2009:031 CIV

MASTER'S THESIS

NURBS Surface Fitting with Gauss-Newton

Nils Carlson

Luleå University of Technology

MSc Programmes in Engineering
Computer Science and Engineering
Department of Mathematics

2009:031 CIV - ISSN: 1402-1617 - ISRN: LTU-EX--09/031--SE

Abstract

This paper shows an approach to nonlinear least squares fitting of NURBS curves and surfaces to measured data by the Gauss-Newton method. A line search with regularisation and trust region method are used to reach global convergence, as well as variable substitution and simple bounds.

Acknowledgements

This masters thesis completes many years of hard work towards a Master of Computer Engineering Science with a specialisation in Applied Mathematics at Luleå University of Technology. The work was conducted primarily at the Mid Sweden University in Sundsvall, who generously provided space for a student from Luleå University of Technology.

I would like to thank Mårten Gulliksson who has been my supervisor during this work and guided me through the process of learning more about optimisation, the Gauss-Newton method, nonlinear least squares and techniques for global convergence. I would also like to thank Inge Söderkvist who put me on track when it comes to learning about optimisation and the rewards of finding a maximum or minimum by creating large matrices and putting them to work.

Nils Carlson

Contents

1 Introduction

Reverse engineering is the art of obtaining information about an object which is not given. When given a physical object of some sort reverse engineering can constitute an attempt to reconstruct a digital representation of that object which may then be manipulated and reproduced in new physical forms.

Splines are continuous parametric functions which may be manipulated through control points and are well suited for design work. Reverse engineering of an object often results in a spline model as this has many advantages such as smoothness, low storage requirements and the possibility of changing the shape through the control points.

The process starts with some form of data-collection, often a laser scanner [1] that is able to measure distance or, commonly now, aerial photographs which can use changes in angle with respect to facets on buildings to reconstruct distances [7]. These techniques give a data-set, which may or may not be ordered in some fashion, consisting of three dimensional points.

The next step normally consists of assigning parameters to the data points, mapping points in three dimensions onto a two-dimensional parametric surface, followed by an iterative fitting. Alternatively fitting and parameter assignment may be done simultaneously, adding one point at a time during each iteration. A thorough discussion of these different strategies with variations may be found in [18].

The fitting stage is the focus of this thesis. Non Uniform Rational B-Splines (NURBS) are used, a special type of spline that allows for shape control through control point movement, adjustment of weights for each control point and modification of a knot vector. This thesis does not attempt to adjust the knot vector, focusing instead on control points and weights.

1.1 Earlier Work

The problem has been extensively explored in the previous twenty to thirty years both from a perspective of computer graphics and mathematics. Among the different approaches several are based on optimisation, among others [18].

These approaches can be divided into several categories of optimisation; those based on linear least squares as in [17], nonlinear least squares ([18], [16] and [14]) and separable nonlinear least squares [8].

Another division may be made along the lines of static and dynamic parameters. The parameters are included nonlinearly, and thus disqualify purely linear methods. On the other hand parameter fitting is expensive as each data-point must be mapped to a parameter which must then be updated throughout the fitting process. As the number of data-points is often far greater than the desired number of control points it is often not desirable to spend computational time on optimising the parameter mapping as it is not part of the end-product spline model. Many methods thus perform only a single parameter mapping and make use of a special error function to compensate for the errors caused by drift of the parameters position on the fitted surface in relation to the data-point [18].

1.2 Method

The approach of this thesis is to apply a full-scale Gauss-Newton optimisation process to parameters, control-points and weights using a squared error function. This approach is computationally expensive, but has the advantage of maintaining orthogonality from surface to data point throughout the fitting and should therefore theoretically provide a better fit.

In order to reach convergence two different strategies are tested, line search with regularisation and a trust region method. Constraints are handled through variable transformation and simple bounds, eliminating a level of complexity from the problem.

Finally testing was done on simple geometrical shapes to see to which degree the NURBS curves and surfaces could adapt.

2 Defining NURBS - Non-Uniform Rational B-Splines

2.1 Introduction

NURBS - Non-Uniform Rational B-Splines are a very versatile form of mathematical function very commonly used in CAD applications. They have many "nice" properties and are a versatile design tool. In order to define NURBS it is simplest to first define B-splines of which NURBS are a weighted, rational extension [15].

2.2 B-splines

B-splines are curves in $\mathbb{R}^2$ built up by basis functions which are multiplied by control points and summed. The resulting value at a parameter u is given by

$$g(u) = \begin{pmatrix} g_1(u) \\ g_2(u) \end{pmatrix} = \sum_{i=0}^{n} N_{i,d}(u) p_i \quad a \leq u \leq b, \tag{2.1}$$

where $N_{i,d}(u)$ are the basis functions and $p_i \in \mathbb{R}^2$ are the control points. B-splines are thus parametric, mapping $u \in \mathbb{R}^1$ into $\mathbb{R}^2$ as in equation (2.1). They posses degree d and a knot-vector μ. The simplest definition for the basis function is recursive and is given by

$$\begin{aligned} N_{i,0}(u) &= \begin{cases} 1 & \text{if } \mu_i \leq u < \mu_{i+1} \\ 0 & \text{otherwise} \end{cases} \\ N_{i,d}(u) &= \frac{u - \mu_i}{\mu_{i+d} - \mu_i} N_{i,d-1}(u) + \frac{\mu_{i+d+1} - u}{\mu_{i+d+1} - \mu_{i+1}} N_{i+1,d-1}(u) \\ \mu_i &\in \mu = \{\mu_0, ..., \mu_m\}. \end{aligned} \tag{2.2}$$

In Figure 2.1 is a plot of several basis functions of degree three. For each segment a single basis function has the largest value. As u tends towards a and b the curves of $i = 0$ and $i = n$ dominate completely and are equal to 1. Considering the consequences of multiplying each basis function and summing as in (2.1) it is possible to infer that for each value of u only a limited number of points and basis functions will contribute to to the sum. In Figure 2.2 a simple B-spline is shown, the control points act as "attractors" of the curve.

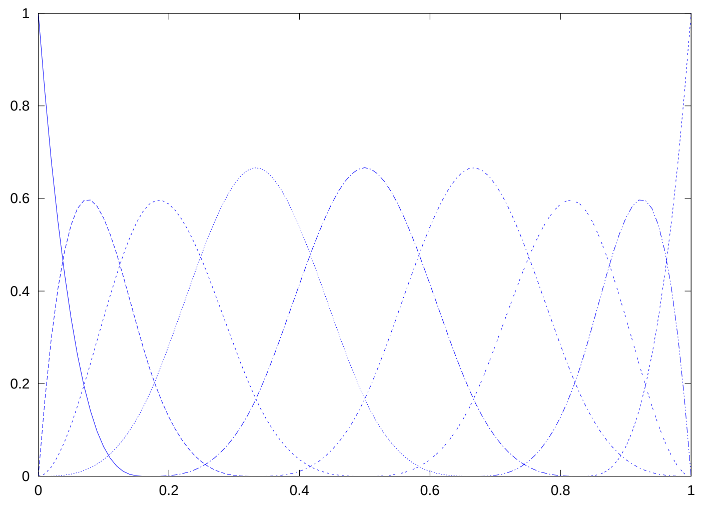


Figure 2.1: A series of basis functions plotted on $0 \leq u \leq 1$ with $\mu = \{0, 0, 0, 0, 1/6, 1/3, 1/2, 2/3, 5/6, 1, 1, 1, 1\}$.

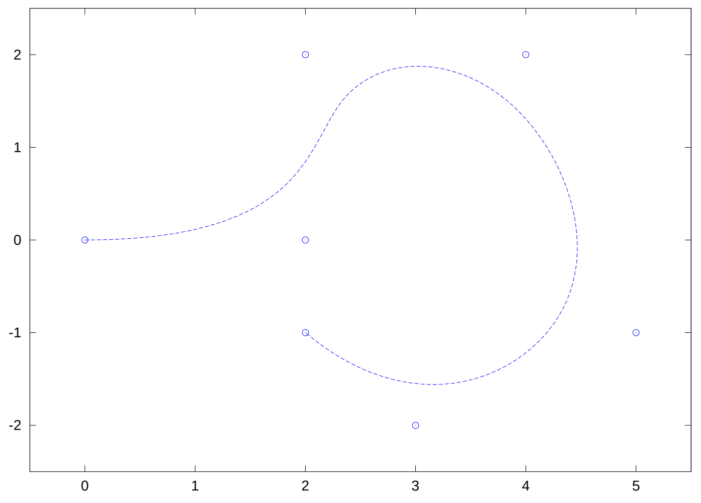


Figure 2.2: A simple B-spline curve, $\mu = \{0, 0, 0, 0, 1/4, 1/2, 3/4, 1, 1, 1, 1\}$, control points are $p = \{(0, 0), (2, 0), (2, 2), (4, 2), (5, -1), (3, -2), (2, -1)\}$

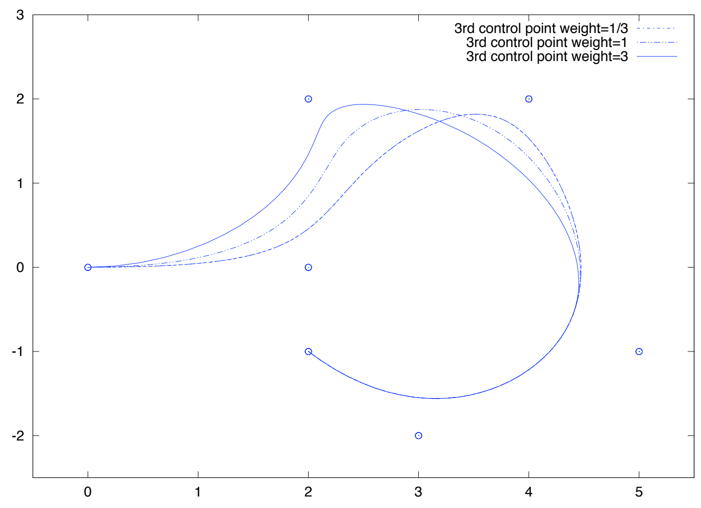


Figure 2.3: A NURBS-curve with varying weights for the third control point.

2.3 NURBS-curves

Non-Uniform Rational B-splines are an extension of B-splines that allow for greater shape control through the use of weights. NURBS are described similarly to B-splines

$$g(u) = \begin{pmatrix} g_1(u) \\ g_2(u) \end{pmatrix} = \frac{\sum_{i=0}^{n} N_{i,d}(u) w_i p_i}{\sum_{i=0}^{n} N_{i,d}(u) w_i} \quad w_i > 0 \quad a \leq u \leq b, \tag{2.3}$$

the denominator acts as a normalising term. Figure 2.3 shows the effect of changing a weight, notice the fact that the curve is modified only locally.

Grouping the basis functions and the weights of equation (2.3) leads to a different form of the NURBS-curve equation, the rational basis form,

$$R_{i,d}(u) = \frac{N_{i,d}(u) w_i}{\sum_{j=0}^{n} N_{j,d}(u) w_j}, \tag{2.4}$$

$$g(u) = \sum_{i=0}^{n} R_{i,d}(u) p_i. \tag{2.5}$$

2.4 NURBS-surfaces

Extending NURBS-curves into NURBS-surfaces is done through a method known as the tensor product scheme. This views the surface as a product of two curves at any given point as

$$g(u,v) = \begin{pmatrix} g_1(u,v) \\ g_2(u,v) \\ g_3(u,v) \end{pmatrix} = \frac{\sum_{i=0}^{n}\sum_{j=0}^{m} N_{i,d_u}(u) N_{j,d_v}(v) w_{i,j} p_{i,j}}{\sum_{i=0}^{n}\sum_{j=0}^{m} N_{i,d_u}(u) N_{j,d_v}(v) w_{i,j}} \quad w_{i,j} > 0 \quad 0 \leq u, v \leq 1. \tag{2.6}$$

The NURBS-surface makes use of two knot vectors μ and ν;

$$\mu = \{\underbrace{0, ..., 0}_{d_u+1}, \mu_{d_u+1}, ..., \mu_{r-d_u-1}, \underbrace{1, ..., 1}_{d_u+1}\}$$

and

$$\nu = \{\underbrace{0, ..., 0}_{d_v+1}, \nu_{d_v+1}, ..., \nu_{s-d_v-1}, \underbrace{1, ..., 1}_{d_v+1}\},$$

where $r = n + d_u + 1$ and $s = m + d_v + 1$.

This function can also be written on a rational basis form

$$R_{i,j}(u,v) = \frac{N_{i,d_u}(u) N_{j,d_v}(v) w_{i,j}}{\sum_{k=0}^{n}\sum_{l=0}^{m} N_{k,d_u}(u) N_{l,d_v}(v) w_{k,l}} \tag{2.7}$$

with

$$g(u,v) = \sum_{i=0}^{n}\sum_{j=0}^{m} R_{i,j}(u,v) p_{i,j}. \tag{2.8}$$

Figure 2.4 shows a simple NURBS-surface.

2.5 Properties of NURBS

NURBS have a large number of very useful properties, here are listed only a few[15].

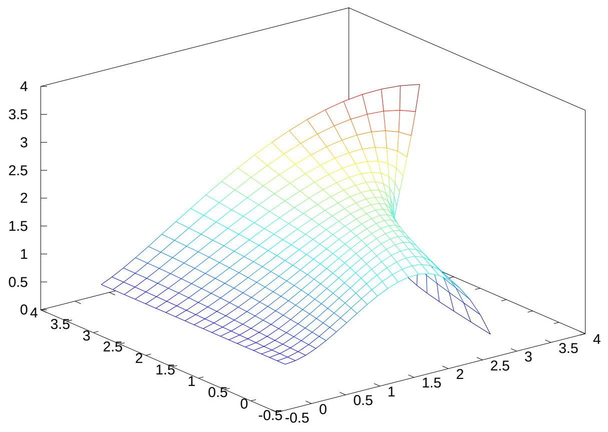


Figure 2.4: A NURBS-surface defined on knot-vectors of length 4 with 4x4 control points.

- Corner point interpolation: $s(0,0) = p_{0,0}$, $s(1,0) = p_{n,0}$, $s(0,1) = p_{0,m}$, $s(1,1) = p_{n,m}$, the same principle applies for the curve.
- Strong convex hull property: If $(u,v) \in [\mu_{i_0}, \mu_{i_0+1}) \times [\nu_{j_0}, \nu_{j_0+1})$ then $s(u,v)$ is in the hull of the control points $p_{i,j}$, $i_0 - d_u \leq i \leq i_0$ and $j_0 - d_v \leq j \leq j_0$.
- Local modification: if $p_{i,j}$ or $w_{i,j}$ is changed only the surface shape of the rectangle $[\mu_i, \mu_{i+d_u+1}) \times [\nu_j, \nu_{j+d_v+1})$.
- Differentiability: $s(u,v)$ is $d_u - k$ or $d_v - k$ times differentiable with respect to u or v at a u or v knot of multiplicty k.

3 Nonlinear Optimisation

3.1 Formulation

A nonlinear optimisation problem is most simply formulated as

$$\min_x F(x) \qquad x \in \mathbb{R}^n, \tag{3.1}$$

where $x = (x_1, ..., x_n)^T$ and F is a nonlinear function. In practice we also limit ourselves to twice-differentiable functions where first and second derivatives are both continuous [13].

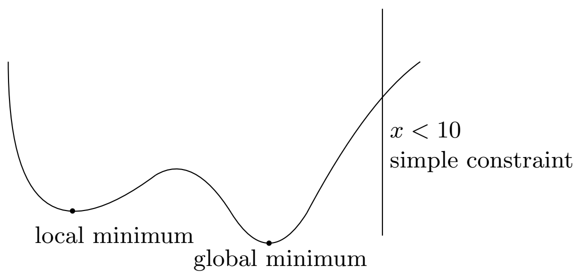


Figure 3.1: One-dimenstional nonlinear problem with local minimum and one constraint

3.2 Necessary Conditions for Optimality

Consider the basic optimisation problem posed in equation 3.1 where $F(x)$ is a nonlinear function. The function may as shown in figure 3.1 have multiple minima of which only one is in fact a global minimum. When, as is most often the case, only a local-perspective is available to an algorithm it is most often impossible to tell whether a minimum is local or global, consequently focused is in most cases placed on finding a local minimum.

A local minima is formally defined as a point x^* such that $F(x^*) \leq F(x), x \in B(x^*, \delta), \delta > 0$ where $B(x^*, \delta)$ is a ball function. Formulated in words this implies that the local minimum $F(x^*)$ has the smallest value within a defined region surrounding it. A local minimum is characterised by two properties:

- It is a stationary point, in other words $F'(x^*) = 0$.

- The second derivative must be positive, $F''(x^*) > 0$.

3.3 Iterative methods

Most methods for optimisation are iterative, implying that a similar procedure is applied multiple times to reach an optimal point. These algorithms are characterised algorithmically by the form

```
while $F(x_k)$ not optimal
    compute a step $p$
    $x_{k+1} = x_k + p$.
```

The step p is normally computed through information obtained from the function value, derivatives and second derivatives.

3.4 The Newton-Rhapson method

The Newton-Rhapson method makes use of a linearisation of a nonlinear equation to find a zero,

$$F(x) = 0. \tag{3.2}$$

The Taylor series

$$F(x_0 + p) = F(x_0) + pF'(x_0) + \frac{1}{2}p^2F''(x_0) + ... + \frac{p^n}{n!}F^{(n)}(x_0) \tag{3.3}$$

provides a polynomial approximation of a curve. Dropping all but the first two terms provides a simplified and less accurate

$$F(x_0 + p) = F(x_0) + pF'(x_0). \tag{3.4}$$

To locate a zero, where $F(x_0 + p)$ it is sufficient to solve the equation

$$0 = F(x_0) + pF'(x_0) \tag{3.5}$$

i.e., $p = -F(x_0)/F'(x_0)$ with the next point $x_1 = x_0 + p$ as a new approximate solution to (3.5). Generally, we set the iterative scheme

$$x_{k+1} = x_k + p_k \tag{3.6}$$

$$\text{where } p_k = -F(x_k)/F'(x_k) \tag{3.7}$$

and x_0 is given.

3.5 Newtons Method for Optimisation

While the Newton method looks for zeros in the value of the function Newtons method for optimisation tries to fulfil the first necessary condition for optimality; that the first derivative be equal to zero. The optimisation version of Newton-Rhapson is

$$\begin{aligned} \text{solve } [\nabla^2 F(x_k)]p_k &= -\nabla F(x_k) \\ x_{k+1} &= x_k + p_k. \end{aligned} \tag{3.8}$$

Equation (3.8) is the numerical matrix version of Newtons method for optimisation. The notation emphasises the fact that p_k is the solution to the equation, and that the inverse of $[\nabla^2 F(x_k)]$ is never calculated.

4 Nonlinear Least Squares

The problem of fitting a curve to measured data can be posed as an optimisation problem. Let $g(u)$ be a parametric curve and consider the problem of fitting $g(u)$ to the measured points $\tilde{g}_1, \tilde{g}_2, ..., \tilde{g}_m$ so that the sum of the square of the distances to the points $g(u_1), g(u_2), ..., g(u_m)$ on the curve is minimised. Let $d(a, b)$ be the distance from a to b then the problem can be posed as a minimisation problem of the form

$$\min_{u \in \mathbb{R}^m} \sum_{i=1}^{m} d(g(u_i), \tilde{g}_i)^2. \tag{4.1}$$

4.1 Orthogonal Distance

The distance from a function can be defined in several ways. Simplest is to assume that the x-axis is precise and that the function must only be fitted in the y-directon. Figure 4.1 illustrates this.

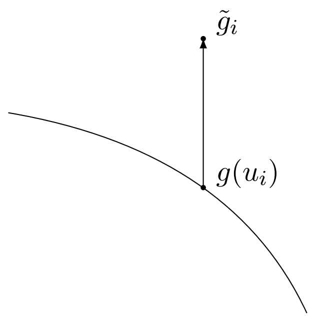


Figure 4.1: Distance on only the y-axis

An alternative is to measure distance orthogonally to the function, resulting in the minimal distance from measured point to function. With points

$$a = \begin{pmatrix} a_1 \\ a_2 \end{pmatrix} \text{ and } b = \begin{pmatrix} b_1 \\ b_2 \end{pmatrix},$$

let the distance function be defined as

$$d(a, b) = \sqrt{(a_1 - b_1)^2 + (a_2 - b_2)^2}.$$

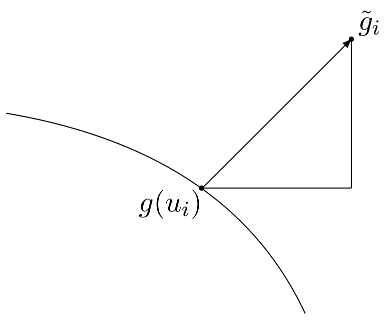


Figure 4.2: Orthogonal distance

With a point on the curve

$$g(u_i) = \begin{pmatrix} g_1(u_i) \\ g_2(u_i) \end{pmatrix}$$

and a measured point

$$\tilde{g}_i = \begin{pmatrix} \tilde{g}_1^{(i)} \\ g_2^{(i)} \end{pmatrix}$$

the distance from the curve to measured point is

$$d(g(u_i), \tilde{g}_i) = \sqrt{(g_1(u_i) - \tilde{g}_1^{(i)})^2 + ((g_2(u_i) - \tilde{g}_2^{(i)}))^2}, \qquad (4.2)$$

as shown in Figure 4.2.

Substituting the orthogonal-distance (4.2) into the minimisation problem (4.1) results in a problem of the form

$$\min_{u \in \mathbb{R}^m} \sum_{i=1}^{m} ((g_1(u_i) - \tilde{g}_1^{(i)})^2 + ((g_2(u_i) - \tilde{g}_2^{(i)}))^2. \qquad (4.3)$$

Let $f(u)$ be defined as the vector

$$f(u) = \begin{pmatrix} g_1(u_1) - \tilde{g}_1^{(1)} \\ g_2(u_1) - \tilde{g}_2^{(1)} \\ g_1(u_2) - \tilde{g}_1^{(2)} \\ g_2(u_2) - \tilde{g}_2^{(2)} \\ \vdots \\ g_1(u_m) - \tilde{g}_1^{(m)} \\ g_2(u_m) - \tilde{g}_2^{(m)} \end{pmatrix}$$

Rewriting the minimisation problem once again using $f(u)$ presents the problem in a new form,

$$\min_{u \in \mathbb{R}^m} f(u)^T f(u), \tag{4.4}$$

or alternatively as a least squares formulation

$$\min_{u \in \mathbb{R}^m} ||g(u) - \tilde{g}||_2^2, \tag{4.5}$$

where

$$g(u) = \begin{pmatrix} g_1(u_1) & g_2(u_1) & g_1(u_2) & g_2(u_2) & \dots & g_1(u_m) & g_2(u_m) \end{pmatrix}^T \tag{4.6}$$

and

$$\tilde{g} = \begin{pmatrix} g_1^{(1)} & g_2^{(1)} & g_1^{(2)} & g_2^{(2)} & \dots & g_1^{(m)} & g_2^{(m)} \end{pmatrix}^T . \tag{4.7}$$

4.2 Gauss-Newton

The Gauss-Newton method is closely related Newton's method for optimisation. From section 3.5 it is known that to minimise a function $f(x)$ using Newtons method requires the first and second derivative. Multiplying the function to be minimised of equation (4.4) by $\frac{1}{2}$ yields

$$F(u) = \frac{1}{2} f(u)^T f(u). \tag{4.8}$$

Let the derivative of the vector $f(u)$ be the Jacobian

$$J(u) = \begin{pmatrix} \frac{df_1}{du_1} & \frac{df_1}{du_2} & \cdots & \frac{df_1}{du_m} \\ \frac{df_2}{du_1} & \frac{df_2}{du_2} & \cdots & \frac{df_2}{du_m} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{df_m}{du_1} & \frac{df_m}{du_2} & \cdots & \frac{df_m}{du_m} \end{pmatrix} .$$

Applying the chain rule to equation (4.8) provides the derivative

$$\frac{d}{du}(\frac{1}{2}f(u)^T f(u)) = J(u)^T f(u).$$

The second derivative is normally given by the Hessian matrix which can be approximated as

$$\begin{aligned} H(u) =& J(u)^T J(u) + \sum f_i(u) \nabla^2 f_i(u) \\ &\approx J(u)^T J(u). \end{aligned} \tag{4.9}$$

The Gauss-Newton method makes use of this approximation of the Hessian in order to reduce the computational complexity of the problem as calculating all second derivatives is prohibitively expensive. With k as an iteration index and disregarding initiation Gauss-Newton is summarised in Algorithm 4.1. Worth noting at this point is that for numerical reasons p is not solved as above and instead of a tolerance some form of halting condition is used, this is described in the coming chapters.

Algorithm 4.1 Gauss-Newton

```
while $f(u^{(k)})^T f(u^{(k)}) >$ tolerance do
    solve $J(u^{(k)})^T J(u^{(k)}) p_k = -J(u^{(k)})^T f(u^{(k)})$
    $u^{(k+1)} \leftarrow u^{(k)} + p_k$
    $k \leftarrow k + 1$
end while
```

5 Regularisation, Line Search and Trust Region

Quasi-Newton methods such as Gauss-Newton often require some additional technique to achieve good results. These techniques are required to compensate for the locality of the model - the Taylor series - and the aim of the optimisation, to achieve global convergence.

The three techniques that will be presented here all aim at limiting the direction or length of the step p. Regularisation attempts to keep the step centred within some area in which solutions are considered probable or desirable. Line search limits the step length during each step, forcing a decrease in each iteration. Trust region methods act like line search but make use of additional information available within the model to take steps which may - or may not - lead to faster global convergence.

5.1 Regularisation

Regularisation is a method for augmenting the problem with an additional term or terms in order to keep the solution within some area where solutions are believed to lie or that in some other way is desirable. The additional term is scaled during each iteration, decreasing its influence as the algorithm converges. Regularisation of a nonlinear least squares problem in its most simple form can be written as

$$\min_{u \in \mathbb{R}^m} f(u)^T f(u) + \alpha ||r(u)||_2^2, \tag{5.1}$$

where the α is the scaling factor and $||r(u)||_2^2$ represents the regularisation term. A common approach to the scaling is to decrease the scaling factor, α, by 1/2 during each iteration.

5.2 Line Search

Gauss-Newton guarantees a descent direction, but depending on the nonlinearity of the problem a step taken at full length may lead to an increase in the value of the function to be minimised. Line search limits the length of the step to such a point that a sufficient decrease in the function value is made.

The choice to be made when using line search is that of scaling factor, which is iteratively multiplied with the step until a sufficient decrease is achieved, a common choice is 1/2. However, more complicated choices can be made, some very elaborate involving a model function[9]. Algorithm 5.1 shows a very simple inner iteration for performing a line search.

Algorithm 5.1 Line Search on Gauss-Newton

```
calculate the full Gauss-Newton step, $p$
$F(u_{k+1}) \leftarrow F(u_k + p)$
$\gamma \leftarrow 1$
while $||F(u_{k+1})|| > |F(u_k)||$ do
  $\gamma \leftarrow \frac{1}{2}\gamma$
  $F(u_{k+1}) \leftarrow F(u_k + \gamma p)$
end while
$k \leftarrow k + 1$
```

5.3 Trust Region

Trust region methods, like line search, start with a full step, but where line search simply shortens the step by a scaling factor trust region methods attempt to make better use of information inherent to the problem. Recalling from section 3.5 that the basis of Newtons method for optimisation is the Taylor series let this same Taylor series be the model

$$m(u+p) = F(u) + pF'(u) + \frac{1}{2}p^2F''(u). \tag{5.2}$$

Generalising the model to higher dimensions gives

$$m(u+p) = F(u) + g^Tp + \frac{1}{2}p^THp \tag{5.3}$$

where g is the gradient and H is the hessian of $F(u)$. Then solving a trust region problem consists of solving (just as in the normal newton method) a series of sub-problems but with an additional constaint;

$$\begin{aligned} &\min_p m(u+p) = F(u) + g^Tp + \frac{1}{2}p^THp \\ &\text{subject to } ||p||_2 \leq \delta. \end{aligned} \tag{5.4}$$

The $||p||_2 \leq \delta$ represents the region for which the model can be trusted. There are then two outstanding issues; how to handle the constraint and which value to assign δ.

5.3.1 The Augmented Hessian

The method of the augmented hessian was developed first by K. Levenberg in 1944 [10] and later expanded upon by D. W. Marquardt in 1963 [11]. It provides a computationally very simple and desirable technique for limiting the steplength by solving an augmented newton step. Let H be the Hessian (in the case of Gauss-Newton composed of $J(u)^TJ(u)$) and g be the gradient (for Gauss-Newton $J(u)^Tf(u)$) then

$$(H + \lambda I)p = -g \tag{5.5}$$

has the solution $p(\lambda)$ with the property $||p(\lambda)||_2 = \delta$ for some λ[9].

5.3.2 The "Hook" Step

Given a specific δ_k there is however no direct way of calculating a suitable λ, instead a nonlinear equation is obtained;

$$\Phi(\lambda) = ||p(\lambda)||_2 - \delta_k = 0, \tag{5.6}$$

$$\Phi'(\lambda) = \frac{p(\lambda)^T(H+\lambda I)^{-1}p(\lambda)}{||p(\lambda)||}. \tag{5.7}$$

It can be shown that solving this problem through Newton-Rhapson will lead to very slow convergence as the step length will be underestimated during each iteration. This is resolved through iteration with Newton-Rhapson like steps of the form

$$\lambda_+ = \lambda_c - \frac{||p(\lambda_c)||_2}{\delta_k}\left[\frac{\Phi(\lambda_c)}{\Phi'(\lambda_c)}\right] \tag{5.8}$$

where λ_+ is the new value of lambda, λ_c is the current value and the scaling term $||p(\lambda)||_2/\delta_k$ acts to compensate for underestimation inherent in the model[9]. The actual algorithm suggested by [9] contains many refinements and does not require a high degree of convergence, an outline of the main loop is given in Algorithm 5.2.

5.3.3 Updating The Trust Region

Identifying the optimal step-length, δ_k, is a matter of compromise, too long a step increases the risk of overshooting the target while too short a step will increase the number of iterations needed. One very intuitive measure of how good the model is at a given step length is given by Moré [12],

$$\rho = \frac{||F(u)||_2^2 - ||F(u+p)||_2^2}{||F(u)||_2^2 - ||F(u)+F'(u)p)||_2^2}. \tag{5.9}$$

As can be seen the numerator consists of the actual decrease in the value of the function and the denominator of the decrease in the model, providing a value ρ which is at 1 for a perfect linearity and approaches 0 as the nonlinearity increases. However, it is normally not interesting to compute the value of ρ; instead a number of criterion and a quadratic model are used to determine δ[9].

Assume that a step has just been calculated, then the simplest criterion for accepting a step of length δ_k is to check that $F(u_{k+1})$ is smaller

Algorithm 5.2 The Hook Step (main loop outline) - finding λ for a given δ_k

```
$low \leftarrow 0.75$
$hi \leftarrow 1.5$
Initiate $\lambda$
**repeat**
  $p \leftarrow -(H + \lambda_c I)^{-1} g$
  $\Phi \leftarrow ||p|| - \delta_k$
  $\Phi' \leftarrow \dfrac{p^T H^{-1} p}{||p||}$
  **if** $((||p|| > low * \delta_k)$ and $(||p|| < hi * \delta_k))$ or $(\lambda_{up} - \lambda_{low} \leq 0)$ **then**
    $done \leftarrow true$
  **else**
    $\lambda_{low} = \max(\lambda_{low}, \quad \lambda_c - \Phi/\Phi')$
    **if** $\Phi < 0$ **then**
      $\lambda_{up} \leftarrow \lambda_c$
    **end if**
    $\lambda_+ \leftarrow \lambda_c - \dfrac{||p||}{\delta_k}\dfrac{\Phi}{\Phi'}$
  **end if**
**until** $done$
```

than $F(u_k)$, or more stringently that

$$F(u_{k+1}) \leq F(u_k) + \alpha g_c^T(u_{k+1} - u_k), \tag{5.10}$$

where α is a very small number (commonly 10^{-4}). The gradient, $g_c^T(u_{k+1} - u_k)$, is included to force a slightly more significant descent.

If the step does not lead to a decrease in $F(u_{k+1})$ then the step must be shortened (though in practice a tolerance is used to limit how short the step can be.) Shortening is accomplished by creating a quadratic model with the currently available information; $F(u_k)$, $g^T(u_{k+1} - u_k)$ (the directional derivative) and $F(u_{k+1})$. The model must thus have the value $F(u_k)$ at $\delta = 0$, an initial slope of $g^T(u_{k+1} - u_k)$ and the value $F(u_{k+1})$ at $\delta = \delta_k$ which yields a model of

$$m(\delta) = F(u_c) + g^T(u_{k+1} - x_k)\delta + (F(u_{k+1}) - F(u_k) - g^T(u_{k+1} - u_k))\delta^2. \tag{5.11}$$

The minimum of $m(\delta)$ can be found at $m'(\delta) = 0$, or solving for δ

$$\delta = \frac{-g^T(u_{k+1} - u_k)}{2(F(u_{k+1}) - F(u_k) - g^T(u_{k+1} - u_k))}. \tag{5.12}$$

If on the other hand the step was accepted then it is possible to use available information to better asses the next step. Recalling the model (5.3), the predicted decrease is modelled by

$$m(u + p) - F(u) = \Delta F_{pred} = g^T s + \frac{1}{2} s^T H s, \tag{5.13}$$

while the actual decrease is given by

$$\Delta F = F(u_{k+1}) - F(u_k) \tag{5.14}$$

If the current step followed the model closely, $|\Delta F_{pred} - \Delta F| \leq 0.1|\Delta F|$, then δ_k may be lengthened and a longer step taken in the same iteration.

Moving onto the next iteration there are three possibilities, $\delta_{k+1} = \delta_k$, $\delta_{k+1} < \delta_k$ and $\delta_{k+1} > \delta_k$. The suggested rules are to decrease δ_{k+1} if $\Delta F \geq 0.1\Delta F_{pred}$, increase if $\Delta F \leq 0.75\Delta F_{pred}$ and otherwise set $\delta_{k+1} = \delta_k$[9].

6 Nonlinear Least Squares Fitting of a NURBS Curve

6.1 Problem Formulation

In order to apply the Gauss-Newton optimisation method to fitting a NURBS curve onto measured data points it is important to decide which of the variables are to be fitted, and which are to be fixed.

Fixing the degree at $d = 3$ is a good compromise as greater degree provides little benefit and lower degree decreases the smoothness of the derivative. The knot vector, μ, is also fixed as adjusting the knots would involve very complicated constraints of the form

$$\mu_0 \leq \mu_1 \leq \ldots \leq \mu_n. \tag{6.1}$$

The remaining variables are the control points, $p \in \mathbb{R}^{2\times n}$, and weights, $w \in \mathbb{R}^n$, and parameters corresponding to measured points, $u \in \mathbb{R}^m$. The optimisation problem may then be written as

$$\min_{u,p,w} \sum_{k=1}^{m} ||g(u_k, p, w) - \tilde{g}_k||_2^2 = f(u, p, w)^T f(u, p, w)$$

$$\text{where } g(u_k, p, w) = \begin{pmatrix} g_1(u_k, p, w) \\ g_2(u_k, p, w) \end{pmatrix} = \frac{\sum_{i=0}^{n} N_{i,d}(u_k) w_i p_i}{\sum_{i=0}^{n} N_{i,d}(u_k) w_i}$$

$$\text{subject to } w_i > 0$$
$$0 \leq u_k \leq 1.$$

6.2 Initial values

In order to start the Gauss-Newton iteration initial values of the variables must somehow be obtained. The greatest dependency is on finding values for u that will correctly reflect the data-set so ordering will be correct, and placement in relation to control points will be suitable for the fitted curve to assume the correct shape. Parameter mapping will not be dealt with by this thesis as this is in itself an very extensive topic, instead a simple parameter mapping is assumed. For a small survey of parameter mapping see [6].

After parameters have be initialised the control points p can be identified by disregarding the weights, in effect setting $w_i = 1, \forall i$ and solving a linear least squares problem. Finally end control points are fixed to the end points of the data set to avoid drift.

Parameter mapping of data points $\tilde{g}_1, \tilde{g}_2, ..., \tilde{g}_m$ results in a vector

$$u = \begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ u_m \end{pmatrix}, \qquad 0 \leq u_i \leq 1. \tag{6.2}$$

As all weights are equal to 1 the NURBS is now a normal B-spline with basis-functions $N_i(u)$ and points combined linearly, allowing for the reformulation of (2.1) as

$$g(u) = \sum_{i=0}^{n} N_i(u) p_i = \begin{pmatrix} N_0(u) & N_1(u) & ... & N_n(u) \end{pmatrix} \begin{pmatrix} p_0 \\ p_1 \\ \vdots \\ p_n \end{pmatrix}, \tag{6.3}$$

where $p_i \in \mathbb{R}^2$. Evaluating the basis functions for each of the parameters and writing in matrix form provides, if $m > n$, gives us a linear least squares problem

$$\min_p \left\| \begin{pmatrix} N_0(u_1) & N_1(u_1) & ... & N_n(u_1) \\ N_0(u_2) & N_1(u_2) & ... & N_n(u_2) \\ \vdots & \vdots & \ddots & \vdots \\ N_0(u_m) & N_1(u_m) & ... & N_n(u_m) \end{pmatrix} \begin{pmatrix} p_d^{(0)} \\ p_d^{(1)} \\ \vdots \\ p_d^{(n)} \end{pmatrix} - \begin{pmatrix} \tilde{g}_d^{(1)} \\ \tilde{g}_d^{(2)} \\ \vdots \\ \tilde{g}_d^{(m)} \end{pmatrix} \right\|, \tag{6.4}$$

where d is the dimension. Solving for the p vector twice, once for each dimension, provides initial values for the control points which may be used in the main optimisation.

6.3 The Derivatives of a NURBS curve

The calculation of derivatives is often a costly part of a Gauss-Newton iteration. Fortunately the derivatives of NURBS curve can be calculated with only minor extra cost.

6.3.1 Derivative with respect to u

From section 2.3 it is known that $g(u)$ is a rational function and thus composed of a polynomial function in both the numerator and denominator. The derivative of a polynomial is simply a lower-order polynomial and with some modification this is true for a NURBS-curve.

The derivative with respect to u of a B-spline is the inner-product of the derivatives of the basis-functions and the points,

$$\frac{\partial g(u,p)}{\partial u} = \sum_{i=0}^{n} N_i'(u)p_i. \tag{6.5}$$

The derivative of a basis-function is

$$N_{i,d}'(u) = \frac{d}{\mu_{i+d}-\mu_i}N_{i,d-1}(u) + \frac{d}{\mu_{i+d+1}-\mu_{i+1}}N_{i+1,d-1}(u), \tag{6.6}$$

which when compared to (2.2) can be seen to be almost identical, thus computed at minor extra cost[15].

Tackling the NURBS let the numerator and denominator respectively be

$$n(u) = \sum_{i=0}^{n} N_i(u)w_ip_i \tag{6.7}$$

and

$$m(u) = \sum_{i=0}^{n} N_i(u)w_i. \tag{6.8}$$

Then the NURBS function can be formulated as

$$g(u,p,w) = \frac{n(u)}{m(u)} = \frac{m(u)g(u)}{m(u)}. \tag{6.9}$$

The derivative of $g(u,p,w)$ with respect to u is then

$$\begin{aligned}\frac{\partial g(u,p,w)}{\partial u} &= \frac{m(u)n'(u) - m'(u)n(u)}{m(u)^2}\\ &= \frac{m(u)n'(u) - m'(u)m(u)g(u)}{m(u)^2}\\ &= \frac{n'(u) - m'(u)g(u)}{m(u)}. \end{aligned}\tag{6.10}$$

6.3.2 Derivative with respect to p_i

Recalling from section 2.3 that $g(u)$, or alternatively $g(p_i)$, can also be formulated as a sum of rational functions (2.5) the derivative with respect to p_i is given by the rational-basis as

$$\frac{\partial g(u,p,w)}{\partial p_i} = R_i(u) = \frac{N_i(u)w_i}{\sum_{j=0}^{n} N_j(u)w_j}. \tag{6.11}$$

6.3.3 Derivative with respect to w_i

As w_i appears in both numerator and denominator the derivative is the derivative of a fraction and as was the case in section 6.3.1 it is simplest to split the problem and then combine. Viewing the numerator this time as a function of w_i the numerator and denominator are given by

$$n(w) = \sum_{i=0}^{n} N_i(u)w_i p_i \tag{6.12}$$

and

$$m(w) = \sum_{i=0}^{n} N_i(u)w_i, \tag{6.13}$$

The derivatives of which are

$$n'(w_i) = N_i(u)p_i \tag{6.14}$$

and

$$m'(w_i) = N_i(u). \tag{6.15}$$

The derivative of a NURBS-curve with respect to w_i then becomes

$$\frac{\partial g(u,p,w)}{\partial w_i} = \frac{m(w)n'(w_i) - m'(w_i)n(w)}{m(w)^2}, \tag{6.16}$$

of which all the constituent parts have already been computed.

6.4 Simple Bounds Constraints and Variable Transformation

The NURBS function of section 2.3 is defined on the parameter $0 \leq u \leq 1$ while the weights belong to $\mathbb{R}_+$ and least-fitting therefore does not truly represent a case of unconstrained optimisation. In order to handle these constraint two different strategies are employed; simple bounds on the parameter u and a variable transformation on w_i.

6.4.1 Simple Bounds

A constraint of the form $u \leq b$, $u \geq a$ or $a \leq u \leq b$ can be handled during the optimisation process by restricting the values of u to those that lie within the constraint by, during each iteration, if u has assumed a disallowed value, setting the value of u to the closest allowable value. Algorithm 6.1 illustrates this.

Algorithm 6.1 Simple Bounds for a NURBS parameter

```
1: if $u_i < 0$ then
2:     $u_i \leftarrow 0$
3: else
4:     if $u_i > 1$ then
5:         $u_i \leftarrow 1$
6:     end if
7: end if
```

6.4.2 Variable Transformation

In order to keep the weights within the open set $(0, \infty)$ a variable transformation is used, $w_i \to e^{w_i}$, neatly mapping $\mathbb{R} \to \mathbb{R}_+ \backslash \{0\}$. The NURBS curve is can then be defined as

$$g(u_k, p, w) = \begin{pmatrix} g_1(u_k, p, w) \\ g_2(u_k, p, w) \end{pmatrix} = \frac{\sum_{i=0}^{n} N_{i,d}(u_k) e^{w_i} p_i}{\sum_{i=0}^{n} N_{i,d}(u_k) e^{w_i}}. \tag{6.17}$$

6.5 Constructing the Jacobian

The Jacobian matrix, as described in section 4.2, is made up of all the previously described derivatives of the NURBS curves, the parameters, the control points and the weights. Letting the vector u be the parameter vector, the first m derivatives are with respect to each parameter, these are followed by the derivatives with respect to the n control points and n weights. Each derivative is defined in two dimensions, providing $2m$ rows.

The Jacobian for a single u_i where u is the vector of parameters corresponding to measured points can be divided into three parts corresponding to each derivative. The first is with respect to the parameter vector u;

$$J_u = \begin{pmatrix} \frac{\partial g(u_1,p,w)}{\partial u} & \begin{matrix}0\\0\end{matrix} & \cdots & \begin{matrix}0\\0\end{matrix} \\ \begin{matrix}0\\0\end{matrix} & \frac{\partial g(u_2,p,w)}{\partial u} & \cdots & \begin{matrix}0\\0\end{matrix} \\ \vdots & \vdots & \ddots & \vdots \\ \begin{matrix}0\\0\end{matrix} & \begin{matrix}0\\0\end{matrix} & \cdots & \frac{\partial g(u_m,p,w)}{\partial u} \end{pmatrix} \in \mathbb{R}^{2m\times m}. \tag{6.18}$$

where $\frac{\partial g(u_i,p,w)}{\partial u_i} \in \mathbb{R}^2$.

The second part of the Jacobian is with respect to the control points $p = (p_1, p_2, ...p_n)$. The derivative with respect to a control point is a scalar, while $p_i \in \mathbb{R}^2$, giving a block matrix of the form

$$B_{p_i}(u_l) = \begin{pmatrix} \frac{\partial g(u_l,p,w)}{\partial p_i} & 0 \\ 0 & \frac{\partial g(u_l,p,w)}{\partial p_i} \end{pmatrix}, \tag{6.19}$$

where the non-zero diagonal elements are with respect to the two dimensions of the control point, p_i. The Jacobian for the control points is

$$J_p = \begin{pmatrix} B_{p_1}(u_1) & B_{p_2}(u_1) & \dots & B_{p_1}(u_n) \\ B_{p_1}(u_2) & B_{p_2}(u_2) & \dots & B_{p_2}(u_n) \\ \vdots & \vdots & \ddots & \vdots \\ B_{p_1}(u_m) & B_{p_2}(u_m) & \dots & B_{p_m}(u_n) \end{pmatrix} \in \mathbb{R}^{2m\times 2n}. \tag{6.20}$$

The third part of the Jacobian is with respect to the weights, $w = (w_1, w_2, ..., w_n)$. The derivative with respect to a weight is in $\mathbb{R}^2$, giving us a Jacobian matrix similar to that of equation (6.18)

$$J_w = \begin{pmatrix} \frac{\partial g(u_1,p,w)}{\partial w_1} & \frac{\partial g(u_1,p,w)}{\partial w_2} & \cdots & \frac{\partial g(u_1,p,w)}{\partial w_n} \\ \frac{\partial g(u_2,p,w)}{\partial w_1} & \frac{\partial g(u_2,p,w)}{\partial w_2} & \cdots & \frac{\partial g(u_1,p,w)}{\partial w_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial g(u_m,p,w)}{\partial w_1} & \frac{\partial g(u_m,p,w)}{\partial w_2} & \cdots & \frac{\partial g(u_m,p,w)}{\partial w_n} \end{pmatrix} \in \mathbb{R}^{2m\times n}. \qquad (6.21)$$

Together (6.18), (6.20), (6.21) give

$$J = (J_u \quad J_p \quad J_w) \in \mathbb{R}^{2m\times(m+3n)}. \qquad (6.22)$$

Then for a well posed problem $2m \geq m + 3n \Rightarrow m \geq 3n$.

6.6 Outline of Algorithm

Two types of algorithms are presented in this section, one making use of line search with regularisation and the other a pure trust region approach. Both algorithms show global convergence for our examples.

6.6.1 Line Search with Regularisation

In order to limit the values which the weights will assume during iteration a regularisation term is introduced,

$$||r(w)||_2^2 = ||w||_2^2, \tag{6.23}$$

where w is the transformed weight of equation (6.17). This regularisation method is described in [5], though adapted for the purpose slightly as it is known that as a rule the weights, e_i^w, should not assume values much greater than 10, or much less than 0.1 as a rule.

The objective function, $f(u,p,w)^T f(u,p,w)$, is then augmented as

$$f(u,p,w)_{reg} = \begin{pmatrix} f(u,p,w) \\ \alpha w \end{pmatrix} \tag{6.24}$$

where

$$w = \begin{pmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \end{pmatrix} \tag{6.25}$$

and α is the scaling term.

This term forces a quadratic penalty when moving away from $w_i = 0$, ensuring that control points will be the primary means of changing the shape of the curve.

During each iteration the regularising term is scaled by a factor α, in this case $\alpha = ||f||_2^2/n$. This causes the penalty on moving the weights away from zero to decrease proportionally to the decrease in the rest of the function as the algorithm converges.

The Jacobian is correspondingly modified through the addition of n rows, the number of weights, only the columns corresponding to J_w are modified. The derivative of each w_i is 1, leading to a new part of the

Jacobian,

$$J_{reg} = \alpha \begin{pmatrix} 1 & 0 & ... & 0 \\ 0 & 1 & ... & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & ... & 1 \end{pmatrix} \in \mathbb{R}^{n \times n}. \tag{6.26}$$

The full Jacobian may now be written as

$$J = \begin{pmatrix} J_u & J_p & J_w \\ 0 & 0 & J_{reg} \end{pmatrix} \in \mathbb{R}^{(2m+n)\times(m+3n)}. \tag{6.27}$$

A full line search based algorithm may now be assembled, consisting of initiation of the variables, starting iteration, creating the augmented Jacobian and evaluationg the function, performing a line search and finally updating the variables with simple bounds and repeating until convergence is attained. Algorithm 6.2 provides an outline of this algorithm.

Algorithm 6.2 NURBS Curve Fitting with Line Search

```
$u \leftarrow$ Perform a simple parameter mapping (parameter to dimension)
$w \leftarrow (0, 0, ..., 0)$ (transformed variables $e^{w_i}$)
$p \leftarrow$ Solve LLS-problem (equation (6.4))
$x \leftarrow (u, p, w)$
**while** $||J(x)p|| > tol$ AND $step - length > mintol$ **do**
  $\alpha \leftarrow ||f||_2^2/n$
  Calculate $J(x)_{reg}$ and $f(x)_{reg}$
  solve $(J(x)_{reg}^T J(x)_{reg})p = -J(x)_{reg}^T f(x)_{reg}$
  $\gamma \leftarrow 1$
  **repeat**
    (Line Search)
    $p_\gamma \leftarrow \gamma p$
    $x_+ \leftarrow x + p_\gamma$
    $x_+ \leftarrow$SimpleBounds($x_+$) (algorithm 6.1)
    $f(x_+)_{reg} \leftarrow f(x + p_\gamma)_{reg}$
    $\gamma \leftarrow \frac{1}{2}\gamma$
  **until** $||f(x_+)|| < ||f(x)||$
  $x \leftarrow x_+$
**end while**
```

6.6.2 Trust Region

Implementing a trust region method on a NURBS curve fitting problem with the Gauss-Newton method is primarily a matter of adapting the the hook-step algorithm described in section 5.3. The hook-step must now update parameters, control points and weights and also apply simple bounds to the parameters during the updating process.

Algorithm 6.3 NURBS Curve Fitting with Trust Region

```
$u \leftarrow$ Parameterise curve
$w \leftarrow (0, 0, ..., 0)$ (transformed variables $e^{w_i}$)
$p \leftarrow$ Solve LLS-problem (equation (6.4))
$x \leftarrow (u, p, w)$
while $||J(x)p|| > tol$ AND $step - length > mintol$ do
  Calculate $J(x)$ and $f(x)$
  $H \leftarrow J(x)^T J(x)$
  $g \leftarrow J(x)f(x)$
  Initialise $\delta$
  repeat
    (Trust Region iterations)
    $p \leftarrow hookstep$ (algorithm 5.2)
    $f(x_+) \leftarrow f(x + p)$ (simple bounds included)
    $\delta \leftarrow UpdateTrustRegion$ (see section 5.3.3)
  until step is acceptable
  $x \leftarrow x_+$
end while
```

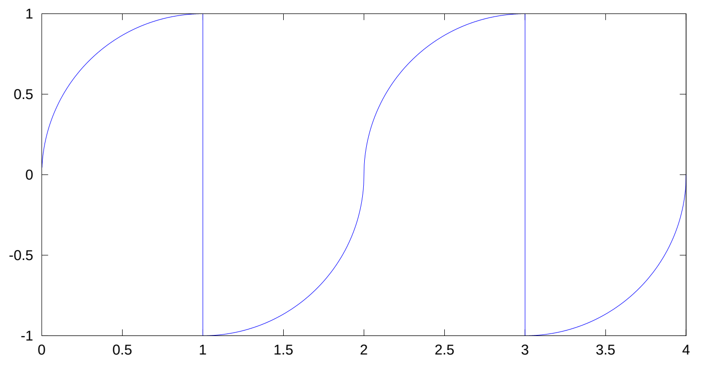


Figure 6.1: A generated test curve with circular curves and sharp corners.

6.7 Results

Comparing one NURBS fitting method to another and evaluating the speed and quality is a subjective matter. Several issues may be important; the number of iterations required, the closeness of the fit, the control point placements and how perturbation affects the final shape. In this comparison the focus is placed on the number of iterations.

For testing purposes a difficult curve was created, consisting of circular shapes combined with corners. This curve is a challenge for the fitting procedure as the weights must assume quite extreme values, the generated curve is shown in Figure 6.1. For fittings to perturbed data the same curve was used, a random number generator supplied normally distributed noise to simulate measurement error.

6.7.1 Without Perturbation

Initial fitting was performed on the generated curve from Figure 6.1 with no perturbation. Plotting the residual against the number of iterations the two methods compared well in speed; with the Trust-Region method converging slightly faster, see Figure 6.2. The initial fitting using linear least squares to find control point values is shown in Figure 6.3. The

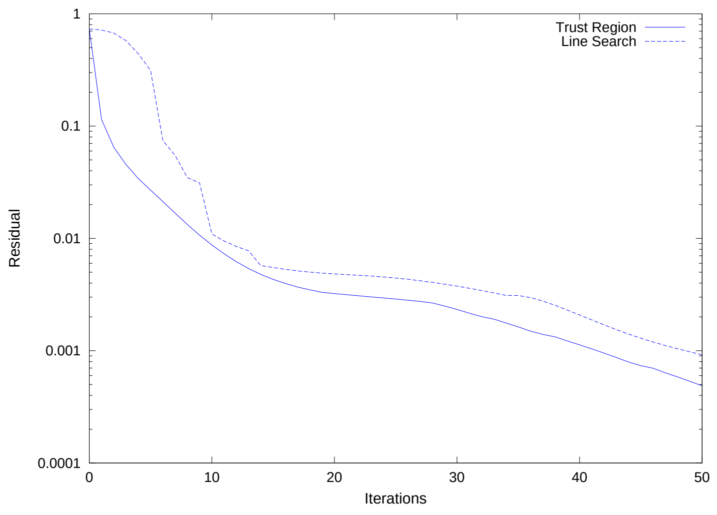


Figure 6.2: A comparison of the convergence speed of the Line Search and Trust Region methods.

Table 6.1: The residual after 20 iterations on a perturbed data set (mean of 20 fittings).

	Standard deviation of perturbation			
Residual after 20 iterations	0	0.01	0.05	0.1
Trust Region	0.0032262	0.012364	0.23926	0.87266
Line Search	0.0048322	0.011701	0.29096	1.1676

final fit is shown in Figure 6.4, with a view of a corner in Figure 6.5.

6.7.2 With Perturbation

When adding perturbation to the measured points it is difficult to determine to which degree the fit is meaningful; how small can the residual be expected to be, and how large can the perturbation be while still obtaining a meaningful degree of convergence? The table 6.1 shows residual for the two methods after 20 iterations for varying normaly distributed perturbations.

For perturbations greater than 0.05, as can be seen in the table, neither method provides a meaningful fit. If the data points are examined

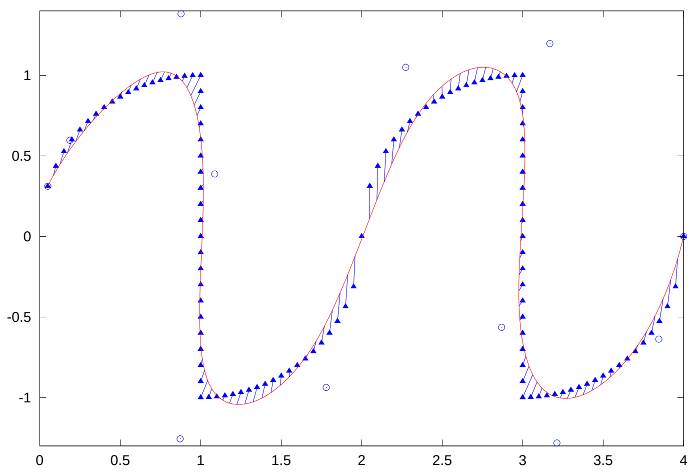


Figure 6.3: Initial linear least squares fit of a NURBS curve to the data points. The triangles represent measured points on the generated curve, the solid line is the NURBS curve and the small circles are the control points.

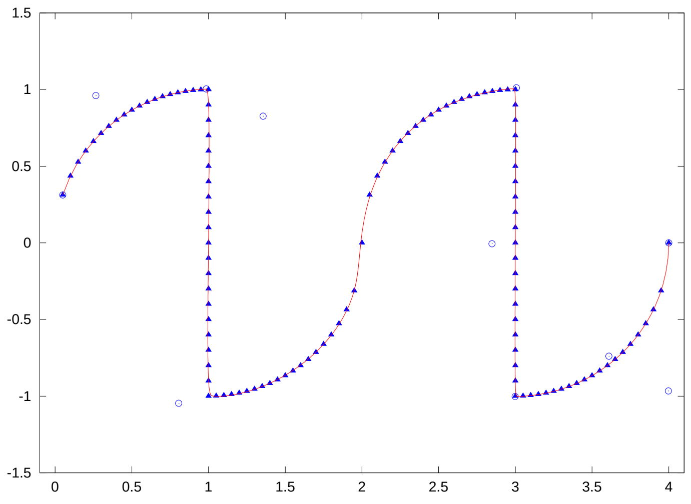


Figure 6.4: The final fit of the NURBS curve to the data points.

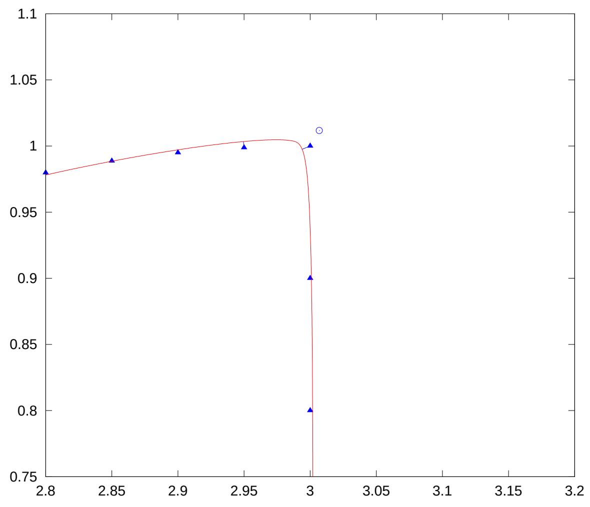


Figure 6.5: A corner in the data point with a fitted NURBS curve.

it is impossible to discern the original curve of figure 6.1. Interesting to note is that line search proved to be slightly superior with a small perturbation when compared to trust region.

7 Nonlinear Least Squares Fitting of a NURBS Surface

Fitting of a NURBS surface is not substantially different from the fitting of a NURBS curve. The added complexity arises from the additional dimensions; where the NURBS curve is defined on a parameter u the NURBS surface is defined on parameters u and v. Once again the degree is fixed at $d_u = d_v = 3$.

7.1 Initial parameters

Given a parameter mapping of the measured points $\tilde{g} = (\tilde{g}_1, \tilde{g}_2, ..., \tilde{g}_l)^T$ into vectors

$$u = \begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ u_l \end{pmatrix} \text{ and } v = \begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ v_l \end{pmatrix} \tag{7.1}$$

of three dimensional points it is again possible to solve a linear least squares problem in order to obtain initial values for the control points.

The NURBS surface

$$g(u, v, p, w) = \frac{\sum_{i=0}^{n} \sum_{j=0}^{m} N_i(u) N_j(v) w_{i,j} p_{i,j}}{\sum_{i=0}^{n} \sum_{j=0}^{m} N_i(u) N_j(v) w_{i,j}} \quad w_{i,j} > 0 \quad 0 \leq u, v \leq 1, \tag{7.2}$$

is a tensor product scheme, meaning that it is the sum of the elements of

$$A(u_i, v_i) = N(u_i) N(v_i)^T = \begin{pmatrix} N_0(u_i)N_0(v_i) & N_0(u_i)N_1(v_i) & ... & N_0(u_i)N_m(v_i) \\ N_1(u_i)N_0(v_i) & N_1(u_i)N_1(v_i) & ... & N_1(u_i)N_m(v_i) \\ \vdots & \vdots & \ddots & \vdots \\ N_n(u_i)N_0(v_i) & N_n(u_i)N_1(v_i) & ... & N_n(u_i)N_m(v_i) \end{pmatrix} \tag{7.3}$$

multiplied by their respective control points, $p_{i,j}$ and weights, $w_{i,j}$ in the numerator and only weights in the denominator. If all weights are equal the NURBS surface is a B-Spline surface

$$g(u, v) = \sum_{i=0}^{n} \sum_{j=0}^{m} N_i(u) N_j(v) p_{i,j}, \tag{7.4}$$

where the control points are included linearly. If

$$B(u,v) = \begin{pmatrix} A_{1,1..m+1}(u_1,v_1) & A_{2,1..m+1}(u_1,v_1) & ... & A_{n,1..m+1}(u_1,v_1) \\ A_{1,1..m+1}(u_2,v_2) & A_{2,1..m+1}(u_2,v_2) & ... & A_{n,1..m+1}(u_2,v_2) \\ \vdots & \vdots & \ddots & \vdots \\ A_{1,1..m+1}(u_l,v_l) & A_{2,1..m+1}(u_l,v_l) & ... & A_{n,1..m+1}(u_l.v_l) \end{pmatrix}, \tag{7.5}$$

then the corresponding vector of measured points $\tilde{g}$ initial control points p may be obtained by solving the linear least squares minimisation problem

$$\min_p ||B(u,v)p - \tilde{g}|| \, . \tag{7.6}$$

This problem is solved three times, once for each dimension.

7.2 The Derivatives of a NURBS Surface

The derivative with respect to a single u or v is similar to that for a NURBS curve. Expressed algebraically the numerator is (as a function of u)

$$n(u,v) = \sum_{i=0}^{n} \sum_{j=0}^{m} N_i(u) N_j(v) w_{i,j} p_{i,j}, \tag{7.7}$$

with a derivative with respect to u as

$$n'(u,v) = \sum_{i=0}^{n} \sum_{j=0}^{m} N_i'(u) N_j(v) w_{i,j} p_{i,j}, \tag{7.8}$$

and the denominator is

$$m(u,v) = \sum_{i=0}^{n} \sum_{j=0}^{m} N_i(u) N_j(v) w_{i,j}, \tag{7.9}$$

with a derivative of

$$m'(u,v) = \sum_{i=0}^{n} \sum_{j=0}^{m} N_i'(u) N_j(v) w_{i,j}. \tag{7.10}$$

Once again we may apply the quotient rule and follow the same path as in section 6.3.1 to find the derivate

$$\frac{\partial g(u,v,p,w)}{\partial u} = \frac{n'(u) - m'(u) g(u)}{m(u)} \quad \in \mathbb{R}^3. \tag{7.11}$$

The derivative with respect to v is derived in the same fashion.

The derivatives with respect to a given control point $p_{i,j}$ are the rational surface functions of (2.7)

$$\frac{\partial g(u,v,p,w)}{\partial p_{i,j}} = R_{i,j}(u,v,p,w) = \frac{N_i(u)N_j(v)w_{i,j}}{\sum_{k=0}^{n}\sum_{l=0}^{m} N_k(u)N_l(v)w_{k,l}} \quad \in \mathbb{R}^{3\times 3}, \tag{7.12}$$

which may be written in the same block form as (6.19). Separating the $g(u,v,p,w)$ into the numerator

$$n(w) = \sum_{i=0}^{n}\sum_{j=0}^{m} N_i(u)N_j(v)w_{i,j}p_{i,j}, \tag{7.13}$$

and denominator

$$m(w) = \sum_{i=0}^{n}\sum_{j=0}^{m} N_i(u)N_j(v)w_{i,j}, \tag{7.14}$$

the derivatives with respect to a specific $w_{i,j}$ are

$$n'(w) = N_i(u)N_j(v)p_{i,j}, \tag{7.15}$$

and

$$m'(w) = \sum_{i=0}^{n}\sum_{j=0}^{m} N_i(u)N_j(v)w_{i,j}. \tag{7.16}$$

The quotient rule may be applied to provide the complete derivative

$$\frac{\partial g(u,v,p,w)}{\partial w} = \frac{n'(w)m(w) - n(w)m'(w)}{m(w)^2} \quad \in \mathbb{R}^3. \tag{7.17}$$

7.3 Constructing the Jacobian

The Jacobian is similar to that of a two dimensional NURBS though the derivatives with respect to control point and weight matrices must be laid out in rows as and additional derivative with respect to v must be inserted. Letting the first part of the Jacobian consist of the derivatives with respect to the parameters u and v it is convenient to construct a block matrix of the form

$$B_i = \begin{pmatrix} \frac{\partial g(u_i,v_i,p,w)}{\partial u} & \frac{\partial g(u_i,v_i,p,w)}{\partial v} \end{pmatrix}. \tag{7.18}$$

This gives a Jacobian component of

$$J_{u,v} = \begin{pmatrix} B_1 & 0_{3\times 2} & ... & 0_{3\times 2} \\ 0_{3\times 2} & B_2 & ... & 0_{3\times 2} \\ \vdots & \vdots & \ddots & \vdots \\ 0_{3\times 2} & 0_{3\times 2} & ... & B_m \end{pmatrix} \in \mathbb{R}^{3l\times 2l}. \tag{7.19}$$

The second part of the Jacobian, J_p contains the derivatives with respect to the control points $p_{i,j}$, elements of the control point matrix with dimensions $n \times m$. The matrix must is constructed as in(7.5), but is otherwise identical to that in section 6.

The third part, J_w contains is with respect to the weights and is constructed in a similar fashion to that for the control points.

The complete Jacobian is then again given by

$$J = \begin{pmatrix} J_{u,v} & J_p & J_w \end{pmatrix} \tag{7.20}$$

7.4 Algorithm

The same trust region approach as in 6.6.2 was used, with modification only to the updating of the parameters.

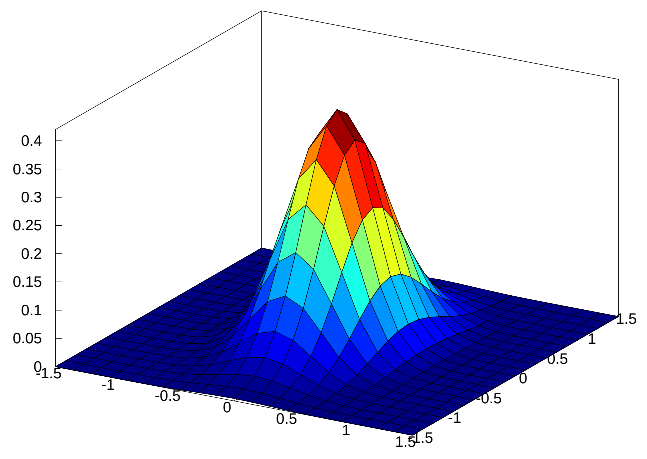


Figure 7.1: A measured Gauss-function.

7.5 Results

For testing purposes two shapes were chosen, a three dimensional Gauss-function and a pyramid. The Gauss-function is shown in Figure 7.1 and the pyramid shown in Figure 7.2 both consist of $21 \times 21 = 441$ measured data point. Each of these is represented by a vertex on the respective figures.

7.5.1 Without Perturbation

After fitting the Gauss-function for 30 iterations and the pyramid for 100 iterations with a 7×7 control point NURBS surface the fitted surfaces of Figure 7.3 and Figure 7.4 are obtained. Figure 7.5 and Figure 7.6 show the residual plotted against the number of iterations. Note the difference in the decrease for a smooth set of data, the Gauss-function, and a non-smooth data-set, the pyramid.

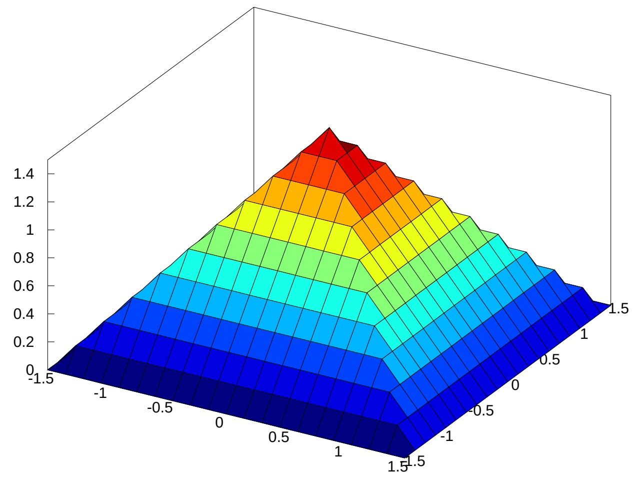


Figure 7.2: A measured pyramid.

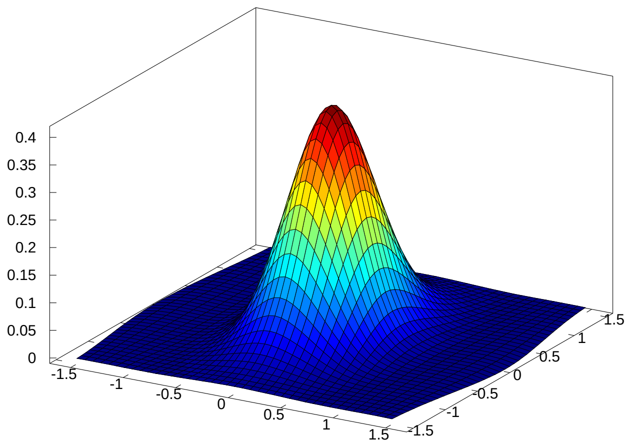


Figure 7.3: A NURBS surface fitted to the measured Gauss-function of Figure 7.1 after 30 iterations.

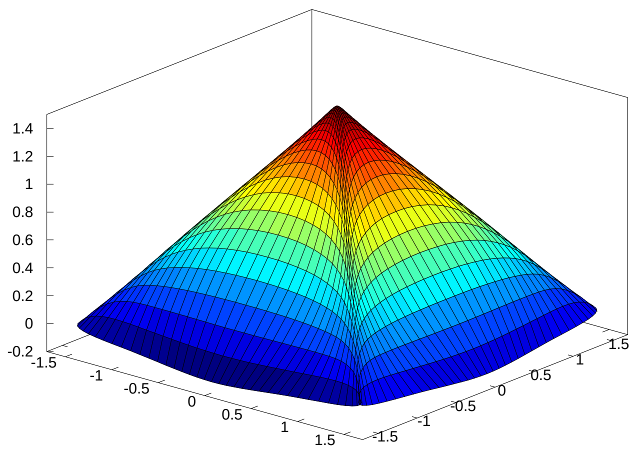


Figure 7.4: A NURBS surface fitted to the measured pyramid of Figure 7.2 after 100 iterations.

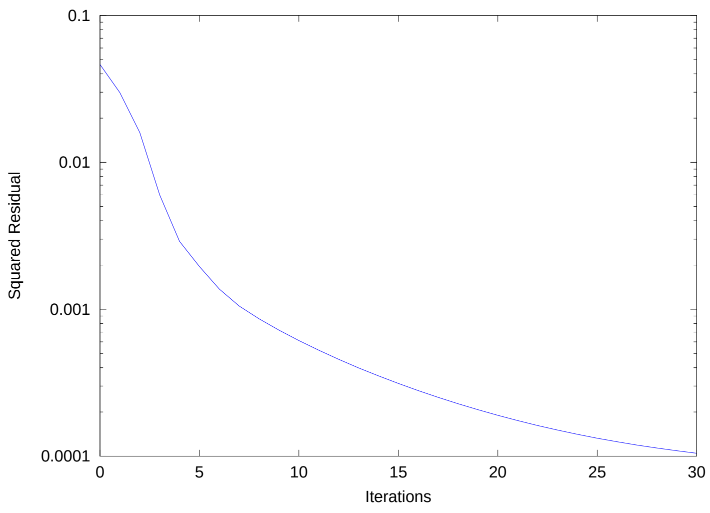


Figure 7.5: The squared residual plotted against the number of iterations for the Gauss-function.

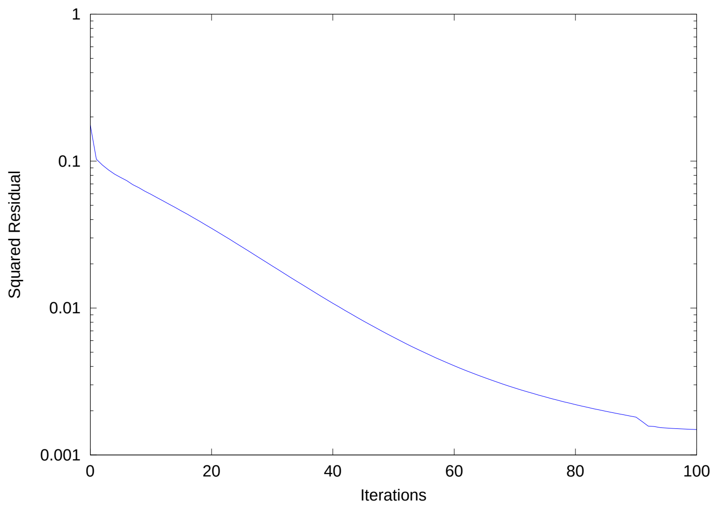


Figure 7.6: The squared residual plotted against the number of iterations for the pyramid.

Table 7.1: The residual after 30 iterations on a perturbed data set for the pyramid (mean of 10 fittings).

	Standard deviation of perturbation			
Residual after 30 iterations	0	0.01	0.05	0.1
Trust Region	0.019351	0.056592	0.93618	3.6003

7.5.2 With Perturbation

Fitting with perturbation was done on the pyramid. Good results were obtained on residuals with a standard perturbation up to 0.05. At a standard error of 0.1 the fitted NURBS Surface no longer provided a meaningful representation of the original figure. Table 7.1 shows the remaining squared residual after 30 iterations.

8 Implementation and Numerical Aspects

The implementation was done in Octave, a free version of MATLAB that has good support for sparse matrices. An experimental implementation of NURBS was also made in a functional language called Haskell in order to illustrate the recursive definition of NURBS.

8.1 Time Consumption

A surface fitting in Octave takes approximately 18 seconds per iteration on a 2GHz machine. Of this time 60% is consumed by evaluating the NURBS Surface points and the Jacobian while 30% is consumed by computations in the trust region implementation. This is mostly due to the slow nature of Octave as an interpreted language.

8.2 Efficient Computation of NURBS Basis Functions

The definition of NURBS basis functions is a recursive sum as given by equation (8.1). A naive implementation is simple to create but extremely inefficient, consider Figure 8.1. The third degree basis function $i = k$ is dependent on the zero degree basis functions $i = k \ldots k + 3$, but only one of these will be non-zero.

$$
\begin{aligned}
N_{i,0}(u) &= \begin{cases} 1 & \text{if } \mu_i \le u < \mu_{i+1} \\ 0 & \text{otherwise} \end{cases} \\
N_{i,d}(u) &= \frac{u - \mu_i}{\mu_{i+d} - \mu_i} N_{i,d-1}(u) + \frac{\mu_{i+d+1} - u}{\mu_{i+d+1} - \mu_{i+1}} N_{i+1,d-1}(u) \qquad (8.1) \\
\mu_i &\in \mu = \{\mu_0, ..., \mu_m\}.
\end{aligned}
$$

However, a point on a NURBS curve or surface is the sum of all basis functions multiplied by their respective control points. Expanding Figure 8.1 to Figure 8.2 it is possible to infer that at each degree d the number of non-zero basis functions will be $d + 1$. At $d = 3$ the non-zero basis functions for a u such that $\mu_k \le u < \mu_{k+1}$ are $N_{k-3,3}(u)$, $N_{k-2,3}(u)$, $N_{k-1,3}(u)$ and $N_{k,3}(u)$. As can be seen in the Figure the basis-functions have a great deal of interdependence and should ideally be calculated simultaneously.

In order to calculate the basis functions efficiently only the non-zero basis functions are calculated, and only once. Moreover, the basis-functions are actually dependent on those below with common elements

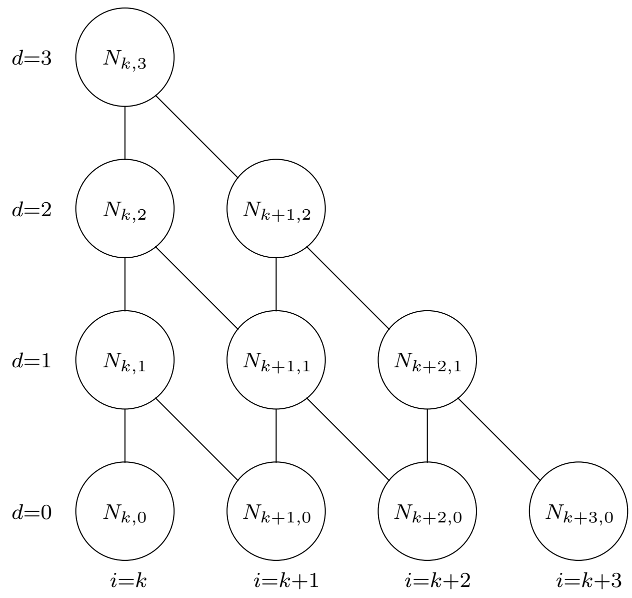


Figure 8.1: The recursive basis function i=k showing all underlying basis functions. The circles represent basis functions and the lines show a dependency.

appearing in several basis functions. Reviewing the second part of equation (8.1);

$$N_{i,d}(u) = \frac{u-\mu_i}{\mu_{i+d}-\mu_i}N_{i,d-1}(u) + \frac{\mu_{i+d+1}-u}{\mu_{i+d+1}-\mu_{i+1}}N_{i+1,d-1}(u), \qquad (8.2)$$

it can be seen that as i goes from $k-d$ to k for a given d the denominators $\mu_{i+d}-\mu_i$ and $\mu_{i+d+1}-\mu_{i+1}$ will recur. The tree given by Figure 8.3 illustrates this.

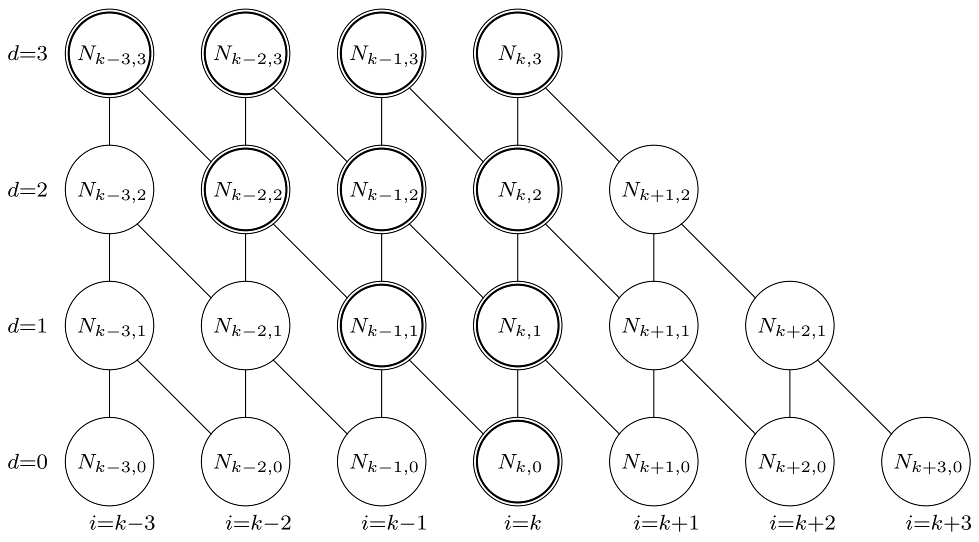
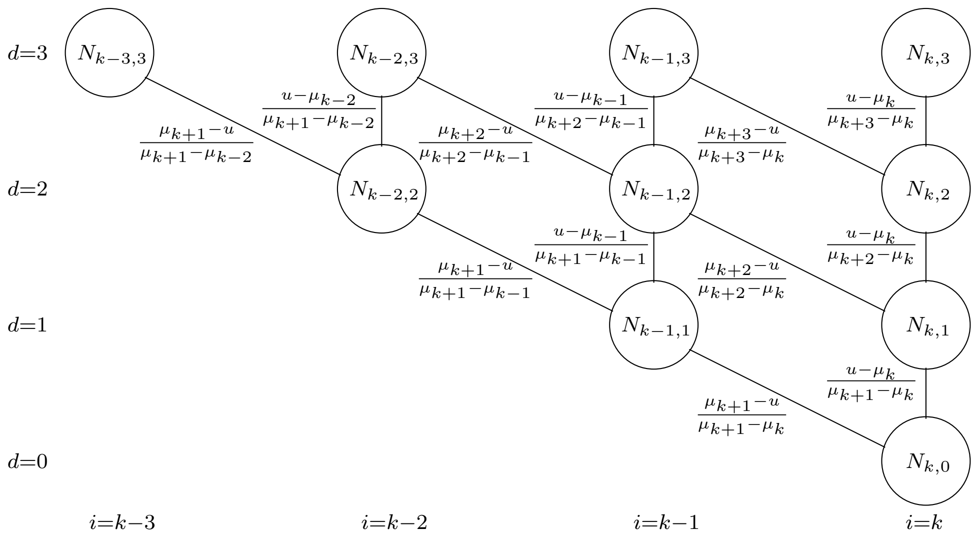


Figure 8.2: The hierarchy of basis functions for $\mu_k{\le}u{<}\mu_{k+1}$, the double circled basis functions have non-zero values.



$d{=}3$
$d{=}2$
$d{=}1$
$d{=}0$
$N_{k-3,3}$
$N_{k-2,3}$
$N_{k-1,3}$
$N_{k,3}$
$\frac{\mu_{k+1}-u}{\mu_{k+1}-\mu_{k-2}}$
$\frac{u-\mu_{k-2}}{\mu_{k+1}-\mu_{k-2}}$
$\frac{\mu_{k+2}-u}{\mu_{k+2}-\mu_{k-1}}$
$\frac{u-\mu_{k-1}}{\mu_{k+2}-\mu_{k-1}}$
$\frac{\mu_{k+3}-u}{\mu_{k+3}-\mu_k}$
$\frac{u-\mu_k}{\mu_{k+3}-\mu_k}$
$N_{k-2,2}$
$N_{k-1,2}$
$N_{k,2}$
$\frac{\mu_{k+1}-u}{\mu_{k+1}-\mu_{k-1}}$
$\frac{u-\mu_{k-1}}{\mu_{k+1}-\mu_{k-1}}$
$\frac{\mu_{k+2}-u}{\mu_{k+2}-\mu_k}$
$\frac{u-\mu_k}{\mu_{k+2}-\mu_k}$
$N_{k-1,1}$
$N_{k,1}$
$\frac{\mu_{k+1}-u}{\mu_{k+1}-\mu_k}$
$\frac{u-\mu_k}{\mu_{k+1}-\mu_k}$
$N_{k,0}$
$i{=}k{-}3$
$i{=}k{-}2$
$i{=}k{-}1$
$i{=}k$

Figure 8.3: The computational hierarchy of basis functions showing basis functions up to $d{=}3$ with all coefficients for dependencies.

```
basisFun :: Double -> [Double] -> Int -> Int -> Double
basisFun u mu d i
         | d==0 = if (u > (mu !! i) && u< (mu !! (i+1)))
                  then 1
                  else 0
         | (u == (mu !! 0) && i == 0) = 1
         | (u >= (last mu) && i == length(mu)-d-2) = 1
         | otherwise = (dZ ((u - (mu !! i))*a)
            ((mu !! (i+d)) - (mu !! i))) +
            (dZ (((mu !! (i+d+1)) - u) * b)
            ((mu !! (i+d+1)) - (mu !! (i+1))))
            where a=basisFun u mu (d-1) i
                  b=basisFun u mu (d-1) (i+1)
                  dZ x y
                     | x == 0 = 0
                     | otherwise = x / y
```

Figure 8.4: A naive NURBS basis function implementation in Haskell. The | signs are called guards and are similar to case switches in imperative languages, testing for a condition. The first one tests for the case where $d == 0$, the second and third test for end points $u == 0$ and $u >= 1$ and the third one is the case for $d > 0$ and not an end-point. The inline function dZ is a divide zero function which handles a special case of 0/0 which must be equal to 0 for NURBS basis functions. The !! are indexing operators, similar to [] in C or () in MATLAB

8.2.1 Implementation of NURBS Basis Functions in Haskell

Haskell is a compiled fully functional language with strong typing and a very clear syntax, making it very suitable for mathematics [4]. Haskell also features lazy evaluation and a limited amount of graph reduction, implying that it will calculate only what is necessary and can recognise if the same computation is performed multiple times and avoid wasting processing cycles. A full description of the Haskell language can be found at `www.haskell.org`.

A naive implementation of the NURBS basis function in Haskell is given in Figure 8.4. This will calculate the full hierarchy of dependencies for a given basis functions and is a direct implementation of Figure 8.1. In order to implement NURBS efficiently the tree of computations should look like Figure 8.3.

As multiple basis functions will be computed the return type will be a list of doubles `[Double]`. As all the non-zero basis functions are to be computed is is assumed that the given index is $i = k - d$ where k is such that $\mu_k \leq u < \mu_{k+1}$. The simplest case is then given by $d = 0$, in Haskell:

```
basisFun :: Double -> [Double] -> Int -> Int -> [Double]
basisFun u mu d i
         | d==0 = [1]
```

The next case is where $d > 0$. In this case a helper function `basisFun'` will be used:

```
         | otherwise = basisFun' u mu d (i+1) b
                    where b = basisFun u mu (d-1) (i+1)
```

This function makes use of the lower degree NURBS basis functions `b` to calculate all the non-zero basis functions at the current degree. Haskell also supports pattern matching, making it very simple to write functions that accept multiple types of input. The first part of the function `basisFun'` matches the case where only only one basis function is non-zero:

```
basisFun' :: Double -> [Double] -> Int -> Int -> [Double] -> [Double]
basisFun' u mu d i [x] =
          [x*(mu !! (i+d) - u )/denom, x*(u - mu !! i)/denom]
          where  denom = (mu !! (i+d)) - (mu !! i)
```

In this case the return value is a list with two elements (separated by a comma) corresponding to a diagonal and a vertical line in Figure 8.3 respectively. The denominator is calculated only once as it is the same for both elements (consider the basis functions $N_{k-1,1}$ and $N_{k,1}$ of Figure 8.3).

The final case is the one where a list is passed to `basisFun'`, containing multiple basis functions of a lower degree. Then there will be cases where a basis function is made up of multiple lower degree basis functions multiplied by their respective coefficients. This is given by:

```
basisFun' u mu d i (x:xs) =
          [x*( mu !! (i+d) - u)/denom, x*(u- mu !! i)/denom + head y]
          ++ tail y
          where denom = (mu !! (i+d)) - (mu !! i)
                y = basisFun' u mu d (i+1) xs
```

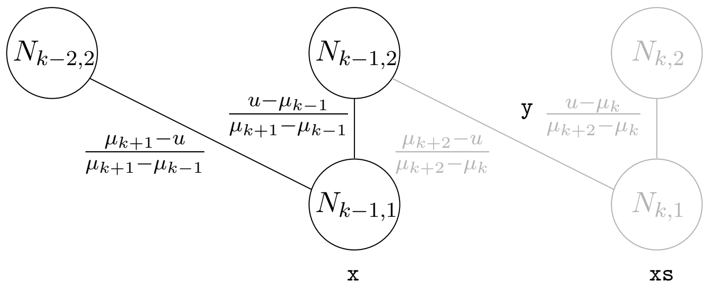

$N_{k-2,2}$
$N_{k-1,2}$
$N_{k,2}$
$\frac{u-\mu_{k-1}}{\mu_{k+1}-\mu_{k-1}}$
y
$\frac{u-\mu_k}{\mu_{k+2}-\mu_k}$
$\frac{\mu_{k+1}-u}{\mu_{k+1}-\mu_{k-1}}$
$\frac{\mu_{k+2}-u}{\mu_{k+2}-\mu_k}$
$N_{k-1,1}$
$N_{k,1}$
x
xs

Figure 8.5: First evaluation of `basisFun'` with an argument of [$N_{k-1,1}$, $N_{k,1}$]

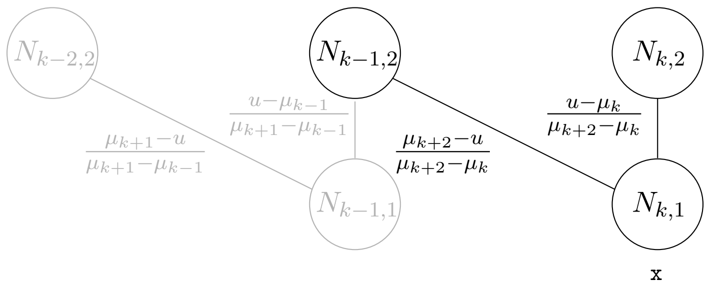

$N_{k-2,2}$
$N_{k-1,2}$
$N_{k,2}$
$\frac{u-\mu_{k-1}}{\mu_{k+1}-\mu_{k-1}}$
$\frac{u-\mu_k}{\mu_{k+2}-\mu_k}$
$\frac{\mu_{k+1}-u}{\mu_{k+1}-\mu_{k-1}}$
$\frac{\mu_{k+2}-u}{\mu_{k+2}-\mu_k}$
$N_{k-1,1}$
$N_{k,1}$
x

Figure 8.6: First evaluation of `basisFun'` with an argument of [$N_{k-1,1}$, $N_{k,1}$]

Referring to Figure 8.3, consider an input to the function `basisFun'` of a list consisting of `(x:xs)`=[$N_{k-1,1}$, $N_{k,1}$]. The element `x` will be equal to [$N_{k-1,1}$ and the basis function $N_{k-2,2}$ (`x*( mu !! (i+d) - u)/denom`) will be calculated as well as the left term of $N_{k-1,2}$ (`x*(u- mu !! i)/denom`), see Figure 8.5. The left term of $N_{k-1,2}$ is summed with the first element of `y` which is evaluated in a recursive call to `basisFun'` with the list [$N_{k,1}$] as an argument evaluating as shown in Figure 8.6 using the first defintion of `basisFun'`.

Comparing the naive approach and the efficient approach on a 2GHz machine the difference is not very substantial. Evaluating the basis functions for 100 000 values of u takes 1.309 seconds with the naive code and 0.902 seconds with efficient code, a relative speedup of factor 1.45. One of the reasons that the speedup is not greater is that Haskell already does some of the the things that we have explicitly coded. The lazy evaluation causes Haskell to never evaluate the denominator if the numerator is 0, thus the size of the tree in Figure 8.1 is somewhat reduced. Furthermore Haskell can do some graph reduction by itself recognising that many of the computations are identical; saving and reusing the result.

8.3 Numerical Aspects

The main numerical concern in the Gauss-Newton method is solving for the step p_k from the equation

$$J(u_k)^T J(u_k) p_k = -J(u_k)^T f(u_k). \tag{8.3}$$

This may be done through any number of factorisations as it is simply a linear system of equations, but recognising that this is in fact the normal equations for a linear least squares problem it is known that solutions will often not be numerically stable and that better solutions can be found [3]. The most common approach is to use QR-factorisation to solve the linear least squares

$$\min_{p_k} ||J(u_k)p_k + f(u_k)||, \tag{8.4}$$

first by factoring

$$J(u_k) = QR \tag{8.5}$$

and then solving for p_k as

$$p_k = R^{-1}(Q^T(-f(u_k))); \tag{8.6}$$

though the matrix inverse is replaced by a backward substitution [3].

In order to apply QR factorisation to the trust region method some small adjustments are required. Recalling from section 5.3 that the augmented Hessian is given as

$$H + \lambda I \tag{8.7}$$

and the Gauss-Newton step is obtained by solving

$$(J(u_k)^T J(u_k) + \lambda I) p_k = J(u_k)^T f(u_k). \tag{8.8}$$

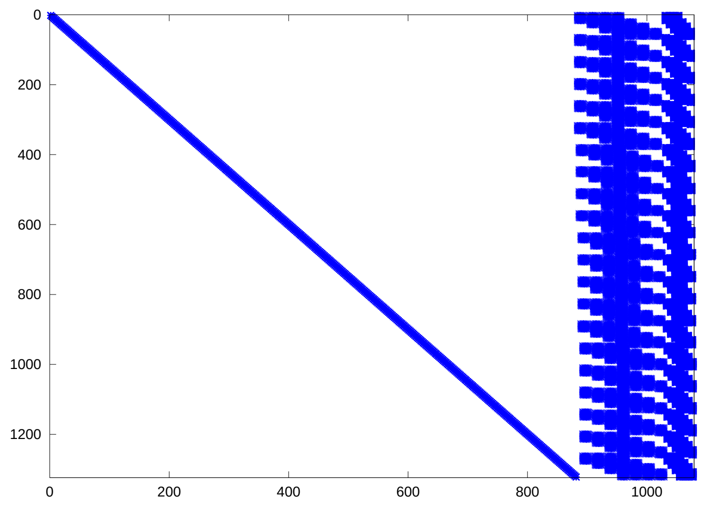


Figure 8.7: The non-zero elements of the Jacobian from a NURBS surface fitting.

In order to turn this into a normal over-determined system the follow composite matrix form is used;

$$\min_{p_k} \left\| \begin{pmatrix} J(u_k) \\ \lambda^{1/2} I \end{pmatrix} p_k + \begin{pmatrix} f(u_k) \\ 0 \end{pmatrix} \right\|, \tag{8.9}$$

this form is described in [12].

8.3.1 Sparse QR-factorisation

The Jacobian is very sparse by nature having a structure as shown in Figure 8.7. Octave makes use of CSparse [2] to calculate the sparse QR-factorisation of $J(u_k)$, substantially decreasing the number of calculations required.

9 Conclusions

Using Gauss-Newton to fully fit a NURBS Surface onto measured data points seems to be fully viable, given a good initial parameter fitting. The additional fitting of weights causes some problems to arise which can lead to oscillations, requiring some form of regularisation to reach convergence. As the weights emphasise the influence of one basis function over another movement in the parameters is necessary to provide a good fit, thus fix parameter techniques are probably not suitable.

A future implementation should probably be done in Haskell or C in order to benefit from faster language, also weak typing of MATLAB and Octave can lead to problems as double values can be converted to complex values without warning. The current implementation is unfortunately not suitable for testing on serious data sets with >2000 measured data points.

9.1 Suggestions for Future Work

Some interesting future work could focus on on-line fitting of NURBS curves, and perhaps even surfaces. Using an on-line algorithm would mean adding one data-point at a time while maintaining a fitted curve, increasing basis functions as needed and thus minimising the number of control points and maximising the ability to modify the curve shape afterwards as fewer control points need to be adjusted. Most current methods set an initial number of control points and increase as needed the number of control points where the residual is greatest, which is very unsatisfactory as unnecessary control points may be present but are difficult to identify.

Another possible area of study would be regional refinement, confining Gauss-Newton iterations to only a limited area of the parameters where the residual is large. This could speed up the algorithm substantially and could even be done in parallel across distributed machines.

References

[1] P. Axelsson. Processing of Laser Scanner Data, Algorithms and Applications. *ISPRS Journal of Photogrammetry and Remote Sensing*, 54(2-3):138–147, 1999.

[2] T.A. Davis. *Direct Methods for Sparse Linear Systems*. Society for Industrial Mathematics, 2006.

[3] J.W. Demmel. *Applied Numerical Linear Algebra*. Society for Industrial Mathematics, 1997.

[4] K. Doets and J. van Eijck. The Haskell Road to Logic, Math and Programming.

[5] H.W. Engl, M. Hanke, and A. Neubauer. *Regularization of Inverse Problems*. Kluwer Academic Publishers, 1996.

[6] M.S. Floater and K. Hormann. Surface Parameterization: A Tutorial and Survey. *Advances in Multiresolution for Geometric Modelling*, 1, 2005.

[7] M. Fradkin, M. Roux, H. Maitre, and UM Leloglu. Surface Reconstruction from Multiple Aerial Images in Dense Urban Areas. In *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, volume 2, pages 262–267, 1999.

[8] G. Golub and V. Pereyra. Separable Nonlinear Least Squares: The Variable Projection Method and its Applications. *INVERSE PROBLEMS*, 19(2):1–26, 2003.

[9] Jr. J. E. Dennis and Robert B. Schnabel. *Numerical Methods for Unconstrained Optimization and Nonlinear Equations (Classics in Applied Mathematics, 16)*. Soc for Industrial & Applied Math, 1996.

[10] Kenneth Levenberg. A method for the solution of certain non-linear problems in least squares. *Quarterly Applied Mathematics*, (2):164–168, 1944.

[11] Donald W. Marquardt. An algorithm for least squares estimation of nonlinear parameters. *Journal Society for Industrial and Applied Mathematics*, 11(2):431–441, June 1963.

[12] Jorge J. Moré. The levenberg-marquardt algorithm: Implementation and theory. In *Numerical Analysis*, volume 630/1978 of *Lecture Notes in Mathematics*, pages 105–116. Springer Berlin / Heidelberg, 1978.

[13] Stephen G. Nash and Ariela Sofer. *Linear and Nonlinear Programming*. McGraw-Hill Science/Engineering/Math, 1995.

[14] L. A. Piegl and W. Tiller. Fitting nurbs spherical patches to measured data. *Engineering with Computers*, 24(2):97–106, June 2008.

[15] Les Piegl and Wayne Tiller. *The NURBS book (2nd ed.)*. Springer-Verlag New York, Inc., New York, NY, USA, 1997.

[16] T. Schütze and H. Schwetlick. Constrained Approximation by Splines with Free Knots. *BIT Numerical Mathematics*, 37(1):105–137, 1997.

[17] T. Speer, M. Kuppe, and J. Hoschek. Global Reparametrization for Curve Approximation. *Computer Aided Geometric Design*, 15(9):869–877, 1998.

[18] W. Wang, H. Pottmann, and Y. Liu. Fitting B-Spline Curves to Point Clouds by Curvature-Based Squared Distance Minimization. *ACM Transactions on Graphics (TOG)*, 25(2):214–238, 2006.