TECHNISCHE UNIVERSITÄT WIEN

VIENNA UNIVERSITY OF TECHNOLOGY

DIPLOMARBEIT

Fitting a Background Map to Spatial Data

unter der Leitung von

O.Univ.-Prof. Dipl.-Ing. Dr.techn. Rudolf Dutter
Institut für Statistik
und Wahrscheinlichkeitstheorie

eingereicht an der

Technischen Universität Wien
Fakultät für Mathematik
und Geoinformation

von

Alexander Kowarik
Matrikelnummer: 0225078
Eyslergasse 36
A-1130 Wien
alex@kowarik.net

Wien, am 13. März 2008 .....

Alexander Kowarik

Kurzfassung

Das Ziel dieser Arbeit ist es, eine Möglichkeit zu bieten, eine gedruckte Karte als Hintergrundkarte eines Datensatzes mit Koordinaten zu verwenden. Dies wird in zweifacher Hinsicht erklärt: erstens wird der theoretische Hintergrund näher beleuchtet und zweitens wird diese Prozedur für Anwender erklärt. Die Daten können aus jedem Anwendungsgebiet stammen, aber jede Beobachtung muss zweidimensionale Koordinaten haben.

Der erste Schritt ist eine Karte im Dateiformat PNM oder PS zu erhalten. Dies kann durch einscannen bzw. durch herunterladen aus dem Internet erfolgen. PNM (Portable Anymap) ist eine Bitmapdatei, PS (PostScript) hingegen ist eine Vektorgraphikdatei.

Nun wollen wir die Hintergrundkarte in das Koordinatensystem der Daten transformieren. Im Fall eines PNM-Bildes starten wir mit einem Koordinatensystem bei dem der Ursprung in der linken unteren Ecke des Bildes liegt und die Einheit ein Pixel ist. Das erste Koordinatensystem einer PS-Datei hängt von Formatierungsoptionen in der PS-Datei ab, z.B. von dem gewählten Papierformat.

Wir müssen uns für eine von drei implementierten Transformationen entscheiden. Um die Parameter der Transformation schätzen zu können, müssen wir nun Referenzpunkte sowohl auf der Karte als auch in den Daten wählen. Die Referenzpunkte sollen nach der richtigen Transformation die selben Koordinaten haben.

Das Problem wird als lineares Modell ausgedrückt, sodass die Schätzung von dessen Parametern die Transformationsparameter ergibt. Dies erfolgt mittels Kleinste-Quadrate-Methode. Wenn wir die Ähnlichkeitstransformation ausgewählt haben, steht auch eine robuste Methode zur Verfügung, nämlich die Kleinste-Quadrate-Methode bei der ein bestimmter Prozentsatz der größten Residuen aus der Summe gestrichen wird.

Der Anpassungsprozess sollte iterativ sein, so dass in jedem Schritt neue Referenzpunkte hinzugefügt oder falsche Referenzpunkte gelöscht werden können. Das wird wiederholt bis eine zufriedenstellende Anpassung erreicht wird.

Die ersten zwei Kapitel dieser Arbeit sind eine Einführung in das Konzept der linearen Regression und der implementierten Transformationen. Danach wird die konkrete Implementierung in R und der grafischen Benutzeroberfläche DAS+R erklärt. Das letzte Kapitel ist eine Anleitung aller nötigen Schritte, um eine angepasste Hintergrundkarte mit DAS+R zu erhalten.

TECHNISCHE
UNIVERSITÄT
WIEN

VIENNA
UNIVERSITY OF
TECHNOLOGY

MASTER THESIS

Fitting a Background Map to Spatial Data

carried out at

Institute of Statistics
and Probability Theory,
Vienna University of Technology

under the supervision of

O.Univ.-Prof. Dipl.-Ing. Dr.techn. Rudolf Dutter

by

Alexander Kowarik
Eyslergasse 36
A-1130 Wien
alex@kowarik.net

Abstract

The goal of this master thesis is to provide and explain a way from a printed map to a fitted background map of a spatial dataset. The variables in this dataset can be from any field of interest, but there have to be two dimensional coordinates for each observation.

The first step is to obtain a picture in the file format PNM or PS. PNM (Portable Anymap) is basically a bitmap image file format, while PS (PostScript) is a vector graphic file format.

Now we want to transform the image to the coordinate system of the data. For PNM images we start with a coordinate system with the origin in the left bottom corner and the scale is one pixel. The first coordinate system for a PS file depends on PostScript format options, for example the paper size.

One of several implemented transformation types has to be chosen now. Furthermore we have to identify reference points, these are points in "map" space and "data" space, which later should have approximately the same coordinates and are used for estimating the transformation parameters.

The estimation of the transformation parameters is done by writing the transformation in terms of a linear model and estimating its parameters. This is done by least squares method or, if the similarity transformation is used, additionally by least trimmed sum of squares estimation, a robust alternative to the default method.

The process of fitting the background map is supposed to be iterative, so that in each step new reference points could be added or wrong reference points could be deleted. We do this until a satisfying level of fitting is achieved.

In the first two chapters the basic ideas of linear regression and the implemented transformations are explained. Afterwards we see how these tools are implemented in and in the graphical user interface DAS+R. The last chapter is an illustration of all required steps for fitting a background map in DAS+R.

Acknowledgments

First and foremost I would like to thank my thesis advisor Rudolf Dutter, who has shown a large and consistent interest in the project during the times. I am grateful for suggestions, comments, and contributions.

It has been a pleasure to study at the Vienna University of Technology during the years. My fellow students are cordially thanked for their friendship and the good teamwork. Also my friends, my family and my patient girlfriend are thanked for their good company and support.

Contents

Chapter 1

Linear Model

The task of fitting a background map to spatial data leads to a regression problem. More details about the adaption of the linear model to each transformation are described in Chapter 2.

Therefore we are looking for a function f fulfilling

$$y = f(x_1, x_2, \ldots, x_{p-1}) + \epsilon$$

where y stands for an observation and ϵ for a random error. We will only consider functions, which are linear in the parameters such as

$$f(x_1, x_2, \ldots, x_{p-1}) = \beta_0 + \sum_{i=1}^{p-1} x_j \beta_j.$$

We write the model in the compact form

$$\boldsymbol{y} = \boldsymbol{X}\boldsymbol{\beta} + \boldsymbol{\epsilon}$$

where
$\boldsymbol{y}$ is a $(n \times 1)$ vector of observations,
$\boldsymbol{X}$ is a $(n \times p)$ design matrix of known form,
$\boldsymbol{\beta}$ is a $(p \times 1)$ vector of parameters,
$\boldsymbol{\epsilon}$ is a $(n \times 1)$ vector of errors,
and where $E(\boldsymbol{\epsilon}) = 0$, $V(\boldsymbol{\epsilon}) = \boldsymbol{I}\sigma^2$. So the elements of $\boldsymbol{\epsilon}$ are homoscedastic and uncorrelated. In most cases the design matrix will look like,

$$\boldsymbol{X} = \begin{pmatrix} 1 & x_{1,1} & x_{1,2} & \ldots & x_{1,p-1} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & x_{n,1} & x_{n,2} & \ldots & x_{n,p-1} \end{pmatrix}$$

where the first column provides a constant term in the linear model and the other $p-1$ columns are the regressors. The goal is to find good values for the parameter vector $\boldsymbol{\beta}$ in the linear model.

The most common way of finding these parameters is a Least Squares (LS)-Regression, which is in some sense the optimal way to go. However the process of selecting reference

points and determine their coordinates is highly error-prone, so that we are in need of a different approach, less sensitive to leverage points and outliers in the response variable. Least trimmed sum of Squares (LTS)-Regression is a robust alternative (Wikipedia, 2007f) to LS-Regression. In this chapter both methods are presented in a short way and without proofs.

1.1 LS-Regression

Today the Least Squares Method, invented by Carl Friedrich Gauss in 1795 (Wikipedia, 2007d), is one of the mostly applied tools in statistical analysis.

1.1.1 Assumptions

Assume we have a model (Draper and Smith, 1998), which can be written in the form

$$\boldsymbol{y} = \boldsymbol{X}\boldsymbol{\beta} + \boldsymbol{\epsilon}$$

and the error term ϵ has the previously defined properties.

1.1.2 Estimation

If we want to find an estimate for $\boldsymbol{\beta}$, we have to minimize the error sum of squares

$$\begin{aligned}\boldsymbol{e}^\top\boldsymbol{e} &= (\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta})^\top(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta}) \\ &= \boldsymbol{y}^\top\boldsymbol{y} - \boldsymbol{\beta}^\top\boldsymbol{X}\boldsymbol{y} - \boldsymbol{y}^\top\boldsymbol{X}\boldsymbol{\beta} + \boldsymbol{\beta}^\top\boldsymbol{X}^\top\boldsymbol{X}\boldsymbol{\beta} \\ &= \boldsymbol{y}^\top\boldsymbol{y} - 2\boldsymbol{\beta}^\top\boldsymbol{X}^\top\boldsymbol{y} + \boldsymbol{\beta}^\top\boldsymbol{X}^\top\boldsymbol{X}\boldsymbol{\beta} \quad .\end{aligned}$$

We now have to differentiate with respect to $\boldsymbol{\beta}$ and set the resulting matrix equation equal to zero. This leads us to the normal equation

$$(\boldsymbol{X}^\top\boldsymbol{X})\boldsymbol{b} = \boldsymbol{X}^\top\boldsymbol{y}$$

where $\boldsymbol{b}$ is a $(p+1 \times 1)$ vector of estimated parameters.

The normal equation consists of p equations. If some of them are dependent, $\boldsymbol{X}^\top\boldsymbol{X}$ is singular and $(\boldsymbol{X}^\top\boldsymbol{X})^{-1}$ does not exist. In this case, there is a need of reducing the number of parameters or assuming additional restrictions of the parameters.

Otherwise if the normal equation consists of p independent equations, the matrix $\boldsymbol{X}^\top\boldsymbol{X}$ is regular and $(\boldsymbol{X}^\top\boldsymbol{X})^{-1}$ exists and can be used to compute

$$\boldsymbol{b} = (\boldsymbol{X}^\top\boldsymbol{X})^{-1}\boldsymbol{X}^\top\boldsymbol{y} \quad .$$

For getting an unique solution we need at least $p+1$ observations, where $p-1$ is the number of regressors and the $+1$ is needed due to the intercept term.

In fact modern software does not compute $(\boldsymbol{X}^\top\boldsymbol{X})^{-1}$ due to numerical reason. The method for solving the normal equation, implemented in R, is described in Chapter 3.

1.1.3 Properties

Minimizing $\boldsymbol{e}^\top \boldsymbol{e}$

The solution $\boldsymbol{b}$ minimizes the sum of squares without assuming any distribution properties of the errors, except homoscedasticity and being uncorrelated. The assumption of normally distributed errors is only needed for carrying out tests such as t- or F-tests.

Optimum

$\boldsymbol{b}$ is the optimal linear and unbiased estimate for $\boldsymbol{\beta}$, which means, it has the minimum variance within all linear, unbiased estimates. This property is contained in the Gauss-Markov Theorem (Wikipedia, 2007b).

1.1.4 Prediction

Now we want to predict the value of y for new values of the variables used as regressors. Suppose we have an $(i \times p)$ matrix $\boldsymbol{X}_0$, where i is the number of new observations. We now obtain the $(i \times 1)$ vector $\hat{\boldsymbol{y}}_0$ of predicted values by multiplying $\boldsymbol{X}_0$ with $\boldsymbol{b}$

$$\hat{\boldsymbol{y}}_0 = \boldsymbol{X}_0 \boldsymbol{b} \quad .$$

1.2 LTS-Regression

In many examples we can see, that just a single regression outlier can make LS-estimation meaningless. The idea of LTS-Regression (Rousseeuw and Leroy, 1986) is to fit the model just to a certain percentage of the data. The breakdown point can be adjusted by the percentage used to estimate the parameters. Moreover the LTS-regression is similar to LS-regression, except that the large residuals are not considered in the summation.

1.2.1 Assumptions

We have a response variable $\boldsymbol{y}$ and $p-1$ regressors collected in the $(n \times p)$ design matrix $\boldsymbol{X}$, where n is the number of observations. Moreover, we assume, that a certain percentage of the data are outliers in $\boldsymbol{y}$ as well as outliers in $\boldsymbol{X}$.

1.2.2 Estimation

The least trimmed squares (LTS) estimator is given by

$$\min_{\boldsymbol{b}} \sum_{i=1}^{h} e^2_{(i)}$$

where
h is the number of observations used for estimating and
$e^2_{(i)}$ are the ordered squared residuals.

If h is set to $[\frac{n}{2}] + 1$, the breakdown point is approximately 50 percent. For details how the estimation is implemented in R see Chapter 3.

1.2.3 Properties

If we want to reach a breakdown point of approximately 50 percent, we need more than twice as many observations as variables to get an unique estimator.

Equivariance

An important property of the LTS-estimator is affine equivariance.

Asymptotic

Furthermore, if the error ϵ follows a normal distribution, the estimator is asymptotically efficient. The convergence speed is proportional to $n^{-\frac{1}{2}}$.

1.2.4 Prediction

Computing the predicted values is basically the same as in LS-regression: Again we have new values in the matrix $\boldsymbol{X}_0$ and with the LTS-estimator $\boldsymbol{b}$ we can obtain $\hat{\boldsymbol{y}}_0$

$$\hat{\boldsymbol{y}}_0 = \boldsymbol{X}_0\boldsymbol{b} \quad .$$

But there are differences between LS- and LTS-regression when it comes to variance, prediction or confidence intervals.

Chapter 2

Transformation

Three different types of transformations are used to transform the coordinate system of the map to the coordinate system of the data: similarity transformation, affine transformation and a simple nonlinear transformation.

An affine transformation (Wikipedia, 2007a) consists of a linear transformation including a translation. A special kind of affine transformation is the similarity transformation with a rotation times a scalar as linear transformation. The name giving property of similarity preserve a circle before transformation to be a circle after the transformation. The more general family of affine transformation just maintain ellipses. A similarity transformation can not be used for an equalization of errors during scanning, for example paper movement. The nonlinear transformation used here is a quadratic transformation.

2.1 Affine Transformation

Details of the idea behind using an affine transformation can be seen in the literature (for example in Kraus, 2004).

2.1.1 Assumptions

A fixed number n of reference points in the map space $\begin{pmatrix} x_{m,i} \\ y_{m,i} \end{pmatrix}$ correspond to given reference points in the data space $\begin{pmatrix} x_{d,i} \\ y_{d,i} \end{pmatrix}$. These reference points should finally pairwise coincide as much as possible.

2.1.2 Transformation

$$x_{d,i} = A + Cx_{m,i} + Ey_{m,i}$$

$$y_{d,i} = B + Dx_{m,i} + Fy_{m,i}$$

$$\begin{pmatrix} x_{d,i} \\ y_{d,i} \end{pmatrix} = \begin{pmatrix} A \\ B \end{pmatrix} + \begin{pmatrix} C & E \\ D & F \end{pmatrix} \begin{pmatrix} x_{m,i} \\ y_{m,i} \end{pmatrix}$$

So $\begin{pmatrix} A \\ B \end{pmatrix}$ is the translation and $\begin{pmatrix} C & E \\ D & F \end{pmatrix}$ is the linear transformation defining the affine transformation.

2.1.3 Properties

Preserving collinearity between points

All points being collinear before the transformations should also be collinear after the transformation. $P_i, i \in \{1,2,3\}$ are points in the original space and $\hat{P}_i$ are the points in the transformed space.

$$P_1 = \begin{pmatrix} x_1 \\ y_1 \end{pmatrix}, \ \hat{P}_1 = \begin{pmatrix} A + Cx_1 + Ey_1 \\ B + Dx_1 + Fy_1 \end{pmatrix}$$

$$P_2 = \begin{pmatrix} x_2 \\ y_2 \end{pmatrix}, \ \hat{P}_2 = \begin{pmatrix} A + Cx_2 + Ey_2 \\ B + Dx_2 + Fy_2 \end{pmatrix}$$

$$P_3 = \begin{pmatrix} x_3 \\ y_3 \end{pmatrix}, \ \hat{P}_3 = \begin{pmatrix} A + Cx_3 + Ey_3 \\ B + Dx_3 + Fy_3 \end{pmatrix}$$

P_1, P_2 and P_3 are collinear, if there is an $s \in \mathbb{R}$ such that

$$\begin{pmatrix} x_1 \\ y_1 \end{pmatrix} + s \begin{pmatrix} x_2 - x_1 \\ y_2 - y_1 \end{pmatrix} = \begin{pmatrix} x_3 \\ y_3 \end{pmatrix} \quad . \tag{2.1}$$

The transformed points $\hat{P}_1$, $\hat{P}_2$ and $\hat{P}_3$ should also be collinear, it means there is an $\hat{s} \in \mathbb{R}$ such that

$$\begin{pmatrix} A + Cx_1 + Ey_1 \\ B + Dx_1 + Fy_1 \end{pmatrix} + \hat{s} \begin{pmatrix} Cx_2 + Ey_2 - Cx_1 - Ey_1 \\ Dx_2 + Fy_2 - Dx_1 - Fy_1 \end{pmatrix} = \begin{pmatrix} A + Cx_3 + Ey_3 \\ B + Dx_3 + Fy_3 \end{pmatrix} \quad .$$

If we use P_3 from (2.1), we see that

$$\begin{pmatrix} Cx_1 + Ey_1 + \hat{s}\,(Cx_2 + Ey_2 - Cx_1 - Ey_1) \\ Dx_1 + Fy_1 + \hat{s}\,(Dx_2 + Fy_2 - Dx_1 - Fy_1) \end{pmatrix}$$

$$= \begin{pmatrix} C(x_1 + s(x_2 - x_1) + E(y_1 + s(y_2 - y_1)) \\ D(x_1 + s(x_2 - x_1) + F(y_1 + s(y_2 - y_1)) \end{pmatrix}$$

$$\begin{pmatrix} Cx_1 + Ey_1 + \hat{s}\,(Cx_2 + Ey_2 - Cx_1 - Ey_1) \\ Dx_1 + Fy_1 + \hat{s}\,(Dx_2 + Fy_2 - Dx_1 - Fy_1) \end{pmatrix}$$

$$= \begin{pmatrix} Cx_1 + Ey_1 + s\,(Cx_2 + Ey_2 - Cx_1 - Ey_1) \\ Dx_1 + Fy_1 + s\,(Dx_2 + Fy_2 - Dx_1 - Fy_1) \end{pmatrix}$$

and conclude $\hat{s}$ equals s. Which means $\hat{P}_1$, $\hat{P}_2$ and $\hat{P}_3$ are collinear.

The next property is a corollary of this result.

Preserving ratios of distances along a line

For distinct collinear points P_1, P_2 and P_3, the ratio between the distances of P_1 and P_2 and of P_2 and P_3 are preserved after an affine transformation.

2.1.4 Linear Model

Because the transformation parameters A, B, C, D, E and F are unknown, we want to shape the affine transformation as a linear model.

$$\underline{\boldsymbol{xy}}_d = \begin{pmatrix} \boldsymbol{x}_d \\ \boldsymbol{y}_d \end{pmatrix} = \boldsymbol{X}\boldsymbol{\beta} + \boldsymbol{\epsilon}$$

where

$$\boldsymbol{X} = \begin{pmatrix} 1 & 0 & x_{m,1} & y_{m,1} & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & 0 & x_{m,n} & y_{m,n} & 0 & 0 \\ 0 & 1 & 0 & 0 & x_{m,1} & y_{m,1} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 1 & 0 & 0 & x_{m,n} & y_{m,n} \end{pmatrix}, \quad \boldsymbol{\beta} = \begin{pmatrix} A \\ B \\ C \\ E \\ D \\ F \end{pmatrix}$$

and n denotes the number of reference points.

Next we can estimate the transformation parameters by one of the methods described in Chapter 1. As a consequence a minimum of three pairs of reference points are needed to determine unique or approximate parameters of the affine transformation.

2.2 Similarity Transformation

If there is no need of deformation, similarity transformation is the right way to go. Scaling, translating and rotating are sufficient for a wide spread field of cases. The similarity transformation is the two dimensional form of a Helmert transformation (for details see for example Wikipedia, 2007c).

2.2.1 Assumptions

A fixed number n of reference points in the map space $\begin{pmatrix} x_{m,i} \\ y_{m,i} \end{pmatrix}$ correspond to given reference points in the data space $\begin{pmatrix} x_{d,i} \\ y_{d,i} \end{pmatrix}$. After the correct similarity transformation the points should be pairwise approximately identical.

2.2.2 Transformation

$$x_{d,i} = A + Cx_{m,i} - Dy_{m,i}$$

$$y_{d,i} = B + Dx_{m,i} + Cy_{m,i}$$

$$\begin{pmatrix} x_{d,i} \\ y_{d,i} \end{pmatrix} = \begin{pmatrix} A \\ B \end{pmatrix} + \begin{pmatrix} C & -D \\ D & C \end{pmatrix} \begin{pmatrix} x_{m,i} \\ y_{m,i} \end{pmatrix}$$

The (2×2) matrix $\begin{pmatrix} C & -D \\ D & C \end{pmatrix}$ has the structure of a rotation matrix multiplied with by a scalar.

2.2.3 Properties

Similarity

The ratios between two distances are the same before and after a Similarity Transformation. In other words the distance between P_1 and P_2 in transformed space must be the distance in non-transformed space times a constant, which does not depend on the points P_1 and P_2.

$$P_1 = \begin{pmatrix} x_1 \\ y_1 \end{pmatrix}, \ \hat{P}_1 = \begin{pmatrix} A + Cx_1 - Dy_1 \\ B + Dx_1 + Cy_1 \end{pmatrix}$$

$$P_2 = \begin{pmatrix} x_2 \\ y_2 \end{pmatrix}, \ \hat{P}_2 = \begin{pmatrix} A + Cx_2 - Dy_2 \\ B + Dx_2 + Cy_2 \end{pmatrix}$$

$$d(P_1, P_2) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

$$(d(\hat{P}_1, \hat{P}_2))^2 = (Cx_1 - Dy_1 - Cx_2 + Dy_2)^2 + (Dx_1 + Cy_1 - Dx_2 - Cy_2)^2$$

$$= (C(x_1 - x_2) - D(y_1 - y_2))^2 + (D(x_1 - x_2) + C(y_1 - y_2))^2$$

$$= C^2(x_1 - x_2)^2 + D^2(y_1 - y_2)^2 - 2CD(x_1 - x_2)(y_1 - y_2) + D^2(x_1 - x_2)^2 + C^2(y_1 - y_2)^2$$
$$+2CD(x_1 - x_2)(y_1 - y_2)$$

$$= (C^2 + D^2)(x_1 - x_2)^2 + (C^2 + D^2)(y_1 - y_2)^2$$

$$= (C^2 + D^2)((x_1 - x_2)^2 + (y_1 - y_2)^2)$$

$\Rightarrow$
$\forall P_1, P_2 \in \mathbb{R}^2, \forall A, B, C, D \in \mathbb{R} : d(\hat{P}_1, \hat{P}_2) = \sqrt{C^2 + D^2} d(P_1, P_2) \Rightarrow$ Similarity

Translation Rotation Scale

For a better understanding the transformation can be split into three separated parts, namely a vector defining the translation, a scalar for the scale and an angle for the rotation.

$$\begin{pmatrix} x_d \\ y_d \end{pmatrix} = \begin{pmatrix} A \\ B \end{pmatrix} + \begin{pmatrix} C & -D \\ D & C \end{pmatrix} \begin{pmatrix} x_m \\ y_m \end{pmatrix}$$

$$= \begin{pmatrix} A \\ B \end{pmatrix} + \sqrt{C^2 + D^2} \begin{pmatrix} \frac{C}{\sqrt{C^2+D^2}} & \frac{-D}{\sqrt{C^2+D^2}} \\ \frac{D}{\sqrt{C^2+D^2}} & \frac{C}{\sqrt{C^2+D^2}} \end{pmatrix} \begin{pmatrix} x_m \\ y_m \end{pmatrix}$$

$$\begin{pmatrix} \frac{C}{\sqrt{C^2+D^2}} & \frac{-D}{\sqrt{C^2+D^2}} \\ \frac{D}{\sqrt{C^2+D^2}} & \frac{C}{\sqrt{C^2+D^2}} \end{pmatrix} \begin{pmatrix} x_m \\ y_m \end{pmatrix} = \begin{pmatrix} \sin\alpha & -\cos\alpha \\ \cos\alpha & \sin\alpha \end{pmatrix} \begin{pmatrix} x_m \\ y_m \end{pmatrix}$$

Translation

$$\begin{pmatrix} A \\ B \end{pmatrix}$$

Scale

$$\sqrt{C^2 + D^2}$$

Rotation

$$\alpha = arcsin\left(\frac{C}{\sqrt{C^2 + D^2}}\right)$$

2.2.4 Linear Model

Again we want to find the correct transformation parameters by estimating a special kind of linear model. Let

$$\underline{\boldsymbol{xy}}_d = \begin{pmatrix} \boldsymbol{x}_d \\ \boldsymbol{y}_d \end{pmatrix} = \boldsymbol{X}\boldsymbol{\beta} + \boldsymbol{\epsilon}$$

where

$$\boldsymbol{X} = \begin{pmatrix} 1 & 0 & x_{m,1} & -y_{m,1} \\ \vdots & \vdots & \vdots & \vdots \\ 1 & 0 & x_{m,n} & -y_{m,n} \\ 0 & 1 & y_{m,1} & x_{m,1} \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 1 & y_{m,n} & x_{m,n} \end{pmatrix}, \quad \boldsymbol{\beta} = \begin{pmatrix} A \\ B \\ C \\ D \end{pmatrix}$$

and n denotes the number of reference points.

Therefore just 2 pairs of reference points are needed to determine the parameters of the similarity transformation.

Linear model with standardized variables

The coordinates of the reference points can be numerically very large, for example if they are given in the Gauss-Krüger coordinate system. Standardization of the coordinates improves numerical properties of the estimation.

The original vectors used in the linear model are
$\underline{\boldsymbol{xy}}_m = \begin{pmatrix} \boldsymbol{x}_m \\ \boldsymbol{y}_m \end{pmatrix}$, $\underline{\boldsymbol{yx}}_m = \begin{pmatrix} -\boldsymbol{y}_m \\ \boldsymbol{x}_m \end{pmatrix}$ and $\underline{\boldsymbol{xy}}_d = \begin{pmatrix} \boldsymbol{x}_d \\ \boldsymbol{y}_d \end{pmatrix}$.
Now we standardize these vectors and get
$\underline{\widetilde{\boldsymbol{xy}}}_d = \frac{\underline{\boldsymbol{xy}}_d - \overline{\underline{xy}}_d \mathbf{1}}{\sigma_d}$, $\underline{\widetilde{\boldsymbol{xy}}}_m = \frac{\underline{\boldsymbol{xy}}_m - \overline{\underline{xy}}_m \mathbf{1}}{\sigma_{\underline{xy}_m}}$ and $\underline{\widetilde{\boldsymbol{yx}}}_m = \frac{\underline{\boldsymbol{yx}}_m - \overline{\underline{xy}}_m \mathbf{1}}{\sigma_{\underline{yx}_m}}$.
We define

$$\underline{\mathbf{10}} = \begin{pmatrix} 1 \\ \vdots \\ 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix}, \quad \underline{\mathbf{01}} = \begin{pmatrix} 0 \\ \vdots \\ 0 \\ 1 \\ \vdots \\ 1 \end{pmatrix}$$

and we can now write the linear model with standardized variables.

$$\underline{\widetilde{\boldsymbol{xy}}}_d = \underline{\mathbf{10}}\tilde{A} + \underline{\mathbf{01}}\tilde{B} + \tilde{C}\underline{\widetilde{\boldsymbol{xy}}}_m + \tilde{D}\underline{\widetilde{\boldsymbol{yx}}}_m$$

$$\frac{\underline{\boldsymbol{xy}}_d - \overline{\underline{xy}}_d \mathbf{1}}{\sigma_d} = \underline{\mathbf{10}}\tilde{A} + \underline{\mathbf{01}}\tilde{B} + \tilde{C}\frac{\underline{\boldsymbol{xy}}_m - \overline{\underline{xy}}_m \mathbf{1}}{\sigma_{\underline{xy}_m}} + \tilde{D}\frac{\underline{\boldsymbol{yx}}_m - \overline{\underline{yx}}_m \mathbf{1}}{\sigma_{\underline{yx}_m}}$$

$$\underline{\boldsymbol{xy}}_d - \overline{\underline{xy}}_d \mathbf{1}$$

$$= \underline{\mathbf{10}}(\tilde{A}\sigma_d + \overline{\underline{xy}}_d) + \underline{\mathbf{01}}(\tilde{B}\sigma_d + \overline{\underline{xy}}_d) + \frac{\tilde{C}\sigma_d}{\sigma_{\underline{xy}_m}}\underline{\boldsymbol{xy}}_m - \frac{\tilde{C}\sigma_d \overline{\underline{xy}}_m \mathbf{1}}{\sigma_{\underline{xy}_m}} + \frac{\tilde{D}\sigma_d}{\sigma_{\underline{yx}_m}}\underline{\boldsymbol{yx}}_m - \frac{\tilde{D}\sigma_d \overline{\underline{yx}}_m \mathbf{1}}{\sigma_{\underline{yx}_m}}$$

We can compute the parameters of the original linear model by using the parameters of the model with the standardized variables.

$$\Rightarrow$$

$$A = \tilde{A}\sigma_d + \underline{\overline{xy}}_d - \frac{\tilde{C}\sigma_d \underline{\overline{xy}}_m}{\sigma_{\underline{xy}_m}} - \frac{\tilde{D}\sigma_d \underline{\overline{yx}}_m}{\sigma_{\underline{yx}_m}}$$

$$B = \tilde{B}\sigma_d + \underline{\overline{xy}}_d - \frac{\tilde{C}\sigma_d \underline{\overline{xy}}_m}{\sigma_{\underline{xy}_m}} - \frac{\tilde{D}\sigma_d \underline{\overline{yx}}_m}{\sigma_{\underline{yx}_m}}$$

$$C = \tilde{C}\frac{\sigma_d}{\sigma_{\underline{xy}_m}}$$

$$D = \tilde{D}\frac{\sigma_d}{\sigma_{\underline{yx}_m}}$$

The predicted values $\underline{\widehat{\widetilde{xy}}}_d$ have to be transformed back to the original space.

$$\underline{\widehat{\widetilde{xy}}}_d = \hat{\tilde{A}}\underline{\mathbf{10}} + \hat{\tilde{B}}\underline{\mathbf{01}} + \hat{\tilde{C}}\underline{\tilde{xy}}_m + \hat{\tilde{D}}\underline{\tilde{yx}}_m$$

$$\underline{\widehat{\widetilde{xy}}}_d = \frac{\underline{\widehat{xy}}_d - \underline{\overline{xy}}_d}{\sigma_d}$$

We easily obtain $\underline{\widehat{xy}}_d$ by multiplying with its standard deviation and adding its mean.

$$\underline{\widehat{xy}}_d = \underline{\widehat{\widetilde{xy}}}_d\sigma_d + \underline{\overline{xy}}_d$$

2.3 Nonlinear Transformation

Several nonlinear transformations could be useful, but due to simplicity only this quadratic transformation is considered. A practical use may be to fit a background map obtained by a satellite picture or a snapshot of a 3d object.

2.3.1 Assumptions

Given reference points in the map space $\begin{pmatrix} x_{m,i} \\ y_{m,i} \end{pmatrix}$ correspond to given reference points in the data space $\begin{pmatrix} x_{d,i} \\ y_{d,i} \end{pmatrix}$.

2.3.2 Transformation

$$x_{d,i} = A + \begin{pmatrix} C & E \end{pmatrix}\begin{pmatrix} x_{m,i} \\ y_{m,i} \end{pmatrix} + \begin{pmatrix} x_{m,i} & y_{m,i} \end{pmatrix}\begin{pmatrix} G & \frac{K}{2} \\ \frac{K}{2} & I \end{pmatrix}\begin{pmatrix} x_{m,i} \\ y_{m,i} \end{pmatrix}$$

$$y_{d,i} = B + \begin{pmatrix} D & F \end{pmatrix}\begin{pmatrix} x_{m,i} \\ y_{m,i} \end{pmatrix} + \begin{pmatrix} x_{m,i} & y_{m,i} \end{pmatrix}\begin{pmatrix} H & \frac{L}{2} \\ \frac{L}{2} & J \end{pmatrix}\begin{pmatrix} x_{m,i} \\ y_{m,i} \end{pmatrix}$$

$$\begin{pmatrix} x_{d,i} \\ y_{d,i} \end{pmatrix} = \begin{pmatrix} A \\ B \end{pmatrix} + \begin{pmatrix} C & E \\ D & F \end{pmatrix}\begin{pmatrix} x_{m,i} \\ y_{m,i} \end{pmatrix} + \begin{pmatrix} G & I \\ H & J \end{pmatrix}\begin{pmatrix} x_{m,i}^2 \\ y_{m,i}^2 \end{pmatrix} + x_m y_m \begin{pmatrix} K \\ L \end{pmatrix}$$

The transformation is a combination of an affine transformation and a quadratic form.

2.3.3 Linear Model

Again we want to find a linear model for this transformation, so that we can easily estimate the parameters.

$$\underline{\boldsymbol{xy}}_d = \begin{pmatrix} \boldsymbol{x}_d \\ \boldsymbol{y}_d \end{pmatrix} = \boldsymbol{X}\boldsymbol{\beta} + \boldsymbol{\epsilon}$$

where

$$\boldsymbol{X} = \begin{pmatrix} 1 & 0 & x_{m,1} & y_{m,1} & 0 & 0 & x_{m,1}^2 & y_{m,1}^2 & 0 & 0 & x_{m,1}y_{m,1} & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & 0 & x_{m,n} & y_{m,n} & 0 & 0 & x_{m,n}^2 & y_{m,n}^2 & 0 & 0 & x_{m,n}y_{m,n} & 0 \\ 0 & 1 & 0 & 0 & x_{m,1} & y_{m,1} & 0 & 0 & x_{m,1}^2 & y_{m,1}^2 & 0 & x_{m,1}y_{m,1} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 1 & 0 & 0 & x_{m,n} & y_{m,n} & 0 & 0 & x_{m,n}^2 & y_{m,n}^2 & 0 & x_{m,n}y_{m,n} \end{pmatrix},$$

$$\boldsymbol{\beta} = \begin{pmatrix} A & B & C & D & E & F & G & H & I & J & K & L \end{pmatrix}^\top \in \mathbb{R}^{12}$$

and n denotes the number of reference points.

Chapter 3

Linear Models in R

A quick and compact way to specify a linear model is the formula interface, implemented in R. The most general form of a formula is `response ~ expression`, where `response` is the response variable and `expression` can be just one regressor or several joined by operators. The objects used in the formula have to be of the same length.

The basic operators are `+`, `-`, `*` or `:`, where
`+` adds a term to `expression`,
`-` removes a term from `expression`,
`:` is for interaction terms `a:b` and
`*` is a shortcut for `a + b + a:b` (for more details, see for example: Ripley and Venables, 2002).

3.1 Least Square Regression in R

For getting the least square estimation of the parameters in a linear model, we just have to define a formula and call the function `lm`. The data can either be attached or has to be specified in the call of `lm`. For example a basic call for a linear model with two regressors and an intercept is `lm(y~x+y)`. There are several other parameters for the call of the function `lm`.

```
lm(formula, data, subset, weights, na.action,
   method = "qr", model = TRUE, x = FALSE, y = FALSE, qr = TRUE,
   singular.ok = TRUE, contrasts = NULL, offset, ...)
```

The default (and at the moment, the only implemented) method for computing the estimation is the QR decomposition (Wikipedia, 2007e). Estimating the parameters is equivalent to solving the normal equation $(\boldsymbol{X}^\top \boldsymbol{X})\boldsymbol{b} = \boldsymbol{X}^\top \boldsymbol{y}$. The difficult task is to invert $(\boldsymbol{X}^\top \boldsymbol{X})$, which is avoided with QR decomposition.

We want to factorize the $(n \times p)$ matrix $\boldsymbol{X}$ into $\boldsymbol{QR}$, where $\boldsymbol{Q}$ is an orthogonal matrix and R is an upper triangle matrix. In R the factorization is made with an algorithm using Householder transformations (for details, see for example: Praetorius, 2005) .

We factorize $\boldsymbol{X}$

$$\boldsymbol{X} = \boldsymbol{QR} \ ,$$

so that

$$\boldsymbol{X}^\top \boldsymbol{X} = \boldsymbol{R}^\top \boldsymbol{Q}^\top \boldsymbol{Q}\boldsymbol{R} = \boldsymbol{R}^\top \boldsymbol{R} \quad .$$

The normal equation

$$(\boldsymbol{X}^\top \boldsymbol{X})\boldsymbol{b} = \boldsymbol{X}^\top \boldsymbol{y}$$

can now be written as

$$\boldsymbol{R}^\top \boldsymbol{R}\boldsymbol{b} = \boldsymbol{R}^\top \boldsymbol{Q}^\top \boldsymbol{y}$$

and directly solved in the form

$$\boldsymbol{R}\boldsymbol{b} = \boldsymbol{Q}^\top \boldsymbol{y} \quad ,$$

because of the upper triangle structure of $\boldsymbol{R}$.

3.2 Least Trimmed Square Regression in R

The function used for computing the LTS estimator is the function `ltsReg` from the package `robustbase`. The call is similar to the call of `lm`

```
ltsReg(formula, data, subset, weights, na.action,
       model = TRUE, x.ret = FALSE, y.ret = FALSE,
       contrasts = NULL, offset, ...)
```

The most basic way to compute the LTS estimator is by drawing trial subsets of h observations and calculate the LS estimator for each subset. This is repeated many times and the subset with the least sum of squares is chosen. For large data sets it is not possible to consider all h-subsets, so many h-subsets are drawn at random.

For large data sets the FAST-LTS algorithm, introduced by Rousseeuw and Van Driessen in 2006, has to be used. This algorithm should provide the exact LTS estimator for small data sets and make LTS regression available for large data sets (for details on computing LTS Regression, see Rousseeuw and Van Driessen, 2006).

Chapter 4

Implementation

4.1 Implementation in R

4.1.1 fitBackground

The function `fitBackground` is the basic function of this package. Its inputs are the model type, used for transformation, the coordinates of the reference points in data space and map space and a Boolean value for the decision whether the variables should be standardized. The output is the transformed map.

```
fitBackground <- function(image.coordinates,data.coordinates,image,
  model.type="sim",standardized=FALSE){
```

The coordinates are converted into the correct form and then saved in an environment called `.SSenv` (see 4.1.5 for further details on the `.SS` functions and environment).

```
  image.coord <- list()
  data.coord <- list()
  image.coord$x <- image.coordinates[,1]
  image.coord$y <- image.coordinates[,2]
  data.coord$x <- data.coordinates[,1]
  data.coord$y <- data.coordinates[,2]
  .SSassignFit("image.coord",image.coord)
  .SSassignFit("data.coord",data.coord)
```

Now the model can be estimated with the function `makeModel` (which is described in the 4.1.3). The model will be written immediately into the `.SSenv` environment.

```
  mod <- makeModel(model.type,standardized)
  .SSassignFit("model",mod)
```

Now we want to transform our map. This is made by predicting the new coordinates for every coordinate of the map (details of `makePrediction` are described in 4.1.4).

```
  image <- makePrediction(mod,image,model.type,standardized)
```

When using LTS-Regression we get a vector of weights of length n, where n is the number of reference points times two. So we have a weight for each coordinate of each

reference point, this can be used to give the user the information which coordinate is left out of consideration.

```
  if(model.type=="robustsim"){
    n <- 1:length(image.coord$x)
    lts.wt.TF <- !mod$lts.wt==1
    x.removed <- n[lts.wt.TF[1:(length(lts.wt.TF)/2)]]
    y.removed
      <- n[lts.wt.TF[(length(lts.wt.TF)/2+1):(length(lts.wt.TF))]]
    .SSassignFit("x.removed",x.removed)
    .SSassignFit("y.removed",y.removed)
    if(length(x.removed)>0){
      x.msg <- paste(x.removed,collapse=", ")
      x.msg <- paste("Point(s) ",x.msg,
        " have zero weight in X-direction\n")
    }else
      x.msg <- ""
    if(length(y.removed)>0){
      y.msg <- paste(y.removed,collapse=", ")
      y.msg <- paste("Point(s) ",y.msg," have zero weight in Y-direction")
    }else
      y.msg <- ""
    if((y.msg!="")||(x.msg!="")){
      msg <- paste(x.msg,y.msg,sep="")
      tkmessageBox(message=msg,icon="info", type="ok")
    }
  }
```

The output of the function is the object `image`, which was returned by the function `makePrediction`:

```
  image
}
```

4.1.2 convertBackground

The function `convertBackground` provides the way to get scanned maps or vector graphic maps into R. The file format of the map has to be Portable Anymap (PNM) or PostScript (PS) (For proceeding PostScript files "Ghostscript" is needed and for windows users the binaries have to be in the working directory or the binary directory has to be assigned to a command path using the DOS command "path"). The maps are saved in the object `.background` in the global environment.

```
convertBackground <- function(filename,type) {
  require("sp")
  require("grImport")
  require("pixmap")
```

If we have a PostScript file, it will be converted into a XML file first. Afterwards it will be imported to R in an object of class `Picture`. The functions used here are from the package `grImport` (for details see Murrell and Richard, 2007).

```
  if(type=="ps"){
    h <- PostScriptTrace(filename)
   filena <- strsplit(filename,"/")
    .background <<- readPicture(paste(filena[[1]][length(filena[[1]])],
      ".xml",sep=""))
```

For pnm files we use the function `read.pnm` from the package `pixmap` (for details see Bivand *et al.*, 2007). A pnm file is a bitmap file, so on each coordinate RGB values are saved. Now we save the information to an object of class `SpatialPointsPictureFrame`, which is the same as `SpatialPointsDataFrame`, except the only data it contains are the three color channels.

```
  }else if(type=="pnm") {
    .background <<- read.pnm(filename)
    .background <<- image2spatial(.background)
  }
  activateMenus()
}
```

4.1.3 makeModel

`makeModel` is the function, which really calls the R functions for estimating. The possible values for `model.type` are "sim", "affine", "nonlinear" and "robustsim". While the first three correspond to the three transformations described in Chapter 2, the last one is a similarity transformation with robust estimated parameters (see 1.2 for LTS-regression).

```
makeModel <- function(model.type,standardized){
  image.coord <- .SSgetFit("image.coord")
  data.coord <- .SSgetFit("data.coord")
```

Now the columns of the design matrix are build, either in standardized version or from the original reference points (details of the design matrix, see 2.2.4). Afterwards the function `lm` is called with a formula as input.

```
  if(model.type=="sim"){
    if(standardized){
      data.coord_xy <- c(data.coord$x,data.coord$y)
      .SSassignFit("meand",mean(data.coord_xy))
      .SSassignFit("sdd",sd(data.coord_xy))
      data.coord_xy
        <- (data.coord_xy - mean(data.coord_xy))/sd(data.coord_xy)
      n <- length(image.coord$x)
      x1 <- c(rep(1,n),rep(0,n))
      x2 <- 1 - x1
      x3 <- c(image.coord$x,image.coord$y)
```

```
    x4 <- c(-image.coord$y,image.coord$x)
    .SSassignFit("sdx3",sd(x3))
    .SSassignFit("meanx3",mean(x3))
    .SSassignFit("sdx4",sd(x4))
    .SSassignFit("meanx4",mean(x4))
    x3 <- (x3 - mean(x3))/sd(x3)
    x4 <- (x4 - mean(x4))/sd(x4)
    model <- lm(data.coord_xy ~ 0 + x1 + x2 + x3 +x4)
  }else{
    data.coord_xy <- c(data.coord$x,data.coord$y)
    n <- length(image.coord$x)
    x1 <- c(rep(1,n),rep(0,n))
    x2 <- 1 - x1
    x3 <- c(image.coord$x,image.coord$y)
    x4 <- c(-image.coord$y,image.coord$x)
    model <- lm(data.coord_xy ~ 0 + x1 + x2 + x3 +x4)
  }
```

The same happens for the robust similarity transformation, except that we call the function `ltsReg` to robustly estimate the parameters.

```
}else if(model.type=="robustsim"){
  require("robustbase")
  if(standardized){
    data.coord_xy <- c(data.coord$x,data.coord$y)
    .SSassignFit("meand",mean(data.coord_xy))
    .SSassignFit("sdd",sd(data.coord_xy))
    data.coord_xy
      <- (data.coord_xy - mean(data.coord_xy))/sd(data.coord_xy)
    n <- length(image.coord$x)
    x1 <- c(rep(1,n),rep(0,n))
    x2 <- 1 - x1
    x3 <- c(image.coord$x,image.coord$y)
    x4 <- c(-image.coord$y,image.coord$x)
    .SSassignFit("sdx3",sd(x3))
    .SSassignFit("meanx3",mean(x3))
    .SSassignFit("sdx4",sd(x4))
    .SSassignFit("meanx4",mean(x4))
    x3 <- (x3 - mean(x3))/sd(x3)
    x4 <- (x4 - mean(x4))/sd(x4)
    model <- ltsReg(data.coord_xy ~ 0 + x1 + x2 + x3 +x4)
  }else{
    data.coord_xy <- c(data.coord$x,data.coord$y)
    n <- length(image.coord$x)
    x1 <- c(rep(1,n),rep(0,n))
    x2 <- 1 - x1
    x3 <- c(image.coord$x,image.coord$y)
```

```
    x4 <- c(-image.coord$y,image.coord$x)
    model <- ltsReg(data.coord_xy ~ 0 + x1 + x2 + x3 +x4)
  }
```

The columns of the design matrix for the affine transformation are different, but afterwards the function `lm` is called again. (Details of the design matrix, see 2.1.4)

```
}else if(model.type=="affine"){
  data.coord_xy <- c(data.coord$x,data.coord$y)
  n <- length(image.coord$x)
  x1 <- c(rep(1,n),rep(0,n))
  x2 <- 1 - x1
  x3 <- c(image.coord$x,rep(0,n))
  x4 <- c(image.coord$y,rep(0,n))
  x5 <- c(rep(0,n),image.coord$y)
  x6 <- c(rep(0,n),image.coord$x)
  model <- lm(data.coord_xy ~ 0 + x1 + x2 + x3 + x4 +x5 + x6)
```

For the nonlinear transformation of order 2, we build the same columns as for the affine transformation, but we call the `lm` function with a different formula.

```
}else if(model.type=="nonlinear"){
  data.coord_xy <- c(data.coord$x,data.coord$y)
  n <- length(image.coord$x)
  x1 <- c(rep(1,n),rep(0,n))
  x2 <- 1 - x1
  x3 <- c(image.coord$x,rep(0,n))
  x4 <- c(image.coord$y,rep(0,n))
  x5 <- c(rep(0,n),image.coord$y)
  x6 <- c(rep(0,n),image.coord$x)
  model <- lm(data.coord_xy ~ 0 + x1 + x2 + x3 + x4 +x5 + x6 +
    x3^2 + x4^2 +x5^2 + x6^2 + x3*x4 + x5*x6)
}
model
}
```

4.1.4 makePrediction

The function `makePrediction` basically builds a data frame of the correct structure, which is used by `predict`.

```
makePrediction <- function(mod,image,model.type,standardized){
  require("sp")
  require("grImport")
  require("pixmap")
```

For maps of class `SpatialPointsPictureFrame` we just read the slot `coords` from the object, to get a vector `x` containing the x-coordinates and a vector `y` for the y-coordinates.

```
if(class(image)=="SpatialPointsPictureFrame"){
  x <- image@coords[,1]
  y <- image@coords[,2]
  n1 <- length(x)
  image.coord <- .SSgetFit("image.coord")
  x <- c(x,image.coord$x)
  y <- c(y,image.coord$y)
```

Each polygonal figure is saved in the list `paths`, which is a slot of an object of class `Picture`. We now concatenate all x- and y-coordinates of the paths into the vectors `x` and `y`.

```
}else{
  x <- vector()
  y <- vector()
  l <- vector()
  for(i in 1:length(image@paths)){
    x <- c(x,image@paths[[i]]@x)
    y <- c(y,image@paths[[i]]@y)
    l[i] <- length(image@paths[[i]]@y)
  }
  x <- as.numeric(x)
  y <- as.numeric(y)
  n1 <- length(x)
  image.coord <- .SSgetFit("image.coord")
  x <- c(x,image.coord$x)
  y <- c(y,image.coord$y)
}
```

We generate a data frame with the structure of the design matrix of the model. If we have used standardized variables, we now have to standardize before prediction also and afterwards we have to transform back the predicted values.

```
if(model.type=="sim"){
  if(standardized){
    meanx3 <- .SSgetFit("meanx3")
    meanx4 <- .SSgetFit("meanx4")
    sdx3 <- .SSgetFit("sdx3")
    sdx4 <- .SSgetFit("sdx4")
    meand <- .SSgetFit("meand")
    sdd <- .SSgetFit("sdd")
    n <- length(x)
    x1 <- c(rep(1,n),rep(0,n))
    x2 <- 1 - x1
    x3 <- c(x,y)
    x4 <- c(-y,x)
    x3 <- (x3 - meanx3)/sdx3
    x4 <- (x4 - meanx4)/sdx4
```

```
      coords.to.predict <- as.data.frame(cbind(x1,x2,x3,x4))
      new.coords
        <- matrix((predict(mod,coords.to.predict)*sdd) + meand,ncol=2)
    }else{
      n <- length(x)
      x1 <- c(rep(1,n),rep(0,n))
      x2 <- 1 - x1
      x3 <- c(x,y)
      x4 <- c(-y,x)
      coords.to.predict <- as.data.frame(cbind(x1,x2,x3,x4))
      new.coords <- matrix(predict(mod,coords.to.predict),ncol=2)
    }
  }else if(model.type=="robustsim"){
    if(standardized){
      meanx3 <- .SSgetFit("meanx3")
      meanx4 <- .SSgetFit("meanx4")
      sdx3 <- .SSgetFit("sdx3")
      sdx4 <- .SSgetFit("sdx4")
      meand <- .SSgetFit("meand")
      sdd <- .SSgetFit("sdd")
      n <- length(x)
      x1 <- c(rep(1,n),rep(0,n))
      x2 <- 1 - x1
      x3 <- c(x,y)
      x4 <- c(-y,x)
      x3 <- (x3 - meanx3)/sdx3
      x4 <- (x4 - meanx4)/sdx4
      coords.to.predict <- as.data.frame(cbind(x1,x2,x3,x4))
```

We use the function `predictLts` for predicting values from a model, estimated by the function `LtsReg`.

```
      new.coords
        <- matrix((predictLts(mod,coords.to.predict)*sdd) + meand,ncol=2)
    }else{
      n <- length(x)
      x1 <- c(rep(1,n),rep(0,n))
      x2 <- 1 - x1
      x3 <- c(x,y)
      x4 <- c(-y,x)
      coords.to.predict <- as.data.frame(cbind(x1,x2,x3,x4))
      new.coords <- matrix(predictLts(mod,coords.to.predict),ncol=2)
    }
  }else if(model.type=="affine"){
    n <- length(x)
    x1 <- c(rep(1,n),rep(0,n))
    x2 <- 1 - x1
```

```
    x3 <- c(x,rep(0,n))
    x4 <- c(y,rep(0,n))
    x5 <- c(rep(0,n),y)
    x6 <- c(rep(0,n),x)
    coords.to.predict <- as.data.frame(cbind(x1,x2,x3,x4,x5,x6))
    new.coords <- matrix(predict(mod,coords.to.predict),ncol=2)
  }else if(model.type=="nonlinear"){
    n <- length(x)
    x1 <- c(rep(1,n),rep(0,n))
    x2 <- 1 - x1
    x3 <- c(x,rep(0,n))
    x4 <- c(y,rep(0,n))
    x5 <- c(rep(0,n),y)
    x6 <- c(rep(0,n),x)
    coords.to.predict <- as.data.frame(cbind(x1,x2,x3,x4,x5,x6))
    new.coords <- matrix(predict(mod,coords.to.predict),ncol=2)
  }
```

The predicted coordinates in `new.coords` are written back into the map object.

```
  if(class(image)=="SpatialPointsPictureFrame"){
    colnames(new.coords) <- colnames(image@coords)
    class(image) <- "SpatialPointsDataFrame"
    image@coords <- new.coords[1:n1,]
    image@bbox[1,] <- c(min(new.coords[1:n1,1]),max(new.coords[1:n1,1]))
    image@bbox[2,] <- c(min(new.coords[1:n1,2]),max(new.coords[1:n1,2]))
    class(image) <- "SpatialPointsPictureFrame"
    image.coord$x <- new.coords[(n1+1):length(new.coords[,1]),1]
    image.coord$y <- new.coords[(n1+1):length(new.coords[,1]),2]
    .SSassignFit("image.coord",image.coord)
    image
```

For maps of class `Picture` this task is more complex, because the coordinates have to be split into every single path again.

```
  }else{
    image.coord$x <- new.coords[(n1+1):length(new.coords[,1]),1]
    image.coord$y <- new.coords[(n1+1):length(new.coords[,1]),2]
    .SSassignFit("image.coord",image.coord)
    new.coords <- new.coords[1:n1,]
    xmin <- min(new.coords[,1])
    xmax <- max(new.coords[,1])
    ymin <- min(new.coords[,2])
    ymax <- max(new.coords[,2])
    for(i in 1:length(l)){
      if(i==1){
        image@paths[[i]]@x <- new.coords[1:l[i],1]
        image@paths[[i]]@y <- new.coords[1:l[i],2]
```

```
      }else{
        s <- sum(l[1:(i-1)])
        image@paths[[i]]@x <- new.coords[(s+1):(s+l[i]),1]
        image@paths[[i]]@y <- new.coords[(s+1):(s+l[i]),2]
      }
    }
    image@summary@xscale <- c(xmin,xmax)
    image@summary@yscale <- c(ymin,ymax)
    image
    }
  }
```

4.1.5 Miscellaneous Workhorses

`.SSgetFit`, `.SSassignFit`, `.SSexistsFit` and `.SSrmFit` are functions to communicate with an environment called `.SSenv`. This environment is used to store data, which should not stuff the global environment. The list `Fit` in this environment is reserved for data coming from functions in connection with map fitting. `.SSgetFit` is for reading a variable out of this list, `.SSassignFit` is for adding or overwriting variables, `.SSexistsFit` is for testing if a variable exists in the list.

```
.SSgetFit <- function(x){
  Fit <- get("Fit", envir=.SSenv, inherits=FALSE)
  Fit[[x]]
}
.SSassignFit <- function(x, value){
  Fit <- get("Fit", envir=.SSenv, inherits=FALSE)
  Fit[[x]] <- value
  assign(paste("Fit", sep=""), Fit, envir=.SSenv)
}
.SSexistsFit <- function(x){
  Fit <- get("Fit", envir=.SSenv, inherits=FALSE)
  TF <- names(Fit)==x
  if(any(TF))
    TRUE
  else
    FALSE
}
.SSrmFit <- function(x){
  Fit <- get("Fit", envir=.SSenv, inherits=FALSE)
  Fit <- Fit[names(Fit)!=x]
  assign(paste("Fit", sep=""), Fit, envir=.SSenv)
}
```

Since there is no applicable method of `predict` for models generated by the function `ltsReg`, `predictLts` is defined here. This function is used by `makePrediction`.

```
predictLts <- function(model,newdata){
```

```
  m.matrix <- newdata
  as.matrix(m.matrix)%*%coef(model)
}
```

The function `makeSpatialdata` creates an object of class `SpatialPointsDataFrame`.

```
makeSpatialdata <- function(xcoord,ycoord,data){
  coords <- SpatialPoints(cbind(xcoord,ycoord))
  spatialdata <- SpatialPointsDataFrame(coords, data)
  spatialdata
}
```

`image2spatial` converts an image of class `pixmapRGB` into an object of class `SpatialPointsPictureFrame`.

```
image2spatial <- function(image){
  xc <- vector()
  for(i in 1:(image@size[2]))
    xc <- c(xc,rep(i,image@size[1]))
  yc <- image@size[1]-rep(0:(image@size[1]-1),image@size[2])
  colors <- data.frame(red=as.vector(image@red),
    blue=as.vector(image@blue),green=as.vector(image@green))
  spat <- makeSpatialdata(xc,yc,colors)
  class(spat) <- "SpatialPointsPictureFrame"
  spat
}
```

`plot.SpatialPointsPictureFrame` is the method for plotting objects of the name giving class. This is done by plotting every pixel of the map with the specified color. Additional features are adding the map to an existing plot by setting the parameter `add` to TRUE and specifying a range for adding. `addRange` should be FALSE or a vector of length four with the coordinates of the left, bottom corner and the right, top corner of the range.

```
plot.SpatialPointsPictureFrame <- function(x,add=FALSE,pch=20,
  add.Range=0,xlab="",ylab="",asp=1,...){
  pic <- x
  col.set <- rgb(pic@data$red,pic@data$green,pic@data$blue)
  if(add==TRUE){
    if(length(add.Range)!=4)
      points(pic@coords[,2]~pic@coords[,1],col=col.set,pch=pch,...)
    else{
      l1.1 <- pic@coords[,1]>=add.Range[1]
      l1.2 <- pic@coords[,1]<=add.Range[3]
      l1.3 <- l1.1+l1.2-1
      l2.1 <- pic@coords[,2]>=add.Range[2]
      l2.2 <- pic@coords[,2]<=add.Range[4]
      l2.3 <- l2.1+l2.2-1
```

```
      l3 <- l2.3+l1.3
      l3[l3<2] <- 0
      l <- as.logical(l3)
      points(pic@coords[l,2]~pic@coords[l,1],col=col.set[l],pch=pch,...)
      }
  }else{
    plot(pic@coords[,2]~pic@coords[,1],xlim=pic@bbox[1,],
      ylim=pic@bbox[2,],xlab=xlab,ylab=ylab,col=col.set,
      pch=pch,asp=asp,...)
    }
}
```

With `setMethod` the function is defined as plot method for objects of the class.

```
setMethod("plot", c("SpatialPointsPictureFrame","missing"),
  plot.SpatialPointsPictureFrame,where=.SSenv)
```

Although there is a function for plotting object of class `Picture` in the package `grImport`, there is a need of a new one, because the default plotting method can be used on a `grid` graphic device only. The function `plot.Picture` basically just plots every path with the designated color. If the parameter `slow` is set to `TRUE`, the paths are plotted like in PostScript. Otherwise the whole path is plotted at once, like every coordinate pair is connected with a `line to` command and `move` commands are ignored.

```
plot.Picture <- function(x,add=FALSE,xlab="",ylab="",asp=1,slow=FALSE,...){
  pic<-x
  xl <- pic@summary@xscale
  yl <- pic@summary@yscale
  if(add==FALSE)
    plot(1,xlim=xl,ylim=yl,type="n",xlab=xlab,ylab=ylab,asp=asp,...)
  for(i in 1:length(pic@paths)){
    if(slow==TRUE){
      for(j in 1:length(names(pic@paths[[i]]@x))){
         if(names(pic@paths[[i]]@x)[j]=="line"){
           lines(pic@paths[[i]]@x[c(j-1,j)],pic@paths[[i]]@y[c(j-1,j)],col=pic@paths[[i]]
             lwd=pic@paths[[i]]@lwd)
         }
      }
    }else
      lines(pic@paths[[i]]@x,pic@paths[[i]]@y,col=pic@paths[[i]]@rgb,
        lwd=pic@paths[[i]]@lwd)
  }
}
```

4.2 Implementation in DAS+R

DAS+R is a graphical user interface based on the Rcommander by John Fox (for details see his paper Fox, 2005). Its menus are written in Tcl/Tk (for details on Tcl/Tk see for example Hobbs *et al.*, 2003). A basic idea is a diversion between functions just for internal use by the GUI and functions supposed to work in batch mode. All functions starting with a small letter work in batch mode. Furthermore all functions with a capital letter first and the word "GUI" at the end, are functions in direct connection with DAS+R, therefore we can not use them in batch mode.

If we start to work with a background map or we want to get a map, the function `ConvertBackgroundGUI` should be started. It clears the `.SSenv` environment for the new map and starts `FitBackgroundGUI` the main function.

```
ConvertBackgroundGUI <- function(){
  require("sp")
  require("grImport")
  require("pixmap")
  if(.SSexistsFit("image"))
    .SSrmFit("image")
  if(.SSexistsFit("clickedOK"))
    .SSrmFit("clickedOK")
  if(.SSexistsFit("data.coord"))
    .SSrmFit("data.coord")
  if(.SSexistsFit("image.coord"))
    .SSrmFit("image.coord")
  if(.SSexistsFit("backup"))
    .SSrmFit("backup")
  if(.SSexistsFit("data.device"))
    .SSrmFit("data.device")
  if(.SSexistsFit("model"))
    .SSrmFit("model")
  if(.SSexistsFit("model.type"))
    .SSrmFit("model.type")
  if(.SSexistsFit("move.p"))
    .SSrmFit("move.p")
  if(.SSexistsFit("p1"))
    .SSrmFit("p1")
  if(.SSexistsFit("p2"))
    .SSrmFit("p2")
  if(.SSexistsFit("xcoord"))
    .SSrmFit("xcoord")
  if(.SSexistsFit("ycoord"))
    .SSrmFit("ycoord")
  FitBackgroundGUI()
}
```

Through the GUI, the background map can be saved to a file with the extension

".Rmap". The function `LoadBackgroundGUI` opens a file menu to load a Rmap file. If a background already exists, it and all entries in the `.SSenv` environment concerning the background map will be deleted.

```
LoadBackgroundGUI <- function(){
  require("sp")
  require("grImport")
  require("pixmap")
  filename <- tclvalue(tkgetOpenFile(filetypes=
    '{"Rmap files" {".Rmap"}} {"All Files" {"*"}}'))
  if(filename!=""){
    if(.SSexistsFit("image"))
      .SSrmFit("image")
    if(.SSexistsFit("clickedOK"))
      .SSrmFit("clickedOK")
    if(.SSexistsFit("data.coord"))
      .SSrmFit("data.coord")
    if(.SSexistsFit("image.coord"))
      .SSrmFit("image.coord")
    if(.SSexistsFit("backup"))
      .SSrmFit("backup")
    if(.SSexistsFit("data.device"))
      .SSrmFit("data.device")
    if(.SSexistsFit("model"))
      .SSrmFit("model")
    if(.SSexistsFit("model.type"))
      .SSrmFit("model.type")
    if(.SSexistsFit("move.p"))
      .SSrmFit("move.p")
    if(.SSexistsFit("p1"))
      .SSrmFit("p1")
    if(.SSexistsFit("p2"))
      .SSrmFit("p2")
    if(.SSexistsFit("xcoord"))
      .SSrmFit("xcoord")
    if(.SSexistsFit("ycoord"))
      .SSrmFit("ycoord")
    command <- paste("load(\"",filename,"\")",sep="")
    doItAndPrint(command)
    activateMenus()
    .SSassignFit("image",".background")
  }
}
```

`FitBackgroundGUI` builds the dialog and the plot to identify reference points. Furthermore it opens the dialog for importing an image file to R.

```
FitBackgroundGUI <- function(){
```

```
require("sp")
require("grImport")
require("pixmap")
```

The first part of the function `getPoints` (before the definition of `click.point`) is for plotting the background map and the location map only.

```
getPoints <- function(imag,spatialdata){
```

At the start the location map of the data and the background map are just plotted overlapping each other. This is done by creating two screens on the graphic device with the function `split.screen`. With `screen(1)` the first screen is set active and with `screen(2)` the second.

```
  if((!.SSexistsFit("deleted"))&&(!.SSexistsFit("added"))){
    if(!.SSexistsFit("clickedOK")){
      if(!.SSexistsFit("data.device")){
```

A new graphic device is opened with `x11()` and its number is saved.

```
        x11()
        data.device <- dev.cur()
        .SSassignFit("data.device",data.device)
        split.screen(c(1,1))
        split.screen(c(1,1))
        screen(1)
        plot(imag)
        p1 <- par(c("xlog","ylog","adj", "bty", "cex", "col",
          "crt", "err", "font", "lab","las", "lty", "lwd",
          "mar", "mex", "mfg", "mgp", "pch","pty", "smo",
          "srt", "tck", "usr", "xaxp", "xaxs", "xaxt", "xpd",
          "yaxp", "yaxs", "yaxt", "fig"))
        .SSassignFit("p1",p1)
```

Here the coordinates of the reference points in the image space are plotted, if reference points already exist.

```
        if(.SSexistsFit("image.coord")){
          points(.SSgetFit("image.coord"),pch=7,col="red")
        }
        screen(2)
        plot(spatialdata)
        p2 <- par(c("xlog","ylog","adj", "bty", "cex", "col",
          "crt", "err", "font", "lab","las", "lty", "lwd",
          "mar", "mex", "mfg", "mgp", "pch","pty", "smo",
          "srt", "tck", "usr", "xaxp", "xaxs", "xaxt", "xpd",
          "yaxp", "yaxs", "yaxt", "fig"))
        .SSassignFit("p2",p2)
```

Here the coordinates of the reference points in the data space are plotted.

```
    if(.SSexistsFit("data.coord")){
      points(.SSgetFit("data.coord"),pch=7,col="blue")
    }
  }else{
    data.device <- .SSgetFit("data.device")
    dev.set(data.device)
  }
}else{
```

If the user already has performed a transformation, the background map and the location map is approximately in the same coordinate system and can be plotted without overlaying screens.

```
if(!.SSexistsFit("data.device")){
  x11()
  data.device <- dev.cur()
  .SSassignFit("data.device",data.device)
  plot(spatialdata)
  plot(imag,add=TRUE)
  plot(spatialdata,add=TRUE)
  if(.SSexistsFit("image.coord")){
    points(.SSgetFit("image.coord"),pch=7,col="red")
    text(.SSgetFit("image.coord"),labels=1:length(.SSgetFit(
      "image.coord")$x),col="yellow")
  }
  if(.SSexistsFit("data.coord")){
    points(.SSgetFit("data.coord"),pch=7,col="blue")
    text(.SSgetFit("data.coord"),labels=1:length(.SSgetFit(
      "image.coord")$x),col="yellow")
  }
}else{
  data.device <- .SSgetFit("data.device")
  dev.set(data.device)
  plot(spatialdata)
  plot(imag,add=TRUE)
  plot(spatialdata,add=TRUE)
  if(.SSexistsFit("image.coord")){
    points(.SSgetFit("image.coord"),pch=7,col="red")
    text(.SSgetFit("image.coord"),labels=1:length(.SSgetFit(
      "image.coord")$x),col="yellow")
  }
  if(.SSexistsFit("data.coord")){
    points(.SSgetFit("data.coord"),pch=7,col="blue")
    text(.SSgetFit("data.coord"),labels=1:length(.SSgetFit(
      "image.coord")$x),col="yellow")
  }
}
```

```
  }
}else{
  if(.SSexistsFit("deleted"))
    .SSrmFit("deleted")
  if(.SSexistsFit("added"))
    .SSrmFit("added")
}
```

In the function `click.point` the reference points are selected with the function `identify` from the location map and with `locator` from the background map.

```
click.point <- function(){
  .SSassignFit("model.type",as.character(tclvalue(modelVariable)))
```

If `clickedOK` does not exist, it means that the background map has not been transformed yet. This makes getting the coordinates a bit more complex, because of the two overlaying screens.

```
  if(!.SSexistsFit("clickedOK")){
```

If there is no graphic device with the plot of background map and location map, it is plotted again.

```
    if(!.SSexistsFit("data.device")){
      x11()
      data.device <- dev.cur()
      .SSassignFit("data.device",data.device)
      split.screen(c(1,1))
      split.screen(c(1,1))
      screen(1)
      plot(imag)
      p1 <- par(c("xlog","ylog","adj", "bty", "cex", "col",
        "crt", "err", "font", "lab","las", "lty", "lwd",
        "mar", "mex", "mfg", "mgp", "pch","pty", "smo",
        "srt", "tck", "usr", "xaxp", "xaxs", "xaxt", "xpd",
        "yaxp", "yaxs", "yaxt", "fig"))
      .SSassignFit("p1",p1)
      if(.SSexistsFit("image.coord")){
        points(.SSgetFit("image.coord"),pch=7,col="red")
      }
      screen(2)
      plot(spatialdata)
      p2 <- par(c("xlog","ylog","adj", "bty", "cex", "col",
        "crt", "err", "font", "lab","las", "lty", "lwd",
        "mar", "mex", "mfg", "mgp", "pch","pty", "smo",
        "srt", "tck", "usr", "xaxp", "xaxs", "xaxt", "xpd",
        "yaxp", "yaxs", "yaxt", "fig"))
      .SSassignFit("p2",p2)
```

```
  if(.SSexistsFit("data.coord")){
     points(.SSgetFit("data.coord"),pch=7,col="blue")
  }
}else{
  data.device <- .SSgetFit("data.device")
  dev.set(data.device)
}
```

With `screen(1)` we set the active screen to the level of the background map and ask for the coordinates with `locator`.

```
screen(1)
par(.SSgetFit("p1"))
if(.SSexistsFit("image.coord")){
  image.coord2 <- locator(1,type="p",pch=7,col="red")
  image.coord <- .SSgetFit("image.coord")
  image.coord$x <- c(image.coord$x,image.coord2$x)
  image.coord$y <- c(image.coord$y,image.coord2$y)
}else{
  image.coord <- locator(1,type="p",pch=7,col="red")
  if(.SSexistsFit("new.coords"))
    .SSrmFit("new.coords")
  }
```

Now we set screen two active and with `identify` we can select the reference points in the data space.

```
screen(2)
par(.SSgetFit("p2"))
if(.SSexistsFit("data.coord")){
  spatialdata <- .spatialdata1
  ind <- identify(spatialdata@coords,n=1,plot=FALSE)
  data.coord <- .SSgetFit("data.coord")
  points(spatialdata@coords[ind,1],spatialdata@coords[ind,2],
    type="p",pch=7,col="blue")
  data.coord$x <- c(data.coord$x,spatialdata@coords[ind,1])
  data.coord$y <- c(data.coord$y,spatialdata@coords[ind,2])
}else{
  spatialdata <- .spatialdata1
  ind <- identify(spatialdata@coords,n=1,plot=FALSE)
  points(spatialdata@coords[ind,1],spatialdata@coords[ind,2],
    type="p",pch=7,col="blue")
  data.coord <- list()
  data.coord$x <- spatialdata@coords[ind,1]
  data.coord$y <- spatialdata@coords[ind,2]
  }
.SSassignFit("image.coord",image.coord)
.SSassignFit("data.coord",data.coord)
```

The selecting process is now defined for the background map and the location map already being in the same coordinate system.

Again we look for the right graphic device and if it does not exist, we create it.

```
}else{
  .SSassignFit("added","TRUE")
  if(!.SSexistsFit("data.device")){
    x11()
    data.device <- dev.cur()
    .SSassignFit("data.device",data.device)
    plot(spatialdata)
    plot(imag,add=TRUE)
    plot(spatialdata,add=TRUE)
    if(.SSexistsFit("image.coord")){
      points(.SSgetFit("image.coord"),pch=7,col="red")
      }
    if(.SSexistsFit("data.coord")){
      points(.SSgetFit("data.coord"),pch=7,col="blue")
      }
```

Now we can simply use `locator` without defining an active screen.

```
    if(.SSexistsFit("image.coord")){
      image.coord2 <- locator(1,type="p",pch=7,col="red")
      image.coord <- .SSgetFit("image.coord")
      image.coord$x <- c(image.coord$x,image.coord2$x)
      image.coord$y <- c(image.coord$y,image.coord2$y)
```

Again we use `identify` for the reference point in the location map.

```
    }else
      image.coord2 <- locator(1,type="p",pch=7,col="red")
    if(.SSexistsFit("data.coord")){
      spatialdata <- .spatialdata1
      ind <- identify(spatialdata@coords,n=1,plot=FALSE)
      data.coord <- .SSgetFit("data.coord")
      points(spatialdata@coords[ind,1],spatialdata@coords[ind,2],
        type="p",pch=7,col="blue")
      data.coord$x <- c(data.coord$x,spatialdata@coords[ind,1])
      data.coord$y <- c(data.coord$y,spatialdata@coords[ind,2])
    }else{
      spatialdata <- .spatialdata1
      ind <- identify(spatialdata@coords,n=1,plot=FALSE)
      points(spatialdata@coords[ind,1],spatialdata@coords[ind,2],
        type="p",pch=7,col="blue")
      data.coord <- list()
      data.coord$x <- spatialdata@coords[ind,1]
      data.coord$y <- spatialdata@coords[ind,2]
```

```
            }
            .SSassignFit("data.coord",data.coord)
            .SSassignFit("image.coord",image.coord)
          }else{
```

Now the same selecting process starts, if the graphic device already exists.

```
            data.device <- .SSgetFit("data.device")
            dev.set(data.device)
            if(.SSexistsFit("image.coord")){
              image.coord2 <- locator(1,type="p",pch=7,col="red")
              image.coord <- .SSgetFit("image.coord")
              image.coord$x <- c(image.coord$x,image.coord2$x)
              image.coord$y <- c(image.coord$y,image.coord2$y)
            }else
              image.coord <- locator(1,type="p",pch=7,col="red")
            if(.SSexistsFit("data.coord")){
              spatialdata <- .spatialdata1
              ind <- identify(spatialdata@coords,n=1,plot=FALSE)
              data.coord <- .SSgetFit("data.coord")
              points(spatialdata@coords[ind,1],spatialdata@coords[ind,2],
                type="p",pch=7,col="blue")
              data.coord$x <- c(data.coord$x,spatialdata@coords[ind,1])
              data.coord$y <- c(data.coord$y,spatialdata@coords[ind,2])
            }else{
              spatialdata <- .spatialdata1
              ind <- identify(spatialdata@coords,n=1,plot=FALSE)
              points(spatialdata@coords[ind,1],spatialdata@coords[ind,2],
                type="p",pch=7,col="blue")
              data.coord <- list()
              data.coord$x <- spatialdata@coords[ind,1]
              data.coord$y <- spatialdata@coords[ind,2]
            }
```

At the end we write the new coordinates into the `.SSenv` environment.

```
            .SSassignFit("data.coord",data.coord)
            .SSassignFit("image.coord",image.coord)
        }
      }
```

We close the dialog "Selecting Reference Points" now and restart it with `getPoints`.

```
      closeDialog()
      getPoints(imag,spatialdata)
    }
```

The function `initializeDialog` is for building a Tcl/Tk window. Here starts the part for building the dialog for selecting reference points by defining the necessary functions.

```
initializeDialog(title=paste("Selecting Reference Points: ",
  ActiveDataSet()))
on.save <- function(){
  filename <- tclvalue(tkgetSaveFile(filetypes=
    '{"Rmap files" {".Rmap"}} {"All Files" {"*"}}'))
  command <- paste("save(",.SSgetFit("image"),",file=\"",filename,
    "\")",sep="")
  justDoIt(command)
  closeDialog()
  }
```

The function `on.restart`, which is later connected with the "Reset" button, is defined. It restores the background map from a backup made at the very beginning and deletes all reference points.

```
on.restart <- function(){
  if(.SSexistsFit("backup")){
    command.restore <- paste(.SSgetFit("image")," <- ",
  ".SSgetFit(\"backup\")",sep="")
    justDoIt(command.restore)
  }
  if(.SSexistsFit("data.coord"))
    .SSrmFit("data.coord")
  else
    tkmessageBox(message="There are no points to delete!",
      icon="error",type="ok")
  if(.SSexistsFit("image.coord"))
    .SSrmFit("image.coord")
  if(.SSexistsFit("data.device")){
    dev.off(.SSgetFit("data.device"))
    .SSrmFit("data.device")
  }
  closeDialog()
  if(.SSexistsFit("step.size"))
    .SSrmFit("step.size")
  if(.SSexistsFit("clickedOK"))
    .SSrmFit("clickedOK")
  if(.SSexistsFit("model"))
    .SSrmFit("model")
  if(.SSexistsFit("moved"))
    .SSrmFit("moved")
  getPoints(justDoIt(.SSgetFit("image")),
    justDoIt(.SSgetFit("varname")))
}
```

The function `on.delete.all` is called when we click on the "Delete all" button. The background map remains at its actual status, but all reference points are deleted.

```
on.delete.all <- function(){
  if(.SSexistsFit("data.coord"))
    .SSrmFit("data.coord")
  else
    tkmessageBox(message="There are no points to delete!",
      icon="error",type="ok")
  if(.SSexistsFit("image.coord"))
    .SSrmFit("image.coord")
  if(.SSexistsFit("data.device")){
    dev.off(.SSgetFit("data.device"))
    .SSrmFit("data.device")
  }
  closeDialog()
  if(.SSexistsFit("step.size"))
    .SSrmFit("step.size")
  if(.SSexistsFit("model"))
    .SSrmFit("model")
  if(.SSexistsFit("moved"))
    .SSrmFit("moved")
  getPoints(justDoIt(.SSgetFit("image")),
    justDoIt(.SSgetFit("varname")))
}
```

Now we define the function `move`, which just moves a reference points on the background map. The corresponding reference point on the location map can not be moved, because it is selected using `identify`, so that its coordinate should be fixed, otherwise the coordinates of an observation would be changed.

```
move <- function(direction){
```

Moving is only available after the first transformation, because otherwise there are two coordinate systems.

```
  if(!.SSexistsFit("clickedOK"))
    tkmessageBox(message=paste("Not yet implemented!",
      "At the moment moving reference points is only available",
      " after clicking on OK!",
      sep=""), icon="error", type="ok")
  else{
```

The number of the reference point, which should be moved, is read in from Tcl.

```
    point.number <- as.numeric(tclvalue(.SSgetFit("move.p")))
    if(point.number == 0)
      tkmessageBox(message=paste("Please select a reference point to",
        " move by clicking on a radio button!",sep=""),
        icon="error", type="ok")
    else{
      .SSassignFit("moved",TRUE)
```

Now we simply add or subtract the step size from the x- or y-coordinate, depending on the direction of the moving.

```
image.coord <- .SSgetFit("image.coord")
data.coord <- .SSgetFit("data.coord")
img <- justDoIt(.SSgetFit("image"))
step.size <- as.numeric(tclvalue(.SSgetFit("step.size")))
old.x <- image.coord$x
old.y <- image.coord$y
if(direction == "up")
  image.coord$y[point.number]
    <- image.coord$y[point.number]+step.size
else if(direction == "down")
  image.coord$y[point.number]
    <- image.coord$y[point.number]-step.size
else if(direction == "left")
  image.coord$x[point.number]
    <- image.coord$x[point.number]-step.size
else if(direction == "right")
  image.coord$x[point.number]
    <- image.coord$x[point.number]+step.size
.SSassignFit("image.coord",image.coord)
xy <- c(image.coord$x[point.number],
  image.coord$y[point.number])
if(class(img)== "Picture"){ # Picture <= PS-File
  plot(justDoIt(.SSgetFit("varname")))
  plot(justDoIt(.SSgetFit("image")),add=T)
  plot(justDoIt(.SSgetFit("varname")),add=T)
  points(image.coord,pch=7,col="red")
  text(image.coord,labels=1:length(image.coord$x),col="yellow")
  points(data.coord,pch=7,col="blue")
  text(data.coord,labels=1:length(image.coord$x),col="yellow")
}else{# SpatialPointsPictureFrame <= PNM-File
  class(img) <- "SpatialPointsDataFrame"
  x.Range <- (img@bbox[1,2]-img@bbox[1,1])/40
  y.Range <- (img@bbox[2,2]-img@bbox[2,1])/40
  xy.Range <- c(xy-c(x.Range,y.Range),xy+c(x.Range,y.Range))
```

For maps from the class `SpatialPointsPictureFrame` we just re-plot a small part of the image, because plotting is very slow.

```
  plot.SpatialPointsPictureFrame(img,add=TRUE,pch=20,
    add.Range=xy.Range)
  plot(justDoIt(.SSgetFit("varname")),add=T)
  points(image.coord,pch=7,col="red")
  text(image.coord,labels=1:length(image.coord$x),col="yellow")
  points(data.coord,pch=7,col="blue")
```

```
          text(data.coord,labels=1:length(image.coord$x),col="yellow")
        }
      }
    }
  }
```

A `move` function has to be defined for each direction, because a function called by a button is not allowed to have parameters.

```
move.up <- function(){
  move(direction="up")
}
move.down <- function(){
  move(direction="down")
}
move.left <- function(){
  move(direction="left")
}
move.right <- function(){
  move(direction="right")
}
```

The function `onOK` is called by the "OK" button. It calls the function `fitBackground` for performing the transformation.

```
onOK <- function(){
```

The transformation type is read in form the radio button in Tcl.

```
  model.type <- as.character(tclvalue(modelVariable))
  n <- length(.SSgetFit("data.coord")$x)
  if(((model.type=="robustsim")&&(n<=4))||
    ((model.type=="sim")&&(n<=1))||((model.type=="affine")&&(n<=2))||
    ((model.type=="nonlinear")&&(n<=3))){
    tkmessageBox(message=paste("Please select the needed number of",
      " points for the chosen model type!",sep=""),icon="error",
      type="ok")
  } else {
```

This backup will be restored, when the "Reset" button is clicked.

```
    if(!.SSexistsFit("backup"))
      .SSassignFit("backup",justDoIt(.SSgetFit("image")))
```

If it is the first transformation, we have to close the two overlaying screens using the function `close.screen`.

```
    if((.SSexistsFit("data.device"))&&(!.SSexistsFit("clickedOK"))){
      dev.set(.SSgetFit("data.device"))
      close.screen(1)
```

```
  close.screen(2)
  dev.off(.SSgetFit("data.device"))
  .SSrmFit("data.device")
}
.SSassignFit("clickedOK",TRUE)
.SSassignFit("model.type",as.character(tclvalue(modelVariable)))
data.coord <- .SSgetFit("data.coord")
image.coord <- .SSgetFit("image.coord")
if(!.SSexistsFit("moved")){
```

The coordinates of the reference points are read from the Tcl text fields, so no changes made by the user get lost.

```
  for(i in 1:length(data.coord$x)){
    image.coord$x[i] <- as.numeric(tkget(image.coord.text.x[[i]],
      "0.0","end"))
    image.coord$y[i] <- as.numeric(tkget(image.coord.text.y[[i]],
      "0.0","end"))
    data.coord$x[i] <- as.numeric(tkget(data.coord.text.x[[i]],
      "0.0","end"))
    data.coord$y[i] <- as.numeric(tkget(data.coord.text.y[[i]],
    "0.0","end"))
  }
}else
  .SSrmFit("moved")
.SSassignFit("data.coord",data.coord)
.SSassignFit("image.coord",image.coord)
closeDialog()
image.x <- paste(as.character(round(image.coord$x,2)),
  collapse=",")
image.y <- paste(as.character(round(image.coord$y,2)),
  collapse=",")
data.x <- paste(as.character(round(data.coord$x,2)),collapse=",")
data.y <- paste(as.character(round(data.coord$y,2)),collapse=",")
xcoord <- .SSgetFit("xcoord")
ycoord <- .SSgetFit("ycoord")
```

If the background map was a PNM file, it is a object of class `pixmapRGB` until the first transformation is made. The function `image2spatial` converts the object to the class `SpatialPointsPictureFrame`. Furthermore the string `command` is build, which contains the call of `fitBackground`.

```
if(class(justDoIt(.SSgetFit("image")))=="pixmapRGB"){
  command.spatial2
    <- paste("image2spatial(",.SSgetFit("image"),")",sep="")
  command <- paste(.SSgetFit("image"),
    " <- fitBackground(image.coordinates=matrix(c(",image.x,",",
    image.y,"),ncol=2),data.coordinates=matrix(c(",data.x,",",
```

```
        data.y,"),ncol=2)",",image=",command.spatial2,',
        model.type="',.SSgetFit("model.type"),'"',")",sep="")
    }else{
      command <- paste(.SSgetFit("image"),
        " <- fitBackground(image.coordinates=matrix(c(",image.x,",",
        image.y,"),ncol=2),data.coordinates=matrix(c(",data.x,",",
        data.y,"),ncol=2),","image=","justDoIt(\"",.SSgetFit("image"),
        "\")",',model.type="',.SSgetFit("model.type"),'"',")",sep="")
    }
```

The `command` is executed with `justDoIt` and then written into the DAS+R script window with the function `logger`.

```
    justDoIt(command)
    logger(command)
    if(class(justDoIt(.SSgetFit("image")))[1] ==
      "SpatialPointsPictureFrame")
      {
      command <- paste("class(",.SSgetFit("image"),
    ") <- \"SpatialPointsDataFrame\"",sep="")
      invisible(justDoIt(command))
      command <- paste(.SSgetFit("image"),"@bbox <- ",.SSgetFit(
    "varname"),"@bbox",sep="")
      justDoIt(command)
      command <- paste("class(",.SSgetFit("image"),
    ") <- \"SpatialPointsPictureFrame\"",sep="")
      invisible(justDoIt(command))
    }else if(class(justDoIt(.SSgetFit("image")))[1] == "Picture"){
      xsca <- justDoIt(.SSgetFit("image"))@summary@xscale
      ysca <- justDoIt(.SSgetFit("image"))@summary@yscale
      bbox <- justDoIt(.SSgetFit("varname"))@bbox
      xsca <- bbox[1,]
      ysca <- bbox[2,]
      command <- paste(.SSgetFit("image"),"@summary@xscale <- c(",
    xsca[1],",",xsca[2],")",sep="")
      justDoIt(command)
      command <- paste(.SSgetFit("image"),"@summary@yscale <- c(",
    ysca[1],",",ysca[2],")",sep="")
      justDoIt(command)
      }
    getPoints(justDoIt(.SSgetFit("image")),
      justDoIt(.SSgetFit("varname")))
  }
}
if((.SSexistsFit("new.coords"))&&(.SSexistsFit("image.coord"))&&
  (.SSexistsFit("data.coord"))){
  .SSrmFit("data.coord")
```

```
  .SSrmFit("image.coord")
  .SSrmFit("new.coords")
}
```

Now the content of the menu "Selecting Reference Points" is build with the Tcl/Tk functions `tkframe`, `tkbutton`, `tklabel`, `tktext`, `tkradiobutton` and finally `tkgrid` to bring the objects on the screen.

```
coord.frame <- tkframe(top)
point.button <- tkbutton(coord.frame,text=paste(
  "Click for: Add/Identify a Reference Point \n",
  "(First click=image, second click=data points)"), fg="green3",
  command=click.point)
tkgrid(point.button,columnspan=8,column=1,row=1)
max.row <- 1
if(.SSexistsFit("image.coord")){
  image.coord <- .SSgetFit("image.coord")
  data.coord <- .SSgetFit("data.coord")
  image.coord.text.x <- list()
  data.coord.text.x <- list()
  image.coord.text.y <- list()
  data.coord.text.y <- list()
  delete.button <- list()
  data.label <- tklabel(coord.frame,text="Data Coordinates",
    fg="blue")
  image.label <- tklabel(coord.frame,text="Image Coordinates",
    fg="blue")
  x.label <- tklabel(coord.frame,text="X")
  y.label <- tklabel(coord.frame,text="Y")
  x1.label <- tklabel(coord.frame,text="X")
  y1.label <- tklabel(coord.frame,text="Y")
  delete.label <- tklabel(coord.frame,text="Delete",fg="blue")
  move.label <- tklabel(coord.frame,text="Move",fg="blue")
  tkgrid(image.label,columnspan=2,column=2,row=2)
  tkgrid(data.label,columnspan=2,column=4,row=2)
  tkgrid(tklabel(coord.frame,text="#"),columnspan=1,column=1,row=3)
  tkgrid(x.label,columnspan=1,column=2,row=3)
  tkgrid(y.label,columnspan=1,column=3,row=3)
  tkgrid(x1.label,columnspan=1,column=4,row=3)
  tkgrid(y1.label,columnspan=1,column=5,row=3)
  tkgrid(delete.label,columnspan=1,column=6,row=3)
  tkgrid(move.label,columnspan=1,column=8,row=3)
  delete.point <- list()
  move.radio <- list()
  move.p <- tclVar("0")
  .SSassignFit("move.p",move.p)
  .SSassignFit("top",top)
```

```
for(i in 1:length(image.coord$x)){
```

For all reference points a function for deleting is build, which is later assigned to the corresponding button.

```
function.command <- paste("function(){","\n",
  "data.coord <- .SSgetFit(\"data.coord\")","\n ",
  "image.coord <- .SSgetFit(\"image.coord\")","\n ",
  "dev.set(.SSgetFit(\"data.device\"))","\n",
  ".SSassignFit(\"deleted\",TRUE)","\n",
  "if(length(data.coord$x)==1)","\n",
  "tkmessageBox(message=\"Please use the Delete All-button!\",
    icon=\"error\", type=\"ok\")","\n",
  "else{","\n",
   "xy <- c(image.coord$x[",i,"],image.coord$y[",i,"])","\n",
   "xy2 <- c(data.coord$x[",i,"],data.coord$y[",i,"])","\n",
  "data.coord$x <- data.coord$x[-",i,"]","\n ","image.coord$x
    <- image.coord$x[-",i,"]","\n ",
  "data.coord$y <- data.coord$y[-",i,"]","\n ","getPoints
    <-.SSgetFit(\"getPoints\")","\n ",
  "image.coord$y <- image.coord$y[-",i,"]","\n ",
  ".SSassignFit(\"data.coord\",data.coord)","\n ",
  "top <- .SSgetFit(\"top\")","\n ",
  ".SSassignFit(\"image.coord\",image.coord)","\n ",
  "if(!.SSexistsFit(\"clickedOK\"))","\n ",
     "tkmessageBox(message=paste(\"At the moment the reference\",
       \"points are deleted from the graph only available after\",
       \" clicking on OK!\",sep=\"\"), icon=\"error\",
       type=\"ok\")","\n",
  "else{","\n",
    "if(class(justDoIt(.SSgetFit(\"image\")))== \"Picture\"){",
     "\n",
      "plot(justDoIt(.SSgetFit(\"varname\")))","\n",
      "plot(justDoIt(.SSgetFit(\"image\")),add=T)","\n",
      "plot(justDoIt(.SSgetFit(\"varname\")),add=T)","\n",
      "points(image.coord,pch=7,col=\"red\")","\n",
      "text(image.coord,labels=1:length(image.coord$x),
        col=\"yellow\")","\n",
      "points(data.coord,pch=7,col=\"blue\")","\n",
      "text(data.coord,labels=1:length(image.coord$x),
        col=\"yellow\")","\n",
    "}else{","\n",
      "x.Range <- (justDoIt(.SSgetFit(\"image\"))@bbox[1,2]
        -justDoIt(.SSgetFit(\"image\"))@bbox[1,1])/50","\n",
      "y.Range <- (justDoIt(.SSgetFit(\"image\"))@bbox[2,2]
        -justDoIt(.SSgetFit(\"image\"))@bbox[2,1])/50","\n",
      "xy.Range",
```

```
            " <- c(xy-c(x.Range,y.Range),xy+c(x.Range,y.Range))","\n",
            "xy2.Range",
            " <- c(xy2-c(x.Range,y.Range),xy2+c(x.Range,y.Range))",
            "\n",
            "plot(justDoIt(.SSgetFit(\"image\")),add=TRUE,pch=20,
              add.Range=xy.Range)","\n",
            "plot(justDoIt(.SSgetFit(\"image\")),add=TRUE,pch=20,
              add.Range=xy2.Range)","\n",
            "plot(justDoIt(.SSgetFit(\"varname\")),add=T)","\n",
            "points(image.coord,pch=7,col=\"red\")","\n",
            "text(image.coord,labels=1:length(image.coord$x),
              col=\"yellow\")","\n",
            "points(data.coord,pch=7,col=\"blue\")","\n",
            "text(data.coord,labels=1:length(image.coord$x),
              col=\"yellow\")","\n",
          "}","\n",
        "}","\n ",
        "closeDialog()","\n ",
        "justDoIt(getPoints(.SSgetFit(\"image\"),",
          ".SSgetFit(\"varname\")))" ,"\n",
        "}","\n","}",sep="")
      delete.point[[i]] <- justDoIt(function.command)
      image.coord.text.x[[i]] <- tktext(coord.frame,height=1,width=7)
      image.coord.text.y[[i]] <- tktext(coord.frame,height=1,width=7)
      data.coord.text.x[[i]] <- tktext(coord.frame,height=1,width=7)
      data.coord.text.y[[i]] <- tktext(coord.frame,height=1,width=7)
      delete.button[[i]] <- tkradiobutton(coord.frame,
        command=delete.point[[i]],variable=tclVar("0"))
      move.radio[[i]] <- tkradiobutton(coord.frame,variable=move.p,
        value=i)
      tkgrid(tklabel(coord.frame,text=i),column=1,columnspan=1,
        row=i+3)
      tkgrid(image.coord.text.x[[i]],column=2,columnspan=1,row=i+3)
      tkgrid(image.coord.text.y[[i]],column=3,columnspan=1,row=i+3)
      tkgrid(data.coord.text.x[[i]],column=4,columnspan=1,row=i+3)
      tkgrid(data.coord.text.y[[i]],column=5,columnspan=1,row=i+3)
      tkgrid(delete.button[[i]],column=6,columnspan=2,row=i+3)
      tkgrid(move.radio[[i]],column=8,columnspan=2,row=i+3)
```

With `tkinsert` the values of the coordinates are written in the corresponding text fields.

```
      tkinsert(image.coord.text.x[[i]], "end",
        round(image.coord$x[[i]],2))
      tkinsert(image.coord.text.y[[i]], "end",
        round(image.coord$y[[i]],2))
      tkinsert(data.coord.text.x[[i]], "end",
        round(data.coord$x[[i]],2))
```

```
    tkinsert(data.coord.text.y[[i]], "end",
      round(data.coord$y[[i]],2))
    max.row <- i+3
    }
  delete.all.button <- tkbutton(coord.frame,text="Delete All",
    command=on.delete.all,fg="red")
  up.button <- tkbutton(coord.frame,text="^",command=move.up,
    fg="darkgreen")
  down.button <- tkbutton(coord.frame,text="v",command=move.down,
    fg="darkgreen")
  right.button <- tkbutton(coord.frame,text=">",command=move.right,
    fg="darkgreen")
  left.button <- tkbutton(coord.frame,text="<",command=move.left,
    fg="darkgreen")
  if(.SSexistsFit("step.size")){
    step.size <- .SSgetFit("step.size")
  }
  img <- justDoIt(.SSgetFit("image"))
```

By dividing the whole range of the map by 50, a reasonable value for the step size for moving is computed.

```
  if(class(img)[1]=="SpatialPointsPictureFrame"){
    x.Range <- (img@bbox[1,2]-img@bbox[1,1])/50
    y.Range <- (img@bbox[2,2]-img@bbox[2,1])/50
    step.size <- tclVar(round(mean(x.Range,y.Range)))
  }else if(class(img)[1]=="Picture"){
    x.Range <- (img@summary@xscale[2]-img@summary@xscale[1])/50
    y.Range <- (img@summary@yscale[2]-img@summary@yscale[1])/50
    step.size <- tclVar(round(mean(x.Range,y.Range)))
  }else
    step.size <- tclVar("5000")
  step.input <- tkentry(coord.frame, width="5",
    textvariable=step.size)
  .SSassignFit("step.size",step.size)
  tkgrid(delete.all.button,row=max.row+1,column=6,columnspan=2)
  tkgrid(up.button,column=8,row=max.row+1,columnspan=2,sticky="n")
  tkgrid(left.button,column=8,row=max.row+2,sticky="w")
  tkgrid(right.button,column=9,row=max.row+2,sticky="e")
  tkgrid(down.button,column=8,row=max.row+3,columnspan=2,sticky="n")
  tkgrid(step.input,column=8,columnspan=2,row=max.row+4,columnspan=2)
  max.row <- max.row+4
  }
trans.label <- tklabel(coord.frame,text="Transformation Type:",
  fg="blue")
if(.SSexistsFit("model.type"))
  modelVariable <- tclVar(.SSgetFit("model.type"))
```

```
    else
      modelVariable <- tclVar("sim")
```

trans.radio1, trans.radio2, trans.radio3 and rotation.radio are the radio buttons for selecting a transformation type.

```
    trans.radio1 <- tkradiobutton(coord.frame,variable=modelVariable,
      value="sim",text="Similarity Transformation ( min : 2 )")
    trans.radio2 <- tkradiobutton(coord.frame,variable=modelVariable,
      value="affine",text="Affine Transformation ( min : 3 )")
    trans.radio3 <- tkradiobutton(coord.frame,variable=modelVariable,
      value="nonlinear",text="Nonlinear Transformation ( min : 6 )")
    rotation.radio <- tkradiobutton(coord.frame,variable=modelVariable,
      value="robustsim",
      text="Robust Similarity Transformation ( min : 5 )")
    tkgrid(tklabel(coord.frame,text=" "),row=max.row+1,column=1,
      columnspan=2)
    tkgrid(trans.label,row=max.row+2,column=1,columnspan=3)
    tkgrid(trans.radio1,row=max.row+3,column=2,columnspan=3,sticky="w")
    tkgrid(trans.radio2,row=max.row+4,column=2,columnspan=3,sticky="w")
    tkgrid(trans.radio3,row=max.row+6,column=2,columnspan=3,sticky="w")
    tkgrid(rotation.radio,row=max.row+5,column=2,columnspan=3,sticky="w")
    save.button <- tkbutton(coord.frame,text="Save Background Map",
      command=on.save,fg="darkgreen")
    reset.button <- tkbutton(coord.frame,text="Reset",command=on.restart,
      fg="red")
    tkgrid(tklabel(coord.frame,text=" "),row=max.row+7,column=1,
      columnspan=2)
    tkgrid(reset.button,row=max.row+8,column=2,columnspan=1,sticky="w")
    tkgrid(save.button,row=max.row+8,column=3,columnspan=3,sticky="w")
    tkgrid(coord.frame,row=1,column=1,sticky="w")
    OKCancelHelp(helpSubject="fitBackground")
    tkgrid(buttonsFrame,row=15,columnspan=3)
    }
  .SSassignFit("getPoints",getPoints)
```

If no image is saved on the object .background, a dialog for opening an image file is generated.

```
  if(!.SSexistsFit("image")){
    image.names <- character()
    dir <- ls(envir=as.environment(1))
    for(i in 1:length(dir)){
      if((class(justDoIt(dir[i]))=="pixmapRGB")||
        (class(justDoIt(dir[i]))=="pixmapGrey")||
        (class(justDoIt(dir[i]))=="pixmapIndexed")||
        (class(justDoIt(dir[i]))[1]=="Picture"))
        image.names[length(image.names)+1] <- dir[i]
```

```
  }
initializeDialog(title="Loading Image into R")
on.ok2 <- function(){
  .SSassignFit("new",TRUE)
  .SSassignFit("new.coords",TRUE)
  onOK()
}
on.okpnm <- function(){
  .SSassignFit("new",TRUE)
  .SSassignFit("new.coords",TRUE)
  .SSassignFit("pnm",TRUE)
  onOK()
}
onOK <- function(){
  varname <- as.character(tclvalue(varname))
  selected <- tclvalue(tkcurselection(list.screen))
  closeDialog()
  if((selected=="")||(.SSexistsFit("new"))){
    if(.SSexistsFit("new"))
      .SSrmFit("new")

  if(.SSexistsFit("pnm")){
      filename <- tclvalue(tkgetOpenFile(filetypes=
        '{"pnm files" {".pnm"}} {"ps files" {".ps"}}
        {"All Files" {"*"}}'))
      .SSrmFit("pnm")
    }else
      filename <- tclvalue(tkgetOpenFile(filetypes=
        '{"ps files" {".ps"}} {"pnm files" {".pnm"}}
        {"All Files" {"*"}}'))
    filena <- strsplit(filename,"")
    if(filename!=""){
      if(filena[[1]][length(filena[[1]])]=="m"){
        command <- paste("convertBackground(\"",filename,"\",
          type=\"pnm\")",sep="")
        doItAndPrint(command)
      }else{
        command <- paste("convertBackground(\"",filename,"\",
          type=\"ps\")",sep="")
        doItAndPrint(command)
      }
      .SSassignFit("image",varname)
    }
  }else{
    selected <- as.numeric(selected)+1
    .SSassignFit("image",image.names[selected])
```

```
      .SSassignFit("next",TRUE)
    }
  }
```

Here the content of the dialog "Loading image into R" is defined.

```
  input.frame <- tkframe(top)
  varname <- tclVar(".background")
  tkgrid(tklabel(input.frame,
    text="Select an existing image or load a new one"))
  tkgrid(tklabel(input.frame,text="  "))
  new.pnm <- tkbutton(input.frame,text="Load PNM-image",
    command=on.okpnm)
  new.ps <- tkbutton(input.frame,text="Load PS-image",command=on.ok2)
  tkgrid(new.pnm,new.ps)
  tkgrid(tklabel(input.frame,text="   "))
  list.screen <- tklistbox(input.frame,selectmode="single",
    background="white",height=max(6,length(image.names)),
    exportselection=FALSE )
  scrollbar <- tkscrollbar(input.frame, repeatinterval=5,
    command=function(...)
  tkyview(list.screen, ...))
  tkconfigure(list.screen,
    yscrollcommand=function(...) tkset(scrollbar, ...))
  tkgrid(tklabel(input.frame, text="Existing images in workspace: "),
    list.screen, scrollbar, sticky="nw")
  tkgrid.configure(scrollbar, sticky="wns")
  tkgrid.configure(list.screen, sticky="ew")
  if(length(image.names)>0){
    for (i in 1:length(image.names)){
      tkinsert(list.screen,"end",image.names[i])
      if(.SSexistsFit("image")){
        if(image.names[i]==.SSgetFit("image"))
          selected.image <- i-1
      }
    }
  }
  if(exists("selected.image")){
    tkselection.set(list.screen,selected.image)
  }
  OKCancelHelp(helpSubject="fitBackground")
  tkgrid(input.frame)
  tkgrid(buttonsFrame, sticky="w")
  dialogSuffix(rows=3, columns=1)
}
if((checkActiveDataSet())&&.SSexistsFit("image")){
```

This is the `onOK` function, which will be started, when the image is ready and x- and

y-coordinates are selected.

```
onOK <- function(){
  if((!.SSexistsFit("xcoord"))||(!.SSexistsFit("ycoord"))){
    xcoord <- getSelection(xBox)
    .SSassignFit("axes",as.logical(as.numeric(tclvalue(axes.yn))))
    if (length(xcoord) == 0){
      errorCondition(recall=FitBackgroundGUI, message=
    "You must select X-coordinates")
      return()
    }else{
      v <- xBox$varlist
      for(i in 1:length(v)){
        if(xcoord==v[i])
          ini <- i-1
      }
      .SSassignFit("xini",ini)
    }
    ycoord <- getSelection(yBox)
    if (length(ycoord) == 0){
      errorCondition(recall=FitBackgroundGUI, message=
    "You must select Y-coordinates")
      return()
    }else{
      v <- yBox$varlist
      for(i in 1:length(v)){
        if(ycoord==v[i])
          ini <- i-1
      }
      .SSassignFit("yini",ini)
    }
    closeDialog()
    .SSassignFit("xcoord",xcoord)
    .SSassignFit("ycoord",ycoord)
  }else{
    xcoord <- .SSgetFit("xcoord")
    ycoord <- .SSgetFit("ycoord")
  }
  varname <- tclVar(".spatialdata1")
  varname <- as.character(tclvalue(varname))
  .SSassignFit("varname",varname)
  command.spatial <- paste(varname,"<-SpatialPoints(cbind(",
    ActiveDataSet(),"$",xcoord,",",ActiveDataSet(),"$",ycoord,"))",
    sep="")
  justDoIt(command.spatial)
  img <- justDoIt(.SSgetFit("image"))
```

```
  spatial.data <- justDoIt(varname)
  getPoints(img,spatial.data)
}
```

If there is an attribute specified for the coordinates, we can go along without manually selecting them.

```
if(!is.null(attr(justDoIt(ActiveDataSet()),"coordinates"))){
  coordi <- attr(justDoIt(ActiveDataSet()),"coordinates")
  .SSassignFit("xcoord",coordi[1])
  .SSassignFit("ycoord",coordi[2])
  onOK()
}else{
```

Otherwise we build a dialog for choosing the variables, which contain the coordinates.

```
  initializeDialog(
    title="Selecting Variables for Fitting Image to Data")
  input.frame <- tkframe(top)
  if(.SSexistsFit("data.device")){
    dev.off(.SSgetFit("data.device"))
    .SSrmFit("data.device")
    }
  if(.SSexistsFit("varname"))
    varname <- tclVar(.SSgetFit("varname"))
  else
    varname <- tclVar(".spatialdata1")
  if(.SSexistsFit("axes"))
    axes.yn <- tclVar( as.character(as.numeric(.SSgetFit("axes"))))
  else
    axes.yn <- tclVar("1")
```

Here two list boxes containing all variables are build.

```
  xBox <- variableListBox(top, Numeric(),
    title="X-coordinates (pick one)",listHeight = 20)
  yBox <- variableListBox(top, Numeric(),
    title="Y-coordinates (pick one)",listHeight = 20)
  OKCancelHelp(helpSubject="fitBackground")
  tkgrid(input.frame,row=1,column=1,columnspan=3)
  space.frame <- tkframe(top)
  spacer <- tklabel(space.frame,text="   ")
  space2.frame <- tkframe(top)
  spacer2 <- tklabel(space2.frame,text="   ")
  tkgrid(spacer)
  tkgrid(spacer2)
  tkgrid(space.frame,row=2)
  tkgrid(getFrame(xBox),row=3,column=1,sticky="w")
```

```
      tkgrid(getFrame(yBox),row=3,column=2,sticky="w")
      tkgrid(space2.frame,row=4)
      tkgrid(buttonsFrame,row=5,column=1,sticky="w",columnspan=3)
    }
  }
}
```

The `PlotBackgroundGUI` opens the menu for plotting a background map. The only changeable parameter controls, if the map should be added to an existing plot or not.

```
PlotBackgroundGUI <- function(){
  onOK <- function(){
    if(exists(".background")){
      addP <- as.logical(as.numeric(tclvalue(addVariable)))
      if(addP){
        plot(.background,add=TRUE)
        logger('plot(.background,add=TRUE)')
      }else{
        plot(.background,xaxt="n",yaxt="n",bty="n")
        logger('plot(.background,xaxt="n",yaxt="n",bty="n")')
      }
    }else
        tkmessageBox(
          message="No background map loaded! Use 'Load' or 'Convert'.",
          icon="error", type="ok")
    tkdestroy(top)
  }
  addVariable <- tclVar(0)
  initializeDialog(title=paste("Plotting Background Map: ",
    ActiveDataSet()))
  addCheckBox <- tkcheckbutton(top, variable=addVariable)
  tkgrid(addCheckBox,column=1,row=1,sticky="e")
  tkgrid(tklabel(top,text="Add to Existing Plot"),column=2,columnspan=1,
    row=1,sticky="w")
  OKCancelHelp(helpSubject="fitBackground")
  tkgrid(buttonsFrame, sticky="w",columnspan=4,column=1)
}
```

Chapter 5

Illustration

In this chapter we want to go the way from an image file, either in PNM or PS format, to a fitted background map. We want to achieve this by using the graphical user interface DAS+R. Its concept is to provide easy access to powerful R functions, with no need of knowing details on the language or having programming skills.

5.1 How-To Manual

First of all we start R by clicking on the logo in a windows environment or by typing `R` in a Linux terminal.

The DAS+R GUI is called by typing `require("DASplusR7")`. As we can see in Figure 5.1 the functions for fitting a background map can be found in the menu "Maps" and "Background".

The first entry "Load" is to load previously, with this tool, saved maps. The map will be saved in the object `.background`. We can now plot the map with the menu entry "Plot". Furthermore we can decide if we want to add the map to an existing plot or if we want to plot it in a new one.

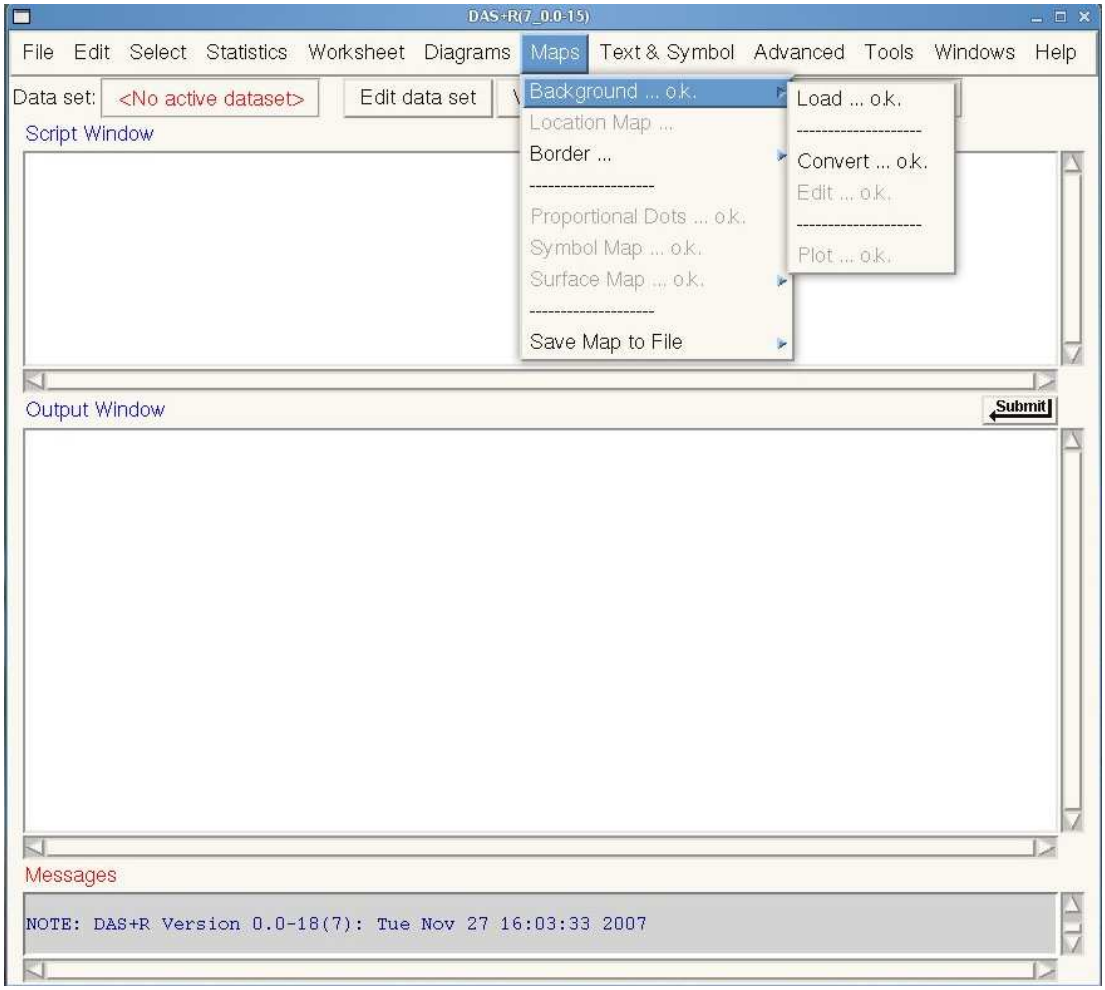

Figure 5.1: DAS+R

"Convert" is for loading an image file into R and starting the process of fitting, for using this entry an active data set is needed (we have used the Dataset `KOLA95_C2MM` from the package `DASplusR7` here). Figure 5.2 shows that we can choose between loading a PNM file or a PS file. The third option is using an image object, which already exists in the workspace (by manually loading it into R). There are several restriction on PS files, for example there should not be text in it (for details see Murrell, 2006).

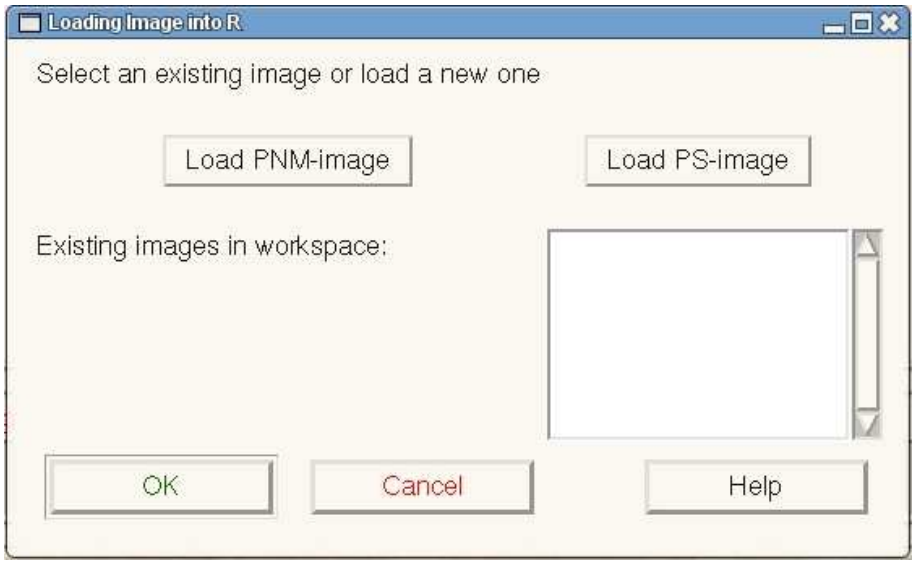

Figure 5.2: Menu for loading a map image

After the loading process the image is plotted overlapped by the location map of the active data set (see Figure 5.3 for a PNM file and Figure 5.4 for a PS file). Moreover a menu (Figure 5.5) is started, it will accompany us through out the fitting process. For adding a reference point, at first we click on the button "Click for: Add/Identify a Reference Point", then we go to the graphic device with our map and the locations and click on the reference point on the image and then on the same reference point in the data. This should be repeated until we have enough reference points selected.

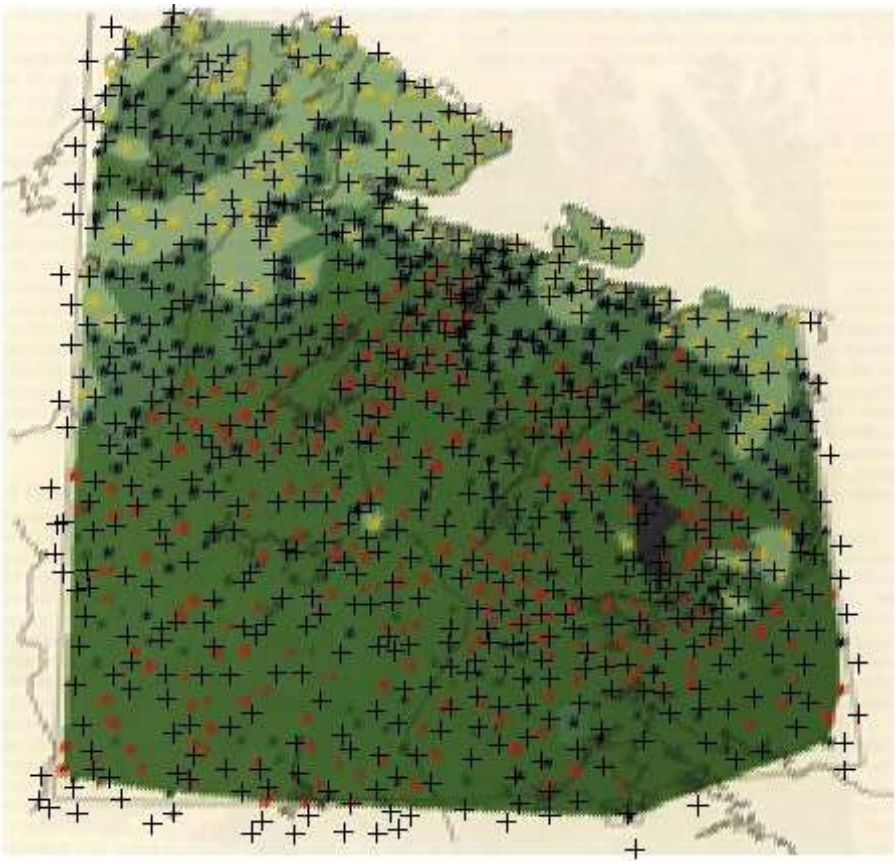

Figure 5.3: Map from a PNM file with the location map

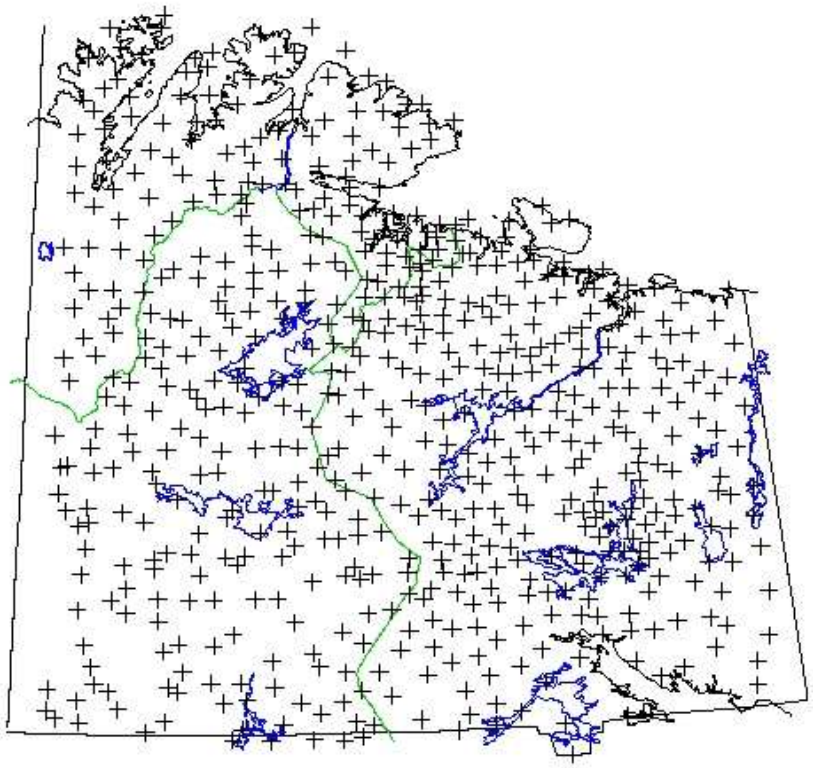

Figure 5.4: Map from a PS file with the location map

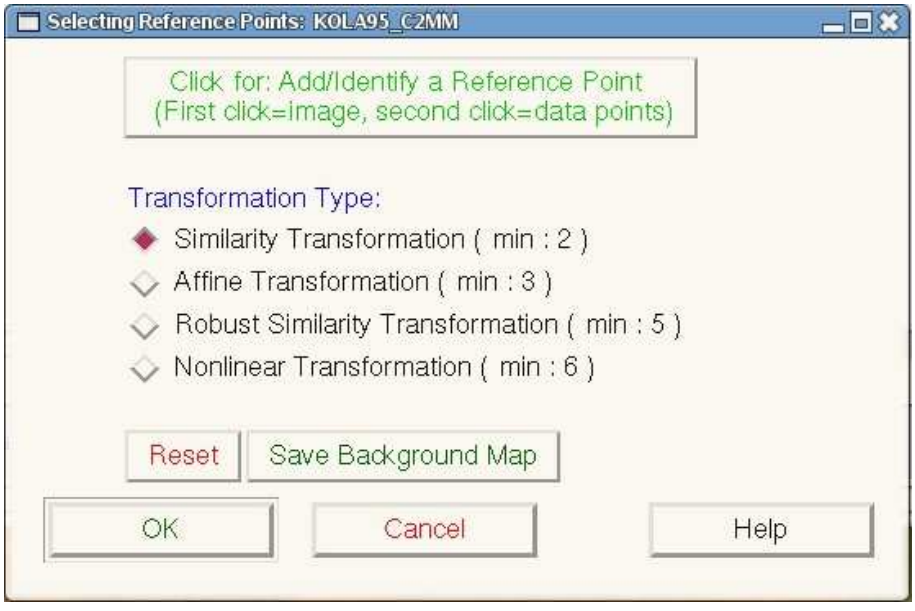

Figure 5.5: "Selecting Reference Points" menu

The reference points are shown in the graphic device (see Figure 5.6) in red for the image and blue for the data points. Furthermore the coordinates of the reference points

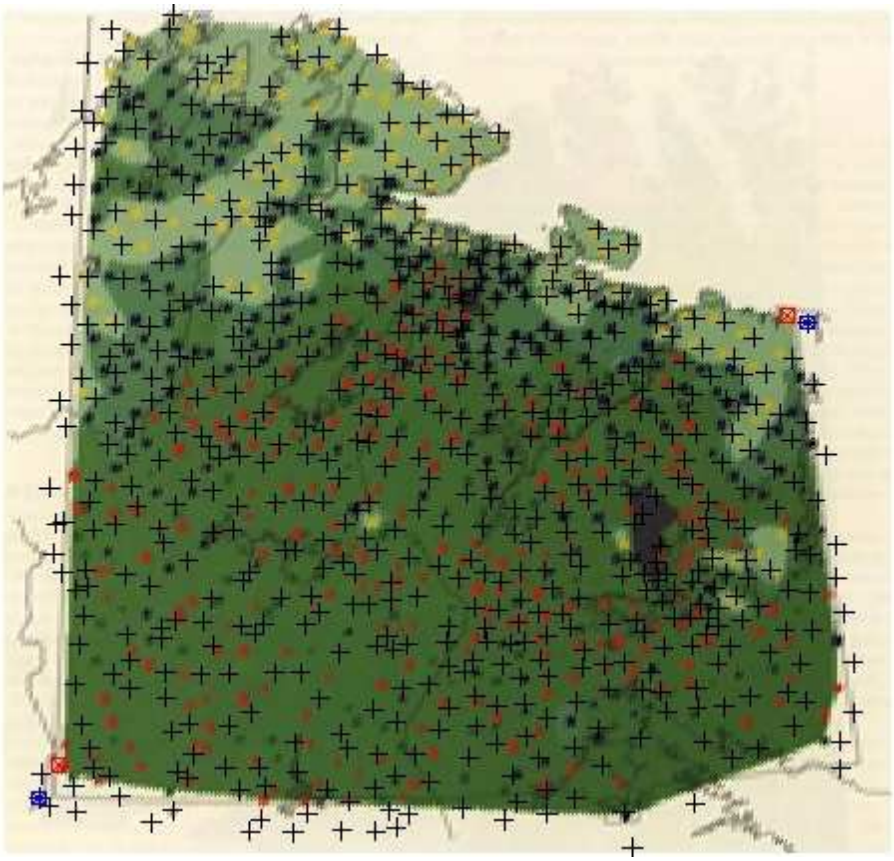

Figure 5.6: Map from a PNM file with reference points

can be seen in the “Selecting Reference Points” menu (see Figure 5.7). Now we choose the transformation type and click on “OK” for starting the transformation.

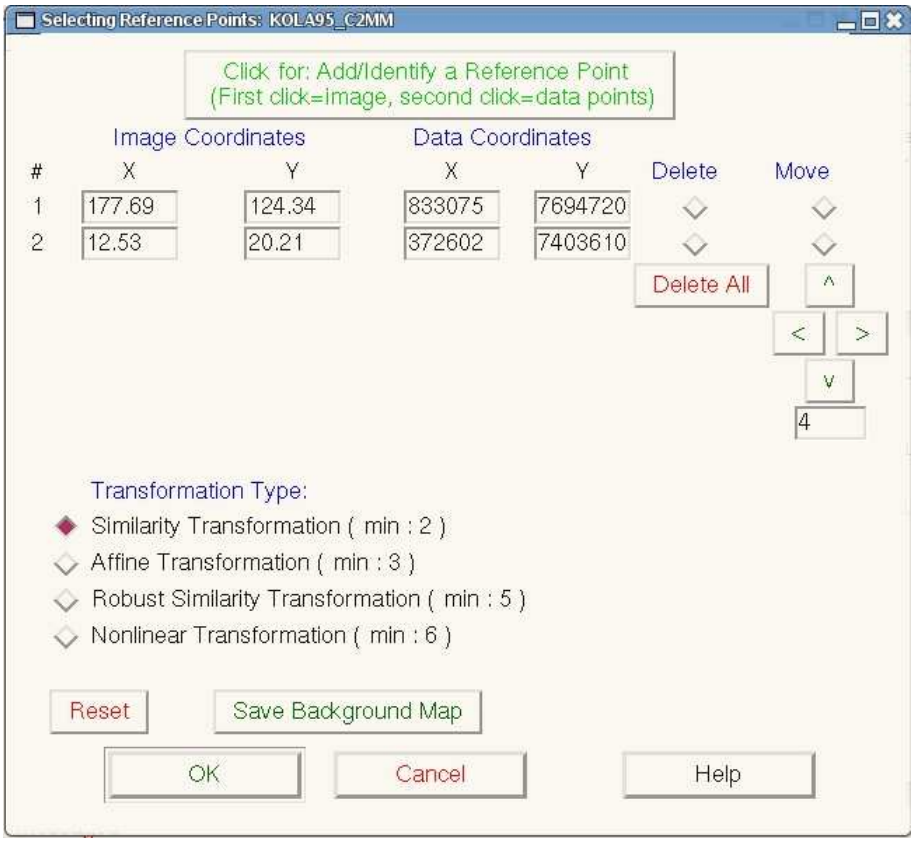

Figure 5.7: "Selecting Reference Points" menu

Our map can look like Figure 5.8 or Figure 5.9 after the first step of fitting. If we are satisfied now, we can click on "Save Background Map" for saving the background map and closing the menu. We can now proceed with other plots and every time we want to plot or add our background map we simply click "Plot" in the menu shown in Figure 5.1. If we are not satisfied yet, we can add more reference points, delete wrong reference points or correct reference points by moving them. For moving we need to click on the radio button in the column "Move" and in the row of the desired reference point. Now we can move the point on the image (not the selected point of the data locations) by clicking on one of the arrows. Below the arrows is a text field for the step size for moving. It is filled automatically with a reasonable value, but can be changed.

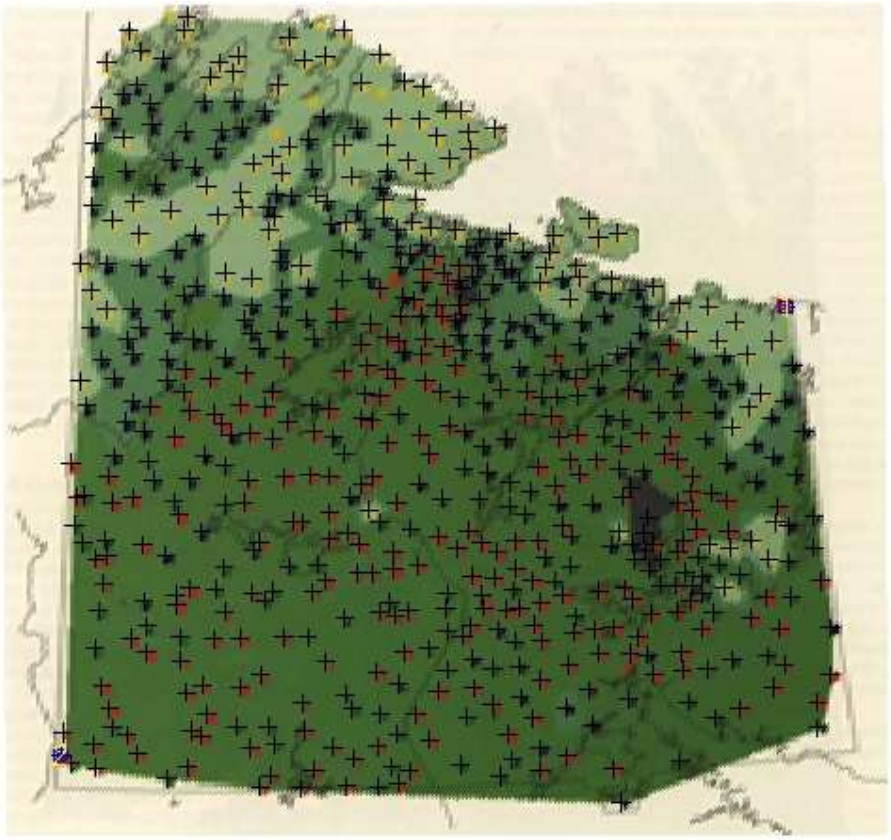

Figure 5.8: Map from a PNM file after a transformation

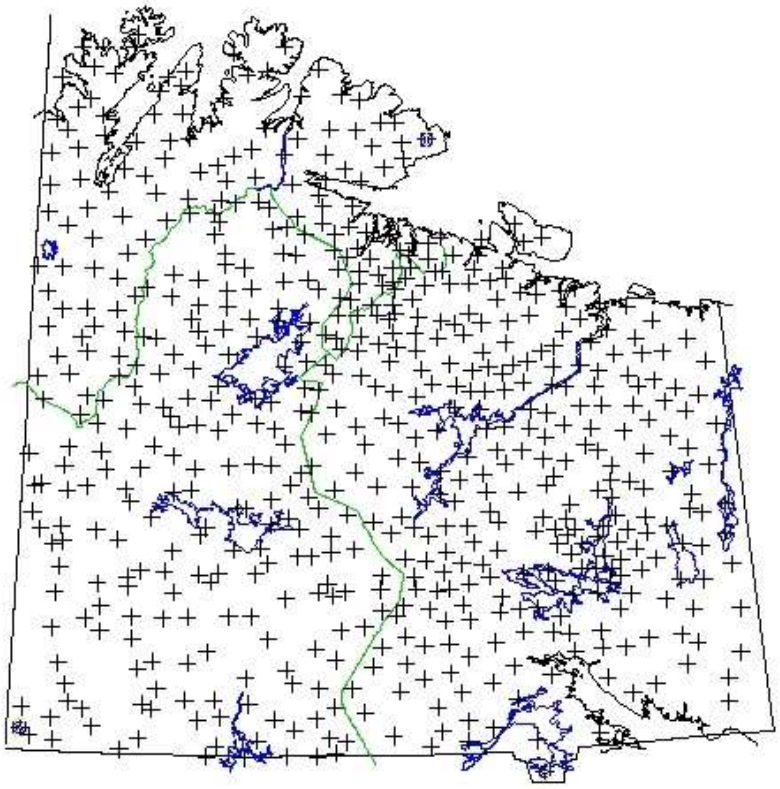

Figure 5.9: Map from a PS file after a transformation

In Figure 5.10 we can see how the script and the output window look like after one transformation step. `data("KOLA95_C2", package="DASplusR7")` is for loading the data. `convertBackground("...",type="pnm")` is for converting the image into a object. `.background <- fitBackground(...)` overwrites the old `.background` with the new background map, which is the output of the function `fitBackground`. `save(.background,file="../bg.Rmap")` saves the background map to the local file `bg.Rmap`.

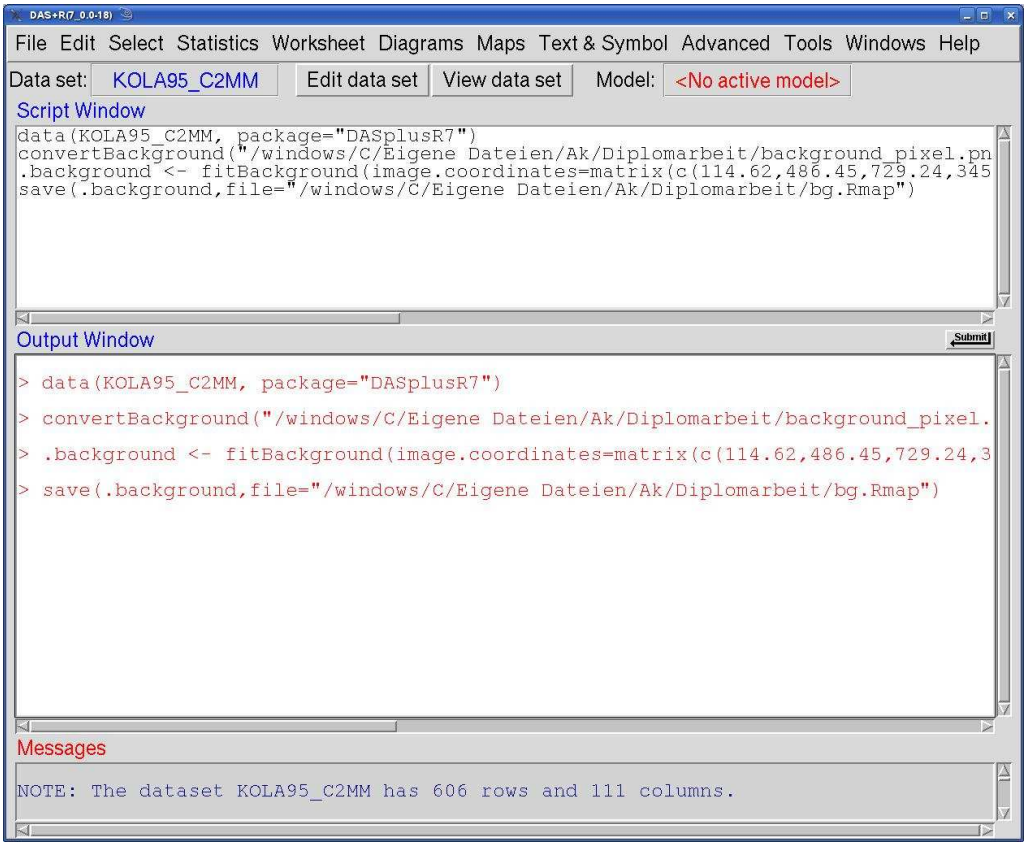

Figure 5.10: DAS+R with script

5.2 Application of the Nonlinear Transformation

We now would like to see what effect the implemented nonlinear transformation has. The idea of this example is using a satellite picture as background for our data. The locations of the data are the coast line of Florida, obtained from a map in Postscript format. In Figure 5.11 we see the satellite picture (downloaded from NASA, 2008) overlayed with the location map. We may now assume, that a similarity or affine transformation is not sufficient for this purpose. This assumption is strengthened by Figure 5.12 and Figure 5.13, because no satisfying level of fitting is achieved.

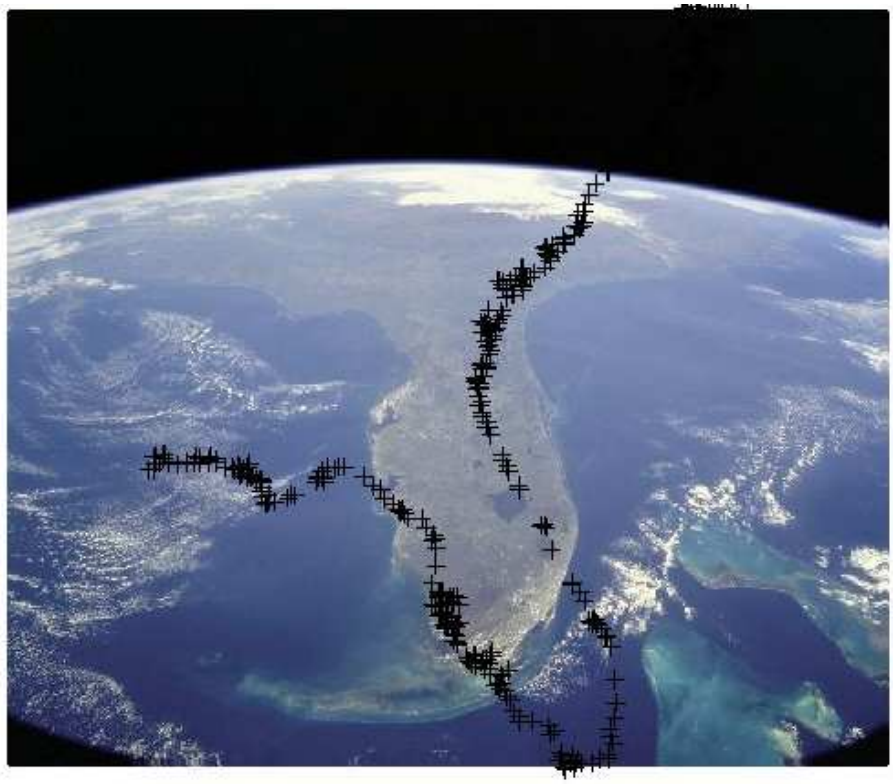

Figure 5.11: Satellite picture of Florida before transformation, overlayed with the points along the coast line

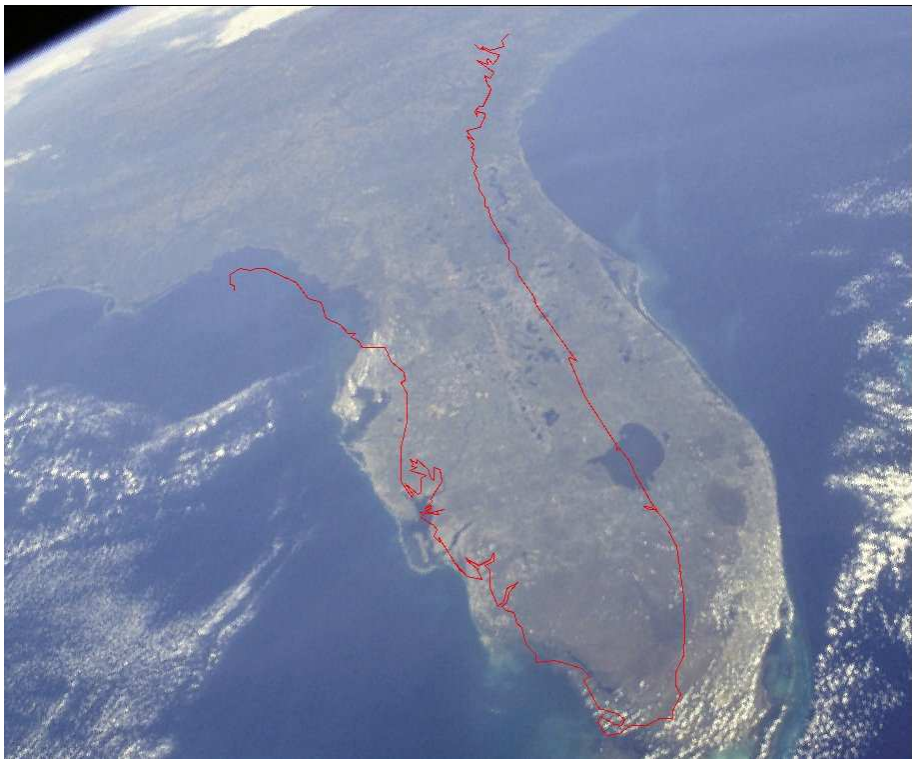

Figure 5.12: Satellite picture after a similarity transformation

Figure 5.13: Satellite picture after an affine transformation

In Figure 5.14 and 5.15 the satellite picture after several steps of adding and deleting reference points and applying a nonlinear transformation is plotted.

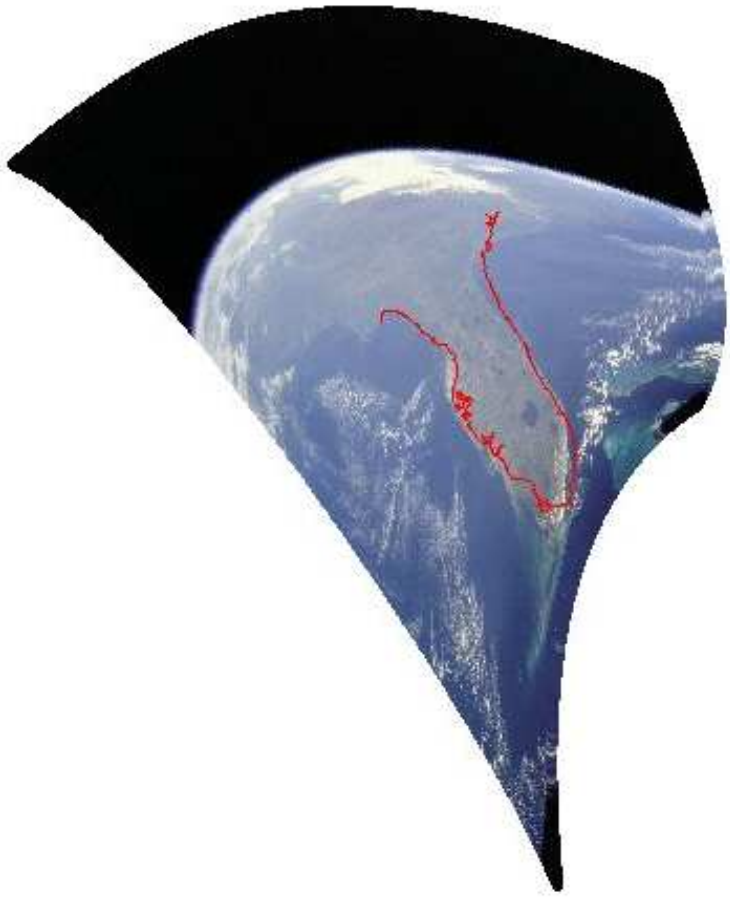

Figure 5.14: Satellite picture after several nonlinear transformations

Figure 5.15: Satellite picture within the range of the data locations, after several non-linear transformations

Bibliography

Bivand, R., Leisch, F., and Mächler, M. (2007). The pixmap package. `http://cran.r-project.org/doc/packages/pixmap.pdf` [Online; accessed 21-February-2008].

Draper, N. R. and Smith, H. (1998). *Applied Regression Analysis*. New York: Wiley, 3rd edition.

Fox, J. (2005). The R commander: A basic-statistics graphical user interface to R. *Journal of Statistical Software*, **14**(9).

Hobbs, J., Jones, K., and Welch, B. B. (2003). *Practical Programming in Tcl and Tk*. New York: Prentice Hall PTR, 4th edition.

Kraus, K. (2004). *Photogrammetrie 1: Geometrische Informationen aus Photographien und Laserscanneraufnahmen*. Berlin: de Gruyter, 7th edition.

Murrell, P. (2006). Importing graphics for statistical plots. `http://www.stat.auckland.ac.nz/~paul/` [Online; accessed 21-February-2008].

Murrell, P. and Richard, W. (2007). The grimport package. `http://www.stat.auckland.ac.nz/~paul/` [Online; accessed 21-February-2008].

NASA (2008). Great images in nasa. `http://grin.hq.nasa.gov/ABSTRACTS/GPN-2000-001182.html` [Online; accessed 21-February-2008].

Praetorius, D. (2005). Lecture notes: Numerische Mathematik.

Ripley, B. and Venables, W. (2002). *Modern Applied Statistics with S*. New York: Springer, 4th edition.

Rousseeuw, P. J. and Leroy, A. M. (1986). *Robust Regression and Outlier Detection*. New York: Wiley, 1st edition.

Rousseeuw, P. J. and Van Driessen, K. (2006). Computing lts regression for large data sets. *Data Min. Knowl. Discov.*, **12**(1), 29–45.

Überhuber, C. W. (1997). *Numerical Computation 2: Methods, Software, and Analysis*. Berlin: Springer, 1st edition.

Wikipedia (2007a). Affine transformation — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/wiki/Affine_transformation` [Online; accessed 21-February-2008].

Wikipedia (2007b). Gaussmarkov theorem — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/wiki/Gauss%E2%80%93Markov_theorem` [Online; accessed 21-February-2008].

Wikipedia (2007c). Helmert-Transformation — Wikipedia, die freie Enzyklopädie. `http://de.wikipedia.org/wiki/Helmert-Transformation` [Online; accessed 21-February-2008].

Wikipedia (2007d). Least squares — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/wiki/Least_squares` [Online; accessed 21-February-2008].

Wikipedia (2007e). QR decomposition — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/wiki/QR_decomposition` [Online; accessed 21-February-2008].

Wikipedia (2007f). Robust regression — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/wiki/Robust_regression` [Online; accessed 21-February-2008].

Index