TODO: NetPoly umbenennen TODO: Ist M invertierbar, falls nur unterschiedliche Basisfunktionen? TODO: Plots mit Definitionslücken durchrechnen

Part I
Deriving necessary equations

1 Problem statement

We want to simulate wave propagation along a certain area. There are several equations, suited for different situations, that deal with those kind of problems (TODO: Referenzen, SWE, Navier-Stokes, etc.). The main problem for computer simulation is that the entire domain has to be discretized, usually in sets of triangles that all contain several fields of information (such as mass/height, velocity, etc.). But even without discretization problems, accurately simulating wave propagation is a challenging task, because several of the involved factors are very hard to calculate. Especially in tsunami-related environments even non-local factors like bathymetry information, tidal forces and the coriolis effect have to be considered to get accurate results.

In this documentation we will focus mainly on the derivation of the shallow water equations, which serve as a starting point for calculating wave propagation, along with an analysis of the discretization problem, as well as numerical solutions for it.

2 Adaptive triangle grid

For our purposes we use an adaptive triangle grid. In areas with many variations, the grid adapts to a finer resolution, allowing for more detailed calculations, while not wasting unnecessary computation time in areas with lower variation.

Using this scheme, we can simplify a few steps in the calculation process. With a suitable library for coordinate projection, we can rotate triangles and edges into a space that's relevant for us. So instead of having to adjust how we apply the functions, based on which edges the triangles are touching on (because different edges suggest different positions in space), we just rotate the triangles into a certain space, so we only need to consider this space when applying funtions and algorithms.

In addition to rotating, we can also scale the edges of the triangles to always be in a certain shape. We'll use that to scale the catheses to length 1 and hence the hypothenuse to length $\sqrt{2}$. This will help with further calculations, because some of them will rely on the length of an edge, which can be conveniently omitted (or set to the constant $\sqrt{2}$ respectively).

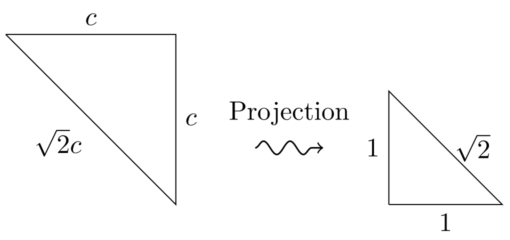


Figure 1: A sample triangle rotated into normal space and scaled down to cathetus length 1.

For further reference, we will be calling the domain of the entire triangle T (the set of all points on the triangle), and the set of edges E, enumerated with e_i, $\forall i \in 1, 2, 3$. Points on the triangle (both on the edge and inside) will be denoted by p. In the implementation we will be referring to the edges `left`, `hyp` and `right` by e_1, e_2 and e_3 respectively.

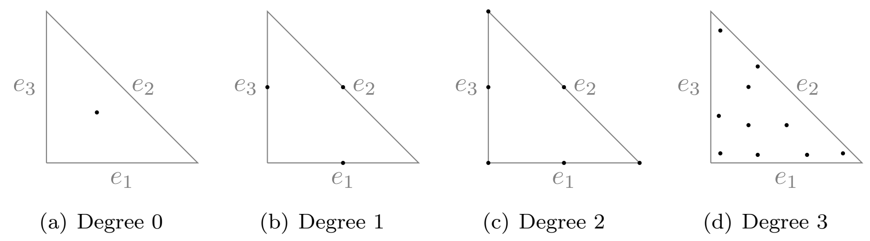


Figure 2: Triangles containing a support points for polynomial degrees 0 through 3.

The points in the triangle are support points. The number depends on the degree of the test functions, as will be discussed in section 3. As it is impossible to calculate the propagation exactly, we need to find a numerical approximation. This will be done with the discontinuous Galerkin method explained in section 4. However, for each test function used for the method we will need one point in the triangle to save the calculated information to. They can be regarded as sample points that are representative for the values within the triangle. The higher polynomial degree we operate on, the more support points we will have. While this increases accuracy, it has a negative impact on efficiency, so a proper balance is desired.

Another set of points are *Gaussian quadrature* points, which we will use later on to approximate integrals along the edges. Accordingly, they will also be located on the respective edges. They will be explained in more detail in section 6.3.

3 Choice of basis functions

For our project, we restricted ourselves to polynomial basis functions (depending on two variables x and y) of variable degree n. The name already gives away the nature of these functions, because they are supposed to be bases for a polynomial space, so that we can construct any polynomial of degree n as a linear combination of these basis functions.

A relatively simple way of constructing basis functions is to use Lagrange interpolation:

- Choose a certain set of n points p_1 through p_n (see 3.1).
- Construct function φ_i such that $\varphi_i(p_j) = \delta_{ij}\ \forall i, j \in \{1 \ldots n\}$ using Lagrange polynomials (with δ_{ij} being the Kronecker delta function).

The (minimum) number of basis functions is not arbitrary, but it depends on the degree of the polynomials. If we only employ polynomials of degree 0 (i.e. constants), we only need one basis function. Increasing the degree by 1 involves introducing variables x and y. Thus, we need three polynomials to construct every polynomial of degree 1, one to create polynomials containing x, one to create polynomials containing y and one to create other constant polynomials.

Generalizing this scheme we observe the following: if we want to use polynomials of degree n, a polynomial can contain terms $x^a y^b$ with $a + b \leq n$. A simple combinatorial arguments indicates that we need $\frac{(n+2)\cdot(n+1)}{2}$ basis functions to span the function space containing all the polynomials up to degree n.

3.1 Choice of support points

As mentioned before, we construct the basis functions with the help of some support points. However, arbitrary choosing those can lead to accuracy loss in the best case, and malconditioned matrices, which can prevent precomputing arrays in the worst.

For low degree polynomials, we choose specific distribution of the support points within the triangle:

Degree 0 We choose the support point at the center of the triangle (at $\left(\frac{1}{3}, \frac{1}{3}\right)$).

Degree 1 We choose the support points at the center of each edge.

Degree 2 We choose the support points at the triangle corners and at the center of each edge.

For higher degree, we choose the points derived by Dunavant (see [2]). The distributions of support points are illustrated in figure 2.

4 Discontinuous Galerkin

The numerical method we are using to solve the shallow water equations on our grid is the discontinuous Galerkin method (DG). It combines ideas from the finite element method and the finite volume method. The first suggests computing solutions for each element locally, which is what the DG method also does. However, it lets adjacent elements communicate relevant information with each other by numerically calculating a flux function, which determines how much of each stored quantity is passed along between elements, which is a method borrowed from finite volume schemes. We can apply this to the general continuity equation, which looks like this in its differential form:

$$\frac{\partial \mathbf{q}}{\partial t} + \nabla \cdot F(\mathbf{q}) = r \tag{1}$$

$\mathbf{q}$ The state vector, containing all relevant information for any given point in space.

F The flux function, applied to $\mathbf{q}$ it results in a vector describing the spatial transport of the quantities stored in $\mathbf{q}$.

r A source term, which is used to offset the calculated quantities. What this quantity means can differ depending on the situation.

Similar to the finite element method, we multiply the equation by a test function, then integrate over the entire domain. This can be done with more than one function as well, and we will use several, depending on the degree of the functions we use. These functions happen to be the basis functions we mentioned in section 3. Essentially, what we obtain is the following (with φ_i being the basis function):

$$\int_\Omega \frac{\partial \mathbf{q}}{\partial t} \varphi_i \, d\Omega + \int_\Omega \nabla \cdot F(\mathbf{q}) \varphi_i \, d\Omega = \int_\Omega r \varphi \, d\Omega \tag{2}$$

We will later use this to transform the shallow water equations into a different form which is easier to compute.

5 Shallow water equations

The shallow water equations can be applied to the general continuity equation by choosig appropriate variables. For the state vector $\mathbf{q}$ we use the following:

$$\mathbf{q} = \begin{pmatrix} h \\ hv_x \\ hv_y \end{pmatrix}$$

Here h is the height of the water at that point, which is proportional to the mass. v_x and v_y are the velocities in the x and y direction. Note that we will occasionally use $\mathbf{v}$, which means a two-dimensional vector consisting of $\begin{pmatrix} v_x \\ v_y \end{pmatrix}$. Next, the flux function can be defined like so:

$$F(\mathbf{q}) = \begin{pmatrix} h\mathbf{v} \\ hv_x\mathbf{v} + \frac{1}{2}gh^2 e_x \\ hv_y\mathbf{v} + \frac{1}{2}gh^2 e_y \end{pmatrix}$$

This takes into account gravitic effects for the respective x and y components. This is accomplished by e_x and e_y, which are $\begin{pmatrix} 1 \\ 0 \end{pmatrix}$ and $\begin{pmatrix} 0 \\ 1 \end{pmatrix}$ respectively, thus only adding the gravitational terms where necessary.

As for the source term, we can interpret it to be the bathymetry in our case, to offset our calculated results from the state and flux function. For simplicity, we will use $r = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$ (i.e. assuming constant even bathymetry).

Having defined those terms to suit our purpose, we can rewrite the above as follows:

$$\mathbf{q} = \begin{pmatrix} h \\ u_x \\ u_y \end{pmatrix} \qquad F(\mathbf{q}) = \begin{pmatrix} \mathbf{u} \\ \frac{u_x}{h}\mathbf{u} + \frac{1}{2}gh^2 e_x \\ \frac{u_y}{h}\mathbf{u} + \frac{1}{2}gh^2 e_y \end{pmatrix}$$

Here we replaced the velocity $\mathbf{v}$ by the impulse $\mathbf{u}$, which, similarly, consists of two dimensions, one in each direction (referred to by u_x and u_y). Occasionally we will need only the x components (or y components) from that vector, for which we will write F^x (and F^y respectively):

$$F^x(\mathbf{q}) = \begin{pmatrix} u_x \\ \frac{u_x^2}{h} + \frac{1}{2}gh^2 \\ \frac{u_x u_y}{h} \end{pmatrix} \qquad F^y(\mathbf{q}) = \begin{pmatrix} u_y \\ \frac{u_x u_y}{h} \\ \frac{u_y^2}{h} + \frac{1}{2}gh^2 \end{pmatrix}$$

At this point we apply the discontinuous Galerkin method as mentioned above. We obtain the following, called the weak form:

$$\int_T \frac{\partial \mathbf{q}}{\partial t}\varphi \, dT + \int_T \nabla \cdot F(\mathbf{u})\varphi \, dT = 0 \tag{3}$$

The operator $\nabla\cdot$ is the divergence, which computes the sum of the x derivative of the first component and the y derivative of the second component. Using the operator ∇ and applying the Gaussian divergence theorem, we can transform the above into the following set of equations:

$$\int_T \frac{\partial \mathbf{q}}{\partial t}\varphi \, dT + \int_{\partial T} F(\mathbf{q}) \cdot \mathbf{n} \, \varphi \, ds - \int_T F(\mathbf{q}) \cdot \nabla\varphi \, dT = 0 \tag{4}$$

Note at this point, that these are actually six equations, one for each component of $\mathbf{q}$, each of which contains two-dimensional vectors for the two space coordinates. The dot product here signifies a scalar product, i.e. multiply the components by row and add up the result. The last integral contains the symbol ∇. This operator takes a function and creates a vector from it containing the derivatives for x and y in the respective components.

The second integral in that equation is a border integral around our triangle. The $\mathbf{n}$ in there denotes the (outward facing) normal at that point. We write ds to denote integration over the (x, y) values along the border. Since it is a border integral of a triangle, we can split it up into three integrals (one for each edge of the triangle), which will aid in computation of the term. This also implies splitting the normal vector $\mathbf{n}$ into three vectors, one for each edge. This helps as well, because the normal vector is constant within each edge.

$$\int_{\partial T} F(\mathbf{q}) \cdot \mathbf{n} \, \varphi \, ds = \sum_{e \in E} \int_e F(\mathbf{q}) \cdot \mathbf{n}_e \, \varphi \, ds = \sum_{e \in E} \int_e F^e \varphi \, ds =: \text{Border integral} \tag{5}$$

We also combined the terms $F(\mathbf{q})$ and $\mathbf{n}$ to one term F^e. This term denotes the quantities given or received from the triangle neighboring on the edge e. The normal will not pose a problem for us, as we rotate the edges into a standardized space, so they can all be computed the same way, regardless of position (this process is described in section 2). As was mentioned in 4, we apply this method for every basis function at our disposal. Since we have one for each support point, it will run from 1 through n, named φ_i accordingly:

$$\int_T \frac{\partial \mathbf{q}}{\partial t}\varphi_i \, dT + \sum_{e\in E}\int_e F^e \varphi_i \, ds - \int_T F(\mathbf{q})\cdot\nabla\varphi_i \, dT = 0 \quad \forall i \in 1\ldots n \tag{6}$$

6 Matrix extraction

Equation 6 gives a lot of room to improve regarding efficiency. Upon closer examination, some terms stand out as being constant, or independent of the values at specific points, which means we can extract them and precompute them to save computation time while the program is running. For that, we will try to extract as much information as possible into matrices.

To do that, we first have to get rid of the various terms containing $\mathbf{q}$. Since $\mathbf{q}$ depends on the position, and that is what we integrate over, we cannot extract it from the integral. To achieve this regardless, we use the following approximation over all support points and corresponding basis functions:

$$\mathbf{q} \approx \sum_{i=1}^{n} \mathbf{q}_i \varphi_i\,(x, y) \tag{7}$$

$\mathbf{q}_i$ are the $\mathbf{q}$ values at the corresponding support points. Since those don't depend on the position, it allows us to move them out of the integral, leaving integrals over various combinations of basis functions, which are all independant of $\mathbf{q}$, and hence can be precomputed.

6.1 Mass matrix

The first integral of (6) shows exactly how this works. As only the basis functions φ_i are dependent on (x, y), we can extract $\mathbf{q}$ and obtain a constant term:

$$\begin{aligned}\int_T \frac{\partial \mathbf{q}}{\partial t}\varphi_i \, dT &\approx \int_T \frac{\partial\left(\sum_{j=1}^{n}\mathbf{q}_j\varphi_j\right)}{\partial t}\varphi_i \, dT = \\ &= \sum_{j=1}^{n}\underbrace{\int_T \varphi_i\varphi_j \, dT}_{m_{ij}}\frac{\partial \mathbf{q}_j}{\partial t}\end{aligned}$$

The elements m_{ij} form the $n \times n$ mass matrix M. That means we have a product between M and a vector containing all $\mathbf{q}$ components differentiated by t. For a shorthand notation, we can define $\tilde{\mathbf{q}} := \begin{pmatrix}\mathbf{q}_1\\ \vdots \\ \mathbf{q}_n\end{pmatrix}$. Using this, we can now translate the above into the following formula:

$$\int_T \frac{\partial \mathbf{q}}{\partial t}\varphi_i \, dT \approx M \cdot \frac{\partial \tilde{\mathbf{q}}}{\partial t}$$

6.2 Stiffness matrix

We will skip the second integral for now and move to the third. We need to treat each of the three components differently. In this section we will see the state vector array splitting up by their coordinates and giving us different results for the x and y components. This means we will be using the $F^x(\mathbf{q})$ and $F^y(\mathbf{q})$ terms defined in 5 for simpler notation.

6.2.1 First line

Using the same method as in 6.1, we obtain:

$$\begin{aligned}
\int_T F_1(\mathbf{q}) \cdot \nabla\varphi \, dT &= \int_T \begin{pmatrix} u_x \\ u_y \end{pmatrix} \cdot \nabla\varphi_i \, dT \approx \\
&= \int_T \begin{pmatrix} \sum_{j=1}^n u_{x,j}\varphi_j \\ \sum_{j=1}^n u_{y,j}\varphi_j \end{pmatrix} \cdot \begin{pmatrix} \dfrac{\partial\varphi_i}{\partial x} \\ \dfrac{\partial\varphi_i}{\partial y} \end{pmatrix} dT \\
&= \sum_{j=1}^n u_{x,j} \underbrace{\int_T \varphi_j \frac{\partial\varphi_i}{\partial x} \, dT}_{s_{ij}^x} + \sum_{j=1}^n u_{y,j} \underbrace{\int_T \varphi_j \frac{\partial\varphi_i}{\partial y} \, dT}_{s_{ij}^y}
\end{aligned}$$

Again we obtain matrices, denoted by the elements s_{ij}^x and s_{ij}^y, stored in S^x and S^y respectively (called the stiffness matrices). As before, this computation can also be regarded as a matrix multiplication.

6.2.2 Second line

An analogous approach to the second and third line gives us:

$$\begin{aligned}
\int_T F_2(\mathbf{q}) \cdot \nabla\varphi \, dT &= \int_T \begin{pmatrix} \frac{u_x^2}{h} + \frac{1}{2}gh^2 \\ \frac{u_x u_y}{h} \end{pmatrix} \cdot \nabla\varphi_i \, dT \approx && (8) \\
&= \int_T \begin{pmatrix} \sum_{j=1}^n \left(\frac{u_{x,j}^2}{h_j^2} + \frac{1}{2}gh_j^2\right)\varphi_j \\ \sum_{j=1}^n \left(\frac{u_{x,j}u_{y,j}}{h_j}\right)\varphi_j \end{pmatrix} \cdot \begin{pmatrix} \dfrac{\partial\varphi_i}{\partial x} \\ \dfrac{\partial\varphi_i}{\partial y} \end{pmatrix} dT && (9) \\
&= \sum_{j=1}^n \left(\frac{u_{x,j}^2}{h_j^2} + \frac{1}{2}gh_j^2\right) \int_T \varphi_j \frac{\partial\varphi_i}{\partial x} \, dT \\
&\quad + \sum_{j=1}^n \left(\frac{u_{x,j}u_{y,j}}{h_j}\right) \int_T \varphi_j \frac{\partial\varphi_i}{\partial y} \, dT
\end{aligned}$$

The resulting terms here, $\int_T \varphi_j \frac{\partial\varphi_i}{\partial x} \, dT$ and $\int_T \varphi_j \frac{\partial\varphi_i}{\partial y} \, dT$, are the same stiffness matrices we computed before, S^x and S^y respectively.

Explanation: Point-wise approximation The transition from (8) to (9) is used as an approximation to simplify further computation. To compute a function $f(h, u_x, u_y)$, instead of computing

$$f\left(\sum_{i=1}^n h_i\varphi_i, \sum_{i=1}^n u_{x,i}\varphi_i, \sum_{i=1}^n u_{y,i}\varphi_i\right),$$

we compute the value

$$\sum_{i=1}^n f(h_i, u_{x,i}, u_{y,i})\varphi_i.$$

The accuracy of this varies depending on f, so we will introduce an error here. However, as shown in [1], this approximation can be used in practice and give results well within the accepted error margin.

6.2.3 Stiffness matrix, third line

The third line is computationally equivalent to the previous, with the respective values switched. The terms S^x and S^y appear again:

$$\begin{aligned}
\int_T F_3(\mathbf{q}) \cdot \nabla\varphi \, dT &= \int_T \begin{pmatrix} \frac{u_x u_y}{h} \\ \frac{u_y^2}{h} + \frac{1}{2}gh^2 \end{pmatrix} \cdot \nabla\varphi_i \, dT \approx \\
&= \int_T \begin{pmatrix} \sum_{j=1}^n \left(\frac{u_{x,j}u_{y,j}}{h_j}\right)\varphi_j \\ \sum_{j=1}^n \left(\frac{u_{y,j}^2}{h_j^2} + \frac{1}{2}gh_j^2\right)\varphi_j \end{pmatrix} \cdot \begin{pmatrix} \frac{\partial\varphi_i}{\partial x} \\ \frac{\partial\varphi_i}{\partial y} \end{pmatrix} dT \\
&= \sum_{j=1}^n \left(\frac{u_{x,j}u_{y,j}}{h_j}\right) \int_T \varphi_j \frac{\partial\varphi_i}{\partial x} \, dT \\
&\quad + \sum_{j=1}^n \left(\frac{u_{y,j}^2}{h_j^2} + \frac{1}{2}gh_j^2\right) \int_T \varphi_j \frac{\partial\varphi_i}{\partial y} \, dT
\end{aligned}$$

6.3 Inspecting the border integral

Evaluating $\sum_{e\in E}\int_e F^e\varphi_i \, ds$ poses a challenge, because the term F^e depends on the edge, and hence cannot be moved out of the integral, so the same techniques employed with the previous two integrals will not work here. Instead, we employ a Gaussian quadrature and integrate numerically. To do that we substitute the inner integral as follows:

$$\int_e F^e\varphi_i \, ds \approx \sum_{k=1}^m F^e(p_k) \underbrace{\varphi_i(p_k) \cdot w_k}_{w'_{ik}}, \tag{10}$$

where m is the number of integration points and w_k denotes the Gauss weight for point p_k (consisting of x and y coordinates). This also contains the edge length (which was normalized to 1, but the hypotenuse still has length $\sqrt{2}$ in that case), which is why it has to be stored seperately for every edge. Similar to the previous examples, the values $\varphi_i(p_k) \cdot w_k$ can be precomputed as well, and added onto the term $F^e(p_k)$ when required. $F^e(p_k)$ itself is called the *numerical flux* and can be computed in different ways, which we will get to in part II.

7 Matrix listing

The matrices we obtained from section 6 are the following:

Mass matrix

$$M = [m_{ij}]_{n\times n} = \int_T \varphi_i\varphi_j \, dT \tag{11}$$

Stiffness matrices

$$S^x = [s^x_{ij}]_{n\times n} = \int_T \varphi_j \frac{\partial\varphi_i}{\partial x} \quad S^y = [s^y_{ij}]_{n\times n} = \int_T \varphi_j \frac{\partial\varphi_i}{\partial y} \tag{12}$$

Weight matrix

$$W = [w'_{ik}]_{n\times m} = \varphi_i(p_k) \cdot w_k \tag{13}$$

Using the approximation $\mathbf{q} \approx \sum_{j=1}^{n} \mathbf{q}_i \varphi_i(x,y)$, along with the previously defined $\tilde{\mathbf{q}}$ (see section 6.1), we can transform our equation 6 using the matrices defined above to look like this:

$$M \cdot \frac{\partial \tilde{\mathbf{q}}}{\partial t} + \sum_{e \in E} \tilde{F^e} W - (F^x(\tilde{\mathbf{q}}) \cdot S^x + F^y(\tilde{\mathbf{q}}) \cdot S^y) = 0 \tag{14}$$

Here $\tilde{F^e}$ means a vector of all calculated F_k^e, similarly to $\tilde{\mathbf{q}}$. This now has the advantage of being easier to handle, both mathematically as well computationally.

8 Computing the integrals

8.1 Explicit Euler

Given an equation of the form

$$\begin{aligned} \frac{\partial x(t)}{\partial t} &= f(t, x(t)), && (15) \\ x(t_0) &= x_0 && (16) \end{aligned}$$

we want to have a calculate $x(t)$ (as shown in [3]).

This is evaluated iteratively by considering a certain timestep size, τ. The smaller the timestep size, the more accurately the resulting values will approximate the exact solution. However, the timestep is left variable because it can be adjusted to be lower when confronted with highly variable state vectors (for example large observed velocities on a small grid), whereas it can be left higher during phases with little variation or similar velocities to save computation time. The new time is calculated simply by adding the variable timestep on the current time: $t_{k+1} = t_k + \tau$.

The state vector for the next timestep is based on the same vector in the current timestep. It can be regarded as taking a small timestep in the direction of the development of the state vector, where the direction is given by $\frac{\partial x(t)}{\partial t}$. We obtain the final term for the new state variable x:

$$x_{k+1} = x_k + \tau f(t_k, x_k), \quad k = 0, 1, 2, \ldots \tag{17}$$

8.2 Applying the euler method to our integrals

We can apply this method to our modified equation (14) once we solve for our desired quantity $\tilde{\mathbf{q}}$:

$$\frac{\partial \tilde{\mathbf{q}}}{\partial t} = M^{-1} \cdot \left(F^x(\tilde{\mathbf{q}}) \cdot S^x + F^y(\tilde{\mathbf{q}}) \cdot S^y - \sum_{e \in E} \tilde{F^e} W \right)$$

This can only be done if M is actually invertible, TODO: Referenz, Cockburn? which is something we can not be sure of. The matrix M depends heavily on the chosen basis functions $\varphi_1, \ldots, \varphi_n$. A diagonal matrix can be acchieved by using *orthogonal* polynomials. For polynomial degree 1, it is no problem to obtain orthogonal polynomials using the technique described in section 3.

We can see that this has the form we are looking for to apply the euler step. The result looks like this:

$$\tilde{\mathbf{q}}_{k+1} = \tilde{\mathbf{q}}_k + \tau \cdot M^{-1} \cdot \left(F^x(\tilde{\mathbf{q}}) \cdot S^x + F^y(\tilde{\mathbf{q}}) \cdot S^y - \sum_{e \in E} \tilde{F^e} W \right)$$

We have to keep in mind that equation (8.2) is actually a collection for the equations for $\mathbf{q}_1$ through $\mathbf{q}_n$, and each $\mathbf{q}_i$ contains three components (h, u_x, u_y), which means we obtain a set of $3n$ equations in total. The equation for $\mathbf{q}_i$ is then used for updating support point i.

Part II
Comparing polynomials arising from exact and approximate solution of the Lax-Friedrich-Flux

9 Setting

When analyzing wave propagation along a grid of triangles, it's essential to know how much of each quantity is passed from one triangle to another. This value is called the flux and it's determined by the so-called flux function. The quantities that interest us are the height (h, proportional to the volume and hence mass), and impulse (u, proportional to velocity q), where the impulse is a vector for every spatial dimension. The latter, however, can be remodeled using a one-dimensional approach by parametrizing the relevant coordinates.

This may sometimes be acquired by exact means, however those computations are long and complex and can result in absurd computation times. On the other hand, approximations through various numerical methods can speed this process up significantly. We want to analyze what kind of accuracy loss that implies and if it's a suitable method for realistic computation.

We have two adjacent triangles (called R and L respectively, although the direction doesn't matter), sharing a common edge E. Each triangle has its own polynomial for every component (h and u), evaluated over coordinates on the triangle. After parametrization along the edge E, this results in a polynomial $[0, 1] \to \mathbb{R}$ for each component and triangle, resulting in four polynomials. What we want to do now is take a number of sample points along the edge and evaluate the polynomials at those places.

9.1 Flux function

Since we are dealing with a one dimensional problem, we have to consider the flux function in one dimension, as well. Hence, we take the following as flux function:

$$F\left(\begin{pmatrix} h \\ u \end{pmatrix}\right) = \begin{pmatrix} u \cdot h \\ \frac{1}{2}gh^2 + u^2 \cdot h \end{pmatrix} \tag{18}$$

9.2 Lax-Friedrich-Flux

The Lax-Friedrich-Flux is defined as follows:

$$F_{LF}(p^R, p^L) = \frac{1}{2} \cdot (F(p^R) + F(p^L)) - \alpha \cdot (p^R - p^L) \tag{19}$$

In (19), p^R and p^L stand for the height and velocity along the right and the left edge, respectively. The term $F(p)$ is defined as in equation (18), and p is a two-component vector. p^R and p^L might be simple static vectors or polynomials depending on a variable (x and y).

To be precise when it comes to these polynomials, they are containing two components (for height and impulse). Note that then p is a function $[0, 1] \to \mathbb{R}$, thus we – being precise – should write $p(x)$ and $F(p(x))$. For simplicity, we alter this cumbersome notation and simply use p and $F(p)$ respectively.

9.3 Original situation: A 2d-problem

The original problem arises when two adjacent triangles share one edge. Each triangle has several support points, that are interpolated to obtain a polynomial for the corresponding triangle.

Figure 3 shows two adjacent triangles and exemplary positions of the support points. In the case shown we would use six support points, resulting in a two-variable polynomial of degree 2. We could use more support points, thereby increasing the degree of the resulting polynomial.

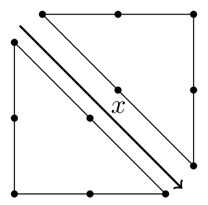


Figure 3: Two adjacent triangles with support points

Since we are interested in the situation along the adjacent edges, which is a linear curve, we can remodel the problem to one dimension, which will reduce the computational complexity. This means that we do not actually construct two-variable polynomials for the whole triangles, but instead consider just polynomials that depend on one variable $x \in [0, 1]$. The variable x simultaneously traverses both adjacent edges.

9.4 Constructing the polynomials

As stated in the introduction, we have to construct p^L and p^R, the polynomials describing the height and impulse of the right and left triangle, respectively. We construct these polynomials by interpolating some points.

We do so as well for the left as the right triangle, and for the height and the impulse. We call the points we've chosen the *support points* for the left and right triangle. For convenience, we have decided to use a naming convention for these points. Assume that we have n points for each triangle. Then we call the (components of the) points of the left triangle

$$\left(x_1, \begin{pmatrix} h_1^R \\ u_1^R \end{pmatrix}\right), \ldots, \left(x_n, \begin{pmatrix} h_n^R \\ u_n^R \end{pmatrix}\right)$$

and the points for the right triangle

$$\left(x_1, \begin{pmatrix} h_1^L \\ u_1^L \end{pmatrix}\right), \ldots, \left(x_n, \begin{pmatrix} h_n^L \\ u_n^L \end{pmatrix}\right),$$

i.e. we introduce a superscript indicating whether we are dealing with points within the left or the right triangle. As you can see, each point consists of two components, first a position on the edge ($x_i, i \in \{1 \ldots n\}$) and second a vector with the corresponding height and impulse, (h_i^R and u_i^R for the left triangle and h_i^L and u_i^L for the right, with $i \in \{1 \ldots n\}$).

At first one may think the support points should be equidistant to each other, but there are better ways to approach that problem. Eventually we'll need to integrate over the resulting function, which is numerically expensive. To compute it efficiently, we'll employ the Gaussian quadrature. This method requires its own support points to be applied, so we will choose our support points to coincide with them for easier computation. The corresponding results are summed up in section 11.

However, we also investigated the situation for an equidistant distribution of support points. You find the corresponding results in 12. A short explanation on why this setting might be useful is given there, as well.

To construct the polynomials assume we have a routine $\mathit{interpolate}(\{(x_i, y_i) \mid i \in \{1 \dots n\}\})$. This routine takes a set of points $(x_1, y_1), \dots, (x_n, y_n)$ and returns the polynomial of degree $n-1$ that goes through all the points within the set (in practice this can be e.g. a Lagrange-Interpolation-Routine, in Maple there's `CurveFitting[PolynomialInterpolation]`). Here we introduce a notational shorthand for this cumbersome construct by simply writing $\mathit{interpolate}(x, y)$ for $\mathit{interpolate}(\{(x_i, y_i) \mid i \in \{1 \dots n\}\})$.

We then generate four polynomials as follows:

- $p_h^L(x) := \mathit{interpolate}(x, h^L)$. Polynomial interpolating the height values for the left triangle.
- $p_u^L(x) := \mathit{interpolate}(x, u^L)$. Polynomial interpolating the impulse values for the left triangle.
- $p_h^R(x) := \mathit{interpolate}(x, h^R)$. Polynomial interpolating the height values for the right triangle.
- $p_u^R(x) := \mathit{interpolate}(x, u^R)$. Polynomial interpolating the impulse values for the right triangle.

We then combine two polynomials into one to obtain the following:

$$p^L(x) := \begin{pmatrix} p_h^L(x) \\ p_u^L(x) \end{pmatrix}, \quad p^R(x) := \begin{pmatrix} p_h^R(x) \\ p_u^R(x) \end{pmatrix}$$

Even if p^L and p^R are – strictly speaking – functions that map $x \in [0, 1]$ onto a vector containing the values of the height and the impulse polynomial for the respective triangle, we still call p^L and p^R polynomials.

9.5 What do we want to know?

What we eventually would like to do is to compare the following two things with each other:

Exact solution We supply the polynomials p^L and p^R (depending on x) into the Lax-Friedrich-Flux, and compute its exact value. The resulting values are again polynomials depending on x. We call these polynomials $N(x)$ (a vector which has a polynomial for the height as its first component, while the second component represents the impulse).

Approximate solution We supply the support points into the Lax-Friedrich-Flux directly, and compute the resulting values. I.e. we compute the following for $i \in \{1 \dots n\}$ (remember: n points along each adjacent edge):

$$F_{LF}\left(\begin{pmatrix} h_i^R \\ u_i^R \end{pmatrix}, \begin{pmatrix} h_i^L \\ u_i^L \end{pmatrix}\right) := \begin{pmatrix} h_i^F \\ u_i^F \end{pmatrix} = x_i^F$$

We introduce shorthand notation x_i^F for the resulting values. Each x_i^F has the same structure as the x_i, only it's evaluated by the flux function, hence the F superscript. Next we interpolate the points (i.e. we interpolate the first components to obtain one polynomial, and we interpolate the second components to obtain another polynomial). The resulting polynomials are called $\mathit{NetPoly}(x)$ (interpreted like $N(x)$ as explained above).

That is, we set

$$\mathit{NetPoly}(x) := \begin{pmatrix} \mathit{interpolate}(x, h^F) \\ \mathit{interpolate}(x, u^F) \end{pmatrix}$$

Our goal will then be to compare $N(x)$ against $NetPoly(x)$ (component-wise, of course). We compare the quality of an approximation by considering the following term:

$$I := \int_{x=0}^{1} |N(x) - NetPoly(x)| \; dx, \tag{20}$$

where the integration is to be understood component-wise.

9.6 How to evaluate I?

It is a nontrivial task to evaluate the integral in equation (20). Even if we delegate most of the work to the symbolic math program *Maple*, we experienced some pitfalls while evaluating the exact approach.

The main problem with exact integration seems to be that Maple basically has to compute the zeroes of $N(x) - NetPoly(x)$, which is, depending upon the degree of the polynomials, a time consuming task.

There are several possibilities to overcome this problem. Here are the two we took into account:

- $N(x)$ and $NetPoly(x)$ intersect when supplied the support points, i.e.

  $$N(x_i^F) = NetPoly(x_i^F), \forall i \in \{1 \dots n\}.$$

  Knowing this, we can already derive *some* zeroes (but possibly not all). If we compute

  $$\sum_{i=1}^{n-1} \int_{x=x_i}^{x_{i+1}} |N(x) - NetPoly(x)| \; dx,$$

  we get a more accurate result as an approximation for I compared to simply computing the integral over $[0,1]$.

  If we moreover divide each interval $[x_i, x_{i+1}]$ into several smaller sub-intervals, and simply integrate over them, we can reduce the error ad libitum.

  While we can tweak this method by simply evaluating more and more sub-intervals, this comes at the price of more computations.

- Another approach was to use Maple's numerical integration facilities. If we tell Maple to

  ```
  evalf(Int(abs(N(x)-NetPoly(x)),x=0..1)),
  ```

  we get a good result – if we wait long enough.

  We experimented with random polynomials and tried a few of them out, comparing the `evalf`-result to the exact integration. We experienced that the `evalf`-result was always equal to the actual result within an error of 0.001.

  However, this approach works only if *all* variables are *numbers*, i.e. if all points are instantiated to concrete values. If we want to plot a graph for a range of values for one or more of the points' components, at least one variable will not be a number, but will range within a certain interval. Thus, the `evalf`-approach has its drawbacks as well.

- Finally, there's an approach that is tought in school. We subdivide the function

  $$|N(x) - NetPoly(x)|$$

  into several stripes, and "manually compute" the area of these stripes. The more stripes we have, the better the result will be. This – of course – comes at the price of more expensive computation

We finally went for the third of the three approaches. It should also be possible to use the second or first one – if one is able to tell Maple that *first* all necessary substitutions are made, and *second* the plot is done. This is not only harder to accomplish but also results in a mess of code, but would not produce more accurate results, which is why we didn't explore this option further.

You can find some more information about the developed tool in section 10.

10 Visualizing the error term I

We are now going to plot (the two components of) the term I as described in equation (20). First, we have to do some considerations concerning plotting of such a complicated term. Afterwards, we present our tool that was developed to simplify plotting as much as possible.

10.1 Some considerations to take into account

Please recall from section 9.4, that the terms p^L and p^R depend upon the chosen support points. Moreover, $N(x)$ depends on these polynomials. Thus, the term $N(x) - NetPoly(x)$ also depends on the polynomials and – in turn – upon the support points. This means, we have $4n$ variables[1] in this term – quite a large number. This implies that I (see equation (20)) also depends on the single points $\begin{pmatrix} h_1^L \\ u_1^L \end{pmatrix}, \dots, \begin{pmatrix} h_n^L \\ u_n^L \end{pmatrix}, \begin{pmatrix} h_1^R \\ u_1^R \end{pmatrix}, \dots, \begin{pmatrix} h_n^R \\ u_n^R \end{pmatrix}$. When it comes to plotting, the sheer amount of variables requires us to later "mask out" some of them to be able to obtain useful plots.

So, we can interpret I as a function over these $4n$ values onto a two-component vector containing the integral from the h- and u-components. If we were precise, we then would write it in function notation $I\left(\begin{pmatrix} h_1^L \\ u_1^L \end{pmatrix}, \dots, \begin{pmatrix} h_n^L \\ u_n^L \end{pmatrix}, \begin{pmatrix} h_1^R \\ u_1^R \end{pmatrix}, \dots, \begin{pmatrix} h_n^R \\ u_n^R \end{pmatrix}\right)$ instead of simply I. However, most of the time we'll prefer I, since we are interested in notational simplicity.

We decided to use 3d-plots, where we have two axes representing two of the $4n$ variables. All the other variables are fixed to certain values. For example, if we fixed all points to $\begin{pmatrix} 10 \\ 0 \end{pmatrix}$ except the first point, this would mean we are interested in

$$I\left(\begin{pmatrix} h_1^L \\ u_1^L \end{pmatrix}, \begin{pmatrix} 10 \\ 0 \end{pmatrix}, \dots, \begin{pmatrix} 10 \\ 0 \end{pmatrix}\right).$$

We then could use a 3d-plot whose x-axis resembles h_1^L and whose y-axis resembles u_1^L. The z-value would then represent the value of I.

10.2 Plotting tool

As you might know, Maple supports kind of "dynamic plotting". That is, you get a window containing one slider for each variable in a mathematical expression. You can then plot this expression and adjust the parameters accordingly.

However, we quickly had to recognize that this method has some downsides. First of all, we always are interested in plotting two error values: One for the height and one for the impulse (remember: I has two components). While it is still possible to have two dynamic plots in parallel in Maple, it turned out to be very cumbersome.

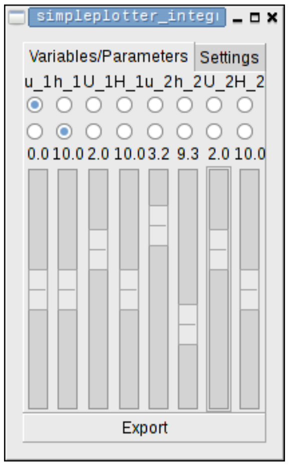
Moreover, we wanted to be able to quickly switch the coordinate axes, so that we can compare plots with each other. Last, Maple's user interface is not intended to be used with that many variables (remember that we are talking about $4n$ variables in the term I).

[1]We have n points for each of the two adjacent edges, each point consisting of a height and a impulse component, resulting in $4n$ variables.

So we decided to develop a tool that helps us visualizing our data properly. We implemented a simple program that is capable of doing the following:

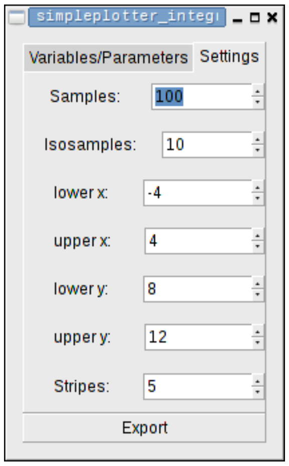
- Read two functions from a file: In our case these functions will be the "raw components" of I. To be more precise: It will be the contents of $N(x) - NetPoly(x)$ (i.e. the integrand within the integral of I). This file is generated by Maple in our case, but it can be user-written as well, of course.
- Extract variable names from the mathematical expressions extracted from the file (in our case, this will be mainly the variables u_i and h_i – of course in a suitable string representation[2]). Moreover, the variable x is treated in a special way since this is the variable that we want to integrate over to compute I.
- Offer a way to dynamically change the values of the variables (i.e. the u_i's and h_i's). This is acchieved by some sliders (see figure 4(a)).
- Numerically evaluate I for the given functions: This is acchieved using the technique described in section 9.6.
- Choose any of the variables to be used as x- resp. y-coordinates for a 2d- or 3d-plot.
- Plot I using gnuplot.
- Export plots and generate a proper tex-file to include in this report.

While our program surely will win no beauty competition, it prooved to be very useful and a real time-saver. You can see what the tool looks like in figure 4.



u_1	h_1	U_1	H_1	u_2	h_2	U_2	H_2
0.0	10.0	2.0	10.0	3.2	9.3	2.0	10.0

(a) Adjusting variables and axes



Samples:	100
Isosamples:	10
lower x:	-4
upper x:	4
lower y:	8
upper y:	12
Stripes:	5

(b) Gnuplot settings/Settings for numerical integration

Figure 4: Plotting tool window.

Subfigure 4(a) shows the page that is used to adjust the variables. As you can see, the tool has detected variables `u_1`, `h_1`, `U_1`, and so on. Moreover you can see that e.g. the parameter `u_2` is set to a value of 3.2, while `u_1` is set to 0.0.

The radio buttons determine which variables are used as axes. In subfigure 4(a), in the first option row, the column for `u_1` is selected, while in the second row, we chose `h_1`. This means that the plotter uses `u_1` as x-axis, and `h_1` as y-axis for the (3d)-plot.

[2]We decided to use a simple scheme for the "string"-variable names: u_1^L becomes `u_1`, while u_1^R becomes `U_1`. Similarly, h_3^L becomes `h_3` and h_4^R becomes `H_4`.

Subfigure 4(b) shows exemplary settings for plotting. Samples and isosamples are gnuplot-specific parameters that basically determine how fine-grained gnuplot is plotting the curve. A hundred samples and ten isosamples have shown to be a good balance between a nice looking result and acceptable waiting times. The settings for lower and upper x/y-values determine the range that is plotted by gnuplot.

Last (and possibly most important), the parameter "stripes" determines how many stripes are used to numerically evaluate the integral I. Again, we tried some values and found out that using less than five stripes is quite inaccurate. We recognized that using five or more stripes does not significantly affect the result.

10.3 How to interpret the pictures

As mentioned before, all the plots are generated using our tool. We said that our tool reads two functions from a file and plots (result of some computation involving) them. Moreover we said that the program obtains the two components of $N(x) - NetPoly(x)$. The tool generates two plots that are labelled "0. Function" and "1. Function". So, in our case "0. Function" represents the error in the h-component (height), while "1. Function" stands for the error in the u-component (impulse).

To understand how our plots work, we can take a look at figure 6. In this example, we used 2 points on each triangle and edge. That is, we have four (relevant) points in total, *two of which* we have plotted here. This is why figure 6 has *two* sub-figures.

Looking at sub-figure 6(a), its caption says that point p_1^L is varying. To be precise, its height and impulse vary along the plot axes.

The first part of this caption tells us that we are fixing *all points except* p_1^L to a specific value (in this case it was $\begin{pmatrix}10\\0\end{pmatrix}$), i.e. we are considering the term $I\left(\begin{pmatrix}10\\0\end{pmatrix}, \begin{pmatrix}h_2^R\\u_2^R\end{pmatrix}, \begin{pmatrix}10\\0\end{pmatrix}, \begin{pmatrix}10\\0\end{pmatrix}\right)$.

The orientation of the axes is displayed in figure 5.

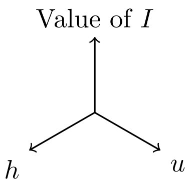


Figure 5: Orientation of the axes used throughout the 3d-plots

To plot u and h we need to know their respective ranges as well. This is noted in the caption of the whole figure. For example, in figure 6, the values for u are in $[8, 12]$ and the values for h in $[-4, 4]$.

11 Results: Distribution of support points according to Gaussian quadrature

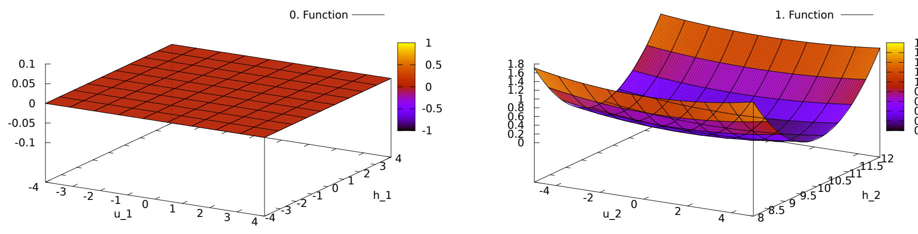
We implemented several scenarios and present the outcome of some of them using the support points one would use when doing a Gaussian quadrature. The x values of these support points are summed up in table 1 and can be read in any book on Gaussian quadrature.

11.1 Setting all support points to the same value

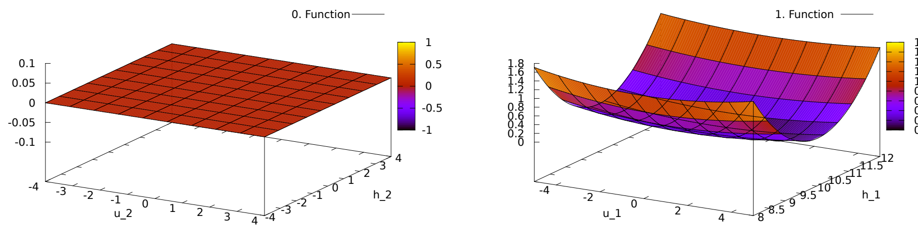
First we set all support points to one single value ensuring homogenous height and impulse. Then, we tackle each point individually and let it's values range over a certain domain.

Order	x-coordinates
2	$\frac{1}{2} - \frac{1}{6} \cdot \sqrt{3}, \frac{1}{2} + \frac{1}{6} \cdot \sqrt{3}$
3	$-\frac{1}{10} \cdot \sqrt{15} + \frac{1}{2}, 0.5, \frac{1}{10} \cdot \sqrt{15} + \frac{1}{2}$

Table 1: x-coordinates of support points for Gauss-Quadrature



(a) Height and impulse for varying point p_1



(b) Height and impulse for varying point p_2

Figure 6: Two points for each triangle. All support points have height 10 and impulse 0. h ranges from 8 to 12, u from -4 to 4. Note that the behaviour is completely symmetric. Varying the points of the other triangle yields the same plots.

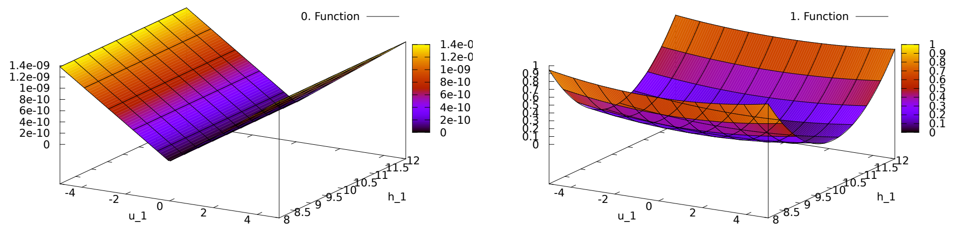
Figure 6 shows the case for two points on each adjacent edge. The sub-figures are interpreted the following way: Each subfigure shows what happens if we fix all but one specific support point. For example, subfigure 6(b) shows what happens if we fix all support points except p_2 (containing the variables h_2 and u_2) and let range h_2 from 8 to 12 and u_2 from -4 to 4. The left part of subfigure 6(b) shows the error in the h-component, while the right half depicts the error in the u component. As you can see, the error in the h-component is always zero (to be precise about 10^{-15}) in the range depicted here. From now on, we will omit plots that plot "zeroes only".

As you can see, the error for the h component ranges up to 2.5.

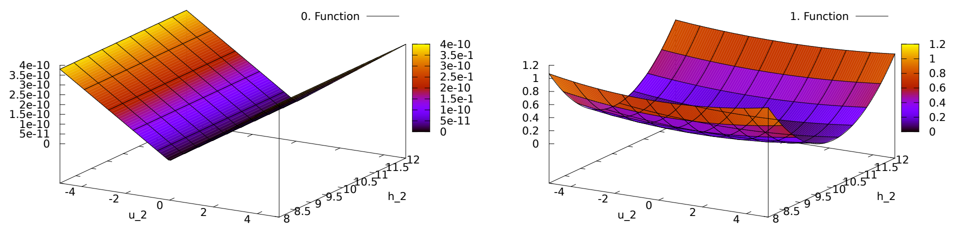
Of course it would be interesting to inspect the same thing for more support points along each edge. So we did for three support points along each edge. You can see the results in figure 7. Please note that the structure of the error plots seems to be the quite similar: The error in the height component is almost zero, while the error in the impulse component looks similar to the ones seen in figure 6.

11.2 One point variation

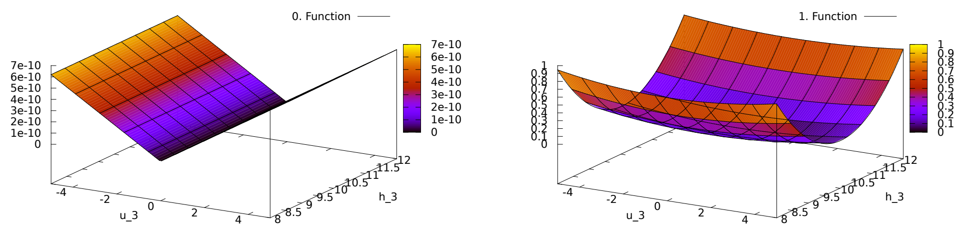
Up to now, we saw what the plots look like if all points (except the one point whose u- and h-coordinates are to be plottet, of course) have the same value. We are now goint to inspect



(a) Height and impulse for point p_1^L resp. p_1^R



(b) Height and impulse for point p_2^L resp. p_2^R



(c) Height and impulse for point p_3^L resp. p_3^R

Figure 7: Three points for each triangle. All points have height 10, impulse 0. The remaining points (P_1, P_2 and P_3) are not plotted since their plots look exactly the same.

what happens if the single points have different values.

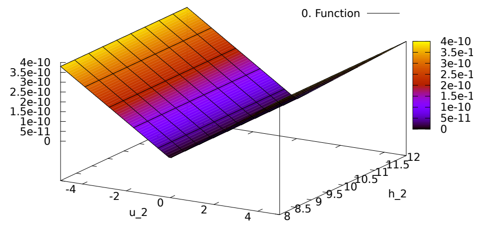
11.2.1 Decreasing the height of p_1

Now we alter the previously described experiment in one detail. We fix all points as before, but we alter *one* single point slightly. We do this in order to find out if specific points might have more impact than others.

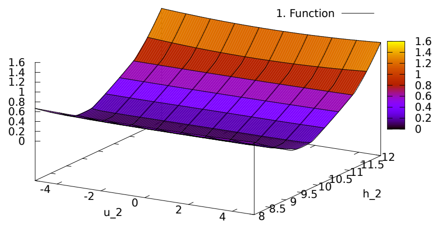
We start off by using three points per edge, setting each point to $\begin{pmatrix}10\\0\end{pmatrix}$ except p_1 wich is set to $\begin{pmatrix}9\\0\end{pmatrix}$ (i.e. we decrease the height-component of p_1).

You see the results of this experiment in figure 8.

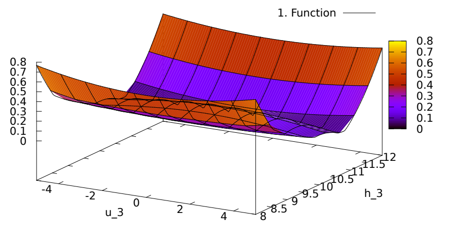
Comparing figure 7 and 8 it's worth noting that especially the plots for p_2, p_3 and – surprisingly – P_3 differ strongly. Additionally, it can be seen that the structure differs as well as the range of the error.



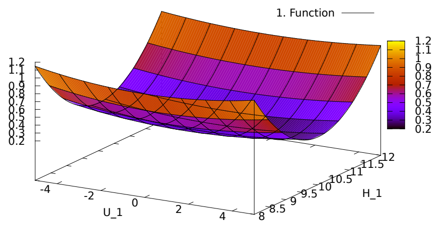
(a) Height for point p_2^L



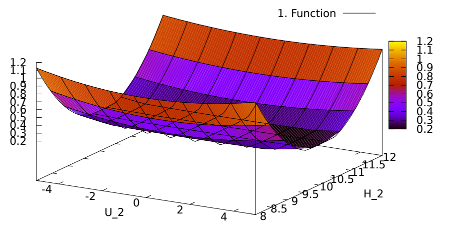
(b) Impulse for point p_2^L



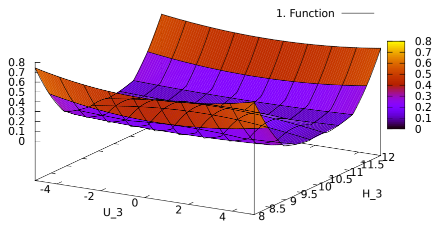
(c) Impulse for point p_3^L



(d) Impulse for point p_1^R



(e) Impulse for point p_2^R



(f) Impulse for point p_3^R

Figure 8: Three points for each triangle. All points except p_1 have height 10, impulse 0. Point p_1 is set to $(9, 0)$. Since the error in the h-component is not very different to the one depicted in subfigure 8(a) (i.e. same shape, magnitude 10^{-15}), we omitted the plots. Especially the plots for p_2^L, p_3^L and p_3^R have changed notably.

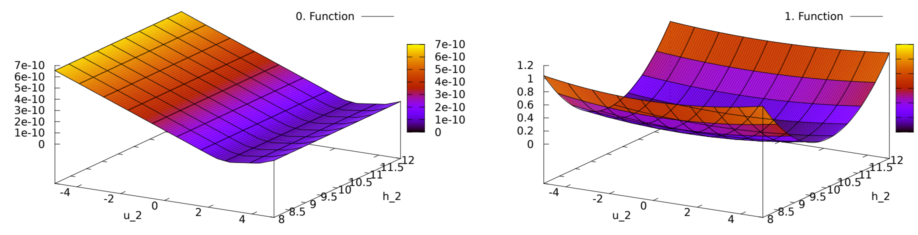
11.2.2 Decreasing the impulse of p_1

While we changed the height in subsection 11.2.1, we are now going to change the impulse. The results are shown in figure 9

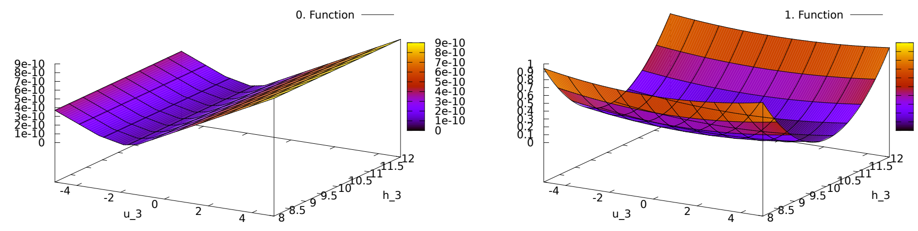
11.2.3 Decreasing the height of p_2

Now we are going to examine if it makes a difference if we decrease the height of another point, namely p_2. We show the results in figure 10. In this case, in particular the graphs for points p_1 and p_3 catch one's eye.

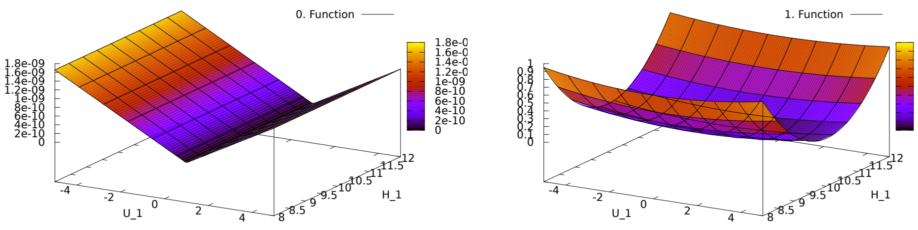
TODO: Diese Bilder genauer untersuchen, ob wir da den Ausschnitt zu früh abschneiden und es zu DefLücken kommt!



(a) Height and velocity for p_2^L resp. p_2^R



(b) Height and velocity for p_3^L res.p p_3^R



(c) Height and velocity for p_1^R

Figure 9: Three points for each triangle. All points except p_1 have height 10, impulse 0. Point p_1 is set to $(10, -1)$. Surprisingly, changing the impulse results in another error concerning the height component more than changing the height component. Subfigure shows the height and impulse errors as well for p_2^L and p_2^R since the plots (at least) look so similar that they can't be distinguished with the naked eye. Moreover we sum up the plots for points p_3^L and p_3^R in subfigure 9(b) for the same reason.

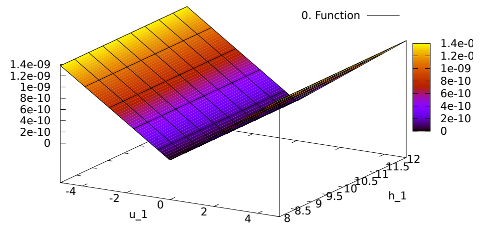
11.2.4 Decreasing the impulse of p_2

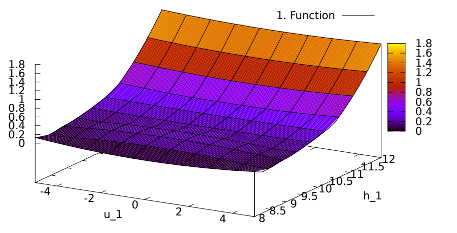
Decreasing the impulse of point p_2 results in the plots depicted in figure 11. In particular, the graphs for p_1 and p_3 show significant structural differences.

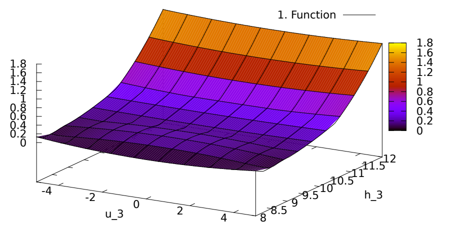
11.3 Discontinouities?

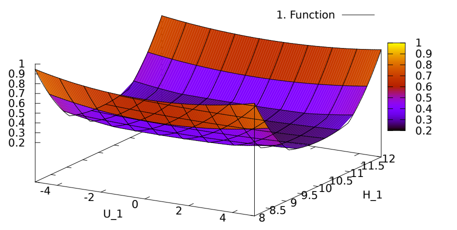
When we look at figure 6(b), we see that the error in the impulse component looks somewhat like a parabola with respect to the variable h_1. However, if we plot a wider range, we are likely to be surprised.

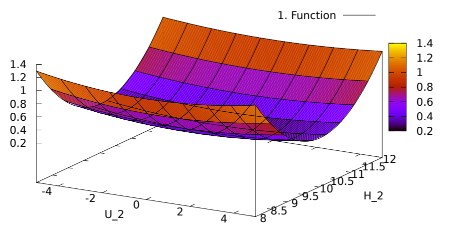
Figure 12 shows what happens here. As you can see there, the error suddenly grows to about 200 – a much larger value t han the one we have seen before! Of course it is interesting


(a) Height of point p_1^L


(b) Impulse of point p_1^L


(c) Impulse of point p_3^L


(d) Impulse of point p_1^R


(e) Impulse of point p_2^R

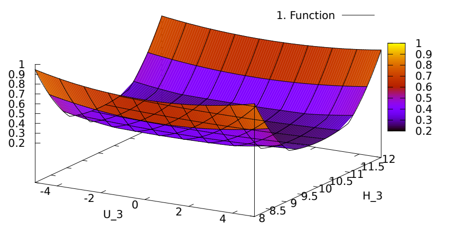

(f) Impulse of point p_3^R

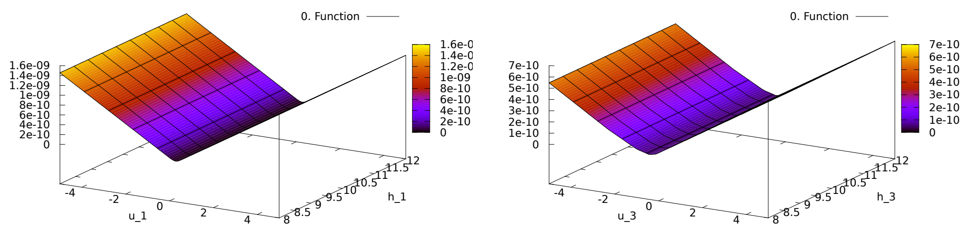
Figure 10: Three points for each triangle. All points except p_2 have height 10, impulse 0. Point p_2 is set to $(9, 0)$. We omit the plots for the h-components for other points than p_1^L since the plot has (roughly) the same shape than the one in subfigure and the error is partially even smaller than 10^{-9} for the other points.

to ask what happens here. To inspect this problem, we first realize that these discontinuities arise if h_1 takes a values of about 1.6 and 0.

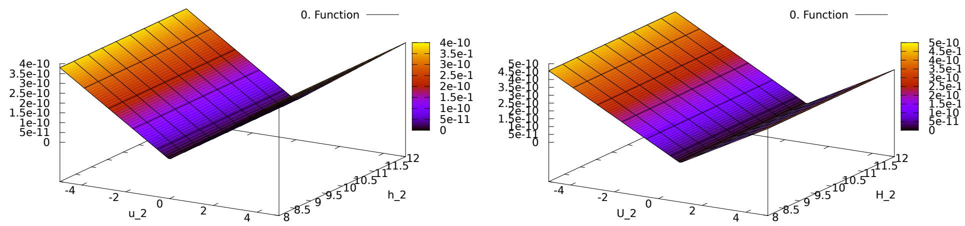
The plots in figure 12 are plotted with all points being $(10, 0)$. We are now going to inspect what the terms used in the function look like.

Without going into too much detail (since the formulae are huge and – let's say – "don't look nice"), if we expand $N(x) - NetPoly(x)$ and view its second component (describing the error in the impulse component), we recognize that there is a fraction involved that looks (approximately due to floating point rounding) as follows:

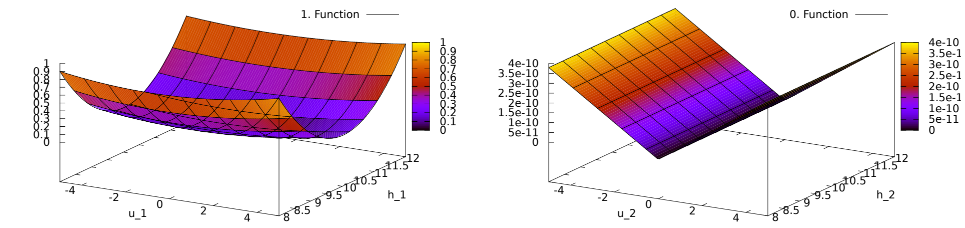
$$-0.5 \cdot \frac{((-1.7321h_1 + 1.7321) \cdot x + 1.3660u_1)^2}{(-1.7321h_1 + 1.7321)x + 1.3660h_1 - 3.6603} =: \texttt{bad}(x)$$



(a) Height for points p_1^L and p_3^L (same for p_1^R and p_3^R respectively).



(b) Height for points p_2^L and p_2^R. Slight difference in scale.



(c) Impulse for points p_1^L (same for p_3^L) and p_2^L.

Figure 11: Three points for each triangle. All points except p_2 have height 10, impulse 0. Point p_2 is set to $(10, -1)$. The resulting error graphs look the same on both triangles for each point (p_i^L and p_i^R), with only slight variation in p_2^L, due to that point's impulse being different to p_2^R's. The shape between the points is the same, only the scale differs.

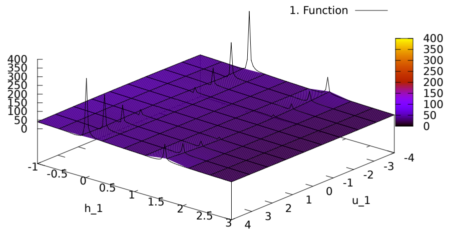


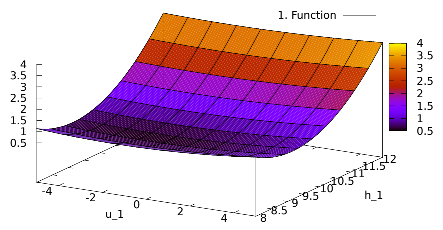
Figure 12: Plotting the impulse error depending on p_1 for a wider range. All other points are $(10, 0)$.

11.4 Random values

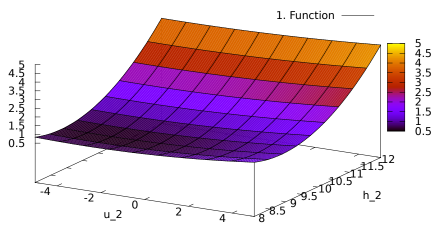
And now for something completely different. Up to now, we assigned specific values to the points and looked what happened. But it might be interesting as well what happens if we supply random values to the points.

11.4.1 Two random points

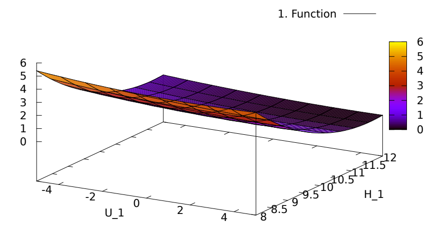
Let's start out with two random points. We see the results of random assignment of values in figure 13. As you can see, there are heavy structural differences. The maximum error concerning the height is about 6.



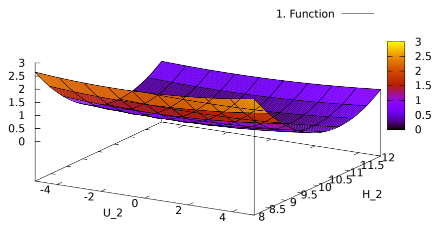
(a) Impulse of point p_1^L



(b) Impulse of point p_2^L



(c) Impulse of point p_1^R



(d) Impulse of point p_2^R

Figure 13: Two points for each triangle. Values for the points are: $u_1^L = -2.9, h_1^L = 8.8, u_1^R = -1.7, h_1^R = 10.6, u_2^L = -1.4, h_2^L = 9.1, u_2^R = 3.8, h_2^r = 11.6$. Height errors are not plotted since they are tiny (roughly 10^{-15}) in the whole area that is plotted.

11.4.2 Three random points

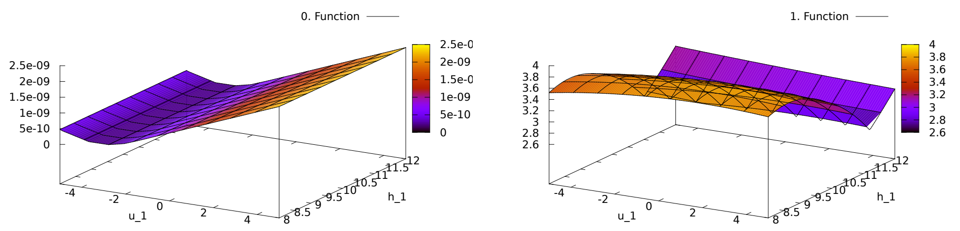
Same game for three random points is shown in figure 14. As you can see, it differs strongly from figure 7, where we set all points to the same value.

12 Equidistant distribution of support points

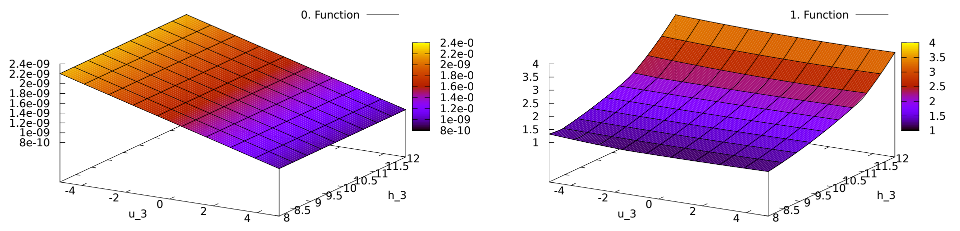
When one considers the figures depicted in section 11, it seems convenient to assume that the error grows if the values for h and u get "more exotic" (i.e. if they diverge more from the average).

One suspicion why this is the case was because of the fact that the polynomials $N(x)$ and $NetPoly(x)$ might diverge towards the borders of $[0, 1]$.

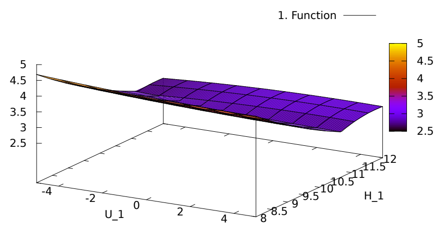
So, we chose the support points equidistant over the interval $[0, 1]$ and conducted some experiments. When we choose the support points equidistant over this interval, 0 and 1 are



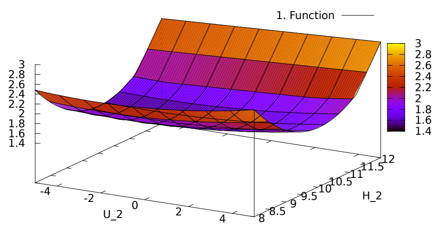
(a) Height and Impulse for point p_1^L



(b) Height and Impulse for point p_3^L



(c) Impulse for point p_1^R



(d) Impulse for point p_2^R

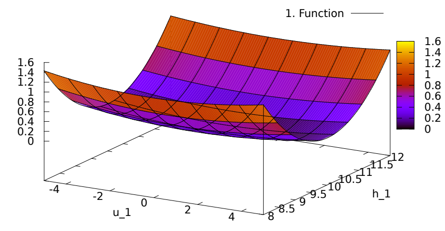
Figure 14: Selected plots when choosing the height and impulse of three points randomly. Height and impulse data for these plots: $u_1^L = 3.9, h_1^L = 11.0, u_1^R = -1.3, h_1^R = 11.3, u_2^L = 1.7, h_2^L = 8.6, u_2^R = 3.5, h_2^R = 11.9, u_3^L = -3.2, h_3^L = 11.4, u_3^R = -3.8, h_3^R = 8.7$

surely support points and we can be sure that $N(x)$ and $NetPoly(x)$ coincide at the values 0 and 1 (i.e. $N(0) = NetPoly(0)$ and $N(1) = NetPoly(1)$).

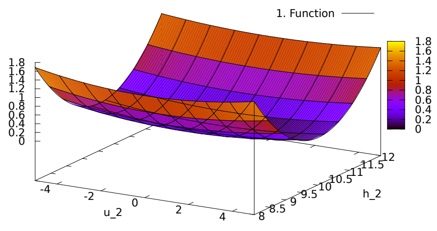
In this section, we show some results that were obtained by choosing the support points equidistant along the edge.

12.1 Three fixed support points

We started as in section 11.1 and set all support points to $\begin{pmatrix} 10 \\ 0 \end{pmatrix}$. You can see the resulting plots in figure 15. Interestingly, these plots – at least somewhat – look like the ones depicted in figure 7, where we considered the same situation for Gauss-Quadrature support points. Even the range of the error is not very different.



(a) Height and impulse of p_1^L, p_1^R, p_3^L resp. p_3^R.



(b) Height and impulse of p_2^L resp. p_2^R.

Figure 15: Three equidistant points. Coordinates of points are (10,0). Errors in height component are ommited since they are tiny (roughly 10^{-15}) for each point.

13 Conclusions from the things seen

What all plots have in common is the fact that it looks as if towards the borders the error grows. Trying to derive an exact statement of how large the error is depending upon the support point values seems a bit witless since there are too many variables that interact with each other.

References

[1] Bernardo Cockburn. Discontinuous galerkin methods for convection-dominated problems. *Lecture Notes in Computational Science and Engineering*, 9:69–224, 1999.

[2] DA Dunavant. High degree efficient symmetrical gaussian quadrature rules for the triangle. *International journal for numerical methods in engineering*, 21(6):1129–1148, 1985.

[3] Johannes Schwaiger. Adaptive discontinuous-galerkin-verfahren zum lösen der flachwassergleichungen mit verschiedenen randbedingungen. Diplomarbeit, Fakultät für Mathematik, TU München, September 2008.