.

[4] `http://www2.jpl.nasa.gov/srtm/`

Hammes [2001] introduced a method for real time landscape visualization based on ecotopes (ecosystems) modeling. This approach plausibly placed vegetation exclusively according to elevation-derived products such as basic elevation, a relative elevation, a slope and a slope direction. An advantage of this method was that it generated realistic landscapes without using real-world land cover data. An advantage is that the rules were figured out ad-hoc.

A method introduced by Ahlberg et al. [2005] used large amount of elevation data to automatically generate detailed 3D scenes. It was fine approach for preprocessing detailed model of small region, but was not practical either for a runtime placement of vegetation objects or processing of larger areas due to the poor length of processing and source data requirements.

Wells [2005] intended to fill gasps between these various approaches. Therefore they employed basic imagery and topographic analysis to automatically construct vegetation-populated terrains. The method was based on readily available source data, such as elevation points or land cover classification, where the land cover images were crucial for the vegetation placement.

We hold here with the idea of the land cover utilization. But a need for finer source data appeared, because the suitable land cover pictures are too coarse to be satisfyingly used in production. Therefore a method to refine these surface data needed to be discovered.

An availability of many distribution sources provides huge mass of data, which represents an opportunity to study various aspects. Unfortunately, there is often conflict between relatively coarse resolution of available data source and the user needs which may require finer resolution of the distribution information. A major challenge for using the coarse data sources is developing appropriate methods for translating information from one scale to another. The distribution information often serves different specific purposes therefore many various methods of downscaling coarse data can be found.

Needed occurrence data are typically much coarser than a grain of many biological processes, which affects the base cause, and which environmental data are available for. Therefore a big potential was shown for methods which combine the occurrence data in coarse resolution with environmental data in fine resolution to predict the base cause occurrence in the fine resolution. The downscaling is processed then within a single statistical modeling framework. Many of proposed downscaling methods using the statistical models shown results of different quality (Keil et al. [2013], Bombi and D'Amen [2012], McPherson et al. [2006]).

First, widely used approach of downscaling is to project distribution models calibrated at a coarse data onto a grid in finer resolution. This *direct approach* is intuitive since the model calibrated at coarse data predicts occurrences using environmental variables from data in finer resolution, it projects the statistical relationship by applying model parameters.

Collingham et al. [2000] used logistic regression analysis to predict the presence or absence of three non-indigenous riparian weeds. Araujo et al. [2005] downscaled various distribution atlases such as plants, birds or mammals to finer resolution by utilizing generalized additive models (GAM). Climate variables and land cover were used as predictors for the GAM. Other authors used this downscaling approach for different purposes such as prediction of animal occurrence (Anderson et al. [2009], Mitikka et al. [2008]), revealing of climate change de-

pendency to plant distribution (RANDIN et al. [2009], TRIVEDI et al. [2008]), or other.

The direct approach is build on an assumption that the occurrences in fine resolution show the same environmental relation as the coarse distribution. This assumption usually does not hold in reality (e.g. Menke et al. [2009]).

The *iterative approach* moves from coarse data square to finer resolution incrementally. Instead of applying the model directly to fine resolution, it first downscales the coarse data to some intermediate resolution. Each step doubles the spatial resolution of predictions (McPherson et al. [2006]). However, the unrealistic assumption still remains.

The third approach, *point sampling*, takes fixed number of random points from high-resolution grid and calibrates the model, for each coarse square (McPherson et al. [2006], Penn Lloyd [1998]). This approach needs an assumption that all areas within the chosen points have suitable conditions for the base cause, which is also unrealistic assumption (Keil et al. [2013]).

A further *clustering approach* runs the pixels of occupied coarse atlas cells through cluster analysis based on environment within the cells. Assuming that each occupied cell contains some favourable habitat and that the favourable habitat is more homogenous from one occupied cell to the next than is a unfavourable habitat (McPherson et al. [2006]). Limitation of this approach is that it ignores absence information. There is also difficult to determine objectively how many cluster to divide data into. Moreover irrelevant environmental attributes can mask differences in important indexes which can cause misleading or inconclusive cluster arrangements.

Several other methods were proposed, which were based on some expert knowledge about the base (coarse) data habitat (Niamir et al. [2011]). In our problem we do not have this kind of expert knowledge.

To avoid the unrealistic assumptions, that species–environment relationships do not change across scales (direct approach), and that all finer-resolution cells host equally suitable conditions for species under study (point sampling), Keil et al. [2013] developed a novel method for downscaling species distribution by combining a Bayesian hierarchical modeling approach with power-law scale–area relationships.

We do not use these methods from following reasons. First, we need a faster method because we want to process huge data sets. Second, we need local models. The big data sets must be processed in batch due to computation and memory limitations. If we would create one global model for each tile then there would be too sharp edges between neighboring files. Moreover global models do not know local contexts. Third, their approaches were aimed for sparse observations – we have sufficient of reliable data.

Moon et al. [2014] proposed a new adaptive rendering algorithm which enhanced the performance of Monte Carlo ray tracing by reducing the noise. They employ local weighted linear regression to smooth an image and then they apply error analysis to guide their adaptive sampling process. Advantage of their method is allowing users to use an arbitrary set of features for the models prediction, including noisy features, because the method computes a subset of features by ignoring the noisy ones and decorrelating them for higher quality.

Moon et al. [2015] modifies the previous method by iteratively building mul-

tiple, but sparse linear models. The method recursively estimates the prediction errors introduces by linear predictions performed with different prediction windows. Then they set the optimal prediction window which minimizes the error for each linear model. They do not use different models for predictions of each pixel, as in the previous method, but each model predicts multiple pixels.

For solving our problem, we combine the cores of the direct approach, the Moon et al. [2014] approach and the Moon et al. [2015] approach and transform it to our downscaling problem. By using local generalized linear regression we avoid the unrealistic assumptions of direct approach. We analyze each pixel similarly to Moon et al. [2014], but we build multiple models for each pixel to choose the optimal prediction window which minimizes the prediction model, according to Moon et al. [2015]. However, we do not build the iterative models recursively and construct multiple new ones for each pixels all over again. This is an deficiency of our algorithm, since it is not optimized.

Description of the remaining chapters

The rest of the thesis is structured as follows. The Chapter 1 describes our method to solve the defined problem. First, we briefly describe the overall approach to get better idea about the solution (Sec. 1.1). Then we present a core theory standing behind (Sec. 1.2) and define the input data more in detail (Sec. 1.3) to prepare the reader for full understanding. Thereafter the whole rest of the chapter thoroughly explains our proposed method (Sec. 1.4). The Chapter 2 includes additional explanations for issues which are not completely covered in the approach chapter. A real implementation with all technical details is then presented in Chapter 3 and is followed by a evaluation in Chapter 4. Future work is presented in Section 4.6 and conclusion in Section 4.6.

1. Approach

In this chapter we describe our method for surface data refinement. First of all, we briefly describe the approach in general (Sec. 1.1). This section is followed by a theory standing behind (Sec. 1.2). Then we describe the input data more in detail (Sec. 1.3). The rest of the chapter explains our proposed approach in detail and also explains how we applied the mentioned theory to solve the problem (Sec. 1.4).

1.1 A brief description of the approach

We propose a method to refine coarse 2D data into a finer resolution data sources at that resolution. The approach is based on statistical modeling. The statistical model is constructed by both the coarse data and the helping data at finer resolution. The model learns dependencies between the surface types occurrence and the given data at fine resolution and use the learned dependencies to predict the surface map at fine resolution.

In general, our method can work with various data sources but our problem is aimed to refine specific surface data utilizing a specific digital elevation model (DEM). The motivation behind why we use only DEM is described in Section 2.2. The detailed description of input data is provided in Section 1.3 but shortly, we refine MODIS Land Cover data (Sec. 1.3.1) which is a surface map distinguishing 17 different classes and covering the whole Earth. As we mentioned earlier (Chap.) we utilize SRTMv3 DEM data set (Sec. 1.3.2) to refine MODIS. The resolution of MODIS is approximately 500 m/px against the resolution of DEM which is approximately 90 m/px. Note that the mentioned resolution corresponds with locations near Equator and that the precise area covered by one pixel varies according to specific longitude and latitude. But in general we can say that the resolutions ratio for MODIS and SRTM3 is approximately $1 : 5$.

The MODIS data distribute surface types into 17 classes, thus the surface map contains pixels with integer values between 0 and 16 where each value relates to specific class. But in general, we can work with various number of surface types. Provided that we have all surface types together in one map, we can split up our problem into particular problems of single surface classes. This means that we look at the problem from the point of view of a single class, separately for each class. Therefore, the method does not work with the surface types map including information about all 17 classes. It works with 17 different occurrence maps which are always related to a specific class, instead. Each occurrence map then carries information only about its surface type presence and absence. Thus, a single occurrence map is then a binary map where the value of 1 signifies that the given class in present on corresponding position and 0 signifies the opposite. We process each class separately.

The DEM has a crucial role in our approach similarly as the approach introduced by Hammes [2001]. He defined relationships between ground characters derived from DEM and rules for placing vegetation. He united the rules for placing vegetation into a type of ecotope, which correspond to our surface maps, and then he assigned a specific ecotope to some area according to its DEM charac-

teristic. In his method, the relationships between DEM and ecotopes are defined ad-hoc by the user whereas the core of our method is to find these relationships via machine learning methods.

We use very similar set of predictors as Hammes [2001], namely an absolute elevation, a relative elevation, a slope (an elevation gradient), an aspect (a direction of the elevation gradient). We provide a complete list of used features with their explanations in Section 1.4.1.

Our method is based on the same idea as the de-noising of rendered pictures which was introduced by Moon et al. [2014]. We use a weighted local linear regression (wLLR) to reveal connections between provided surface types occurrences and the particular features (Sec. 1.4.2).

A linear predictor is sufficient when used locally contrary to linear predictor used globally which totally fails (Moon et al. [2014]). The global model is to weak because conditions for surface types occurrence are variable according to specific longitude and latitude. Moreover, vegetation species represented by given surface type are variable therefore it is not possible to fit the surface types using global model. For example, a forest surface class would express different ecosystem in South America than in Europe. Thus, we construct many local models using the wLLR instead of constructing one global model. For each pixel of the target resolution, we take a local window around that pixel and construct the particular local model. Then we predict a result value for that pixel by using this model prediction.

Determination the right size of the local neighborhood is a problem which we solve by constructing several models with different window sizes. We then choose a model with the smallest prediction error. This problematic is described more in Section2.3.2.

The creation of a wLLR model may require a high cost of computation as we consider more types of features. The bigger model is the bigger cost is. Moreover, the features can be linearly/almost-linearly dependent to each other which breaks the linear model assumption. Other problem is that common local regression methods assume that input predictor variables (our features) are not noisy but our features can carry a significant amount of noise. Instead of working in all features dimensional space, we propose to generate an appropriate reduced dimensional feature space to solve the mentioned problems. To achieve the reduced feature space we use truncated singular value decomposition (tSVD) (Sec. 1.4.3).

By evaluating created models we obtain probability probability of a surface type presence. For each surface type, we then create a map which contain a distribution of presence probability for the corresponding class. The final result surface map is created from these single probability maps by taking the surface type which is the most probable on a given position (Sec. 1.4.4).

1.2 Terms and theory

1.2.1 Local linear regression

In this section we give a background on local linear regression (Bishop [2006]). Its application is described in Section1.4.2.

In general, a *regression* is an approach for modeling a relationship between a scalar dependent variable y and one or more independent variables $\mathbf{x}$. More specifically, regression analysis helps one understand how the typical value of the dependent variable changes when any one of the independent variables is varied, while the other independent variables are held fixed.

The *regression model* relates y to a function of $\mathbf{x}$ and $\boldsymbol{\beta}$ as follows:

$$y \approx f(\mathbf{x}, \boldsymbol{\beta}),$$

where

- $\mathbf{x}$ stands for independent variables (often called as predictor variables, regressors, explanatory variables or predictor variables)
- y stands for dependent variable (often called as response variable, regressand, endogenous variable or measured variable)
- $\boldsymbol{\beta}$ represents vector of regression model parameters (often called as effects or regression coefficients)

Linear regression has one essential specification in comparison with general regression. Linear regression model assumes that the relationship between the dependent variable y_i and vector of independent variables $\mathbf{x_i}$ is linear. This relationship is modeled through an error variable ε_i which adds noise to the linear relationship between the dependent variable and independent variables. The noise has zero mean and bounded variance. The model takes the form as follows:

$$y_i = \beta_1 x_{i1} + \cdots + \beta_p x_{ip} + \varepsilon_i = \mathbf{x}_i^T \boldsymbol{\beta} + \varepsilon_i, \quad i = 1, \ldots, N; p = 1, \ldots, D,$$

where i is the index of particular observation, N is the count of statistical units, D is the dimension of independent variables vector $\mathbf{x_i}$, and $\mathbf{x_i^T}\boldsymbol{\beta}$ is inner product between vectors $\mathbf{x_i^T}$ and $\boldsymbol{\beta}$. When we stack all N equations together then they can be written in vector form as follows:

$$\mathbf{y} = \mathbf{X}\boldsymbol{\beta} + \boldsymbol{\varepsilon},$$

where $\mathbf{y} \in \mathbb{R}^N$, $\mathbf{X} \in \mathbb{R}^N \times \mathbb{R}^D$ and $\boldsymbol{\beta} \in \mathbb{R}^D$, $\boldsymbol{\varepsilon} \in \mathbb{R}^D$

Notice that in linear regression the dependent variable $\mathbf{y_i}$ is a linear combination of the model parameters but it does not mean that the linearity must be kept along vector of independent variables $\mathbf{x_i}$. For example, assume that for the following equation:

$$y_i = \beta_0 + \beta_1 a_i + \beta a_i^2 + \varepsilon_i, \quad n = 1, \ldots, N,$$

the right side of the equation is linear in the parameters β_0, β_1 and β_2. Therefore it is a linear regression even though the independent variable $\mathbf{x_i} = \{1, a_i, a_i^2\}$ is quadratic in its items. The constant term $\beta_0 \cdot 1$ is know as an *intercept* term. It has various meanings in applications.

The model assumes that — above that the response variable is a linear combination of the parameters and the predictor variables — independent variables $\mathbf{x_i}$ is a noise-free $\mathbb{R}^D$ vector. Further the model assumes that $\mathbf{y}$ is a homoscedastic $\mathbb{R}^N$ vector of dependent variables which means that different dependent variables

have the same variance in their errors. Moreover, it is assumed that the errors of the dependent variables are uncorrelated with each other. Another important assumption is lack of multicollinearity in the independent variables matrix $\mathbf{X} \in \mathbb{R}^N \times \mathbb{R}^D$. Multicollinearity is a phenomenon in which two or more independent variables in a multiple regression model, columns of $\mathbf{X}$, are highly correlated, meaning that one can be linearly predicted from the others with a substantial degree of accuracy. Presence of some of named bad phenomenons can cause wrong estimation of the model.

These assumptions are often not valid for real data therefore we use a special method for their achieving (Sec. 1.4.3).

Local regression (Cleveland and Loader [1996], Ruppert and Wand [1994]) is a smoothing method for fitting parametric curves or surfaces, $f(\mathbf{x})$, based on a neighborhood of $\mathbf{x}_c$. Its underlying statistical model is the same as for general regression $y = f(\mathbf{x}) + \varepsilon$, but one difference is that independent variables $\mathbf{x}$ are observations from a neighborhood of some center observation, $\mathbf{x}_c$.

Local linear regression is then linear regression based on a neighborhood of $\mathbf{x}_c$.

A large number of procedures have been developed for model parameters estimation and inference in linear regression. These methods differ in computational simplicity of algorithms, robustness, theoretical assumptions and in many other factors. Let us point out one which is frequently used — the least squares approach.

Ordinary least squares

Let us denote the value of the dependent variable predicted by the model by $\bar{y}_i = \mathbf{x_i}^T \bar{\boldsymbol{\beta}} = f(\mathbf{x_i})$ and true value of dependent variable by y_i. The *residual* is then defined as $e_i = y_i - \bar{y}_i$.

The ordinary least squares (OSL) is model estimation method which minimizes the sum of squared residuals, denoted by SSE (sum-of-squares error) (Bishop [2006], Kutner et al. [2005]), as follows:

$$\min_{\bar{\boldsymbol{\beta}}} SSE = \min_{\bar{\boldsymbol{\beta}}} \sum_{i=1}^{N} e_i^2 = \min_{\bar{\boldsymbol{\beta}}} \sum_{i=1}^{N} (y_i - \bar{y}_i)^2 = \min_{\bar{\boldsymbol{\beta}}} \sum_{i=1}^{N} (y_i - f(\mathbf{x_i}))^2,$$

and leads to a closed form expression for the estimated value of the unknown parameter $\bar{\boldsymbol{\beta}}$ as follows:

$$\bar{\boldsymbol{\beta}} = (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{y},$$

where $\bar{\boldsymbol{\beta}} \in \mathbb{R}^D$, $\mathbf{X} \in \mathbb{R}^N \times \mathbb{R}^D$ and $\mathbf{y} \in \mathbb{R}^N$.

Weighted least squares

Weighted least squares (WLS) is an extension of OLS. WLS minimizes a weighted analogue to the sum of squared residuals from OLS as follows:

$$\min_{\bar{\boldsymbol{\beta}}} \sum_{i=1}^{N} w(\mathbf{x_i}) \cdot e_i^2 = \min_{\bar{\boldsymbol{\beta}}} \sum_{i=1}^{N} w(\mathbf{x_i}) \cdot (y_i - f(\mathbf{x_i}))^2,$$

and leads to a closed form expression for the estimated value of the unknown parameter $\bar{\boldsymbol{\beta}}$ as follows:

$$\bar{\boldsymbol{\beta}} = (\mathbf{X}^T \mathbf{W} \mathbf{X})^{-1} \mathbf{X}^T \mathbf{W} \mathbf{y},$$

where $\bar{\boldsymbol{\beta}} \in \mathbb{R}^D$, $\mathbf{X} \in \mathbb{R}^N \times \mathbb{R}^D$ and $\mathbf{y} \in \mathbb{R}^N$. Diagonal matrix $\mathbf{W} \in \mathbb{R}^N \times \mathbb{R}^N$ has its elements defined by the weight function $w(\mathbf{x_i})$.

WLS enables the efficient estimation of $\bar{\boldsymbol{\beta}}$ even if heteroscedasticity or correlations are present among the error terms of the model.

1.2.2 Singular value decomposition

The singular value decomposition (SVD) is a factorization of a real or complex matrix. Let $\boldsymbol{M}$ be the $n \times D$ matrix, and let the rank of $\boldsymbol{M}$ be r. The factorization has a form of

$$\boldsymbol{M} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^*,$$

where $\boldsymbol{U}$ and $\boldsymbol{V}$ are $n \times n$ and $D \times D$ unitary matrices[1] and $\boldsymbol{V}^*$ stands for conjugate transpose[2] of $\boldsymbol{V}$. The matrix $\boldsymbol{\Sigma}$ is $n \times D$ diagonal matrix, where diagonal entries, σ_m, are non-negative ordered singular values in non-increasing order, $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_r > 0$.

Because $\boldsymbol{U}$ and $\boldsymbol{V}$ are unitary, the columns of each of them form a set of orthonormal vectors. We can consider these vectors as basis vectors and call them left-singular vectors, respectively right-singular vectors, of the corresponding singular values of $\boldsymbol{\Sigma}$.

For the decomposition, SVD exploits the linear combination of rows and columns of A.

There are many applications of the SVD. One of them is finding a pseudo-inverse of a matrix. Another can be solving homogeneous linear equations or total least square minimization. SVD gives us precise statement about range, null space and rang of factorized matrix. We can name many other useful applications of SVD but let us point out one which is used in this work (Sec. 1.4.3), it is low-rank matrix approximation.

Truncated singular value decomposition

Some practical applications need to solve the problem of approximation of matrix $\boldsymbol{M}$ with another matrix $\bar{\boldsymbol{M}}$ with specific rank k. Approximated matrix $\bar{\boldsymbol{M}}$ is often called truncated matrix and can be received as

$$\bar{\boldsymbol{M}} = \boldsymbol{U}\bar{\boldsymbol{\Sigma}}\boldsymbol{V}^*,$$

where $\bar{\boldsymbol{\Sigma}}$ is the same matrix as $\boldsymbol{\Sigma}$ except that it contains only first k singular values. The rest of singular values are set to zero.

In SVD of approximated matrix $\bar{\boldsymbol{M}}$, only k columns of $\boldsymbol{U}$ and $\boldsymbol{V}$ corresponding to the k largest singular values carry the information used in $\bar{\boldsymbol{M}}$, the rest of the $\boldsymbol{U}$ and $\boldsymbol{V}$ can be discarded. Let us denote these cropped matrixes by $\boldsymbol{U_k}$ and $\boldsymbol{V_k}$. *Truncated singular value decomposition* (tSVD) is defined as:

$$\bar{\boldsymbol{M}} = \boldsymbol{U}\bar{\boldsymbol{\Sigma}}\boldsymbol{V}^* = \boldsymbol{U_k}\boldsymbol{\Sigma_k}\boldsymbol{V_k}^*,$$

[1] Matrix $\boldsymbol{U}$ is unitary if its conjugate transpose $\boldsymbol{U}^*$ is also its inverse. It means that $\boldsymbol{U}\boldsymbol{U}^* = \boldsymbol{U}^*\boldsymbol{U} = \boldsymbol{I}$, where $\boldsymbol{I}$ is identity matrix.

[2] Conjugate transpose of $\boldsymbol{V}$ with complex entries is a matrix $\boldsymbol{V}^*$ which was created by taking the transpose of $\boldsymbol{V}$ and then taking complex conjugate of each entry. Conjugate transpose of $\boldsymbol{V}$ with real entries is just simple transpose of this matrix, $\boldsymbol{V}^{\boldsymbol{T}}$.

where $\bar{\boldsymbol{M}}$ is $n \times D$, $\boldsymbol{U_k}$ is $n \times k$, $\boldsymbol{V_k}$ is $D \times k$ and $\boldsymbol{\Sigma_k}$ is $k \times k$ diagonal matrix with k singular values from original $\boldsymbol{\Sigma}$ (Hansen [1987]).

Note that tSVD is no longer an exact decomposition of the original matrix $\boldsymbol{M}$, but the approximate matrix $\bar{\boldsymbol{M}}$ is in a very useful sense the closest approximation to $\boldsymbol{M}$ that can be achieved by a matrix of rank k.

TSVD is commonly used for matrix rank reduction, whether from memory demand reasons, algorithm speed reasons or it can be used for removing dependencies from base matrix.

1.3 The description of used data sets

Before we start to describe the core of our approach, it would be useful to fully understand the data sets we are working with – MODIS surface map and SRTM elevation data.

1.3.1 The surface map (MODIS)

A surface map is a two-dimensional array with integer values which unequivocally identify specific surface types. We use MODIS land cover data in our approach.

MODIS[3] (Moderate-resolution Imaging Spectroradiometer) is a scientific instrument for capturing spectral data of various ranges. It was launched into Earth orbit by NASA on board of Terra and Aqua satellites in 1999, 2002 respectively. NASA provides outcomes of captured data by MODIS in several data sets. For instance NASA provides cloud products, atmosphere products, ocean products such as sea surface temperature or available radiation, furthermore various land products such as surface reflectance, burned area product, land cover product and much more. Our interest is in the last mentioned data set, land cover product.

The MODIS Land Cover Type product[4] contains five classification schemes where we use the primary land cover scheme (Land Cover Type 1) which identifies 17 different classes defined by International Geosphere-Biosphere Programmme and exist in resolution of 500 m in sinusoidal projection.

For the purpose of our work we use re-projected land cover data[5] by equirectangular projection, known also as geographic projection. The geographic projection has uniformly spaced elements across latitude longitude within a grid. Thus, it is mosaic and easy to manipulate with.

Also for the purpose of our work, we split data into files which cover one per one decimal degree on Earth ground. The reason is to have unified data for easier manipulation. Thus, one land cover file has resolution 240×240 pixels, where each pixel acquires values between 0 and 16 corresponding to specific land cover type.

We provide a visualized example for a the MODIS Land Cover file in Figure 1.1. Be aware that the file is colorized for our purposes. When it is opened with other programs, e.g. qGIS, it would look differently because the original file contains only integer values saved as gray-scaled picture. The meaning of each value/color is explained in Figure A.1.

[3] `http://modis.gsfc.nasa.gov`

[4] `http://lpdaac.usgs.gov/dataset_discovery/modis/modis_products_table/mcd12q1`

[5] `http://glcf.umd.edu/data/lc/`

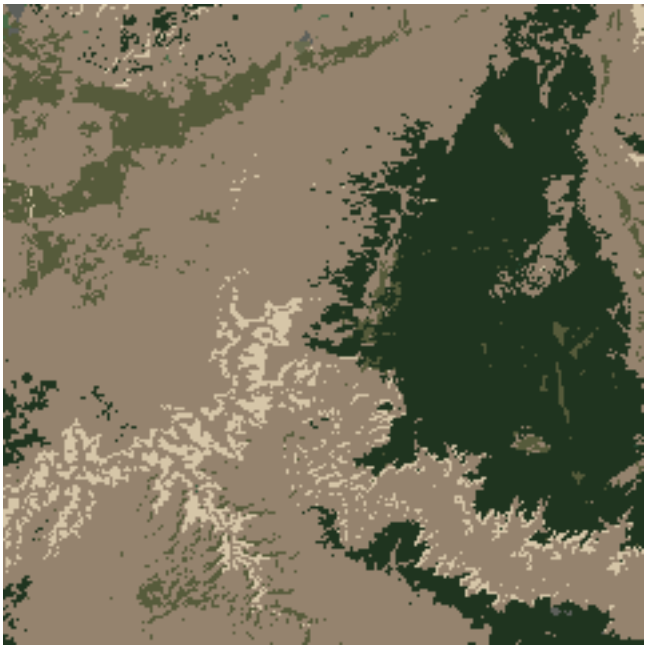


Figure 1.1: A visualization of MODIS file for Grand Canyon: 36-37 northern latitude, 113-112 western longitude.

1.3.2 The elevation map (SRTM)

An elevation map (DEM) is a two-dimensional array whose elements determine an absolute elevation above sea level. The extent of the elevation map must correspond with the extent of the surface map. Thus just as the surface map, the elevation data are split into files where each file contains a grid in geographic projection which covers one per one decimal degree on Earth ground. We use two file formats during the data processing – ".png" and ".hgt" formats.

We assume that PNG files contain integer values saved in 8-bit gray-scale image. Many libraries exist for handling the well known PNG format[6]. We use `libpng` for C++ implementation and `png` for R implementation.

In our approach we use mostly the HGT files since it is the format used by the most commonly accessible global DEM dataset – SRTM[7]. The HGT files contain elevation values saved as uncompressed array of signed integers. There exists the GDAL library[8] commonly used for geospatial data formats but reading of HGT files is so simple that we do not use any special library for their loading. Because we handle the loading of HGT files by ourselves, we describe below the specific content and format of the HGT files more in detail.

As a matter of principle, it is possible to use any other file format or GDAL.

HGT files

The HGT files have ".hgt" extension and are consisted from signed two-byte integers (shorts), where the bytes of one short are in big-endian order. Data within one file are linearized 1201×1201 array, where elements flow row-by-row

[6] `http://www.w3.org/TR/PNG/`
[7] `http://www2.jpl.nasa.gov/srtm/`
[8] `http://www.gdal.org/`

and the first row is the northernmost one. Data files are divided into one by one degree latitude and longitude tiles in the geographic projection. The names refer to the latitude and longitude of the bottom left corner of the tile – for instance, a file "N44W111.hgt" has its bottom left corner an 44 degrees north latitude and 111 degrees west longitude. The files have one pixel overlap between each other, which is on the right and top side of each file. We use the global SRTM3 data with a resolution of approximately 90×90 meters per pixel related to locations on Equator. The distance between to samples varies according to changing longitude and latitude. Values of the pixels (heights) are in meters referenced to the `WGS84/EGM96` geoid. Data voids have assigned the value -32768.

We create 1200×1200 arrays of floats from these data. This means that there is no overlap between our tiles.

More detailed elevation maps than 90 m/px per pixel exist, such as 30 m/px, 5 m/px, 1 m/px or even better, but according to the problem definition, we use exclusively the 90 m/px data. It is an arbitrary limitation of our case study. As a matter of principle, there is no limitation to use data with different resolution than ours. But in general, the bigger scale factor between MODIS and DEM resolutions is, the weaker model prediction is (Bombi and D'Amen [2012]).

We provide a visualization of elevation file in Figure 1.2

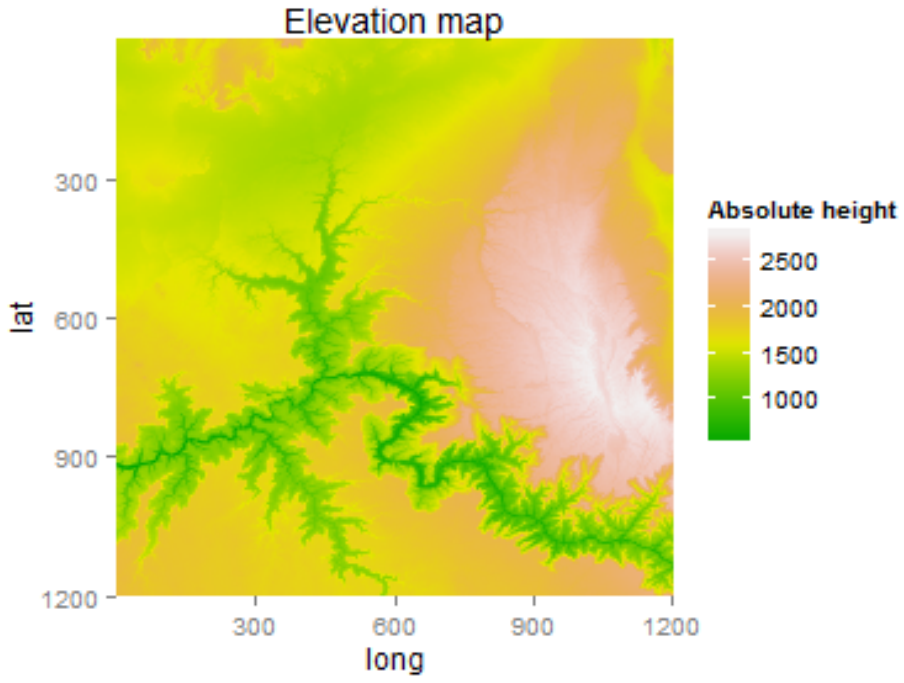


Figure 1.2: *HGT file.* A visualization for a SRTMv3 elevation map in HGT format of Grand Canyon (N36, W113).

1.4 A detailed description of the approach

1.4.1 Input and output data managing

The input of our approach is one *surface map* at the coarse resolution and one *elevation map* at the fine resolution. The output is one surface map at the fine resolution. The approach does not take into consideration any specific resolutions. Because we solve our problem from the single surface type point of view for each type separately, we split the surface map accordingly.

Surface map

The surface map (Sec. 1.3.1) (Fig. A.1) is a 2-dimensional array and involves information about surface types distribution, meaning that in each field is a value determining specific surface type . Let $m \times n$ stand for resolution of surface map and C stand for number of surface types. For each surface type, we create an occurrence map in the way that if type c is present in original surface map at some position the value 1 is set to occurrence map of type c at corresponding positions otherwise the value 0 is set. By this splitting up, we receive C maps of $m \times n$ resolution, which tells us where are different surface types in base surface map. We can see the types distribution in Figure 1.3.

Elevation map as a feature

The elevation map (Sec. 1.3.2) (Fig. 1.2) is 2-dimensional array and includes values of absolute elevation above the sea level and is used as a *feature* in the linear regression core of our method. Almost any data source with including information about some relationship between surface types and itself may be used as a feature, e.g. an average rainfall, a soil composition, average temperature etc. The relationships are later used by the regression for a creation of regression models and following predictions. We can say that the features are predictors, which predict occurrence of surface types in refined surface map. Apart from elevation map, there are more features used in our algorithm which are derived from the DEM. Each feature has different impact on the result surface map according to a strength of its relationships.

Additional features

Relative elevation map is one of additional features based on primary elevation feature. Let us denote relative elevation value at c position by $\bar{p_c}$ as follows:

$$\bar{p_c} = p_c - mean(\Omega_c^-) = p_c - \frac{\sum_{i \in \Omega_c^-} p_i}{|\Omega_c^-|},$$

where p_c is primary elevation map value at position c and Ω_c^- stands for neighborhood of c, which means all positions close to c, but excluding c.

An elevation map result can be altered by changing the neighborhood size. In our method, we use window size which is approximately twice bigger (in one dimension) then area covered by one pixel in coarse resolution.

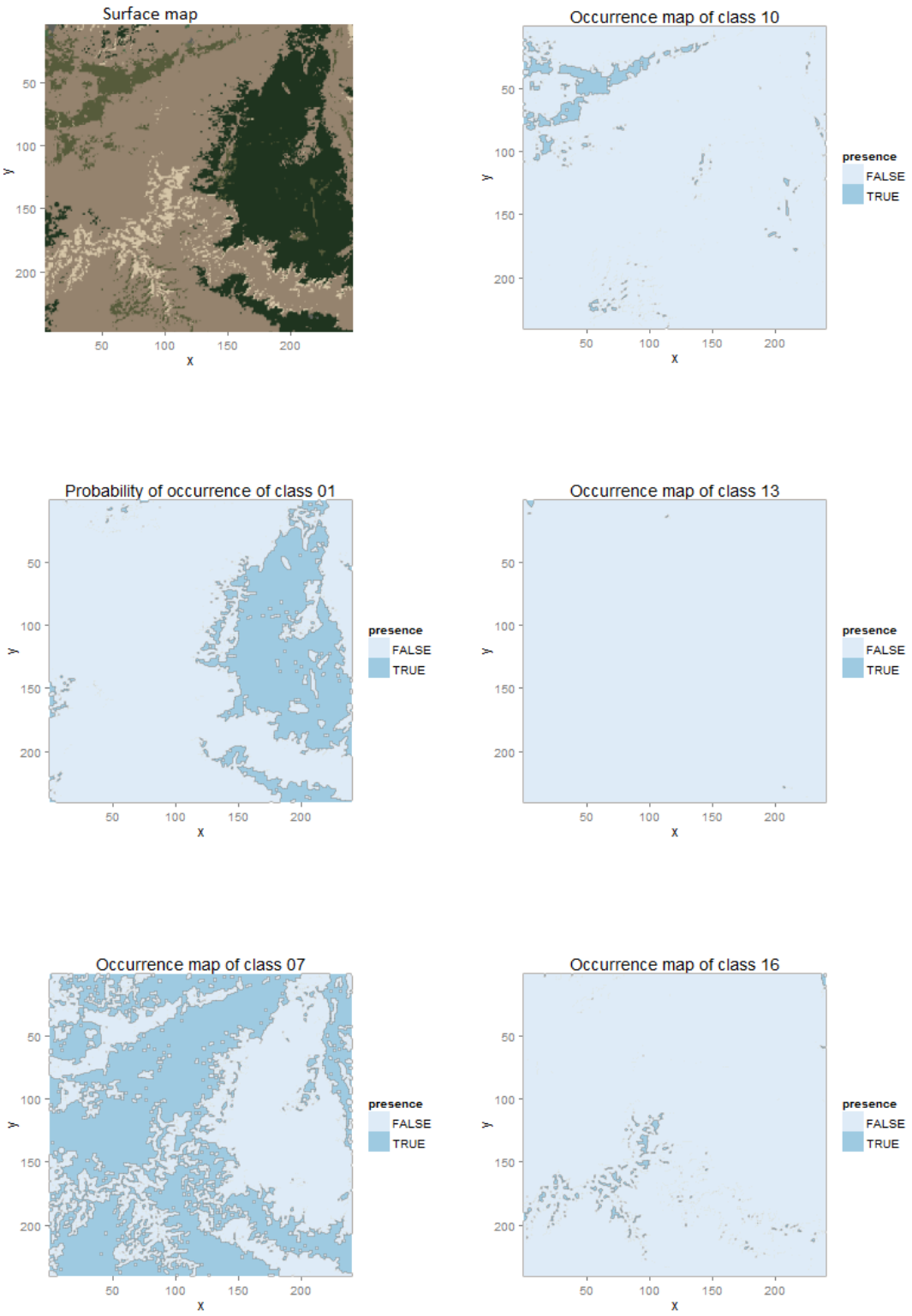


Figure 1.3: *A division of surface map into occurrence maps.* The surface map of Grand Canyon (top left picture) is distributed into occurrence maps (lower pictures) of present surface types (only the most significant ones are showed).

For example, a given surface map has 240×240 resolution and elevation map is 1200×1200. Then one pixel from surface map resolution covers 5×5 pixels in elevation map resolution. The neighborhood is then approximately $2 \cdot 5 \times$

$2 \cdot 5 = 10 \times 10$ in elevation map resolution. Because kernel methods have odd resolution, caused by the center pixel, we need to make the neighborhood size even, therefore the example resolution is $(10+1) \times (10+1) = 11 \times 11$.

Let w_f and w_c stand for the width of the finer and the coarse resolution, respectively, and h_f and h_c denote the heights, likewise. We define the formula for determining size of the neighborhood as follows:

$$neighborhood\,resolution = ((P \cdot 2) + 1) \times ((P \cdot 2) + 1),$$

where the P is determined as follows:

$$P = max\left(\frac{w_f}{w_c}, \frac{h_f}{h_c}\right).$$

We show a visualization of the relative elevation map in Figure 1.4.

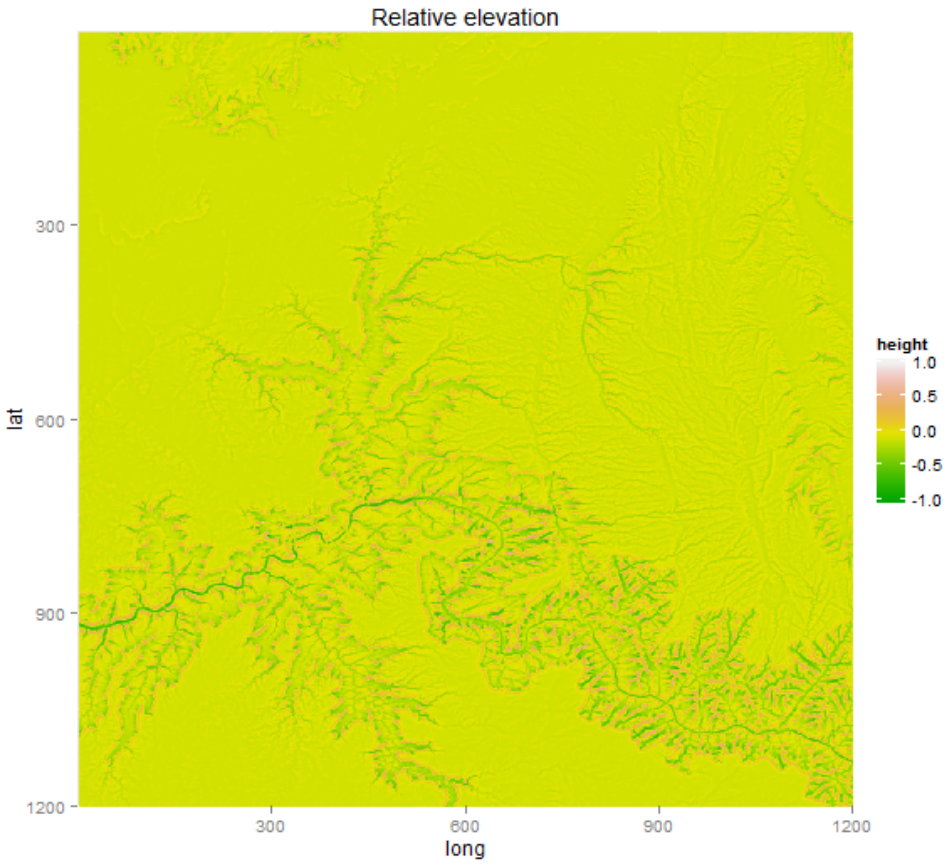


Figure 1.4: The relative elevation map for Grand Canyon.

Slope map is essentially the magnitude of the elevation map gradient. We use it as an additional feature in our approach. It can be computed from the DEM as well. The slope calculates the maximum rate of value changes from a cell to its neighbors. The slope gains values from 0 to 90 degrees. Every better geographic information system provides a tool to count the slope, e.g. GDAL DEM Tools.

We show a visualization of the slope map in Figure 1.5.

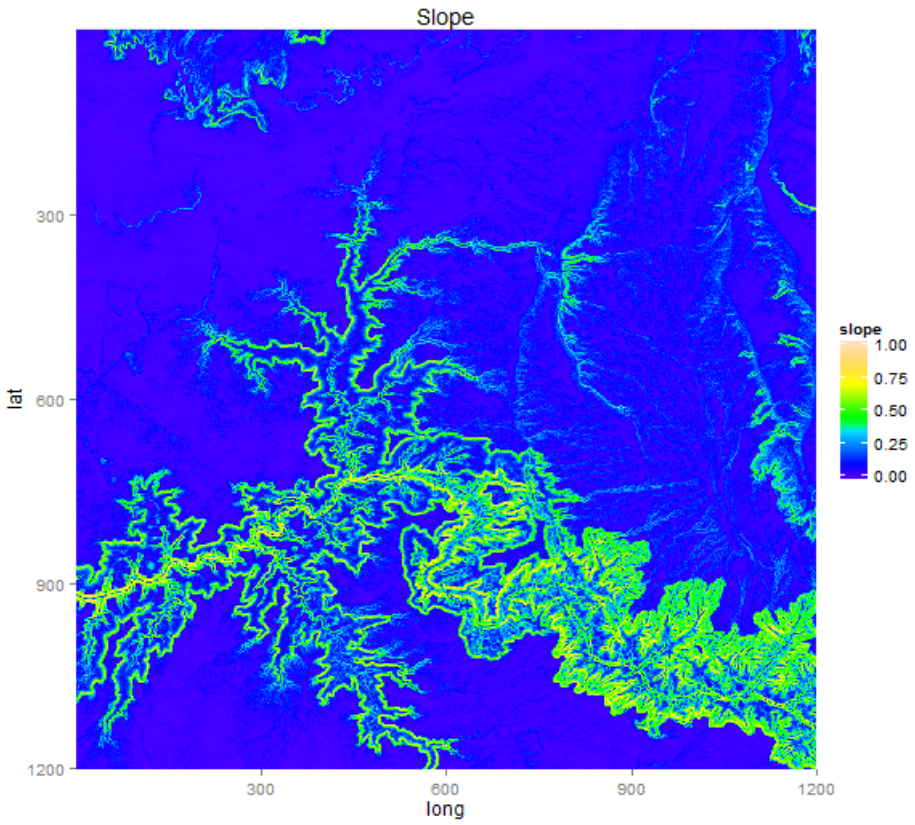


Figure 1.5: The slope map for Grand Canyon.

Aspect map is another feature which can be derived from the DEM. The aspect is the horizontal direction of the slope normal line, or just simply the slope gradient direction. The map gains values from 0 to 360. Every better geographic information system provides a tool to count the aspect, either, e.g. GDAL DEM Tools.

We show a visualization of the aspect map in Figure 1.6.

Coordinate feature is the last additional feature which carries information about a position in the window. More specifically, we create two arrays, one for horizontal direction and second one for vertical direction, which include the gradient — values between 0 and 1 uniformly interpolated from left side to right side for the horizontal feature, respectively bottom to top for the vertical feature.

We do not demonstrate any visualization of this feature because it is just a gradient.

A motivation and discussion on the topic about which additional features should be chosen, if any, can be found in Section 2.2.

1.4.2 Usage of weighted local linear regression

We use statistical models to reveal relationships between the occurrence map and given features to predict the situation at the bigger resolution. We aim for plausible results that resemble reality.

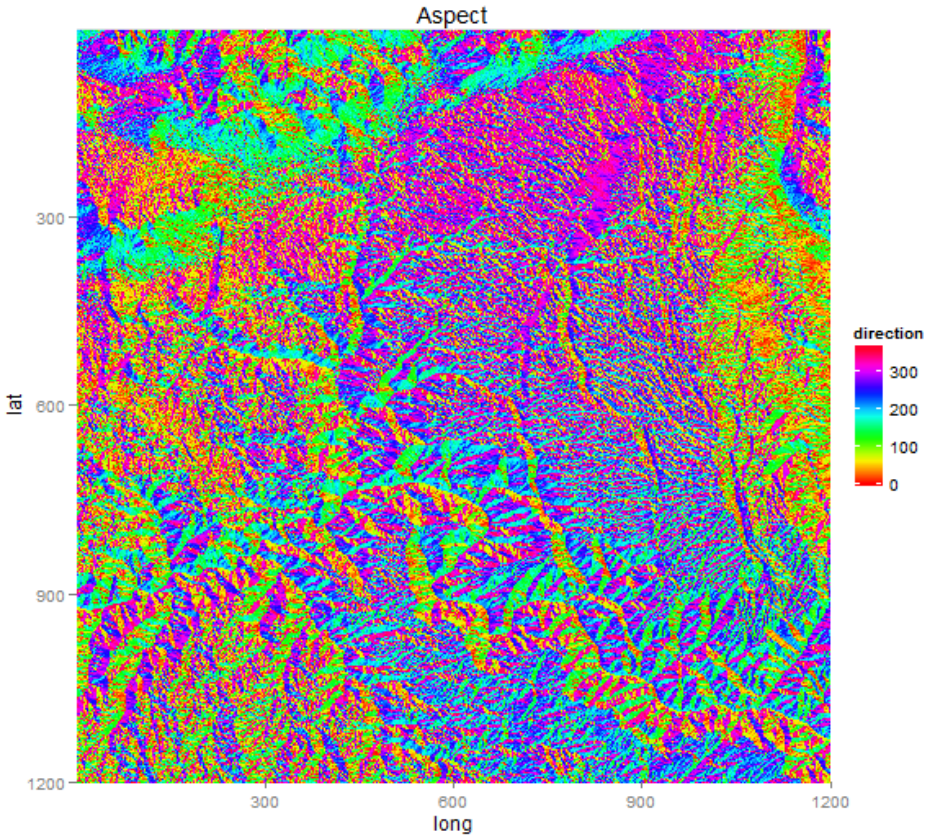


Figure 1.6: The aspect map for Grand Canyon.

In this section we describe how to apply wLLR (Sec. 1.2.1) in our algorithm which is the core of the whole computation. To describe the effects of the local regression, it is a smoothing method for fitting parametric curves (Cleveland and Loader [1996], Ruppert and Wand [1994]). Let us view our refinement refinement problem as a local regression, where the occurrence map correspond to the mentioned curve and the features correspond to curve parameters. The underlying statistical model we want to get has following form:

$$y = f(\mathbf{x}) + \varepsilon,$$

where $y \in \mathbb{R}^2$ denotes the occurrence data, the predictor $\mathbf{x} \in \mathbb{R}^D$ is a feature vector and ε stands for an additive random noise which has zero mean and bounded variance. The term $f(\mathbf{x}) \in \mathbb{R}^2$ is a model prediction of the specific class occurrence according to given $\mathbf{x}$. The vector $\mathbf{x}$ is D dimensional and includes values from all used features (the elevation map, the relative elevation map, etc.). As mentioned, many various features can be used here and the features selection is discussed more in Section 2.2.

The statistical model assumes that features vector $\mathbf{x}$ is noise-free and is not multi-collinear (Sec. 1.2.1), but unfortunately it is not true in reality from multiple reasons. First, real data carry a significant amount of noise caused by reality factors, imprecise data capture and additional data manipulation. Second, some features can be highly correlated with each other which is intelligible. For simple example, relative elevation is the same as absolute elevation within flat areas,

except for a value shift. A global model would probably contain not correlated features but with more local point of view the features become more correlated, therefore their de-correlation is needed. Third, more types of features require high cost of computation although not all of them bring information gain, as they do not have to be related with the surface type distribution. Because of these mentioned artifacts we utilize tSVD for a dimensional reduction of the features space which is described in Section 1.4.3.

Let us write $\dot{\boldsymbol{\beta}}$ for the estimated model $D+1$ dimensional coefficients. The $\dot{\boldsymbol{\beta}}$ has the additional dimension to include an intercept term. Therefore it is $D+1$ dimensional instead of D dimensional. To determine the unknown model coefficients we formulate weighted least squares minimization as follows:

$$\min_{\dot{\boldsymbol{\beta}}} \sum_{i \in \Omega_c} w(i,c) \cdot (y_i - \dot{\mathbf{x}}_\mathbf{i}^\mathbf{T} \dot{\boldsymbol{\beta}})^2, \tag{1.1}$$

where y_i is value from occurrence map which corresponds to position i. The set Ω_c includes all positions within a square window region around a center position c, in features resolution. Feature vector $\dot{\mathbf{x}}_\mathbf{i} \in \mathbb{R}^{(D+1)}$ is $[1, \mathbf{x}_\mathbf{i}^\mathbf{T}]^T$, where vector $\mathbf{x_i} \in \mathbb{R}^D$ includes values of all features at i-th position. The constant value 1 is added into the vector to retrieve intercept term $\dot{\beta}_0$ as the model coefficient. The term $\dot{\mathbf{x}}_\mathbf{i}^\mathbf{T} \dot{\boldsymbol{\beta}}$ has the meaning of a prediction of the regression model with corresponding $\dot{\boldsymbol{\beta}}$ parameters. We can also write $f(\dot{\mathbf{x}}_\mathbf{i})$ instead.

The kernel $w(i,c)$ is commonly chosen to be some symmetric decreasing function with raising distance from center position c. This kernel function, or let us call it weight function, can be realized by many various functions such as Epanechnikov function or Gaussian function both very well known in the statistics Moon et al. [2014]. However, we use Tricubic function which is defined as:

$$W(u) = \frac{70}{81}(1 - |u|^3)^3 \, \mathbf{1}_{\{|u| \leq 1\}}, \tag{1.2}$$

where $\mathbf{1}_{\{|u| \leq 1\}}$ is indicator function. We use the Tricubic function with values of $u \in [0,1]$ where the center position c has $u = 0$ and the furthest positions of the window have $u = 1$. Moreover, some filtering methods use uniform weighting, i.e. $W(u) = 1$.

We define the set Ω_c as a set of pixels within a window which contribute to the creation of the regression model. The window has nature position of square due to the chosen decreasing weight function and easy data manipulation. Important part is to determine the appropriate size of this window for the local regression well, while a knowledge about the number of features is known.

Because choosing the right size of the window is fundamental, we dedicate the whole Section 2.3 to this problem. We also describe a method based on an adaptive size of the window which creates more windows for one position and chooses the one which fits the input data the best, there.

Nevertheless, we split up the pixel set within the local window into two distinct groups. One group, let us call it learning set, is strictly aimed for the regression model creation, which is being described here, and the second group, let us call it validation set, is strictly aimed for a validation of the model. The model validation reveals a rate of model inaccuracy – how well the model fits the input data. We use the validating process for choosing the proper window size (the best model), which we discuss in Section 2.3.2.

In case of static window size, the whole window is used as learning set leaving the validation set empty since the validation is never used in this static window size method (Sec. 2.3.1).

The local approximation of an unknown occurrence function $f(\mathbf{x})$ is done with a low order polynomial in a feature space $\mathbf{x} \in \mathbb{R}^D$. This approach can be seen as counter-intuitive, since the unknown function has discontinuities and therefore is typically non-linear. Nonetheless, observations of textures tell us that the non-linearity in occurrence map space can be well approximated by the local linear model in high dimensional features space (Moon et al. [2014]). These observations are valid for textures in rendering context but we assume that they can be transfered to our problem and our data. Each function can be well linearly approximated from some point when the local area shrinks. Other alternatives, such as non-linear regression, were taken into consideration but *"the local linear regression strikes an excellent balance between denoising quality and computational efficiency, while preserving feature edges well."*(Moon et al. [2014]).

Let us write n for the size of the set Ω_c^+. The model estimation by solving least squares minimization (Equation 1.1) has following closed form solution:

$$\dot{\boldsymbol{\beta}} = (\dot{\mathbf{X}}^{\mathbf{T}}\mathbf{W}\dot{\mathbf{X}})^{-1}\dot{\mathbf{X}}^{\mathbf{T}}\mathbf{W}\mathbf{y},$$

where $\dot{\mathbf{X}}$ is $n \times (D+1)$ design matrix, whose i-th row is the feature vector $\dot{\mathbf{x}}_\mathbf{i}^{\mathbf{T}}$, and vector $\mathbf{y} \in \mathbb{R}^n$ is build from the occurrence map values, y_i. The matrix $\mathbf{W}$ is a $n \times n$ diagonal matrix, whose diagonal elements are defined by the weight function $w(i, c)$.

The model estimation problem as stated above works with features vectors defined in a global space $\mathbb{R}^D$. However, from multiple reasons mentioned in this chapter, such as highly correlated features or high cost of computation, we do not work exactly in D dimensional features space. We utilize truncated singular decomposition (tSVD) instead to solve our model estimation problem in a reduced dimensional features space.

1.4.3 Features dimensionality reduction

Since our features do not satisfy well assumptions for common local regression models, we construct a reduced features space with a help of truncated singular value decomposition to solve the Equation 1.1. Definitions of both the SVD and tSVD are described in Section 1.2.2.

Before solving the minimization problem to estimate the regression model we must use the TSVD to create reduced feature space which is locally defined in each local window. Then all used feature vectors $\mathbf{x} \in \mathbb{R}^D$ in the global space are transformed into feature vectors $\mathbf{z} \in \mathbb{R}^k$ in a reduced space. The least squares are then solved in the reduced features space and we receive a local linear regression model for the current local window also in the reduced space.

The coordinate transformation provided by SVD transforms features vectors into space with different basis vectors. These new basis vectors are linearly combined with some of the original basis vectors. It means that the transformation creates new feature types which are result of linear combinations of original feature types. Moreover, the SVD orders these new feature types by importance,

and reveals a correlation between original features. The new feature ordering by importance is the core for the following dimensionality reduction.

Let us map the SVD to our problem and denote the $n \times D$ features design matrix by $\mathbf{X}$, whose i-th row $\mathbf{x_i}$ is a feature vector which is consisted of all D global features. We expect $n >> D$. The application of SVD to $\mathbf{X}$ leads to a factorization form as follows:

$$\mathbf{X} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^\mathbf{T},$$

where $\mathbf{U}$ is an $n \times n$ unitary matrix of left-singular vectors and $\mathbf{V}$ is a $D \times D$ unitary matrix of right-singular vectors. The matrix $\mathbf{\Sigma}$ is $n \times D$ rectangular diagonal matrix with non-negative real numbers σ_m, known as singular values.

The key property of singular values is that they are ordered as $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_r > 0$, where r stands for rank of $\mathbf{X}$. Thus, if 0 appears on the diagonal, some of features are perfectly correlated. Small singular values sign high correlation know as matrix multicollinearity or can be a result of corruption from noise. These small corruptions can significantly affect the least square solution. To solve the multicollinearity and noise problem, we apply tSVD (Moon et al. [2014]).

The matrix $\mathbf{X}$ is approximated with truncated matrix $\bar{\mathbf{X}}$, which can be expressed by compact TSVD form as:

$$\mathbf{X} \approx \bar{\mathbf{X}} = \mathbf{U_k}\mathbf{\Sigma_k}\mathbf{V_k^T},$$

where $\mathbf{U_k}$ and $\mathbf{V_k}$ are $x \times k$ and $D \times k$ reduced unitary matrices respectively, which were created from original $\mathbf{U}$ and $\mathbf{V}$ by keeping only first k columns. The matrix $\mathbf{\Sigma_k}$ is $k \times k$ diagonal square matrix, with first k non-zero singular values of $\mathbf{\Sigma}$.

The proper choise of k

The approximated matrix $\bar{\mathbf{X}}$ highly depends on the right choice of k. We would like to reduce matrix dimensional space but with minimal loss of information.

Intuitively, we can set k to match the rank r of the matrix $\mathbf{X}$. With this presumption, the TSVD is an exact factorization of the original matrix $\mathbf{X}$, where SVD matrices are dimensionally smaller but $\mathbf{X}$ is without any information loss. The "$k = r$" reduction brings a performance boost to our minimization problem (Eq. 1.1) but only in case that some of features are perfectly correlated, which we do not really meet if features are chosen wisely. Any such wise choise would in particular include choosing two or more different but never the same sources. In reality we have never perfect correlation but a high correlation and the noise corruption. In this case, the singular matrix $\mathbf{\Sigma}$ includes some small singular values and we choose k in way that $k < r$. The singular values property, i.e. $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_k \geq \cdots \geq \sigma_r > 0$, guarantees that the first k singular values are bigger than or equal to the discarded ones. The singular value σ_i directly relates to the informativeness of the corresponded i-th new dimension. High singular values determine dimensions of the new reduced space where examples have more variability and therefore more informativeness. On the other hand, small singular values correspond to dimensions whereas examples have smaller variability and therefore smaller informativeness. These dimensions corresponding to small singular values can be then hardly used as efficient features for learning.

"The possibility of computing approximated versions of matrices gives a powerful method for feature selection and filtering as we can decide in advance how many features or, better, linear combination of original features we want to use" (Francesca and Zanzotto [2009]).

We have considered more approaches how to determine k. Probably the simplest method is to set k hardly but it has no regard to a possible high importance of more then k new features types. Most of common approaches use a threshold value, where all singular values smaller then this threshold are cut off. The thresholding handles the problem of a changeable count of the important new features. The threshold can be set hardly, e.g. in ratio $4/\sqrt{3}$(Gavish and Donoho [2014]) or can be adaptive. The adaptive thresholding belongs to a family of methods which determines the value of k with respects for the features variability. Witten et al. [2009] leave out random subsets of the entries in the data matrix, measure the differences between the fitted values and the original values for those entries, and choose the threshold levels that minimize the differences. Other commonly used approaches include various forms of cross-validation (Owen and Perry [2009], Lee et al. [2010], Yang et al. [2014]). Another example of method for finding k is based on perturbation theory (Moon et al. [2014], Hansen [1987]), which unfortunately needs a closer knowledge about the noise distribution.

We are using a method based on a criteria how much energy can the singular matrix loose (Rajaraman and Ullman [2011]). Let us define the *energy* of the singular matrix $\mathbf{\Sigma}$ as a function:

$$energy(\mathbf{\Sigma}) = \sum_{i=1}^{r} \sigma_i,$$

where σ_i are the singular values of $\mathbf{\Sigma}$. In our method we cut off singular values one by one and compare an energy ration between original singular matrix $\mathbf{\Sigma}$ and the actual reduced singular matrix $\mathbf{\Sigma_j}$ with j remaining singular values. The cutting off stops when the energy ratio between $\mathbf{\Sigma}$ and $\mathbf{\Sigma_j}$ would fall below a threshold τ. *"A useful rule of thumb is to retain enough singular values to make up* 90 % *of the energy in* $\mathbf{\Sigma}$*"*(Rajaraman and Ullman [2011]) therefore we set $\tau = 0.9$. The amount of kept energy τ is a variable in our algorithm. Our observations showed that values higher than 0.9 can be used accompanied with presenting satisfiable results with preservation of significant decrease of the dimensionality (Sec. 4.5.3).

A features vector projection

The earlier mentioned feature transformation into local reduced feature space can be taken as a projection into the k dimensional space defined by columns of $D \times k$ unitary matrix $\mathbf{V_k}$ as all its vectors are orthonormal – thus are new basis. This projection can be formulated as follows:

$$\mathbf{z_i} = \mathbf{V_k^T x_i},$$

where $\mathbf{z_i} \in \mathbb{R}^k$ stands for projected feature vector $\mathbf{x_i} \in \mathbb{R}^D$, which was projected from D dimensional global space into k dimensional local reduced space.

Using these projected features vectors $\mathbf{z_i}$ in our optimization problem (Equation 1.1) leads to its modification:

$$\min_{\dot{\boldsymbol{\beta}}} \sum_{i \in \Omega_c^+} w(i,c) \cdot (y_i - \dot{\mathbf{z}}_\mathbf{i}^\mathbf{T} \dot{\boldsymbol{\beta}})^2,$$

where $\dot{\boldsymbol{\beta}} \in \mathbb{R}^{k+1}$ stands for the coefficients of the regression model in the k-dimensional reduced features space. The vector $\dot{\mathbf{z}}_\mathbf{i} \in \mathbb{R}^{k+1}$ has form of $[1, \mathbf{z}_\mathbf{i}^\mathbf{T}]^T$. We can also write this minimization problem in closed form:

$$\dot{\boldsymbol{\beta}} = (\dot{\mathbf{Z}}^\mathbf{T} \mathbf{W} \dot{\mathbf{Z}})^{-1} \dot{\mathbf{Z}}^\mathbf{T} \mathbf{W} \mathbf{y}, \tag{1.3}$$

where $\dot{\mathbf{Z}}$ is the $n \times (k+1)$ design matrix projection whose i-th row is $\dot{\mathbf{z}}_\mathbf{i}^\mathbf{T}$.

By projecting original design matrix into reduced features space and by solving the optimization problem in reduced features space, we retrieve a local regression model which is also in the reduced features space. This model is specifically defined for the window Ω_c with c as the window position. With an assistance of this constructed model we can now perform predictions on projected features to get a smoothed occurrence map — to refine the occurrence map.

1.4.4 From regression models to the surface map

Finally, let us look how to utilize the regression models to refine given occurrence maps and then how to merge them into resulting surface map. Let us summarize all available resources first. On the one hand, we have the coarse surface map which is divided into C different occurrence maps, accordingly to each class. The resolution of the occurrence maps is $m \times n$. On the other hand, we have list of features each of which has $M \times N$ resolution forming the final resolution for our surface map refinement. Let D stand for the features count.

Within one occurrence map we must proceed position-by-position in the final resolution and construct a relevant regression model for each position – or better said models, in case of adaptive window method, where we then use the best fitting one (Sec. 2.3.2). The models are strongly dependent on the size of the local window around a position p which is being processed. Section 2.3 shows us how to choose proper size of the window.

Within coverage of this local window, $n \times D$ features design matrix is constructed and immediately projected into reduced dimensional space, thus $n \times k$ local design matrix $\mathbf{Z}$. Note that n stands for the count of within the window used feature vectors, which are called predictor vectors in the regression terminology.

Thereafter for each model, we construct n-dimensional regressand vector $\mathbf{y}$ from the local window cut-out of the occurrence map. The occurrence map cut-out corresponds to local window area even if different resolution for occurrence map and features maps are used. That means that more values are taken from one field of occurrence map by way that both the y_i and $\mathbf{z_i}$ must correspond to the same absolute position i.

As the last thing before everything is set-up for estimation of the linear model, the diagonal $n \times n$ weight matrix $\mathbf{W}$ needs to be created. Weight values are counted by weight function (Eq. 1.2) according to relative positions within the local window. Then the weight values are put onto weight matrix diagonal in

the same position order as we did for both the regressands in regressand vector and the predictor vectors in design matrix.

Finally, with the presence of the regressand vector $\mathbf{y}$, the reduced design matrix $\mathbf{Z}$ and the weight matrix $\mathbf{W}$, the estimation of unknown coefficients for local linear model can be achieved by solving Equation 1.3.

The smoothing of the response value y_i on the relevant i-th position of the local window is realized by applying the prediction on the reduced feature vector $\mathbf{x_i}$ from relevant i-th position. Let us denote this smoothed occurrence value by $\bar{y}_i$. After such denotation from the definition of linear regression, the model prediction is a dot product function f as follows:

$$\bar{y}_i = \dot{\mathbf{z}}_\mathbf{i}^T \dot{\boldsymbol{\beta}} = f(\dot{\mathbf{z}}_\mathbf{i}), \tag{1.4}$$

noting that $\dot{\mathbf{z}}_\mathbf{i} \in \mathbb{R}^{k+1}$ stands for $[1, \mathbf{z_i^T}]^T$ and model coefficients vector $\dot{\boldsymbol{\beta}}$ includes intercept term as its first element. Be aware that the prediction uses model is defined in the reduced local space and therefore the used predictors must be also from the reduced space.

Any prediction within the local window can be done but the most important one relates to the center position of the window. This center position of local window clearly corresponds to position p, which is being processed. On that account the resulting refined occurrence map can be constructed from the predictions as soon as all positions of the $M \times N$ output resolution are processed.

In comparison with the base occurrence map of some surface type, the refined occurrence map does not involve only 0 and 1 values but results from the models predictions which can be, technically speaking, any real numbers. Most of the predicted values fit between 0 and 1 but it is not a rule. This artifact can be caused by the fact that the regression smoothly fits a parametric function through some samples. Furthermore the function is linear in our case. To predict values into $[0, 1]$, generalized linear model (GLM) can be used which transform the linearity of dependent values according to given function (Bishop [2006]). But we do not use GLM because in the end we do not need to fit the values into the $[0, 1]$ since we use the values from occurrence maps just to compare between each other. We would need the GLM in case that real probability interpretation would be needed. More about this problem and the occurrence map interpretation is discussed in Section 2.1.

The final synthesis of the resulting surface map is now simple, since we have the refined occurrence map for all surface types. To process a position p of the output resolution, we must only write a number of the most probable surface type standing on that position into the output surface map. The most probable surface type is a type the value of which is the highest at the position p of the refined occurrence maps.

This method creates the surface map in resolution of the given occurrence maps which is exactly what we wanted to achieve at the beginning of the whole process. Thus the refinement is done. A pseudo-code for our overall algorithm is provided in Algorithm 1.1.

Algorithm 1.1: A pseudocode for our overall algorithm.

```
1  input: surface map, height map
2  output: refined surface map
3
4  begin
5    Split up the surface map into occurrence maps o_i (Sec. 1.4.1)
6    Create additional features from the height map (Sec. 1.4.1)
7
8    foreach position p in final resolution do
9    {
10     // In case of static window, the window is only one
11     foreach local window w surrounding the position p do
12     {
13       Create reduced features f_w within the local window w (Sec. 1.4.3)
14       foreach occurence map o_i do
15       {
16         Create the linear model for f_w and o_i within the w (Sec. 1.4.2)
17       }
18     }
19
20     foreach occurence map o_i do
21     {
22       if case of adaptive window method then
23       {
24         // The created models are validated and the one with the best results is chosen
25         Choose the right window w̄ according to the validation (Sec. 2.3)
26       }
27       Predict the value for the middle point of chosen window w̄ (Sec. 1.4.4)
28       Write the predicted value into particular position p of refined o_i (Sec. 1.4.4)
29     }
30   }
31
32 // Note that in case of adaptive version with random validation/learning set distribution,
33 // we must create more occurrence maps o_i and average them
34
35   Synthesize the refined surface map from the refined occurrence maps (Sec. 1.4.4)
36 end
```

2. Additional approach details

This chapter includes additional explanations for issues which are not completely solved in the previous chapter.

2.1 Occurrence maps interpretation

In Section 1.4.4, we mentioned, how to refine occurrence maps with the aid of the local linear regression. There are coarse occurrence maps as an input. By using the local linear regression we are able to refine these maps into much finer resolution. The occurrence maps consist of values either 1 or 0, where these true-false values has meaning of presence of given surface type at corespondent places. This meaning is well defined. On the other hand, the output refined maps are made up of regression models predictions. Here, the meaning is blurred, since the values can generally become any real number.

Intuitively, we can consider the value of an occurrence map to be a probability of an event, which determines if a given surface type is present at a given position. Then the occurrence maps are a probability distribution for each surface type. Based on this information, let us have a look at the refining problem as at a redistribution of the base probability distribution.

From the decomposition of the original coarse surface maps, it looks that at one position can happen only very one event, which determines the surface type presence. However, it is a result of some classification problem where more types are present in the same cell. The MODIS classification method simply chose the most dominant type and assigned it into the related cell. Our method tries to find the original fine grain distribution of all possible types within the cell. The predicted value says how is probable that given class is present in given cell.

One huge imperfection of this point of view is that the prediction realized by the linear model can shoot values out of the $[0, 1]$ scale. It does not happen often, but it happens. The first offered solution to solve this inconvenience is to project the predicted values into the wanted interval. To transform a sample x, we can use specific form of logistic function f which project real number into $[0, 1]$ interval, defined as follows:

$$f(x) = \frac{1}{1 + \mathrm{e}^{(0.5-x)}}.$$

Unfortunately this approach is not really convenient either since the model coefficients are gained on condition that there is linear dependency between the coefficients and response variables in the model. The logistic transformation adds non-linearity between them. The model must count with the non-linearity already in the moment of the model creation. Thus, we cannot use ordinary linear regression model for this probability perception. The suitable solution is an usage of generalized linear model (GLM) which is flexible generalization of the ordinary linear regression and allows the dependent variables to have arbitrary distribution and also allows the dependent variable not to be exactly linearly related to the predictors (Bishop [2006]). More especially, the model assumes that an arbitrary function of the response variable varies linearly with the predicted values,

rather than that the response itself must vary linearly. This function is called link function and must be provided for the model creation. We can use Logit[1] function as the link function to cover the wanted distribution.

However in our main algorithm, we can leave this problem out of any consideration. At every position of the surface map should be the most probable surface type. There is no need to adapt our algorithm to probability theory because we do not need to know an exact probability, we only need to know the most probable surface type. The models predictions implicitly order the types by a comparison of their results. Therefore as the most probable type can be taken the one with the highest result of the prediction.

For our practical usage we discovered that only one class with the most significant prediction is needed. However, if anyone would want to use the probability maps as the predicted data, then this simple method cannot be used and GLM should be used instead. An example of pure probability maps usage can be that the plants can be placed into scene directly according to their occurrence probability, not according to chosen surface class (ecotope).

2.2 A motivation for the features selection

An approach introduced by Hammes [2001] used DEM character (elevation, slope, aspect etc.) to determine rules for placing vegetation. The rules were defined in different ecotope types, therefore his method was basically to find suitable ecotope type for given location according to DEM character. It was successful method but the disadvantage of this approach was that the dependencies between DEM character and chosen ecotope were figured out ad-hoc. We took the idea of assigning an ecotope to some area according to DEM character but we use the coarse surface map and the regression to discover such dependencies. His ecotopes have the same meaning here as our surface types.

Other features then the DEM character can be used for the surface type prediction. We use, similarly as Hammes [2001], only features derived from DEM because of availability and the fact that the features should be close to the fine resolution (for model construction). More complicated features, such as average temperature or rainfall, usually do not satisfy these demands therefore are not much suitable for our case of study.

Another motivation for using only DEM character is that our method can be used in the future to fully synthesize geotypical landscape. The dependencies can be learned and then applied to the generated terrain which is pure DEM.

From our problem definition, we use the height map as our leading feature. It is intuitive that absolute elevation affect the distribution of surface types. Apart from the height map we can use any other feature, which can bring some dependency information, as an additional feature. An usage of additional features is not essential but results show an improvement in comparison with usage of only the height map as a feature. The result is significantly affected by the features selection and the learning data set (McPherson et al. [2006], Bombi and D'Amen [2012],Keil et al. [2013]). In some cases, an expert knowledge could help, but in other cases could harm (McPherson et al. [2006]). All used feature maps are

[1] `http://mathworld.wolfram.com/LogitTransformation.html`

described in Section 1.4.1.

As the first additional feature we mention a relative elevation map. The assumption and reason why we use this feature is that relative elevation reveals nature phenomenon such as valleys. As we know, a valley is a geomorphological shape which was formed usually by impact of river flowing. Of course, the river does not have to be always the cause, for example we mention a glaciation or tectonic processes. Anyway, the vegetation typically stays the same along the valley and changes with a distance from the valley. Moreover, usually there runs a river in the middle. Also weather conditions can affect surroundings of the valley and cause different surface types. These assumptions and easy computation lead us to use the relative elevation map as a fully-fledged feature (Hammes [2001]). The calculation is done by using a neighborhood to calculate a mean values around map points. Note that a size of this neighborhood significantly affects created result.

The second mention feature is a slope map. We expect that a strength of slope, to a certain extent, affects the surface type distribution. One example is that trees will rarely grow on places where the slope is 80°. This feature can reveal this dependency even if commonly does not contribute on the result much. It can be easily computed from the height map and the linear regression filters the slope map out (Sec. 1.4.2) at places without any contribution, therefore we use the slope map as a feature, too (Hammes [2001]).

An additional feature which reveals strong dependencies is an aspect map. There is strong dependency between an aspect and the surface types distribution because the aspect has an influence over the sun shining and also weather conditions, thus different temperature of the ground. An aspect map usually brings a big gain to the result surface map (Hammes [2001]). Note that flat areas do not have an aspect value well defined. Usually it is 0° or 360°. This strongly depends on the aspect calculation implementation. Therefore this can sometimes confuse the statistical model which needs to be taken in consideration.

Last feature, which we use is information about a position. We call it a coordinate feature. The idea behind an usage of a position information is that different latitude carries different climate. Also the fact if a location has continental or maritime climate depends on a location. This observation leads us to use the coordinate feature. This motivation was taken in a globally scale and since we apply only local observations it does not have so big importance. On the other hand, it can reveal some dependencies in case of some distinct object, natural phenomenons, significant borders or other anomalies. One example of such an anomaly is caused by men – a logging of rain forests. Strong difference can be observed on the edge of a logged area. Therefore we decided to keep the coordinate feature in our approach.

As we mentioned, for our approach can be used many other various features which would improve the result of the refinement. A good feature should have strong relation with some surface types and be able to differ them. For example, a ground moisture map or a humidity map seem to be potentially good candidates, also as a land surface temperature. Another features can be for instance vegetation index maps, burned area maps, surface reflectance maps or orthophoto maps. Since our workflow is adapted to reduce the feature dimensional space, the amount of features is not a real obstacle for our method.

2.3 A local window size

In our approach, we create linear regression models based on a neighborhood of some center position p (Sec.1.4.2). This allows us to process large areas locally. We use square window to define such neighborhood. We discovered that the size of this local window has huge impact on linear model estimation and thus the result image. The choice of the right window size is then a key problem for our method. We offer two different approaches as a solution. The first one, which we use in our base algorithm, is using a static window size for the whole picture, the second one selects the window size dynamically according to an accuracy of corresponding local linear regression models.

2.3.1 A static window

The approach for a static window keeps the same window size along the processing of the whole map. Since the size is constant for the whole process, it must be chosen at the beginning of the algorithm and the determining of the right size is crucial (Sec. 4.3)). The bigger window is used the more global information is used for the creation of the regression model. It can seem right to receive wider information but it can quickly happen that the model becomes under-fitted. That too much information is used and the model does not fit well the surroundings any more. On the other hand, the smaller window is used the more local information is used. In this case we can easily encounter an information deficiency, which causes the wrong model fit as well.

Let P denotes approximately how much bigger distance represents a side of one pixel at coarse resolution in comparison with a side of a pixel at fine resolution. An example visualization of the P-value is shown in Figure 2.1. The P is determined as follows:

$$P = max\left(\frac{w_r}{w_{nr}}, \frac{h_r}{h_{nr}}\right),$$

where w_r and w_{nr} stand for the width of the refined and the not-refined resolution, respectively, and h_r and h_{nr} denote the heights, likewise.

Let $Coef$ stand for coefficient which multiplies the P, then the local window size is determined by this multiplication and is expressed in the refined resolution.

$$local\,window\,size = (P \cdot Coef) \times (P \cdot Coef).$$

The Figure 2.1 also shows an example of window size determination by $Coef$.

Our tests showed that the optimal choice of the static window size is with values of $Coef$ between 3 and 4 (Sec. 4.3). Through observation, we learned that too small coefficients simply copy the input and that to big coefficients wipe out sparse occurrences.

An advantage of this method is the computation speed and the algorithm simplicity.

A disadvantage is that the window size can not be determined adaptively therefore the models fit data somewhere better and elsewhere worse. Accordingly to how rapidly the predictors and input data change there.

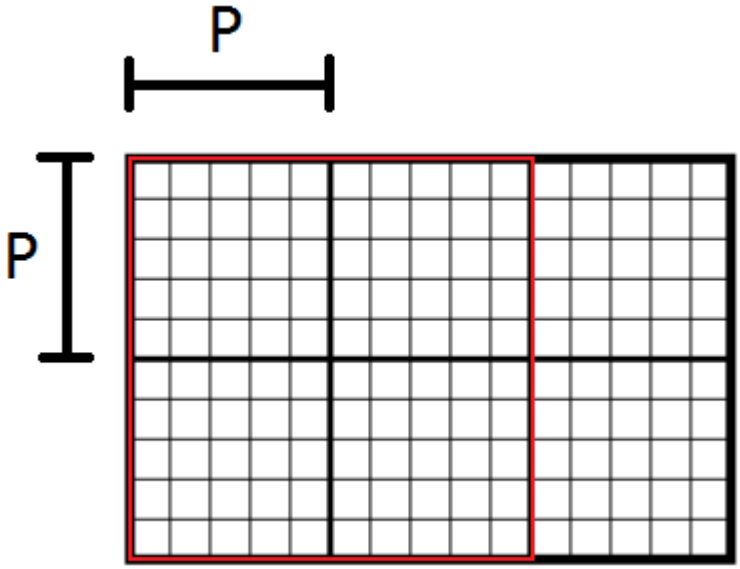


Figure 2.1: **A visualization of local window size.** The coarse resolution is represented by thicker black lines, the finer one with thiner black lines. The side of the pixel at coarse resolution is five times longer then the side of the pixel at finer resolutions therefore $P = 5$. The red square window is then determined by $Coef = 2$ $(2 \cdot 5 = 10)$.

2.3.2 An adaptive window

We provide a method to detect the appropriate size of the local window as an extension to our base algorithm. The window in this approach extension changes its size according to the best neighborhood for the model learning.

Each window size defines different regression model with different predictions. We validate these predictions and estimate an accuracy of each model. The model which is signed as the most precise is then used for the final prediction of the position in the middle of the model window.

Let us have a local window Ω_c which surrounds a specific position c. For the purpose of the validation, we split up the items within the window into two groups with different purposes. The purpose of the first group is to determine the regression model, contrary to the second group purpose which is to serve the validation. We call the first group a learning set and denote it by Ω_c^L. The second group is then called validation set and we write Ω_c^V for this set. It holds that $\Omega_c^L \cup \Omega_c^V = \Omega_c$ and $\Omega_c^L \cap \Omega_c^V = \emptyset$.

A linear regression model validation

A validation is a process to reveal a rate of a model inaccuracy. We use it to choose a model which fits the best given data. This model is then used for the prediction of the position p, which is being processed.

In general, we want to minimize the bias and variance of a model. As an quality assessment of the model fitting we use an error value which relates to the intensity of bias and variance in the model. Therefore we then choose the model with the smallest error value (Geman et al. [1992], Kutner et al. [2005]).

We offer different error types (defined below) which contain the bias and

variance information. Let y_i stand for observed value at i-th position, $\bar{y}_i$ for predicted value for i-th position and Y for mean of observed values.

Mean squared error (MSE) is defined as follows (Kutner et al. [2005]):

$$\mathrm{MSE} = \frac{1}{|\Omega_c^V|} \sum_{i \in \Omega_c^V} (y_i - \bar{y}_i)^2.$$

Weighted mean squared error (wMSE) is defined as follows:

$$\mathrm{wMSE} = \frac{1}{|\Omega_c^V|} \sum_{i \in \Omega_c^V} (w_i(y_i - \bar{y}_i))^2,$$

where w_i is a weight for i-th position. This error estimation takes farther samples as less important.

Coefficient of determination ($\mathbf{R^2}$) is defined as follows (Kutner et al. [2005]):

$$\mathbf{R^2} = 1 - \frac{\sum_i (y_i - \bar{y}_i)^2}{\sum_i (y_i - Y)^2}.$$

Note that value of 1 indicates that the regression line perfectly fits the data. Lower value determines bigger error, therefore we maximize $\mathbf{R^2}$ instead of minimization. We do not use this error estimation because it is spuriously increasing when extra independent variables are added into model. Instead, we use adjusted $\mathbf{R^2}$ which corrects this phenomenon.

Adjusted $\mathbf{R^2}$ ($\mathbf{adjR^2}$) is defined as follows (Kutner et al. [2005]):

$$\mathbf{adjR^2} = 1 - (1 - R^2)\frac{n-1}{n-p-1},$$

where n is the count of samples and p is feature vector dimension (without the intercept term). In contrast to $\mathbf{R^2}$, the $\mathbf{adjR^2}$ takes account variable number of parameters in regression model. Similarly to $\mathbf{R^2}$, the bigger value determines better fit.

Results for usage of these error types can be found in Section 4.4.

A local window division

There are many ways how to split up the local window into validation and learning set. Different divisions have different impact on the result therefore we introduce and discuss few of them in this chapter. The splitting patterns are shown in Figure 2.2 However, static patterns create significant bias therefore the only one statistically right solution is usage of some random distribution. We use the static patterns only for speed and experimental reasons.

A fully learning set is the simplest case, where $\Omega_c = \Omega_c^L$ and thus $\Omega_c^V = \emptyset$. Obviously this case cannot be used for the validation, since no examples for validation exist. We are mentioning this just because we use it in our base algorithm with static window size, where as much as possible samples are needed for the model estimation, therefore all window items are assigned to the estimation.

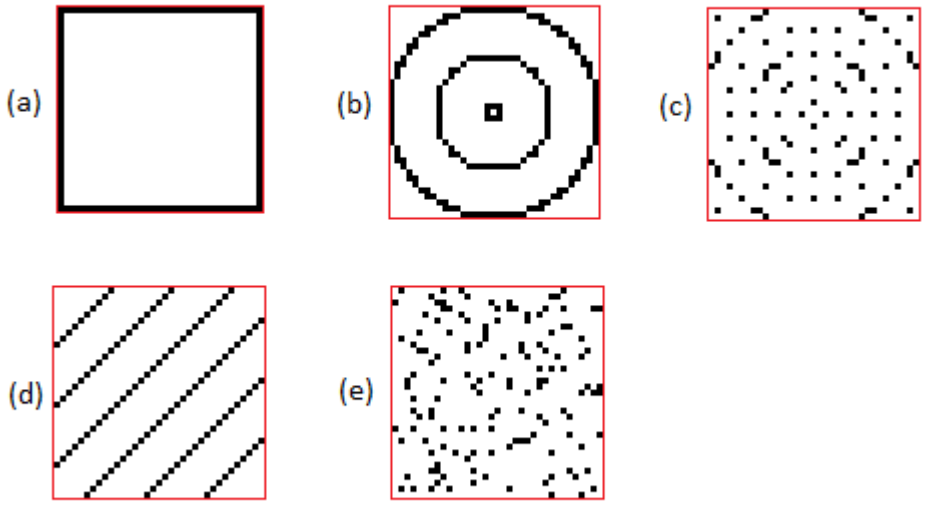


Figure 2.2: **Local window division patterns:** On this picture, we show patterns how we can split pixels of the local window into learning and validation sets: (a) the border separation, (b) homocentrical circles, (c) homocentrical dotted circles, (d) straight strips, (e) random distribution. Black dots correspond to the validation set, the white rest belongs to learning set. Note that the random distribution is different for each window and from statistical point is much better then the other showed distributions.

A border separation is a division, where all positions from the edge of the window belong to validation set and the rest belong to learning set. This distribution is used by Moon et al. [2015]. This approach expands the window without a recalculation of the whole window but recursive models calculation is used there according to the new window increment. The estimation of the prediction error is calculated recursively, too. They introduce the iterative error prediction mostly for performance reasons. We tested this divisions and discover that it does not serve well for our purposes.

Straight strips is the case of the window division, where the two sets are distributed into parallel lines. The validation strips are thiner then learning strips in a ratio approximately $1 : 9$. If only windows with strips division is used, the result involves artifacts in specific direction which are noticeable to human eye, therefore we do not use this method.

Homocentrical circles are very similar to the strips case except fore that the strips are not parallel lines but homocentrical circles. We use the validation-learning ratio as $1 : 9$, too. The circle approach corrects the direction artifacts thus is not noticeable by human eye any more.

Homocentrical dotted circles is an improvement of basic homocentrical circles, but shows better results.

Random distribution is a case where validation and learning points are set randomly. We use again the ratio 90 % for learning set and 10 % for validation

set. Note that this case is the only one statistically right but it brings significant amount of noise therefore more random windows must be created and then averaged.

3. Implementation

In this chapter we look closer at the actual implementation of our method. We describe some technical details, provide explaining diagrams. We also mention used third party software, design patterns and special coding techniques. First we implemented our method as a prototype in R which allowed us to rapidly develop and tune the method. Then we created optimized implementation using C++. The Section thus concerns an implementation of the prototype in R, the Section then discusses an the optimized implementation in C++ in detail.

3.1 A list of used tools and technologies

First we introduce references for tool and technologies which we used to accomplish the project.

R for Windows [1] is an environment for statistical computing which we used for our implementation in R.

Microsoft Visual Studio [2] (MSVS) is an integrated development environment which we used for our C++ implementation.

OpenMP [3] is an API, de-facto a standard for parallel programming. We use it in our C++ implementation to speed up the processing.

Libpng [4] a library which we use for loading and saving files in PNG format.

Eigen library [5] is a C++ template library for linear algebra. We use it in our implementation for matrices representation and linear algebra operations such as SVD or least-squares fit.

GDAL [6] is a geospatial data abstraction library which we used for a while for spatial data processing but we do not use this library in final implementation any more. However, this library is not-directly used by the R prototype through the `raster`[7] package.

QGIS [8] is an open source geographic information system. We used QGIS application for preparing and visualizing spatial data in the beginning of our research.

[1] `https://cran.r-project.org/`
[2] `http://www.visualstudio.com`
[3] `http://www.openmp.org/`
[4] `http://www.libpng.org/pub/png/libpng.html`
[5] `http://eigen.tuxfamily.org/`
[6] `http://www.gdal.org/`
[7] `https://cran.r-project.org/web/packages/raster/index.html`
[8] `http://www.qgis.org/`

3.2 A prototype in R

We started with an implementation in R on the fly when we were still composing the main algorithm. We chose the R language because it is high-level programming language which is aimed for a statistical computing and supports many functionality such as easy work with geographical data, graphics etc. R is easy to use within its software environment due a provision of very specific high-level functions which are distributed by goal specific packages. The packages are distributed commonly within the software environment, therefore a getting of a new package is really fast and simple. These were reasons for the usage of R. Other alternatives for the prototyping could be for example Python or MATLAB, which would provide a suitable environment too. In the following rest of this section, we describe special techniques available in R.

We represent most of the data as matrices which is common R type. The whole algorithm is written procedurally an basically we just apply special functions to sub-matrices of matrices, where the base matrices stands for input and output occurrence maps and features maps and where the sub-matrices has local window point of view to these base maps.

An important package for our algorithm is named `raster`. This package offers high-level functions for reading, writing, manipulating, analyzing and modeling of gridded spatial data. Rasters can be easily created from files of various formats or from other array data structures. Rasters have implemented a functionality for creating aspect and slope maps and can handle various projections. Due to rasters and base R function, a loading of input data and feature creation is really simple. We provide a sample code for a creation of aspect and slope feature from a HGT file (Listing B.1).

The core of our approach – the linear regression and the SVD – uses standard statistical R functions `lsfit` and `svd`, respectively. These high-level functions practically do the whole work for us, we just prepare particular local window data, which we apply these functions on. Example codes for linear model coefficient estimation and for features dimensionality reduction are shown in Algorithm B.2 and Algorithm B.3.

Any other workload is done by standard R functions – especially matrices operations, `mean` function and others.

For other pictures operation such as a saving of the output surface map, we use `png` and `pixmap` packages.

The adaptive window extension is not implemented in R version, but only in C++.

3.3 The optimized C++ implementation

A crucial disadvantage of the R implementation is its computation speed. A processing of one 1200×1200 image took approximately a day for R which we compute approximately 20 s in C++. Of course, many optimizations could be done but it is a fact that R cannot be faster then C++ for our purposes by concept, therefore we decided to do the final implementation in C++.

The optimized C++ implementation was developed for Windows using MSVS, however, the code should be portable to other platforms with minor modifications.

In this chapter, we look closer into the C++ implementation, describe more the important parts of the algorithm, provide class diagrams and introduce used libraries and special techniques of C++. We decided to use commonly known object oriented programming, therefore let us start with classes descriptions.

3.3.1 A description of used classes

class Raster

Class *Raster* represents real number values in a regular grid and represents the backbone of the whole program. We use this class to represent various data – both the refined and original surface map, occurrence maps, elevation maps and all the additional features. For some of named data, we use slightly different classes but most of them are inheritors or at least contain a raster objects or even objects. The base of Raster is shown in Figure 3.1.

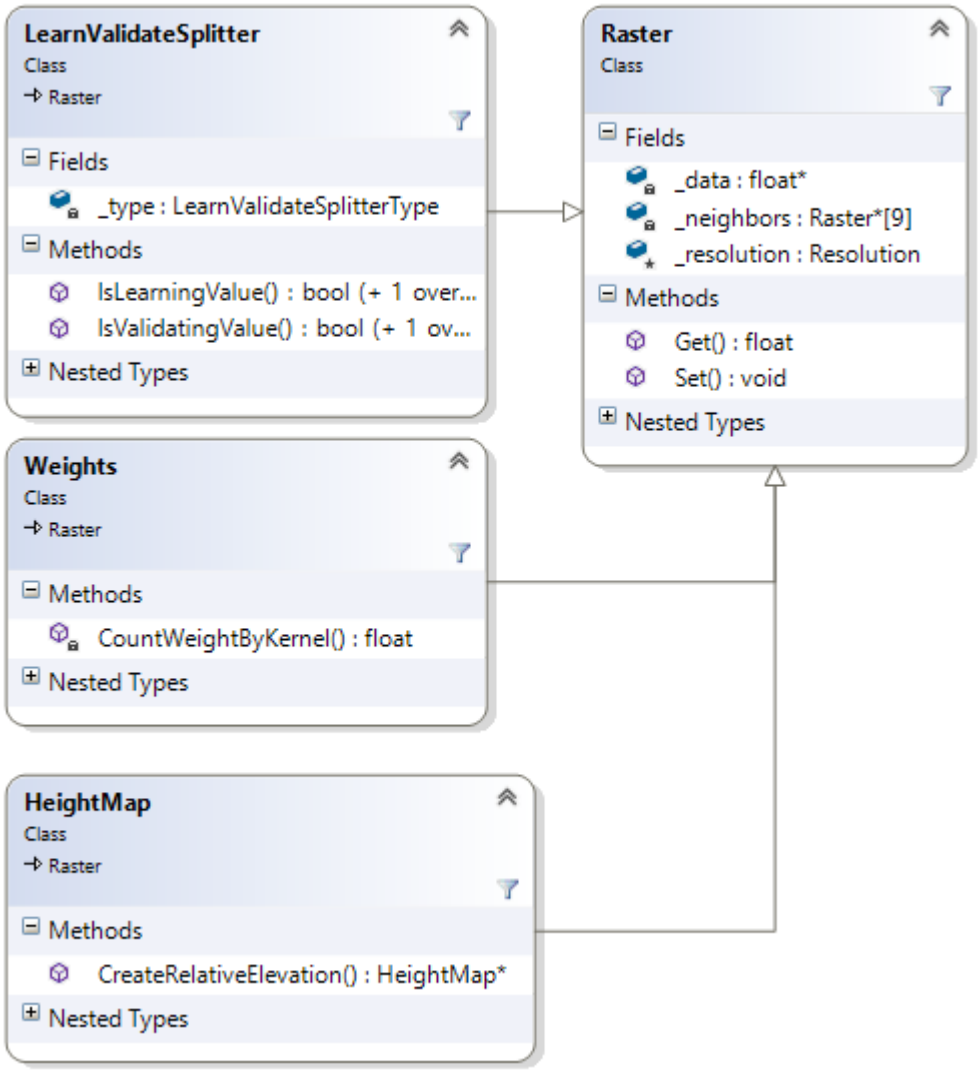


Figure 3.1: The core of Raster class and its derived classes

The Raster is a smarter container to hold two-dimensional data and to their easier accessing. Meaning that it provides both the data setting and getting by

both a position and a linear index.

Because we use local windows for local regression, we need to access out-of-border data while processing edges of a grid, otherwise we would not be able refine whole raster. We could extend the the grid by needed size, but we solve this problem little bit differently – we keep an information about current grid neighbors, which are also Rasters, and we access the neighbors data when we need a value of a position which is not in a current raster. We chose this solution due to easier data manipulation for batch files processing – specific rasters are associated with specific files therefore the neighboring rasters are also the neighboring files.

Raster also provides saving of if its content to PNG files. For operating with PNG files we use a third-party library called `libpng`.

class HeightMap

The *HeightMap* inherits from the *Raster* class and represents an elevation map. The HeightMap class provides a loading ability from both PNG and HGT file formats for a spatial files structure. The HeightMap has also ability to create a new relative elevation map from its data. We show the base diagram of the class in Figure 3.1.

class Landcover

The *Landcover* class is a container of rasters. Landcover represents a surface map, thus included rasters can be interpreted as occurrence maps for all surface types. Landcover is also used as a container for the output refined occurrence maps (pseudo-probability maps). This class can load a surface map with its logical neighbors from PNG files and categorize the surface types into its rasters. Important function for local regression is to get not whole grid but only a local section of wanted occurrence map – the function is called *CreateLocalWindow*. Lancover class structure is indicated with diagram in Figure 3.2

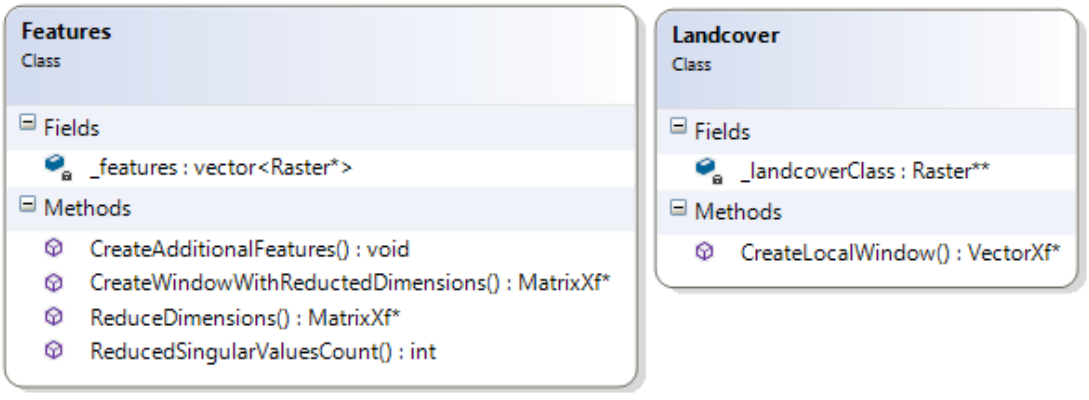


Figure 3.2: The base of classes utilizing Raster

class Features

The *Features* class is container for all features of our approach. The features are saved in Raster objects. As we know, we use the elevation map, relative the

elevation map, the aspect map, the slope map and the coordinates features. The first two mentioned features, i.e. elevation map and relative elevation map, is created by HeightMap class but all the rest is created within this Features class. Similarly to Landcover class, Features class can provide a local cut-out of whole grid. The local window of each feature is linearized and put into a features matrix. As discussed in Section 1.4.3 and Section 1.4.4 the dimensionality of the features matrix can be reduced, which is handled by *ReduceDimension* function of the Features class. The base class diagram is shown in Figure 3.2.

class Weights

The *Weights* class is a raster with values between 0 and 1 which determine weights of corresponding positions. We can choose specific kernel, which is set by *CountWeightByKernel* function, but as is said in Section 1.4.2, we use the Tricubic function (Equation 1.2). In program mode where we do not want to use weights, we just fill the weight matrix with 1. The class diagram is shown in Figure 3.1.

class LearnValidateSplitter

The *LearnValidateSplitter* is a raster which differs its values into two sets. Section 1.4.2 mentioned that we split up the local window into two sets with a different aim. Refer to detailed explanation to Section 2.3. We call the set with all position within the local window as Ω_c, which is centered around a position c. The window is divided into the learning set Ω_c^L and the validation set Ω_c^V. The Section 2.3.2 detailedly describes the division types. Our LearnValidateSplitter raster expresses the local window division. The raster include two special values which determine what set the corresponding position belongs to. Before a LearnValidateSplitter object is created we choose the type of distribution and then we create the splitter accordingly. After the position belongings are distributed, we use the splitter object to answer questions if a given position is aimed for validation or regression model learning. The basic class diagram is shown in Figure 3.1.

class LinearRegression

The *LinearRegression* class represents a linear regression model. The model estimation is done in a constructor according to provided regressors, regressands, weights and a splitter, where only points from learning set are used. The LinearRegression class keeps the information about model coefficients $\boldsymbol{\beta}$, which defines the linear regression model. According to provided predictors, a prediction can by done within this class by function *Predict*. The Figure 3.3 shows the basic class diagram. We provide details about the implementation of the model estimation and the prediction algorithms in Section 3.3.

3.3.2 A description of important methods

In this chapter we describe C++ implementation of important methods and point out few special techniques. Note that we use `Eigen` library for the algebra such as matrices, vectors, SVD, QR decomposition.

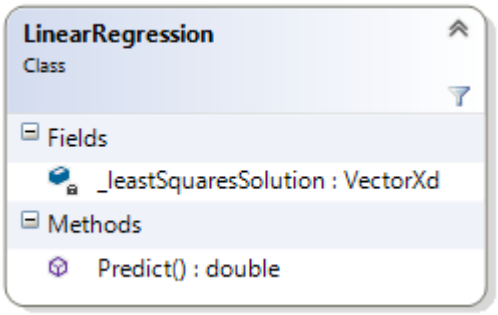


Figure 3.3: The base of LinearRegression class

The linear regression model estimation

They key part of our approach is to estimate models of linear regression for local windows. Assume that we want to count the appropriate model coefficients in the moment when design predictors matrix is already constructed and relevant regressand vector is provided, too. The weights are already counted by Tricubic kernel (Eq. 1.2) and the local window is already split into learning and validation sets (Sec. 2.3.2).

There is a special class named LinearRegression (Sec. 3.3.1) which is aimed for the model estimation (Sec. 1.4.2). The code behind the coefficient obtaining is nested in the LinearRegression constructor and shown in Algorithm 3.1. The coefficients are then kept in the *_leastSquaresSolution* member of LinearRegression object.

Algorithm 3.1: An estimation of linear regression model.

```
1  // Estimates the linear regression model
2  LinearRegression::LinearRegression(
3    const VectorXf& regressand,
4    const MatrixXf& regressors,
5    const Weights& weights,
6    // The splitter which determines learning positions, which are used for the estimation.
7    const LearnValidateSplitter& learnValidateSplitter)
8  {
9    // The count of all the points within the local window
10   int pointsCount = weights.GetResolution().GetItemsCount();
11   // All points from the learning set of the local window
12   int usedPointsCount = learnValidateSplitter.GetLearningValuesCount();
13   // The weighted matrix of regressors and intercept terms
14   MatrixXf weightedRegressorsWithIntercept(usedPointsCount, regressors.cols() + 1);
15   // Weighted regressand vector
16   VectorXf weightedRegressand(usedPointsCount);
17   // Current index of position in learning set
18   int learningSetIndex = 0;
19   // Traverses the local window and initializes the used weighted regressors matrix
20   // and regressand vector
21   for (int i = 0; i < pointsCount; i++)
22   {
23     // Uses only values from the learning set which is aimed for the model estimation
24     if (learnValidateSplitter.IsLearningValue(i))
25     {
26       // Modifies the regressand by weights
27       weightedRegressand(learningSetIndex) = regressand(i) * weights.At(i);
```

```
28          // Adds weighted "intercept term" into the zero−th column of the design matrix
29          weightedRegressorsWithIntercept(learningSetIndex, 0) = weights.At(i);
30          // Adds the weighted features vector behind into the design matrix
31          for (int feature = 1; feature < regressors.cols() + 1; feature++)
32          {
33            // Adds the separate features vector items into the design matrix
34            weightedRegressorsWithIntercept(learningSetIndex, feature) =
35              regressors(i, feature − 1) ∗ weights.At(i);
36          }
37          // Raises the index of learning set position to procces another one
38          learningSetIndex++;
39        }
40    }
41    // Solves the least−square fit
42    _leastSquaresSolution =
43      weightedRegressorsWithIntercept.colPivHouseholderQr().solve(weightedRegressand);
44  }
```

Note that the main problem — the weighted least-squares fit — is done by specific functions from Eigen library. The *colPivHouseholderQr* function counts the fit via QR decomposition of a matrix[9]. This overall model estimating problem we describe in Section 1.2.1. We have also considered to use Eigen *JacobiSVD* class for solving of the least-squares but results showed that QR decomposition is faster.

Features dimensionality reduction

Before we create an appropriate regression model using the LinearRegression class, we reduce the dimensionality of the design matrix. We use the tSVD for the reduction which is analyzed in detail in Section 1.4.3. The class which is responsible for features managing is analogously named Features. More details about this class are provided in Section 3.3.1. Features class has two important functions for a creation of the design matrix of a regression model which are *CreateFeaturesWindow* and *ReduceDimensions*. The *CreateFeaturesWindow* creates a data cut-out for a local window and order it into matrix, whose each column includes a linearized cut-out of different feature. In other words, each row of the matrix is a features vector at a particular position and the matrix has as many rows as the local window has its positions. The *ReduceDimensions* function then uses tSVD and reduces the column space of this matrix. We show the implementation of the features matrix dimensionality reduction in Algorithm 3.2.

Algorithm 3.2: Features dimensionality reduction.

```
1   // Creates a matrix projection into a reduced matrix space of given matrix.
2   MatrixXf∗ Features::ReduceDimensions(
3     // The features cut−out according to a local window.
4     // Each column relates to different feature cut−out.
5     // Each row is features vector relating to specific position of the local window.
6     const MatrixXf∗ window
7     )
8   {
9     // Creates the SVD of the current window matrix (window = U ∗ S ∗ V')
10    JacobiSVD<MatrixXf> svd(∗window, ComputeThinV);
```

[9] `http://eigen.tuxfamily.org/dox-devel/group__LeastSquares.html`

```
11    // The matrix of right singular vectors
12    const MatrixXf* V = &svd.matrixV();
13    // The singular values (the diagonal of S). They are always sorted in decreasing order.
14    const VectorXf* s = &svd.singularValues();
15    // Determines how many singular values should be used for TSVD
16    int reductedSingularValuesCount = Features::ReducedSingularValuesCount(s);
17    // Crops the V accordingly to reductedSingularValuesCount
18    const Block<const MatrixXf>* reducedV =
19      &V->block(0, 0, V->rows(), reductedSingularValuesCount);
20    // Projects the features matrix into reduced space.
21    // Projection of features vector is done by z = V' * x.
22    // Then all features ordered in row (transpose of window) can be projected as:
23    // Z' = V' * window' = (window * V)' => Z = window * V
24    MatrixXf* reducedWindow = new MatrixXf((*window) * (*reducedV));
25
26    return reducedWindow;
27  }
```

In the agorithm we use a function called *ReducedSingularValuesCount* which gives us a proper count of singular values used in tSVD. We choose the right value according to Section 1.4.3 where we describe the actual method – we keep an energy of the singular values matrix over a threshold value, which is defined by user. We have set this threshold to 0.9 which relates to 90 % of the matrix energy.

Note that in this algorithm, we use special Eigen functionality, too. Apart from vector and matrix operations, we use a special Eigen class *JacobiSVD* which is aimed for SVD.

The prediction with a model

We know, that the constructed regression model is represented by an instance of LinearRegression class (3.3.1). Many various predictions can be done with constructed regression model. The prediction formula is written in Equation 1.4. In Algorithm 3.3, we show our C++ implementation of the prediction function.

Algorithm 3.3: A prediction based on a regression model instance.

```
1   // Counts a predicted value according to the current model and given predictors
2   float LinearRegression::Predict(
3     // The vector of predictors. It does NOT include 1 at its first place (intercept term).
4     const Block<MatrixXf, 1 , -1, false>& predictors) const
5   {
6     // Intercept term of model coefficients (model_coefficients[0])
7     float interceptTerm = _leastSquaresSolution(0);
8     // The prediction value based on coefficients without the intercept term
9     float predictorsResult =
10      predictors.dot(_leastSquaresSolution.bottomRows(_leastSquaresSolution.size() - 1));
11    return interceptTerm + predictorsResult;
12  }
```

The model validation

While processing a position p and in case of adaptive local window, there the window size must chosen. In this moment we have a set of constructed regression models which corresponds with different window sizes. From this set of models,

we must choose the best one which is used for the prediction of the position p. The best model is the one which has the lowest prediction error value (Sec. 2.3.2). This value is counted by *CountPredictionError* function showed in Algorithm 3.4 (MSE and wMSE version).

Algorithm 3.4: **The models validation.** A model with the lowest error value is chosen for the prediction of the window center. This code shows MSE and wMSE error version.

```
1  float CountPredictionErrorMSE(
2    LinearRegression* linearModel,
3    Eigen::MatrixXf* regressors,
4    Eigen::VectorXf* regressand,
5    Weights* weights,
6    LearnValidateSplitter* learnValidateSplitter)
7  {
8    double predictionError = 0;
9    for (int y = 0; y < resolution._width; y++)
10     for (int x = 0; x < resolution._height; x++)
11       if (learnValidateSplitter->IsValidatingValue(x, y))
12       {
13         int index = y * resolution._width + x;
14
15         float prediction = linearModel->Predict(regressors->row(index));
16         float realData = (*regressand)(index);
17         double localError = (realData - prediction) * (realData - prediction);
18         // Modification for weighted MSE
19         if (Globals::validationFunction == Globals::ValidationFunction::wMSE)
20           localError *= weights->Get(x, y) * weights->Get(x, y);
21
22         predictionError += localError;
23       }
24   predictionError /= learnValidateSplitter->GetValidatingValuesCount();
25   return predictionError;
26 }
```

The main algorithm

We provide a core overview about surface map refinement in Algorithm 3.5. We refine one surface map according to given features by *RefineLandcover* function which is described in this algorithm.

Our program handles multiple files processing, which is done by finding corresponding elevation maps to given surface map files, loading their data and then calling the *RefineLandcover* function to the corresponding pairs. We will not introduce the files managing algorithms because it is not a core of our approach.

Algorithm 3.5: The general surface refinement algorithm.

```
1  // Refined the surface map according to given features
2  void RefineLandcover(
3    // It represents input occurent map
4    const Landcover& landcover,
5    // It represents all used features
6    const Features& features,
7    // Output refined surface map
8    Landcover& refinedLandcover)
```

```
{
  // Traverses all positions from output resolution and
  // estimate models and predictions for the positions
  #pragma omp parallel for
  for (int yPosition = 0; yPosition < Globals::outputResolution._height; yPosition++)
  {
    #pragma omp parallel for
    for (int xPosition = 0; xPosition < Globals::outputResolution._width; xPosition++)
    {
      // Creates linear regression models for all windows around current position
      for (int windowI = 0; windowI < WINDOWS_COUNT; windowI++)
      {
        // Actual resolution of processing window
        const Resolution& windowResolution = *windowSize_resolution[windowI];

        // Creates design matrix (with reduced features dimensionality)
        regressors = features.CreateWindowWithReductedDimensions(
          xPosition - windowResolution._width / 2,
          yPosition - windowResolution._height / 2,
          windowResolution);

        // Assigns the design matrix with particular window
        windowSize_regressors[windowI] = regressors;

        // Creates regression models for all surface types
        for (int sType = 0; sType < Landcover::GetClassesCount(); sType++)
        {
          if (landcover.ClassIsUsed(sType))
          {
            // Creates regressands -
            // a cut-out of the occurence map of particular surface type
            regressand = landcover.CreateLocalWindow(
              sType,
              xPosition - windowResolution._width / 2,
              yPosition - windowResolution._height / 2,
              windowResolution);

            // Assigns the regressands to particular surface type and window
            windowSize_landcoverClass_regressand[sType][windowI];

            // Gets weights for current window
            const Weights& weights = *windowSize_weights[windowI];
            // Gets the window splitter for current window
            const LearnValidateSplitter& learnValidateSplitter =
              *windowSize_learnValidateSplitter[windowI];

            // Creates the local linear regression model
            windowSize_landcoverClass_linearModel[sType][windowI] =
              new LinearRegression(*regressand, *regressors, weights,
                  learnValidateSplitter);
          }
        }
      }

      // Gets the best model and predict the output value for current position
      for (int sType = 0; sType < Landcover::GetClassesCount(); sType++)
      {
        // Gets the index of best model (window) of particular surface type.
```

```
66            if (ADAPTIVE_WINDOW)
67             selectedWindow = SelectWindow(
68               windowSize_landcoverClass_linearModel[sType],
69               windowSize_regressors,
70               windowSize_landcoverClass_regressand[sType],
71               windowSize_weights,
72               windowSize_learnValidateSplitter,
73               WINDOWS_COUNT)
74
75            // The chosen model
76            LinearRegression* chosenModel =
77              windowSize_landcoverClass_linearModel[sType][selectedWindow];
78
79            // The middle point index of the selected window
80            int middlePointIndex =
81              windowSize_resolution[selectedWindow]->GetItemsCount() / 2;
82
83            // The prediction for the current position (the middle of the window)
84            float prediction = chosenModel->Predict(
85              windowSize_regressors[selectedWindow]->row(middlePointIndex));
86
87            // Assigns predicted value to output refined occurence map
88            refinedLandcover.Set(sType, xPosition, yPosition, prediction);
89          }
90        }
91      }
92    }
```

Note that processing of different positions is not dependent to each other therefore we can process them in parallel. For the parallelism, we use OpenMP, as we can notice in the mentioned algorithm. When we use adaptive window with random validation/learning set distribution, we create more Landcover objects and average their values at corresponding position to avoid a noise.

4. Evaluation

In this chapter, we present the results of our method and discuss them. At first, we describe an evaluation method to compare results with expectations (Sec. 4.1). Afterwards in Section 4.2, we show how the method works on a given example. And after that, we show more of the real results with their validation (Sec. 4.3 and Sec. 4.4). At the end of this chapter, we point out and discuss special observed behavior of some parts of the approach (Sec. 4.5).

We applied our method on several locations in the North America, more especially United States of America. The reason for why we chose specifically North America is that the results validation process needs real data in finer resolution. Our method is aimed for MODIS Land Cover dataset which is available for the whole Earth but exists only at the coarse resolution. Nonetheless, US Land Cover, which covers United States of America, is very similar to MODIS Land cover and exist at finer resolution. Therefore we use US Land Cover for results validation.

4.1 Results validation method

By refining a coarse picture we want to get closer to reality therefore it is intuitive to compare the refinements with some real data. We use the US Land Cover data set which were already constructed for the finer resolution and therefore respond much better the reality. US Land Cover and MODIS Land Cover are very similar but use different classificatiors and different classification schema. Therefore we cannot compare them directly. Instead, we leave the MODIS data and we construct coarse surface maps from US Land Cover data set. This upscaling is done in a way that simulates the results of the MODIS dataset creation pipeline. It means that each coarse pixel includes a surface type which has the most significant occurrence within related area at the fine resolution. By this upscaling of US Land Cover we get coarse data which we refine using our algorithm and try to get the originals back. Then we compare the refined artificially constructed coarse data with the originals in fine resolution. This comparison determines an error value of the refinement, which we use as a measurement of success.

To compare two land cover pictures we use a relative difference. The error value Err has a meaning of how percentually different the two given land covers are and is defined as follows:

$$Err = \frac{1}{N}\sum_{i=1}^{N} f(p_i, \bar{p_i}), \quad f(p_i, \bar{p_i}) \begin{cases} 1 & \text{if } p_i \neq \bar{p_i} \\ 0 & \text{if } p_i = \bar{p_i} \end{cases}$$

where N stands for a total number of pixels from given surface map. Terms p_i and $\bar{p_i}$ are values on i-th position from original and rescaled surface maps, respectively. The function $f(p_i, \bar{p_i})$ is an indicator function and together with the sum they determine a number of differing values at same positions. The error value acquires values between 0 and 1 where 0 stands for identical pictures and 1 determines two completely different pictures.

This definition assumes that the two given surface maps have the same resolution. In case that they do not have, pixels from finer resolution need to be

compared with pixels at projected positions into the map in coarser resolution. The total number of pixels is then taken from finer resolution.

4.2 An example of the method application

Before presenting any particular result, let us we show a commented example of refining using area from Yellowstone National Park. In this demonstration, we use static window method where we have set the window size coefficient $Coef = 4$ (Sec. 2.3.1) and the threshold for features dimensionality reduction $\tau = 0.9$ (Sec. 1.4.3). In Figure 4.1, we can observe both the coarse surface map and the fine one we want to get close to. We refine the coarse distribution map according to features and try to get closer to the truth surface map. The used features

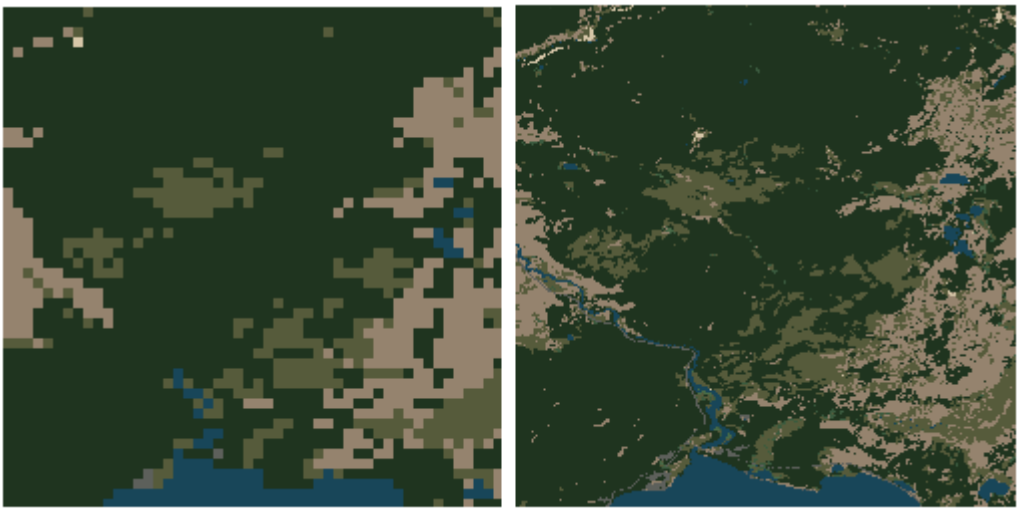


Figure 4.1: From left, the coarse surface types distribution and the truth type distribution for Yellowstone National Park.

are shown in Figure 4.2. By applying our method, we demonstrate a refined probability distribution of few surface classes in Figure 4.3. A meaning of each class showed in Figure A.1. The final refined surface map is then showed in Figure 4.4. We noticed that during the processing, the design feature matrix was reduced to contain only one linearly combined feature. The rest of the projected features were too weak to contribute the final result and were abandoned by the features reduction algorithm.

We measure an error value with the error function described in Section 4.1. The error value between the truth and the input coarse surface map is 0.19450. It means that 19.45% of pictures area is different. Our test showed that a common difference between coarse pictures and truth fine pictures is approximately 20%. The error between the truth surface map and the by-our-method-refined map is 0.19985. We can see that we did not come closer to the truth map. We even went little bit farther, but not too much. However, the result is more plausible for human eye, since the coarse grid is not so much recognizable any more. Let us now introduce more results and discuss their potential.

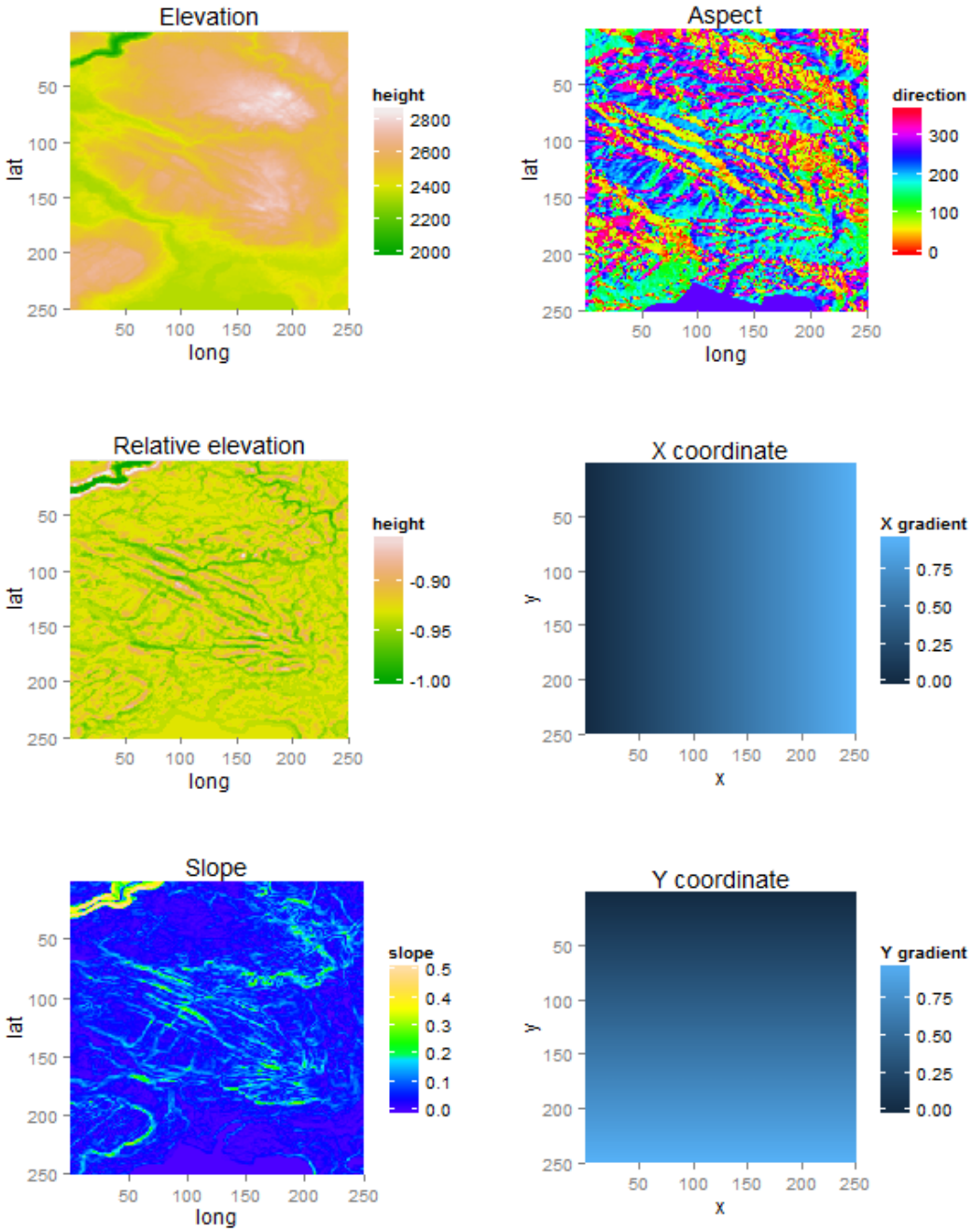


Figure 4.2: The used features visualization for Yellowstone National Park.

4.3 The static window method results

Here, we show results for the approach with static window. In general, the best behavior is noticed when $Coef$ is around 3 as is shown in Table 4.1. The results constructed with this specific $Coef$ values keep slightly better error values compared with the input coarse data. The improvement in error value is not so striking but the results are more plausible compared with the coarse maps by a human eye.

When the $Coef$ is lower then 2 (e.g. $Coef = 2.1$), the error value might be better then for the coarse surface map but the results keep visible coarse grid

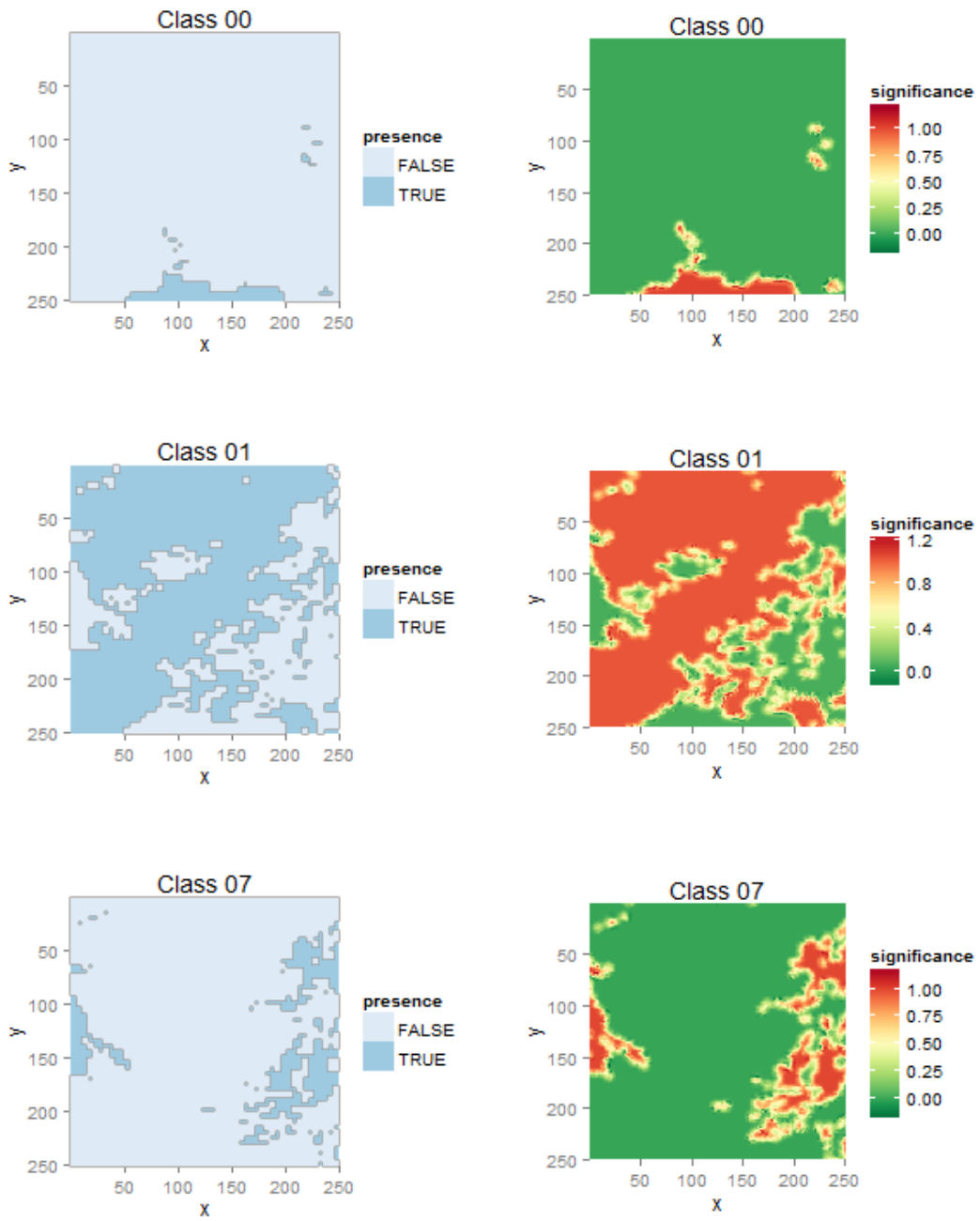


Figure 4.3: The refined "pseudo-probability" distribution (significance) of some classes for Yellowstone National Park. ($Coef = 4$, $\tau = 0.9$)

(Fig. 4.5) which is the main cause for the refinement, therefore we consider these results as worse in general.

On the other hand, refined maps which use $Coef > 4$ do not show better error values either. It is questionable whether the results are more plausible because a stronger class usually wipe out weaker classes with raising area for single regression. Larger window can catch more global information but the classes with sparse occurrences disappear. Therefore the results may seem not to be so much plausible for human eye. The dilemma about loosing more global information with smaller windows is more discussed in Chapter 4.5.1.

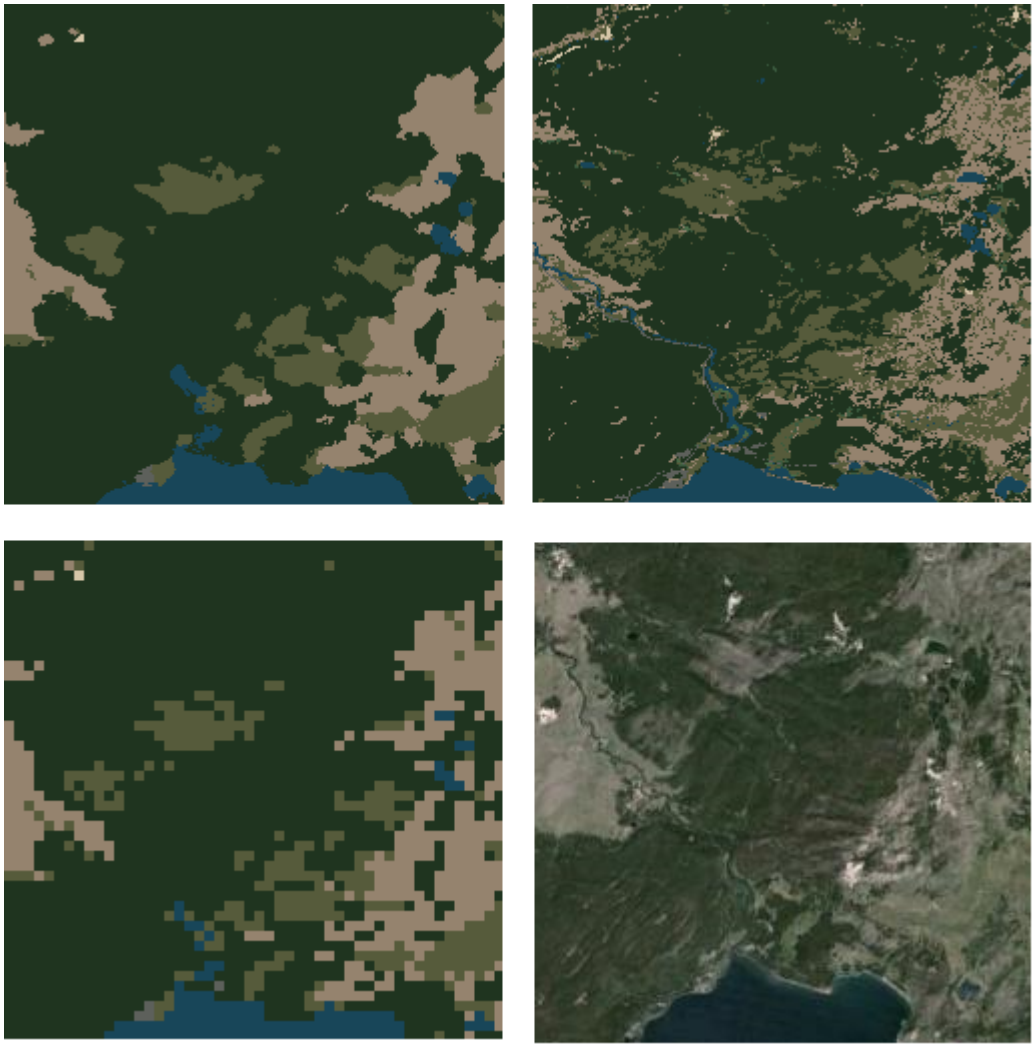


Figure 4.4: *Left top:* refined map, *right top:* truth map, *left bottom:* coarse map, *right bottom:* satellite picture (source – Google Earth, 12/2015) for Yellowstone National Park. ($Coef = 4$, $\tau = 0.9$)

A visualization of surface maps with VBS engine[1] is provided for three locations – Grand Canyon (Figures A.2, A.3, A.4, A.5), Navajo mountain (Figures A.6, A.7, A.8, A.9) and Yellowstone National Park (Figures A.10, A.11, A.12, A.13). For each location, we provide a coarse surface map which is suppose to be refined, a truth surface map which the refinement wants to get closer to and the refinement with static window with $Coef = 3$ and $Coef = 10$.

It is visible that the refined data can not reveal the ground truth picture, but in general they show more visibly pleased results. One of the reasons is that our method breaks the significant grid of coarse data. The reality resembling is affected by a scale factor where the smaller factor of enlargement is the closer to reality the results are.

Sample resulting pictures for Navajo mountain are shown in Figure A.14.

[1] `https://bisimulations.com/`

Static window method	Yellowstone National Park	Grand Canyon	Navajo Mountain
$Coef = 1$	0,19449	0,13233	0,21678
$Coef = 2$	0,19186	0,13093	0,21508
$Coef = 3$	0,19248	0,13147	0,21754
$Coef = 4$	0,19842	0,13505	0,22489
$Coef = 5$	0,20972	0,14113	0,23605
$Coef = 6$	0,21681	0,14491	0,24244
$Coef = 7$	0,22604	0,14934	0,24996
$Coef = 8$	0,23160	0,15264	0,25549
$Coef = 10$	0,24052	0,16162	0,26543
Coarse map error:	0,19450	0,13234	0,21681

Table 4.1: Error values for static window method. $Coef$ determines a size of the local window for whole picture in refinement and the coarse map error is an error value of the coarse surface map which suppose to be refined.

4.4 The adaptive window method results

We tested three different versions of the adaptive window method. The versions differ in validation function which is used for choosing the best model (window size) for a given position and where the chosen model predicts result value for this position. As mentioned in Section 2.3.2, we use three validation functions – MSE, wMSE and $adjR^2$.

In general, the results show slightly worse error value in comparison with input coarse surface map (Tab. 4.2). The wMSE error seems to have the error value the lowest. The results do not keep the visible coarse grid artifact (Fig. 4.5) and do not wipe out surface types with sparse occurrence therefore we can consider them also as visually more plausible. The results are very similar to results for static windows with $Coef = 3$. Note that it is caused by the fact, that we set the minimal local window to have size of $Coef = 2.5$. We investigate more the behavior about the adaptive window size change in Section 4.5.2.

Sample resulting pictures for Navajo mountain are shown in Figure A.15.

Adaptive window method	Yellowstone National Park	Grand Canyon	Navajo Mountain
MSE	0,20517	0,13740	0,22828
wMSE	0,19469	0,13103	0,21645
$adjR^2$ (random[1])	0,20378	0,13696	0,22823
Coarse map error:	0,19450	0,13234	0,21681

Table 4.2: Error values for results of the adaptive window method. Note that the algorithm was choosing between windows according to $Coef$ from range between 2.5 and 8.

4.5 Special behavior

In this section, we discuss special behavior of our method.

4.5.1 Global information vs. local information

Someone can expect that our method reveals strong relations which can easily expand some phenomenons, for example that river flows through a canyon. To demonstrate this example, imagine two water clusters laying in a canyon but not to close to each other. A human brain can process this water-in-canyon information and deduce that the water should probably be along the whole canyon therefore our method could be expected to do the same. Note that the human brain does not process only information only about few pictures but has other information. For example can know that the MODIS data are to coarse to cover too narrow river. Or the brain can know that rivers shape canyons. From this knowledge and observing the canyon shape can be deduced the river shape (the connection of our two mentioned water clusters).

Our model does not have any of this extra knowledge therefore cannot make decisions like this. The models knows only information which is get from the local area at the current spot. Here is strong dependence on the size of the window. Larger windows reveal more global context in comparison with local windows. We can see that the difference between local and more global context in Figure 4.6. To reveal the phenomenons models need to have a proper window sizes. Bigger part of the phenomenon the window covers, the better can be the prediction of the phenomenon. At least some part of the phenomenon must be present in the window because our models cannot figure out new information which is not present within the window. Here is a following problem:

If the algorithm uses bigger window, which is big enough to capture both clusters, then the information about water in canyon is usually overridden with much stronger and more global information. The sparse information is taken as some kind of noise and is therefore tend to be vanished than to be expanded. We can say then that one behavior of big windows is to suppress sparse information. On the other hand, when we do not take the sparse information in consideration, the big window learn more global information and distribute the probabilities according to the features much more visibly. Therefore the output transform includes shapes more easily noticeable in features. We provide an example of probability distribution taken with bigger window in Figure 4.6. We can notice by look, that the probability for bigger $Coef$ is distributed according to given elevation map there, which is not a property of smaller windows. The Figure 4.7 shows the final refinement where we can notice the absence of sparse information for bigger $Coef$.

When the algorithm uses smaller window, then sparse information does not disappear. But the model is too local to affect wider context (Fig. 4.6) – for example to fill the whole canyon with water or similar phenomenons. In the example case of two water clusters in canyon, the water clusters do not disappear but also do not expand much either. We can see on Figure 4.7 the presence of sparse classes for small $Coef$. The smaller window is, the more similar to the base distribution the refined result is.

Our models are used for prediction of their central point. An big sparse information expand could happen when our local models would be used for prediction of different places than just the center. Or an expert knowledge could help to provides model with extra information to reveal phenomenons.

4.5.2 Behavior of adaptive window size

In this section, we briefly summarize how the window changes its size along the refinement with the adaptive window method.

The idea behind the selection of appropriate model is that the predictions which fit the best are done by model with the smallest bias and variance. Therefore here is a need to minimize them, which is known bias-variance tradeoff problem (Geman et al. [1992]). Our validation functions try to minimize them (Sec. 2.3.2).

Constant areas where the surface type does not change have easy solution. Since all models perfectly predict the value (the only possible value), meaning that the models are unbiased and have null variance, the model corresponding with the largest window is chosen because was trained with the highest count of samples.

When the algorithm gets closer to a border of two constant areas, the selected window shrinks its size. It is intuitive because smaller windows still have the constant environment inside, therefore have null prediction error and therefore are preferred to bigger windows which contain variable values. Because size matters, the biggest window with constant area is chosen.

The interesting part of the window selection is held on the edge between different surface types – where all windows contain variable values. There, the window selection truly depends on the models predictions.

A demonstration of the window size selection is shown in Figure 4.8.

4.5.3 Features reduction

Our test showed that all design matrices for all used local windows were reduced to contain only one linearly combined feature. Even if we raised a threshold τ over 0.99 (Sec. 1.4.3), the reduction always kept only one dimension for features. This new single feature is linear combination of all the features. A detailed study of this topic could be done in the future.

4.5.4 Up-to-date data reliance

One character of this method is that the refinement it is completely dependent on provided data. We use data corresponding to statistics up to 2006. Therefore any significant change since then is not considered – such changes could be for example cut down forest or burnt areas. The changes can be revealed by providing newer data set. This is not a defect of our method but an trait of all data driven methods which a user should take into account.

4.6 Future work

Many possible future work is available for our method improvement, starting with a speed optimization.

Our C++ implementation uses OpenMP to parallelize the main algorithm for-cycle because calculation of each pixel is independent to each other. Nevertheless, our algorithm implementation offers much more places where a usage of parallelism would be suitable, e.g. a data preparation. A counting of more difficult operations such as matrix operations could be mover to GPU which would bring a calculation speed gain.

Another speed-up can be done by detecting "constant" models – models with constant dependent variables. These models predict always the same value and a work with them can be optimized.

In the introduction chapter we mentioned that Moon et al. [2015] use iterative adaptive windows which build regression models recursively. The method also provides a recursive determination of prediction error for the models. The advantage of this recursive method is that only the new part of a window needs to be counted. Our method creates new models for each window from scratch. Therefore a usage of this recursive method would rapidly speed-up our algorithm.

The static window method works as expected and gives a relatively plausible results when right $Coef$ is chosen. On the other hand, the adaptive window version was expected to work slightly better. Therefore here is a possible closer investigation of the adaptive method behavior which could be done in future. The research would be aimed for choosing different validation functions and its impact. Also would be interesting to use different regression approach than wLLR.

While our testing, the tSVD kept reducing the feature space to a single dimension. A closer investigation to confirm right behavior or to reveal an implementation fault should be done. There is also a possibility to change the method for determining the right features dimensionality (Sec. 1.4.3).

Possible future approach improvement could use different set of features then only features derived from DEM. For example moist maps, temperature maps, vector data about forest border or river profiles and others. The problem is an availability of such data.

Another improvement of our algorithm could be in the batch file processing. We did not speak about the batch file processing much in this document because it is not important in comparison with the general refinement problem. However, our implementation can process files also as batch. The refinement is done one file/tile at a time where each processed file needs to load also its neighboring files to get an overlap. Much better solution would be to cache already loaded files for their reusing when needed.

An extension of our evaluation method could be done by performing a user study which would validate our results. A point of view from human perspective could bring a better lead to recognizing which pictures are plausible and which not. Our error value does not respond this criteria.

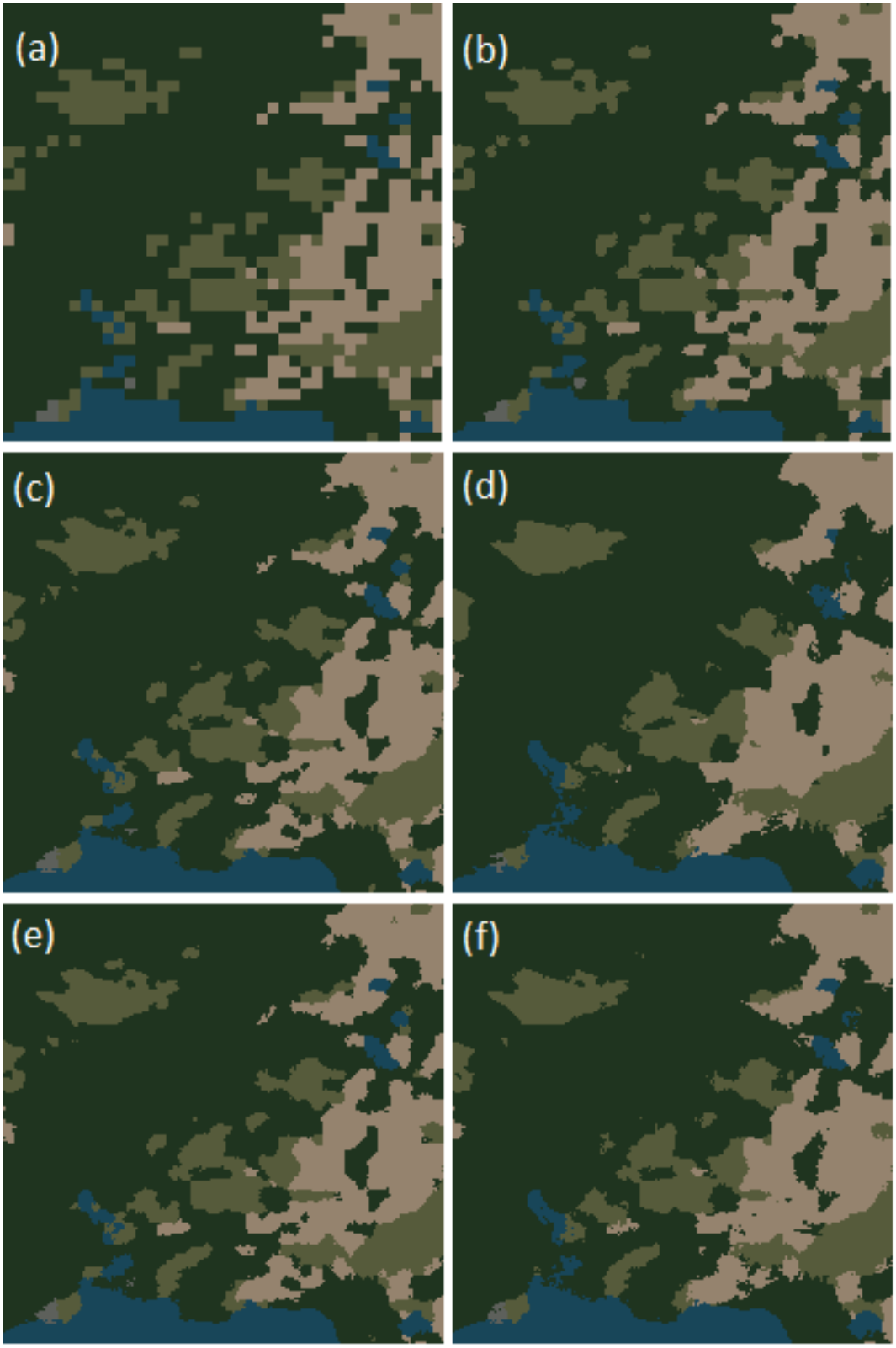


Figure 4.5: Individual pixels are strongly noticeable at coarse surface map (a), we call it a grid artifact. In reading order, (b) the $Coef = 2$ surface map keeps the artifact strong, (c) the $Coef = 3$ map contains the artifact really lightly and (d) the $Coef = 6$ one contains no grid artifact defect. The picture (e), by the wMSE adaptive method refined map, is really similar to (c) with slight defect. The (f) MSE adaptive method do not contain the grid artifact. Pictures correspond to an area in Yellowstone National Park.

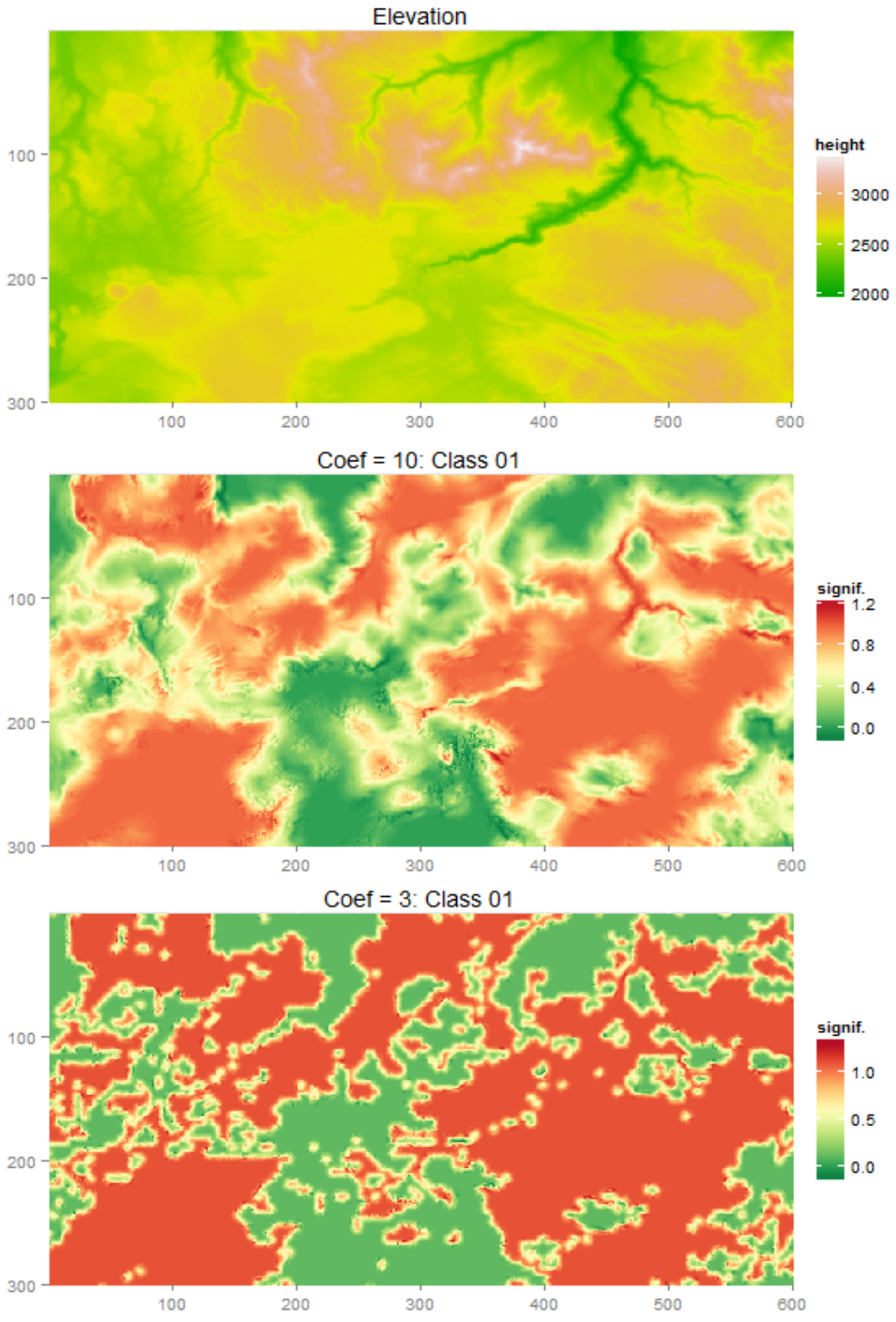


Figure 4.6: **Global vs. local information.** Pseudo-probability maps for Evergreen Needleleaf forest in Yellowstone National Park. Bottom picture corresponds to the small window models predictions and the middle picture corresponds to the bigger window models predictions. Notice by comparing the pseudo-probability maps with the elevation map that the bigger window map contains information in more global context than the small window one. Pictures correspond to an area in Yellowstone National Park.

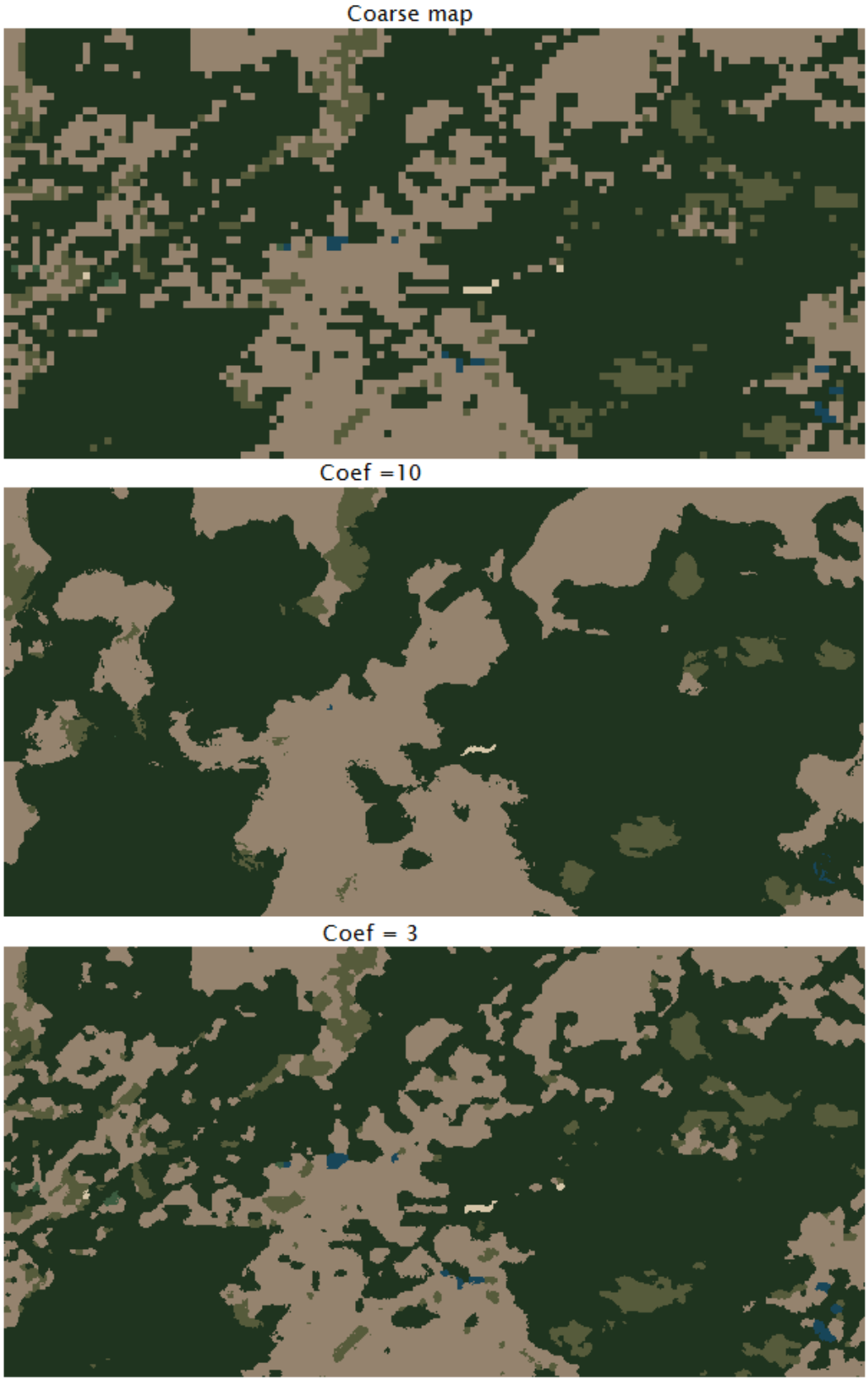


Figure 4.7: **Global vs. local information – results.** Refined surface maps for relatively small and big windows. Pictures correspond to an area in Yellowstone National Park.

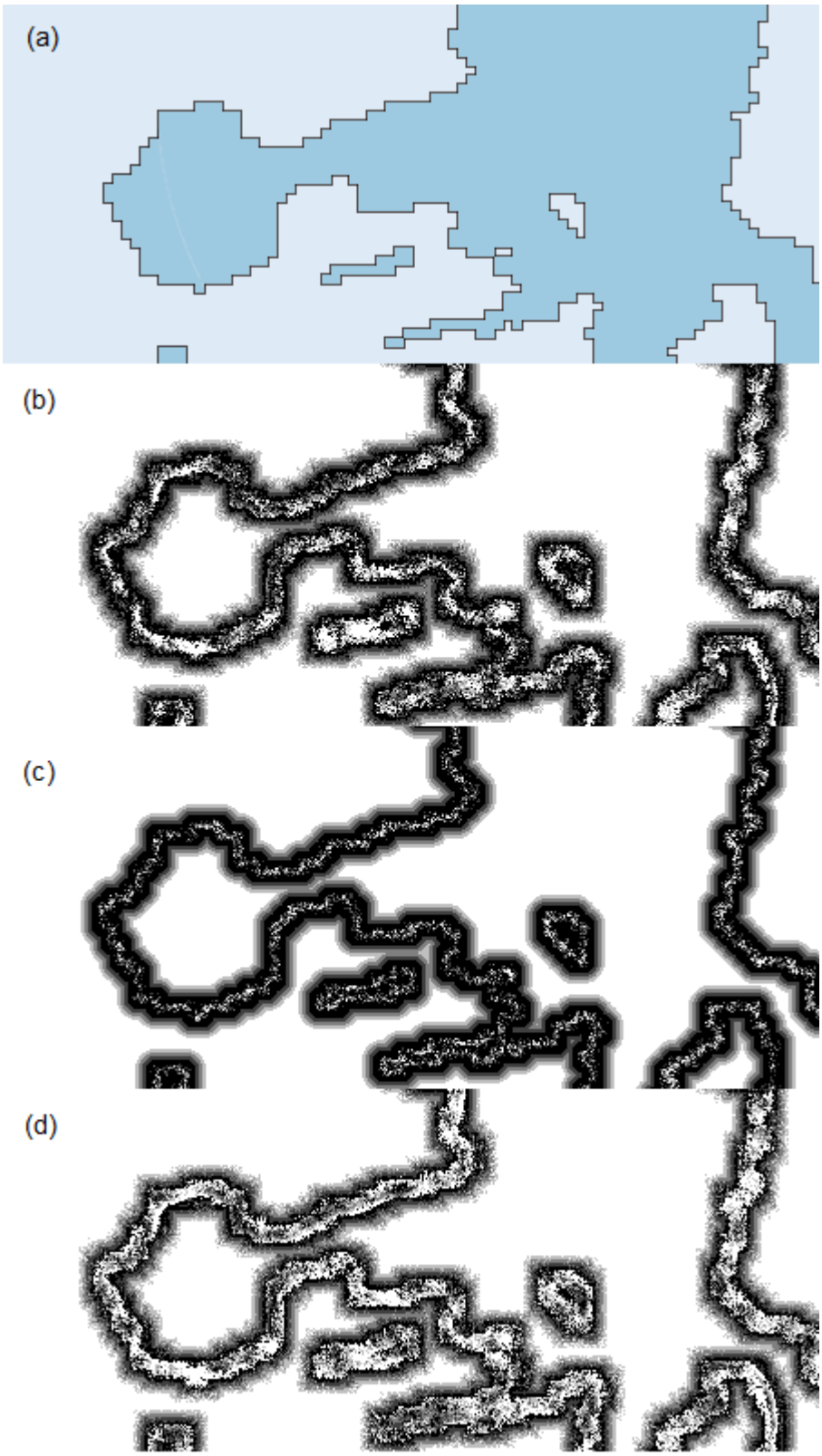


Figure 4.8: **An adaptive window size usage.** Pictures from up, (a) coarse class 0 occurrence map for an area of Yellowstone National Park, usage of window size for (b) MSE, (c) wMSE, (d) adjR^2 adaptive method. Black corresponds to $Coef = 2.5$ window size, white to $Coef = 6$, gray determines intermediate sizes.

Conclusion

Finer surface data are needed for procedural generation of landscape. We have implemented a method that refines a surface data at the coarse resolution into finer resolution with an assistance of DEM which is already at the better resolution. The approach is based on statistical modeling which reveals dependencies between the surface data and the DEM. Our wLLR models are constructed by both the surface and the DEM data at finer resolution. Then the local models predict values at finer resolution from the learned dependencies.

The precision of a model prediction strongly depends on the data which were used for the model creation. Our approach uses variable sizes of the models which changes the resulting behavior. The method can work in two modes, first, the models keep the same size along whole the refinement, or second, model sizes are adjusted to fit the best currently processed data.

Our method can work with many utility data. We work strictly with DEM derived features but an addition of any other is not a problem for the approach since operations with the features are done in reduced feature space. We utilize the tSVD for the features dimensionality reduction.

We observed that our method works well under relatively small and static models size. In this case the refinement looks much more plausible and usually resembles the reality more than the coarse data. A grid artifact which is visible in coarse data disappears with the refinement. When models are too small then the results show improvement in resembling reality but the human eye fails validation since the grid artifact stays too strong. On the other hand when the models are too big then our prediction error is too big as well. In this case the results also seem not so much plausible because the method wipes out sparse classes of the surface data. However, the refined data are more pleasing since the grid artifact is gone.

The adaptive model size version was expected to work much better then static size since the models predictions correspond to better fits but we observed that the models mostly keep lower sizes therefore the results are very similar to the results of small static model size.

The method is highly modifiable. In comparison with the ground truth data our results look bad but compared to the coarse data the refinements look much better. The static method shows good enough results for the need of a practical use therefore is being evaluated for real applications.

There would be contributive to perform a user study which would bring an objective opinion on results plausibility evaluation.

The results depends on scale factor where the bigger enlargement factor is the more predicted results are and therefore farther from reality. This method is more suitable for barren areas than for variable or sparse data such as rivers or urban areas.

In comparison with methods which generate results on base of ad-hoc figured out rules (Hammes [2001]), our method resemble the reality much more because it predicts the results according to dependencies learned from real data.

Bibliography

S. Ahlberg, U. Soderman, A. Persson, and M. Elmqvist. Synthetic natural environments from high resolution sensor data. In *Proceedings of the 2005 IMAGE Conference*, pages 46–53, 2005.

Barbara J. Anderson, Beatrice E. Arroyo, Yvonne C. Collingham, Brian Etheridge, Javier Fernandez-De-Simon, Simon Gillings, Richard D. Gregory, Fiona M. Leckie, Innes M. W. Sim, Chris D. Thomas, Justin M. J. Travis, and Steve M. Redpath. Using distribution models to test alternative hypotheses about a species' environmental limits and recovery prospects. *Biological Conservation*, 142(3):488–499, 2009. ISSN 0006-3207. doi: 10.1016/j.biocon.2008.10.036.

MB Araujo, W Thuiller, PH Williams, and I Reginster. Downscaling european species atlas distributions to a finer resolution: implications for conservation planning. *GLOBAL ECOLOGY AND BIOGEOGRAPHY*, 14:17–30, 2005. doi: 10.1111/j.1466-822X.2004.00128.x. URL `http://dx.doi.org/10.1111/j.1466-822X.2004.00128.x`.

Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006. ISBN 0387310738.

Pierluigi Bombi and Manuela D'Amen. Scaling down distribution maps from atlas data: a test of different approaches with virtual species. *Journal of Biogeography*, 39(4):640–651, 2012. ISSN 1365-2699. doi: 10.1111/j.1365-2699.2011.02627.x. URL `http://dx.doi.org/10.1111/j.1365-2699.2011.02627.x`.

W. S. Cleveland and C. L. Loader. *Smoothing by Local Regression: Principles and Methods*, pages 10–49. Springer, New York, 1996.

Y.C. Collingham, R.A. Wadsworth, B. Huntley, and P.E. Hulme. *Predicting the Spatial Distribution of Non-indigenous Riparian Weeds: Issues of Spatial Scale and Extent*, volume 37. 2000. doi: 10.1046/j.1365-2664.2000.00556.x.

Oliver Deussen, Pat Hanrahan, Bernd Lintermann, Radomír Měch, Matt Pharr, and Przemyslaw Prusinkiewicz. Realistic modeling and rendering of plant ecosystems. In *Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '98, pages 275–286, New York, NY, USA, 1998. ACM. ISBN 0-89791-999-8. doi: 10.1145/280814.280898. URL `http://doi.acm.org/10.1145/280814.280898`.

Fallucchi Francesca and Fabio Massimo Zanzotto. Svd feature selection for probabilistic taxonomy learning. In *Proceedings of the Workshop on Geometrical Models of Natural Language Semantics*, GEMS '09, pages 66–73, Stroudsburg, PA, USA, 2009. Association for Computational Linguistics. URL `http://dl.acm.org/citation.cfm?id=1705415.1705424`.

Matan Gavish and David L. Donoho. The optimal hard threshold for singular values is \(4/\sqrt {3}\). *IEEE Transactions on Information Theory*, 60(8):

5040–5053, 2014. doi: 10.1109/TIT.2014.2323359. URL http://dx.doi.org/10.1109/TIT.2014.2323359.

Stuart Geman, Elie Bienenstock, and René Doursat. Neural networks and the bias/variance dilemma. *Neural Comput.*, 4(1):1–58, January 1992. ISSN 0899-7667. doi: 10.1162/neco.1992.4.1.1. URL http://dx.doi.org/10.1162/neco.1992.4.1.1.

Johan Hammes. Modeling of ecosystems as a data source for real-time terrain rendering. In Caroline Y. Westort, editor, *Digital Earth Moving, First International Symposium, DEM 2001, Manno, Switzerland, September 5-7, 2001, Proceedings*, volume 2181 of *Lecture Notes in Computer Science*, pages 98–111. Springer, 2001. doi: 10.1007/3-540-44818-7_14. URL http://dx.doi.org/10.1007/3-540-44818-7_14.

Per Christian Hansen. The truncated svd as a method for regularization. *BIT*, 27 (4):534–553, October 1987. ISSN 0006-3835. doi: 10.1007/BF01937276. URL http://dx.doi.org/10.1007/BF01937276.

Philipp Hirtz, Hilko Hoffmann, and Daniel Nuesch. Interactive 3d landscape visualization: Improved realism through use of remote sensing data and geoinformation. *Computer Graphics International Conference*, 0:101, 1999. ISSN 1530-1052. doi: http://doi.ieeecomputersociety.org/10.1109/CGI.1999.777922.

Petr Keil, Jonathan Belmaker, Adam M. Wilson, Philip Unitt, and Walter Jetz. Downscaling of species distribution models: a hierarchical approach. *Methods in Ecology and Evolution*, 4(1):82–94, 2013. ISSN 2041-210X. doi: 10.1111/j.2041-210x.2012.00264.x. URL http://dx.doi.org/10.1111/j.2041-210x.2012.00264.x.

M. H. Kutner, C. J. Nachtsheim, J. Neter, and W. Li. *Applied Linear Statistical Models*. McGraw Hill, 5th edition, 2005.

Brendan Lane and Przemyslaw Prusinkiewicz. Generating spatial distributions for multilevel models of plant communities. In *In: Proceedings of Graphics Interface*, pages 69–80, 2002.

Mihee Lee, Haipeng Shen, Jianhua Z. Huang, and S. Marron J. Biclustering via sparse singular value decomposition. *Biometrics*, 66(4):1087–1095, 2010. URL http://EconPapers.repec.org/RePEc:bla:biomet:v:66:y:2010:i:4:p:1087-1095.

Jana M. McPherson, Walter Jetz, and David J. Rogers. Using coarse-grained occurrence data to predict species distributions at finer spatial resolutions–possibilities and limitations. *Ecological Modelling*, 192(3-4):499 – 522, 2006. doi: DOI:10.1016/j.ecolmodel.2005.08.007. URL http://www.sciencedirect.com/science/article/B6VBS-4H68T7W-3/2/70e912c609d8db75548a96fea45520f8.

SB Menke, DA Holway, RN Fisher, and W Jetz. Characterizing and predicting species distributions across environments and scales: Argentine ant occurrences in the eye of the beholder. *Global Ecology and Biogeography*, 18(1):50–63, 2009.

Varpu Mitikka, RistoK. Heikkinen, Miska Luoto, MiguelB. Araújo, Kimmo Saarinen, Juha Pöyry, and Stefan Fronzek. Predicting range expansion of the map butterfly in northern europe using bioclimatic models. *Biodiversity and Conservation*, 17(3):623–641, 2008. ISSN 0960-3115. doi: 10.1007/s10531-007-9287-y. URL http://dx.doi.org/10.1007/s10531-007-9287-y.

Bochang Moon, Nathan Carr, and Sung-Eui Yoon. Adaptive rendering based on weighted local regression. *ACM Trans. Graph.*, 33(5):170:1–170:14, September 2014. ISSN 0730-0301. doi: 10.1145/2641762. URL http://doi.acm.org/10.1145/2641762.

Bochang Moon, Jose A. Iglesias-Guitian, Sung-Eui Yoon, and Kenny Mitchell. Adaptive rendering with linear predictions. *ACM Trans. Graph.*, 34(4):121:1–121:11, July 2015. ISSN 0730-0301. doi: 10.1145/2766992. URL http://doi.acm.org/10.1145/2766992.

Aidin Niamir, Andrew K. Skidmore, Albertus G. Toxopeus, Antonio R. Muñoz, and Raimundo Real. Finessing atlas data for species distribution models. *Diversity and Distributions*, 17(6):1173–1185, 2011. ISSN 1472-4642. doi: 10.1111/j.1472-4642.2011.00793.x. URL http://dx.doi.org/10.1111/j.1472-4642.2011.00793.x.

Art B. Owen and Patrick O. Perry. Bi-cross-validation of the svd and the non-negative matrix factorization. *Ann. Appl. Stat.*, 3(2):564–594, 06 2009. doi: 10.1214/08-AOAS227. URL http://dx.doi.org/10.1214/08-AOAS227.

Antony R. Palmer Penn Lloyd. Abiotic factors as predictors of distribution in southern african bulbuls. *The Auk*, 115(2):404–411, 1998. ISSN 00048038, 19384254. URL http://www.jstor.org/stable/4089199.

Anand Rajaraman and Jeffrey David Ullman. *Mining of Massive Datasets*. Cambridge University Press, New York, NY, USA, 2011. ISBN 1107015359, 9781107015357.

CHRISTOPHE F. RANDIN, ROBIN ENGLER, SIGNE NORMAND, MASSIMILIANO ZAPPA, NIKLAUS E. ZIMMERMANN, PETER B. PEARMAN, PASCAL VITTOZ, WILFRIED THUILLER, and ANTOINE GUISAN. Climate change and plant distribution: local models predict high-elevation persistence. *Global Change Biology*, 15(6):1557–1569, 2009. ISSN 1365-2486. doi: 10.1111/j.1365-2486.2008.01766.x. URL http://dx.doi.org/10.1111/j.1365-2486.2008.01766.x.

D. Ruppert and M. P. Wand. Multivariate locally weighted least squares regression. *The Annals of Statistics*, 22:1346–1370, 1994.

Ruben M. Smelik, Tim Tutenel, Rafael Bidarra, and Bedrich Benes. A survey on procedural modelling for virtual worlds. *Computer Graphics Forum*, 33(6):31–50, 2014. ISSN 1467-8659. doi: 10.1111/cgf.12276. URL http://dx.doi.org/10.1111/cgf.12276.

MANDAR R. TRIVEDI, PAMELA M. BERRY, MICHAEL D. MORECROFT, and TERENCE P. DAWSON. Spatial scale affects bioclimate model projections of climate change impacts on mountain plants. *Global Change Biology*, 14 (5):1089–1103, 2008. ISSN 1365-2486. doi: 10.1111/j.1365-2486.2008.01553.x. URL `http://dx.doi.org/10.1111/j.1365-2486.2008.01553.x`.

William D. Wells. Generating enhanced natural environments and terrain for interactive combat simulations (genetics). In *Proceedings of the ACM Symposium on Virtual Reality Software and Technology*, VRST '05, pages 184–191, New York, NY, USA, 2005. ACM. ISBN 1-59593-098-1. doi: 10.1145/1101616.1101655. URL `http://doi.acm.org/10.1145/1101616.1101655`.

Daniela M. Witten, Trevor Hastie, and Robert Tibshirani. A penalized matrix decomposition, with applications to sparse principal components and canonical correlation analysis. *Biostatistics*, 2009.

Dan Yang, Zongming Ma, and Andreas Buja. A sparse singular value decomposition method for high-dimensional data. *Journal of Computational and Graphical Statistics*, 23(4):923–942, 2014. doi: 10.1080/10618600.2013.858632. URL `http://dx.doi.org/10.1080/10618600.2013.858632`.

List of Figures

List of Algorithms

List of Abbreviations

LLR	local linear regression
wLLR	weighted local linear regression
GAM	generalized additive model
GLM	generalized linear model
SVD	singular value decomposition
tSVD	truncated singular value decomposition
OSL	ordinary least squares
WSL	weighted least squares
MODIS	moderate-resolution imaging spectroradiometer
SRTM	shuttle radar topography mission
DEM	digital elevation model
MSE	mean-squared error
wMSE	weighted mean-squared error
R^2	coefficient of determination
$adjR^2$	coefficient of determination

Attachments

A. Figures

Color	MODIS value	IGBP label
	0	Water
	1	Evergreen Needleleaf forest
	2	Evergreen Broadleaf forest
	3	Deciduous Needleleaf forest
	4	Deciduous Needleleaf forest
	5	Mixed forest
	6	Closed shrublands
	7	Open shrublands
	8	Woody savannas
	9	Savannas
	10	Grasslands
	11	Permanent wetlands
	12	Croplands
	13	Urban and built-up
	14	Cropland / Natural vegetation mosaic
	15	Snow and ice
	16	Barren or sparsely vegetated
	255	Fill value / Unclassified

Figure A.1: An explanation for MODIS values. Pictures in this thesis are colored with the colors from the left column. The colors correspond to MODIS value shown in middle column. The last column expresses the meaning for the values/colors.

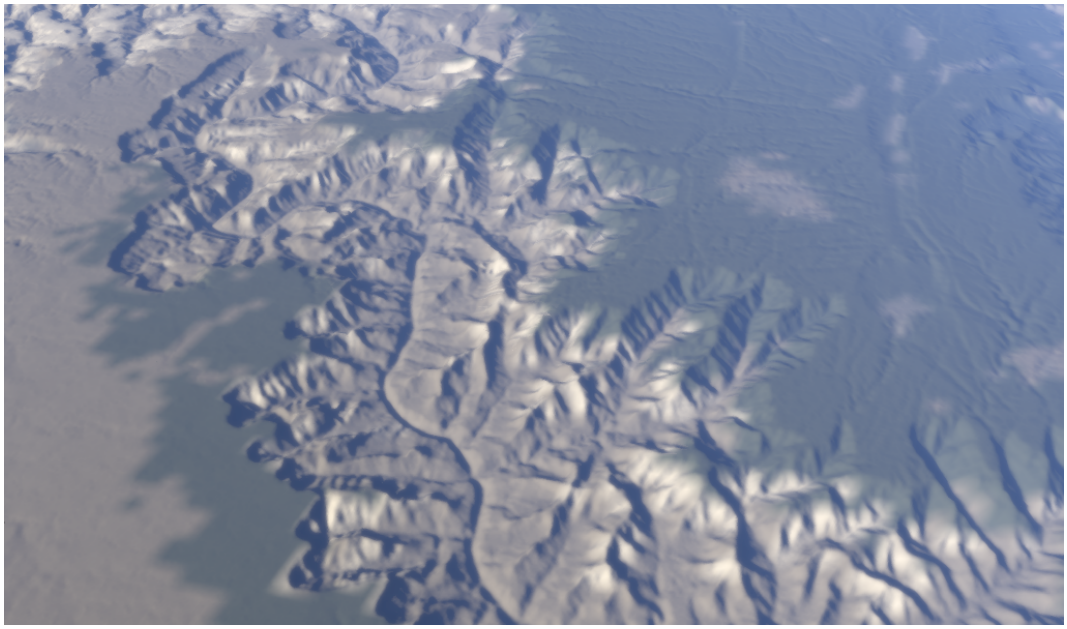


Figure A.2: **The coarse surface map.** A visualization of Grand Canyon by VBS engine. The applied surface map is the coarse surface map which is suppose to be refined. Because the map includes visible coarse grid, the colors of neighboring pixels are interpolated.

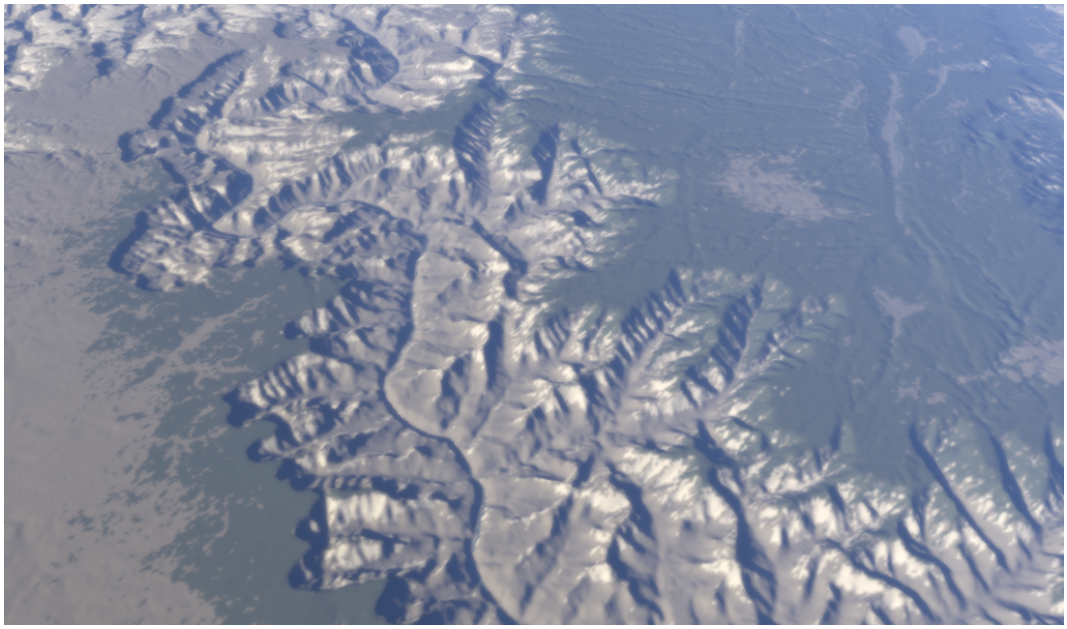


Figure A.3: **The truth surface map.** A visualization of Grand Canyon by VBS engine. The applied surface map is the truth surface map which we want to get closer by refining to.

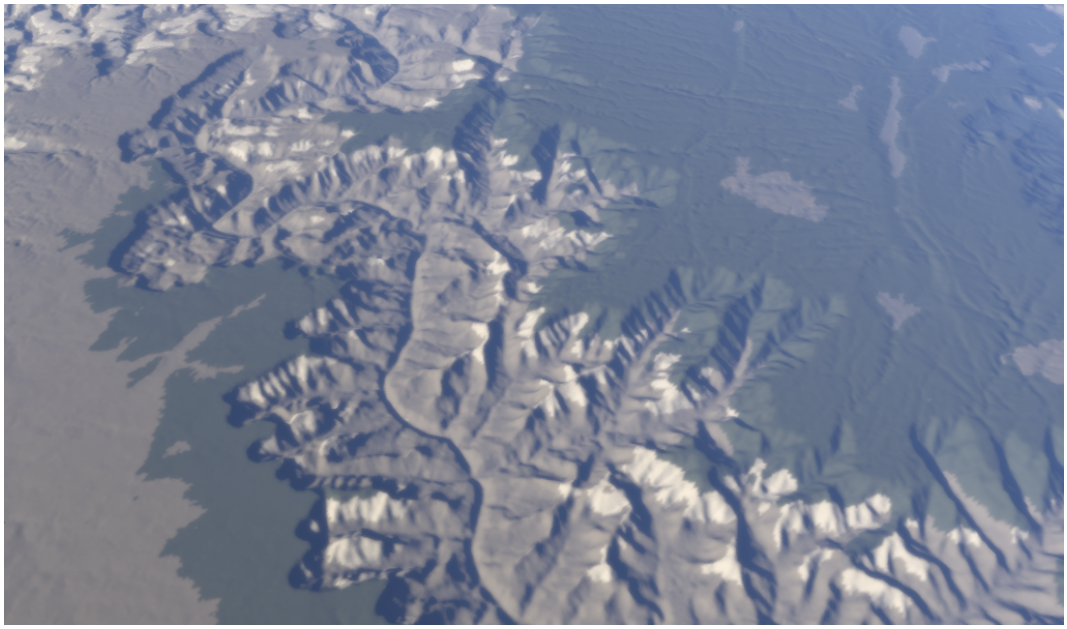


Figure A.4: **The C3 refined surface map.** A visualization of Grand Canyon by VBS engine. The applied surface map is the refined surface map by static method with $Coef = 3$.

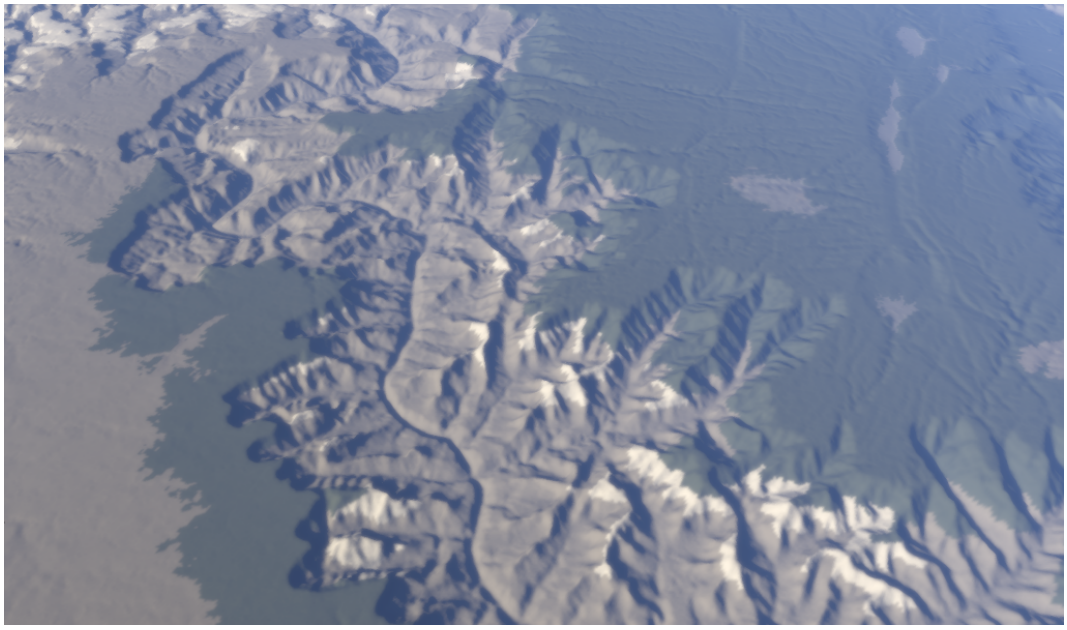


Figure A.5: **The C10 refined surface map.** A visualization of Grand Canyon by VBS engine. The applied surface map is the refined surface map by static method with $Coef = 10$. We can observe that sparse classes mostly disappeared.

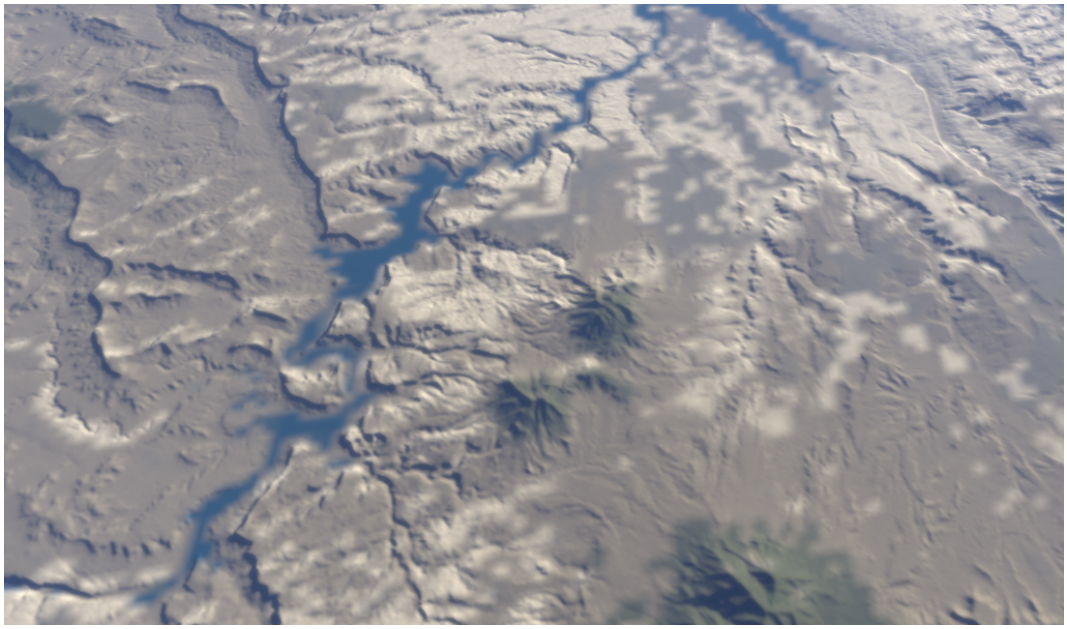


Figure A.6: **The coarse surface map.** A visualization of location near Navajo mountain by VBS engine. The applied surface map is the coarse surface map which is suppose to be refined. Because the map includes visible coarse grid, the colors of neighboring pixels are interpolated.

Figure A.7: **The truth surface map.** A visualization of location near Navajo mountain by VBS engine. The applied surface map is the truth surface map which we want to get closer by refining to.

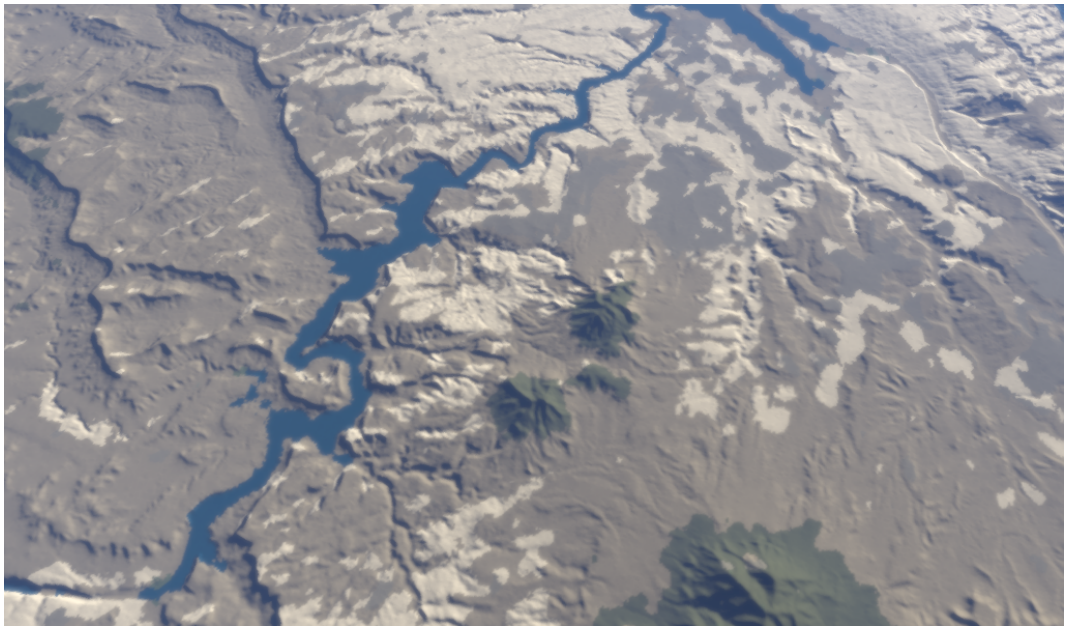


Figure A.8: **The C3 refined surface map.** A visualization of location near Navajo mountain by VBS engine. The applied surface map is the refined surface map by static method with $Coef = 3$. We can observe an improvement for example in the river shape.

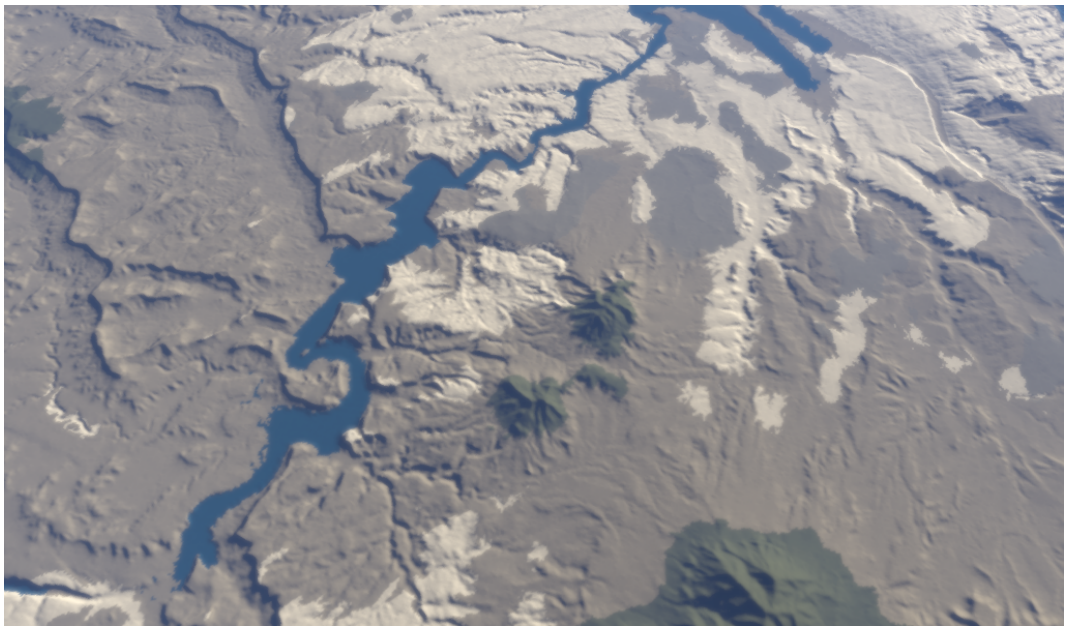


Figure A.9: **The C10 refined surface map.** A visualization of location near Navajo mountain by VBS engine. The applied surface map is the refined surface map by static method with $Coef = 10$. We can observe that sparse classes mostly disappeared. We can observe an improvement for example in the river shape.

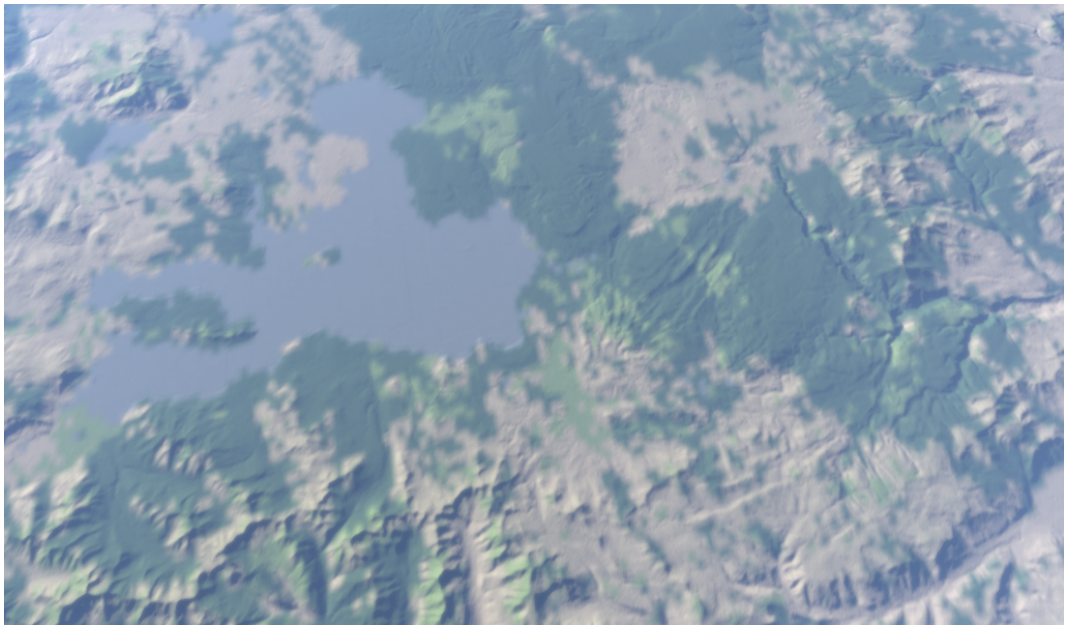


Figure A.10: **The coarse surface map.** A visualization of Yellowstone Park by VBS engine. The applied surface map is the coarse surface map which is suppose to be refined. Because the map includes visible coarse grid, the colors of neighboring pixels are interpolated.

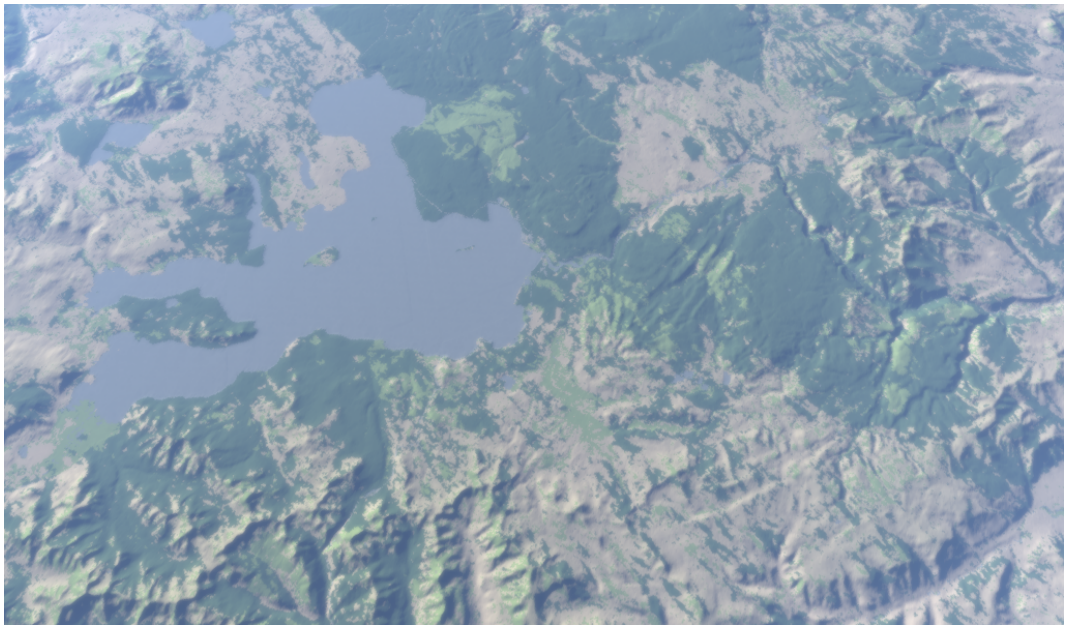


Figure A.11: **The truth surface map.** A visualization of Yellowstone Park by VBS engine. The applied surface map is the truth surface map which we want to get closer by refining to.

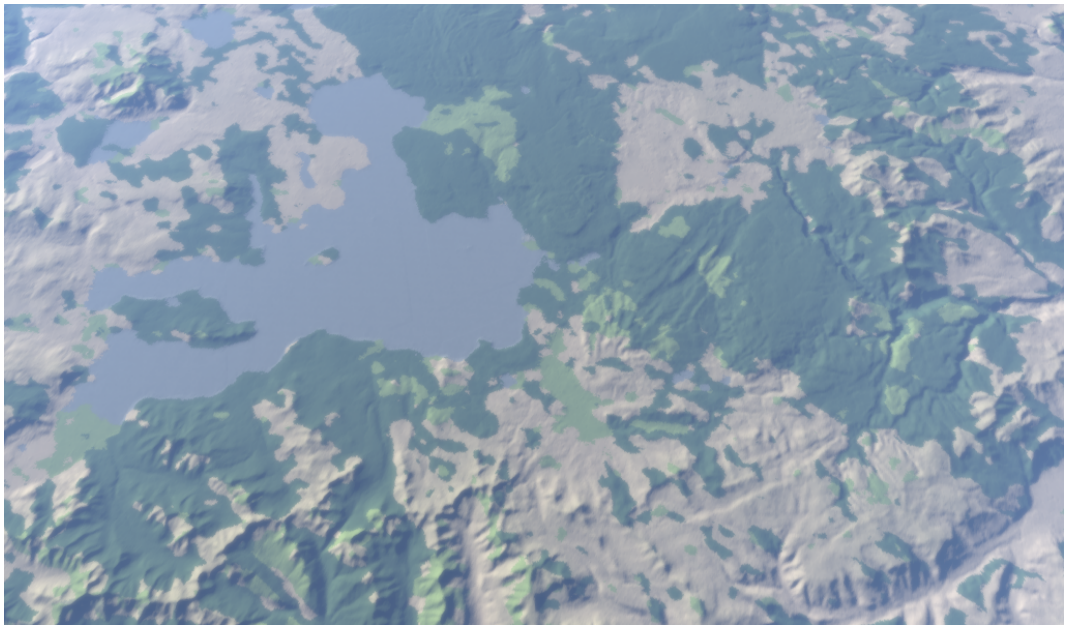


Figure A.12: **The C3 refined surface map.** A visualization of Yellowstone Park by VBS engine. The applied surface map is the refined surface map by static method with $Coef = 3$.

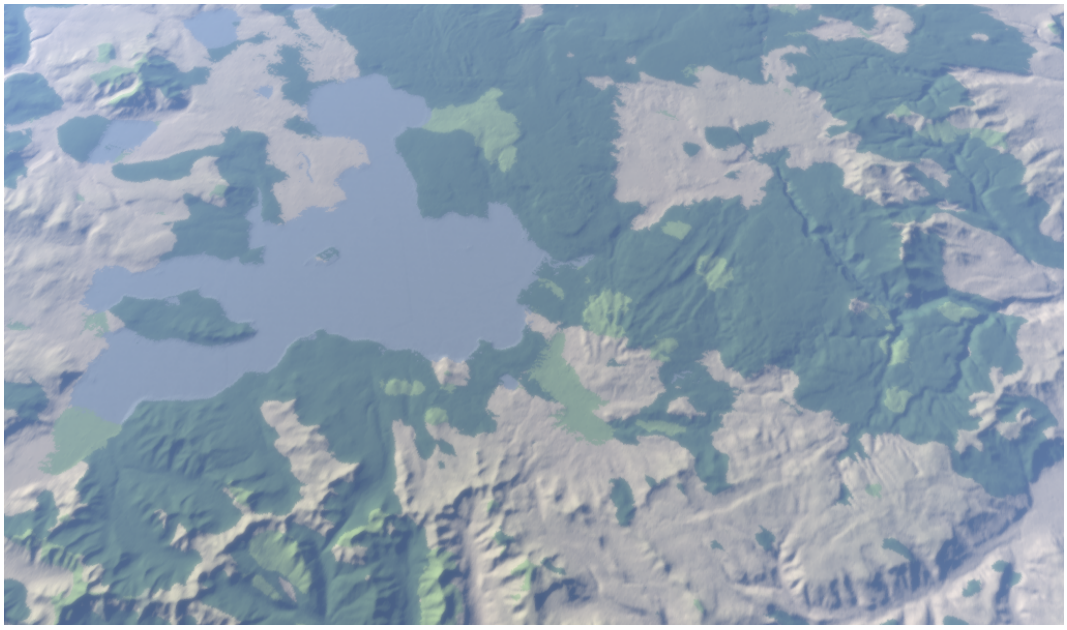


Figure A.13: **The C10 refined surface map.** A visualization of Yellowstone Park by VBS engine. The applied surface map is the refined surface map by static method with $Coef = 10$. We can observe that sparse classes mostly disappeared.

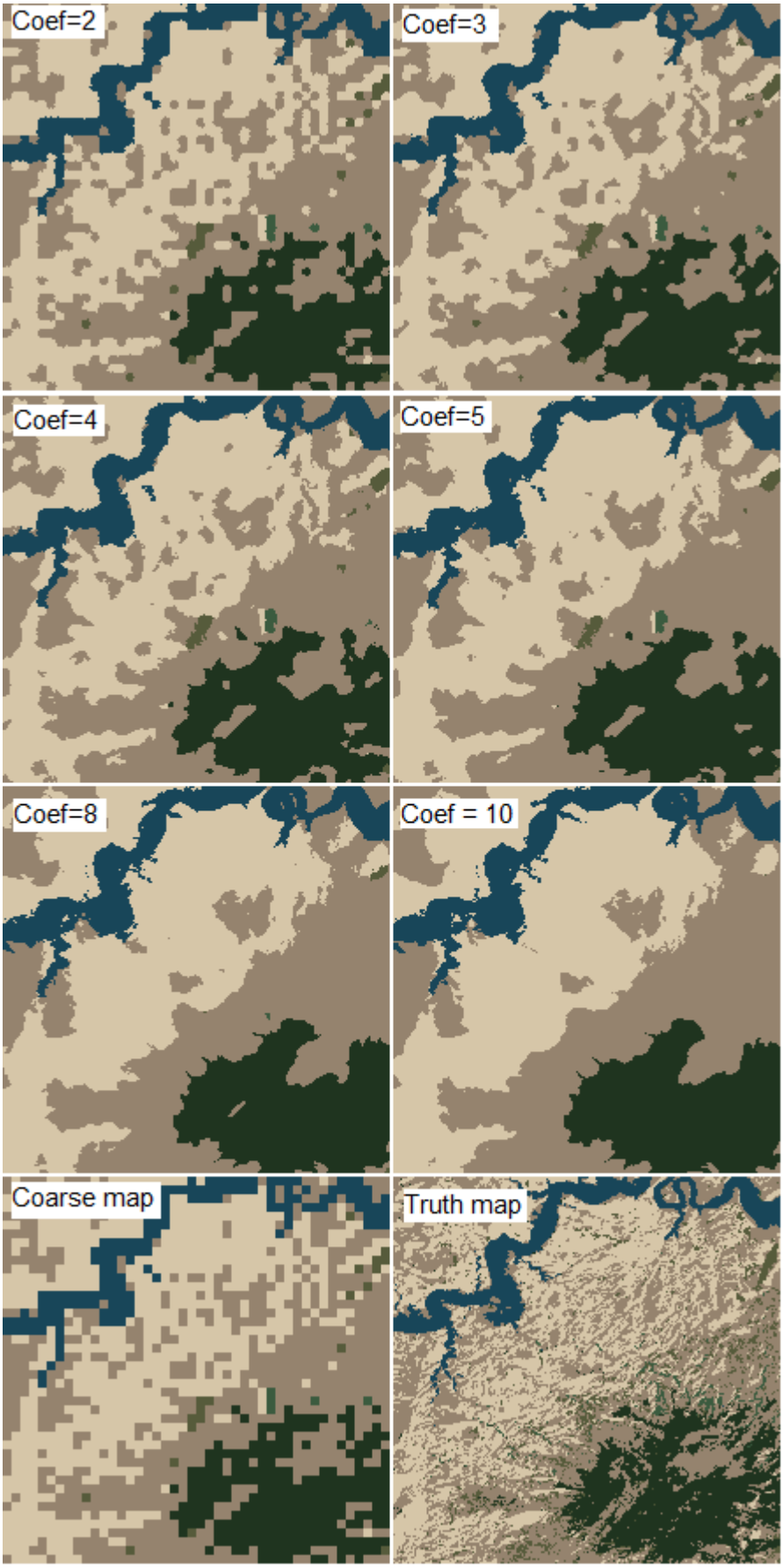


Figure A.14: Navajo mountain - static window

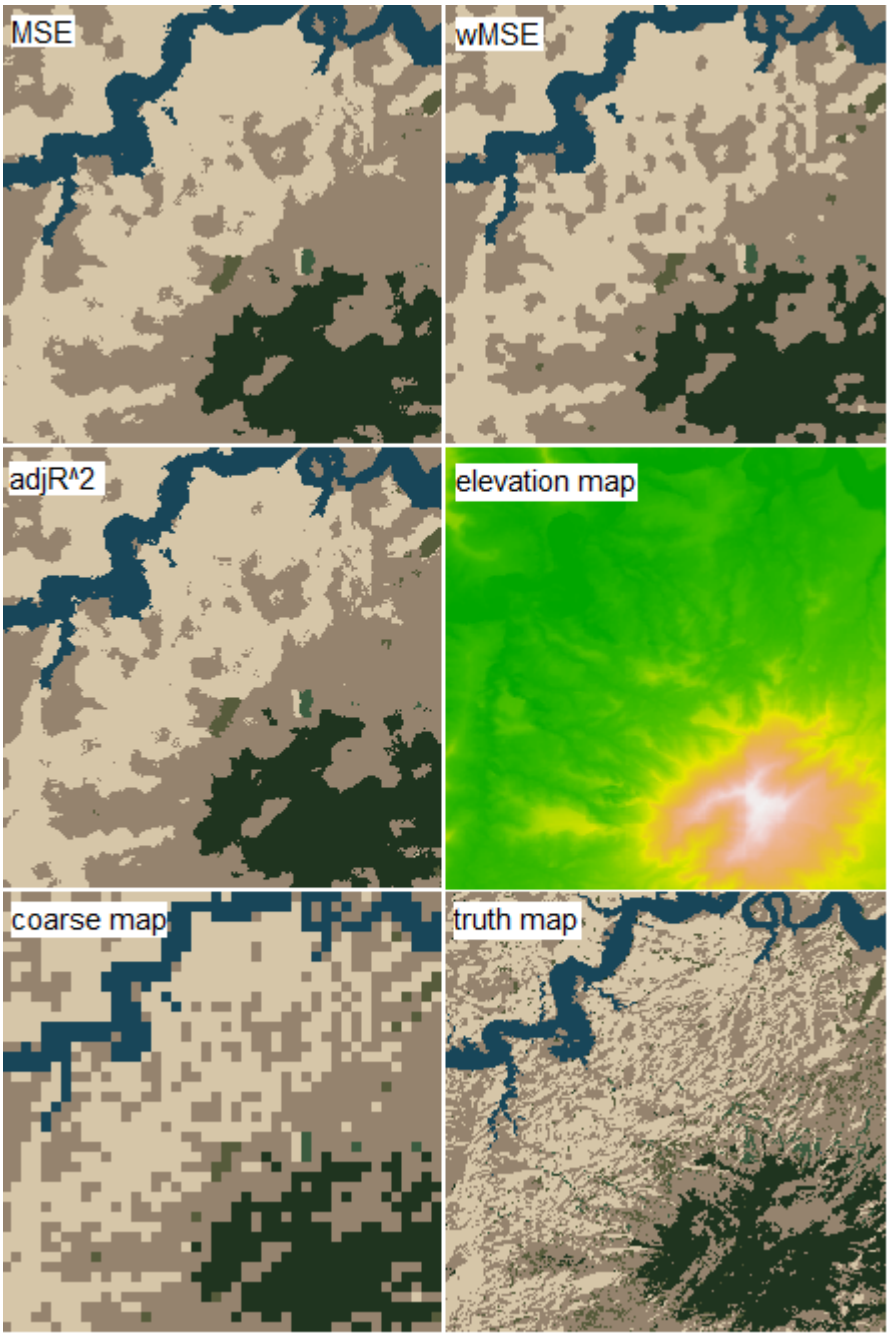


Figure A.15: Navajo mountain - adaptive window

B. Algorithms

Algorithm B.1: R sample of creating slope and aspect features.

```
# Resolution for HGT files is 1021 x 1021
resolution = 1201;
# File stream representing ''N44W111.hgt'' file
fileStream = file(''N44W111.hgt'', "rb");
# Reads data as signed integers in big endian
data = readBin(fileStream, integer(), n = resolution * resolution, size = 2, signed =
    TRUE, endian = "big");
# Closes the file stream
close(fileStream);

# Represent data as a matrix. Transposition is needed since R matrix data are in column
    -by-column order in comparison with HGT row-by-row order
heightMatrix = t(matrix(data, nrow = resolution, ncol = resolution));
# Creates a raster from the height matrix
heightsRaster = raster(retMatrix);
# Specific the extent to the raster (for ''terrain'' function)
extent(heightsRaster) = extent(0, 1, 0, 1);
# Specific the HGT projection to raster (for ''terrain'' function)
crs(heightsRaster) = "+proj=longlat +ellps=WGS84 +datum=WGS84 +no_defs +lat_
    0=44 +lon_0=-111";

# Creates the slope feature
slopeRaster = terrain(heightsRaster, opt = "slope", unit = "degrees");
# Creates the aspect feature
aspectRaster = terrain(heightsRaster, opt = "aspect", unit = "degrees");
```

Algorithm B.2: R sample of linear regression.

```
# Creates linear model and predicts the middle window value
# Returns pair of predicted value of local window and number of used dimensions
doLinearRegression <- function(
  occurenceMap, # The matrix of occurrences for one surface type
  featuresMaps, # The list of features matrices
  width, # The width of the local window
  height, # The height of the local window
  weights, # The weights matrix for the local window
  reduceDimensions # A switch for dimensionality reduction:
                   # -1 - Uses all features
                   # 0 - Detects optimal number of features using SVD
                   # N - Uses exactly N features.)
{
  # A count of features for regression
  featuresCount = length(featuresMaps)

  # Linearizes the 2D arrays
  regressand = c(occurenceMap)
  weightsLinear = c(weights)
  # Creates matrix of linearized features. Each column is different linearized feature.
  regressors = vector();
  for (featureIndex in 1:featuresCount){
    regressors = c(regressors, c(featuresMaps[[featureIndex]]))
  }
  regressors = matrix(regressors, ncol = featuresCount)
```

```
26
27    # Reduces the dimensionality of the features matrix
28    if (reduceDimensions >= 0){
29      # The reduced features
30      regressors = reduceMatrixDimensions(regressors, reduceDimensions)
31      # The actual number of used features
32      featuresCount = dim(regressors)[[2]]
33    }
34
35    # Estimates the linear regression model
36    lsfitResult <- lsfit(regressors, regressand, wt = weightsLinear)
37    # Coefficients of the model
38    lsCoefs = lsfitResult$coefficients
39
40    # The middle point coordinates
41    y = ceiling(height / 2)
42    x = ceiling(width / 2)
43    # Predicts the middle point value
44    outputValue = lsCoefs[[1]]
45    for (featureIndex in 1:featuresCount){
46      outputValue = outputValue + lsCoefs[[featureIndex + 1]] * regressors[y + (x - 1) *
            height, featureIndex]
47    }
48
49    # Returns a pair of the predicted value and the used features count
50    return(c(outputValue, featuresCount))
51  }
```

Algorithm B.3: R sample of feature dimensionality reduction.

```
1   # Reduces the dimensionality of the features matrix
2   # Returns the features matrix projected into the reduced space
3   reduceMatrixDimensions <- function(
4     linearizedFeatures, # The matrix where each column is different linearized feature
5     reduceDimensions # A switch for dimensionality reduction
6                                   # 0 - Detects optimal number of features using SVD
7                                   # N - Uses exactly N features.
8     # Applies SVD on the features matrix M. Form of M = U*D*V'. The matrix V of
          right singular vectors has (n x n) dimensionality.
9     svdM = svd(linearizedFeatures, nv = dim(linearizedFeatures)[[2]])
10    # The matrix of right singular vectors
11    V = svdM$v
12
13    # Determines features dimension to be used
14    if (reduceDimensions == 0){
15      # Determines the proper features count from the energy of the singular values matrix
16      featuresCount = reduceSingularValuesCount(svdM$d)
17    } else {
18      featuresCount = reduceDimensions
19    }
20
21    # Transposed and cropped matrix V (right singular vectors)
22    Vkt = t(V[1:(dim(V)[[1]]),1:featuresCount])
23    # Projects features into the local space
24    result = t(Vkt %*% t(linearizedFeatures))
25
26    return(result)
27  }
```