Introduction to discontinuous Galerkin spectral element methods

Sven Berger

Supervisor: Michael Schlottke,
Institute of Aerodynamics,
RWTH Aachen

06. 05. 2013

Contents

1 Introduction to discontinuous Galerkin methods

In the following sections an introduction to the discontinuous Galerkin (DG) method, an important method for solving partial differential equations, is given. This method was introduced by Reed and Hill in 1973 [2], for the solution of the steady-state neutron transport equation

$$\sigma u + \nabla \cdot (au) = f \tag{1}$$

where σ and $a\,(x)$ are given and u is the unknown. They considered a 2D grid with a piecewise polynominal representation of the neutron flux and compared methods involving a continuous as well as a discontinuous flux at the element boundaries, for which the discontinuous method proved to be the superior one.
DG combines features of finite elements and finite volumes which makes it very suitable for computations of supersonic shock waves (see Figure 1.1) or acoustic waves for instance. DG is called discontinuous because two adjacent elements (an element is a finite subdomain which is used for the discretisation of the geometry), i.e. elements sharing an element boundary, each of which can possibly have a different solution on the shared boundary. In the case that the solutions are different on the element boundary the solution is discontinuous over this element boundary.
This discontinuity of the element boundary solutions allows to implement features which enable DG to resolve fine feature sizes, e.g. shock waves, for which a very fine grid is necessary. Since using a very fine grid for the whole computational domain would be inefficient, effective refinement and parallelisation methods are required to make solutions feasible. The aforementioned discontinuity between element boundaries facilitates effective means for local refinement and parallelisation. In addition, it allows very complex geometries, since elements do not need to be connected continuously, and each element order can be easily increased to yield a high order method as well as the availability of local limiters (for details see [5]).
Disadvantages are typical high-order problems, i.e. oscillation at the domain boundaries or stability issues, and less developed theory compared to finite elements and finite volumes.

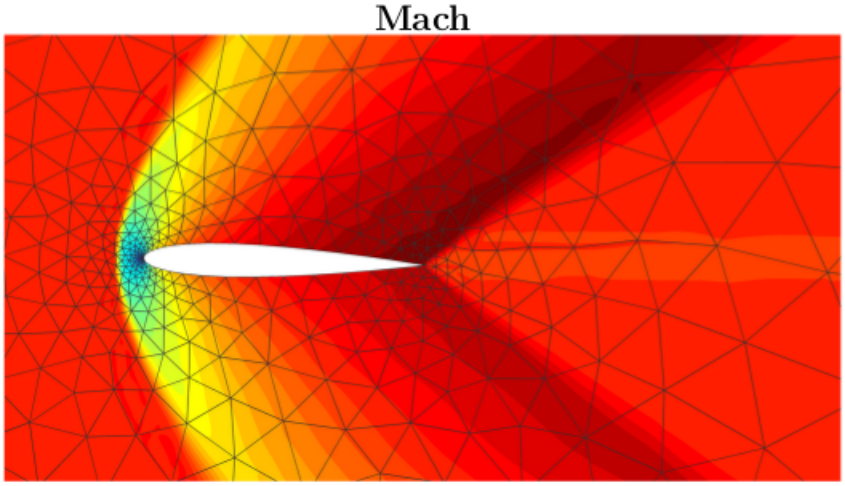

Figure 1.1: Resolved shockwave around a profile [4]

2 Theory of discontinuous Galerkin methods

The discontinuous Galerkin method is a method to convert a continuous problem (i.e. a partial differential equation (PDE)) to a discrete problem. An alternative formulation, the so-called weak formulation (see Section 2.1.1), is used to get a representation as a piecewise polynominal, in which the polynominals are discontinuous at the element boundaries.

2.1 Important numerical methods and principles used by the discontinuous Galerkin method

The discontinuous Galerkin method is based on the weak formulation, which is introduced in the following subsection. For the approximate evaluation of integrals, the Gauss-Legendre quadrature is used, for which a description is given in Subsection 2.1.2.

2.1.1 Weak formulation

The weak formulation of a differential equation yields a weak solution satisfying the differential equation in some specified sense. This is done so as to obtain solutions to problems which contain discontinuities and are thus generally not differentiable. In order to find solutions under those circumstances, differentiable test functions v, such as polynomials for example, can be used for the weak formulation. A differential equation e.g.

$$u \in C^0,\ f \in \mathbb{R},\ x \in \mathbb{R}^+ \tag{2}$$

$$\frac{du}{dx} = f \tag{3}$$

can be solved by multiplying both sides with the test function v on the Euclidean space Ω

$$\Omega = \{[a,b] \in \mathbb{R},\ ||\cdot||_2\} \tag{4}$$

using the L^2 scalar product

$$(u,v) = \int_\Omega uv\,dx. \tag{5}$$

This yields

$$\int_\Omega \frac{du}{dx} v\,dx = \int_\Omega fv\,dx. \tag{6}$$

Using integration by parts, the differential operator can be moved from u – which might not be differentiable – to the differentiable test function v, i.e.

$$[uv] - \int_\Omega u\frac{dv}{dx}\,dx = \int_\Omega fv\,dx. \tag{7}$$

Equation 7 is referred to as a weak formulation and can now be solved to obtain a weak solution of the original differential equation (Equation 3). This solution is not unique since the solution depends on the used test function and an infinite number of test functions exist. Uniqueness cannot be theoretically proven for all problems (see also [5]) but uniqueness can be guaranteed if an entropy condition is satisfied [5].

2.1.2 Gauss-Legendre quadrature

Gaussian quadrature methods can be used to approximate the value of a definite integral. One of those methods is the Gauss-Legendre quadrature which uses the orthogonal Legendre polynomials as base functions. For an n point quadrature, the obtainable results will be accurate for polynomials

up to degree $2n-1$. The Gaussian quadrature with n points assumes that the exact integral of the function f

$$\int_{-1}^{1} f(x)\, dx \tag{8}$$

can be approximated by a polynominal $g(x)$, i.e.

$$\int_{-1}^{1} f(x)\, dx \approx \int_{-1}^{1} g(x)\, dx \tag{9}$$

where the integral can be solved using n weights w_i and function values $g(x_i)$, e.g.

$$\int_{-1}^{1} g(x)\, dx \approx \sum_{i=1}^{n} w_i g(x_i) \tag{10}$$

$$\int_{-1}^{1} f(x)\, dx \approx \sum_{i=1}^{n} w_i f(x_i). \tag{11}$$

The weights w_i are used to normalise the polynomial so that $g(1) = 1$. The weights w_i are determined from the derivatives of the Legendre polynominal g. An element is discretised by the Legendre-Gauss points x_i which are the roots of the Legendre polynomial g, see also [6].

2.2 Formulation of a discontinuous Galerkin spectral element method for the 1D transport equation

In this section, a discontinuous Galerkin spectral element method for the transport equation (also known as the first-order wave equation) is developed – a hyperbolic PDE. The following example is an extended version of the one found in Kopriva [3]. The transport equation can be written as

$$\frac{\partial u}{\partial t} + \frac{\partial u}{\partial x} = 0 \tag{12}$$

$$\Leftrightarrow u_t + u_x = 0. \tag{13}$$

The goal is to find a semi-discrete form, where only the time-derivative remains. The time-derivative can then be integrated for example by using a Runge-Kutta method. Since the only other derivative that appears in Equation 13 is the spatial derivative, it is determined by integrating over x, i.e.

$$\int_{\Omega} [u_t + u_x]\, dx = 0 \tag{14}$$

which can be discretised over K elements (finite domains) $\Omega = \bigcup \Omega_k$ for $k = \{1...K\}$ with $\Omega_k = [x_{k-1},\, x_k]$ in the following fashion

$$\sum_{k=1}^{K} \int_{\Omega_k} [u_t + u_x]\, dx = 0 \tag{15}$$

Equation 15 could now be solved by using the finite volumes method, which solves the integral over each domain by using a quadrature formula so as to obtain a single solution of the time-derivative u_t for each domain. A weak solution is obtained by multiplying the equation by a test function ϕ and solving the integral. Here, Lagrange polynomials of degree N are chosen as test functions,

$$\phi(x) = \sum_{i=0}^{N} \phi_i l_i(x) \tag{16}$$

with nodes at the Legendre-Gauss points (see also [7]). Lagrange polynomials are chosen since orthogonality can be easily used as the following holds

$$l_j(x_i) = 0,\ j \neq i \tag{17}$$

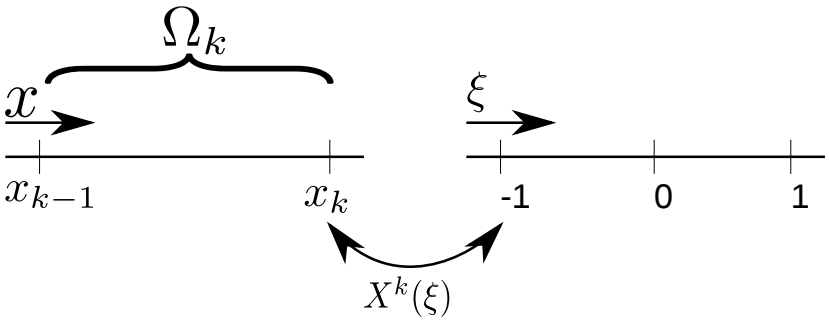

Figure 2.1: Affine mapping

$$l_j(x_j) = 1. \tag{18}$$

Including the test function ϕ_k, for the element k, into Equation 15 yields the integral equation

$$\int_{\Omega_k} [u_t + u_x]\phi_k \, dx \tag{19}$$

The x coordinates are mapped to a reference element $\xi \in [-1, 1]$ by an affine mapping (see Figure 2.1)

$$x = X^k(\xi) = x_{k-1} + \frac{\xi + 1}{2}(x_k - x_{k-1}), \tag{20}$$

since the quadrature formulation is defined on this reference element. Using the quadrature over a reference element reduces the amount of necessary calculations during each time step and simplifies the quadrature routine, too. In order to use the orthogonality properties (see also [5]) of the Lagrange polynominals, the solution and fluxes also need to be approximated as Lagrange polynominals of degree N. In this example, the approximated values of the spatial derivative u_x shall be denoted F_ξ while the time-derivative u_t shall be denoted Q_t. It follows that, with

$$Q_t^k(\xi) = \sum_{j=0}^{N} Q_{t,,j}^k l_j^k(\xi) \tag{21}$$

$$u_t^k\left(X^k(\xi)\right) \approx \frac{x_k - x_{k-1}}{2} Q_t^k(\xi) = \frac{x_k - x_{k-1}}{2} \sum_{j=0}^{N} Q_{t,j}^k l_j^k(\xi) \tag{22}$$

$$u_x^k\left(X^k(\xi)\right) \approx F_\xi^k(\xi) = \sum_{j=0}^{N} u_x^k\left(Q_{t,j}^k\right) l_j^k(\xi). \tag{23}$$

The integral over the reference element is then given by

$$\frac{x_k - x_{k-1}}{2} \int_{-1}^{1} Q_t^k \phi^k \, d\xi + \int_{-1}^{1} F_\xi^k \phi^k \, d\xi = 0. \tag{24}$$

Using integration by parts, the spatial derivative can now be moved from F^k to the test function ϕ^k

$$\frac{x_k - x_{k-1}}{2} \int_{-1}^{1} Q_t^k \phi^k \, d\xi + \left[F^k \phi\right]_{-1}^{1} - \int_{-1}^{1} F^k \phi_\xi^k \, d\xi = 0. \tag{25}$$

The integral over the reference element is determined by applying the Gauss-Legendre quadrature (see Section 2.1.2) which yields

$$Q_t^k = \dot{Q}^k \tag{26}$$

$$\frac{x_k - x_{k-1}}{2} \int_{-1}^{1} Q_t^k \phi^k \, d\xi \approx \frac{x_k - x_{k-1}}{2} \sum_{j=0}^{N} \dot{Q}_j^k w_j \phi_j^k l_j^k \tag{27}$$

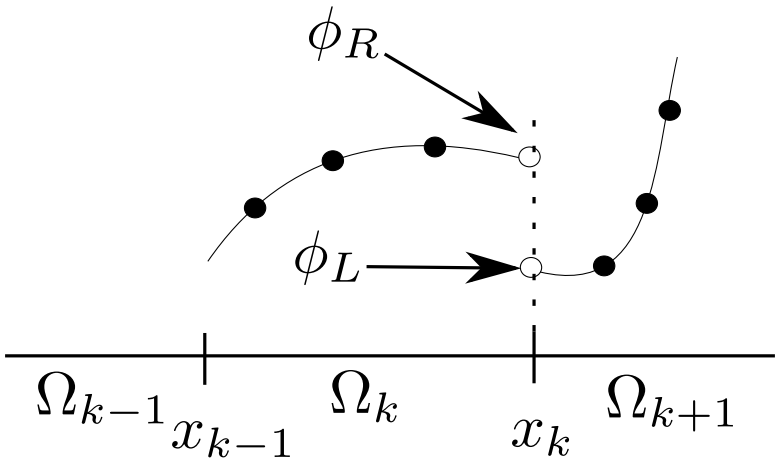

Figure 2.2: Boundary values

since the ϕ_j are linearly independent, hence can be omitted

$$\Leftrightarrow \frac{x_k - x_{k-1}}{2} \int_{-1}^{1} Q_t^k \phi^k \, d\xi \approx \frac{x_k - x_{k-1}}{2} \sum_{j=0}^{N} \dot{Q}_j^k w_j l_j^k \tag{28}$$

dividing by w_j then yields, for $j = 0...N$

$$\frac{x_k - x_{k-1}}{2} \int_{-1}^{1} Q_t^k \phi^k \, d\xi \approx \frac{x_k - x_{k-1}}{2} \dot{Q}_j^k \tag{29}$$

$$[F\phi]_{-1}^{1} = \frac{[F(1) - F(-1)]\, l_j^k}{w_j} \tag{30}$$

$$\int_{-1}^{1} F^k \phi_\xi \, d\xi \approx \sum_{n=0}^{N} F_n^k \frac{w_n l_j^{k'}(\xi_n)}{w_j} \tag{31}$$

which gives

$$\Rightarrow \frac{x_k - x_{k-1}}{2} \dot{Q}_j^k + \underbrace{\frac{[F(1) - F(-1)]\, l_j^k}{w_j}}_{\text{flux at the element boundary}} - \sum_{n=0}^{N} F_n^k \frac{w_j l_j^{k'}(\xi_n)}{w_n} = 0, \; j = 0...N. \tag{32}$$

At element boundaries, the flux from the left and right element may differ (since continuity is not enforced). In order to get a conservative scheme, a numerical flux $F^*$is used. The numerical flux F^* can be determined by using a Riemann solver or approximate solutions, e.g. the local Lax-Friedrich flux (see also [5]). This numerical flux depends on the two values ϕ_R and ϕ_L at each of the element boundaries (see Figure 2.2). For this simple case a Riemann solver/solution (see Equations 33 and 34) can easily be determined and is hence used here. Therefore, the flux F at the element boundaries is replaced by the numerical flux F^*. Assuming the flux to be incident from the left (and thus transportation to the right) and a unity transport velocity, the solution to this Riemann problem is then given by

$$F^{k*}(-1) = F_L^k = \phi_R^{k-1}, \tag{33}$$

$$F^{k*}(1) = F_R^k = \phi_R^k. \tag{34}$$

Finally, the semi-discrete discontinuious Galerkin method is given by, for $j = 0...N$

$$\frac{x_k - x_{k-1}}{2} \dot{Q}_j^k + \left[\frac{F^{k*} l_j^k}{w_j} \right]_{-1}^{1} - \sum_{n=0}^{N} F_n^k \frac{w_n l_j^{k'}(\xi_n)}{w_j} = 0 \tag{35}$$

and introducing the solution to the Riemann problem (see Equations 33 and 34) for the numerical flux F^{k*}, for $j = 0...N$

$$\Rightarrow \frac{x_k - x_{k-1}}{2}\dot{Q}_j^k + \frac{F_L^k l_j^k(-1)}{w_j} + \frac{F_R^k l_j^k(1)}{w_j} - \sum_{n=0}^{N} F_n^k \frac{w_n l_j^{k'}(\xi_n)}{w_j} = 0. \tag{36}$$

For each time step and element k, the time-derivative $\dot{Q}^k$ can be calculated, for $j = 0...N$

$$\dot{Q}_j^k = \frac{2}{x_k - x_{k-1}} \left(- \left[\frac{F^{k*} l_j^k}{w_j} \right]_{-1}^{1} + \sum_{n=0}^{N} F_n^k \frac{w_n l_j^{k'}(\xi_n)}{w_j} \right), \tag{37}$$

$$\dot{Q}_j^k = \frac{2}{x_k - x_{k-1}} \left(\frac{-F_L^k l_j^k(-1)}{w_j} - \frac{F_R^k l_j^k(1)}{w_j} + \sum_{n=0}^{N} F_n^k \frac{w_n l_j^{k'}(\xi_n)}{w_j} \right). \tag{38}$$

3 Implementation and results of a one-dimensional discontinous Galerkin spectral element method

In the following sections, a short overview of the general implementation of a one-dimensional DG method as well as results obtained for a one-dimensional transport equation as described in Section 2.2 are provided.

3.1 Overview of a one-dimensional discontinous Galerkin implementation

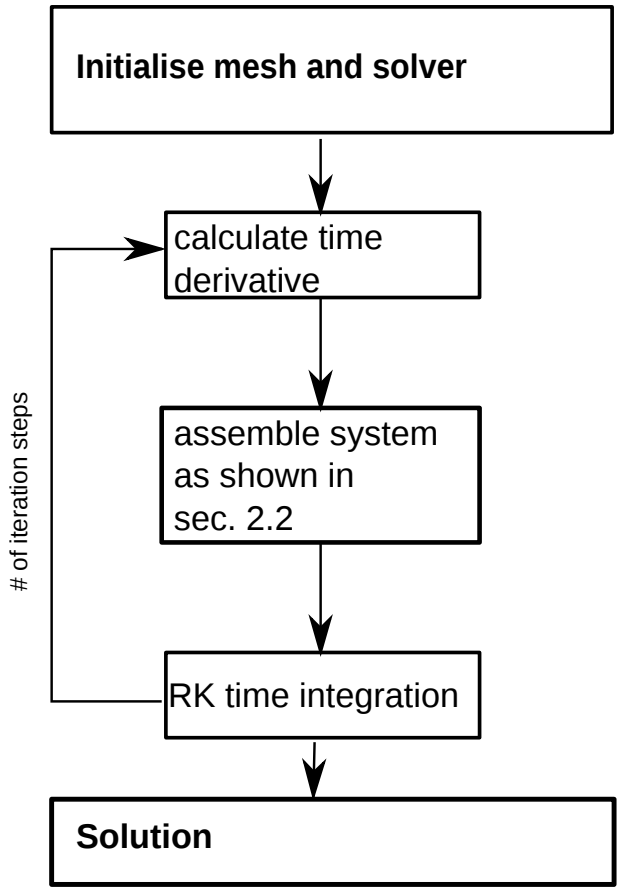

Figure 3.1: main loop of a DG solver

The main loop of a DG solver consists of two main steps; first calculating the time derivative and then solving the time derivative by an Runge-Kutta method (see Figure 3.1). The time derivative is calculated for each time step (see Section 2.2), which is then integrated using a Runge-Kutta (RK) method. The RK method used in this work is a low-storage variant that was introduced by Williamson in 1980 [1]. This loop is repeated with a certain time step size which can be statically set or changed during calculation. The time step size determines the number of steps necessary for the solver to reach the desired solution time.
The calculation of the time-derivative assembles the DG system which was derived in Section 2.2 for a one-dimensional transport equation. This part is the essence of the DG method and shall be explained in more detail.

To calculate the time derivative the following steps are necessary at each time step:

1. On each element the inter-element boundary values are calculated by interpolating the solution values from the solution of the elements inner points. This needs to be done since the solution at the element's inner points can change each time step.

2. The values at the domain boundaries are set to the appropiate boundary conditions. The functions providing these values depend on the defined boundary condition types, e.g. von-Neumann condition or an exact solution. For the simple problem studied an exact solution is provided.

3. The numerical flux through the inter-element boundaries can now be estimated or an exact solution can be provided in the case that a Riemann solution is available. In order to determine the numerical flux, in either case the values of steps 1 and 2 have to be used.

4. Finally, the intermediate results of step 1 to 3 are assembled to form the system that needs to be solved.

3.2 Results for a one-dimensional transport equation

In this section, results are shown for the one-dimensional transport equation that has been presented in Section 2.2 solved by the described DG method (for the implementation, see 4). In Figures 3.2 – 3.4, the polynomial degree N per element k was increased while the number of elements K stayed constant at 5 elements. The points were initialised with the exact solution at time 0, the time step Δt was chosen as 0.0025, the finish time was chosen as 2.5, and the domain boundary conditions are set to the exact solution (see Equation 39).

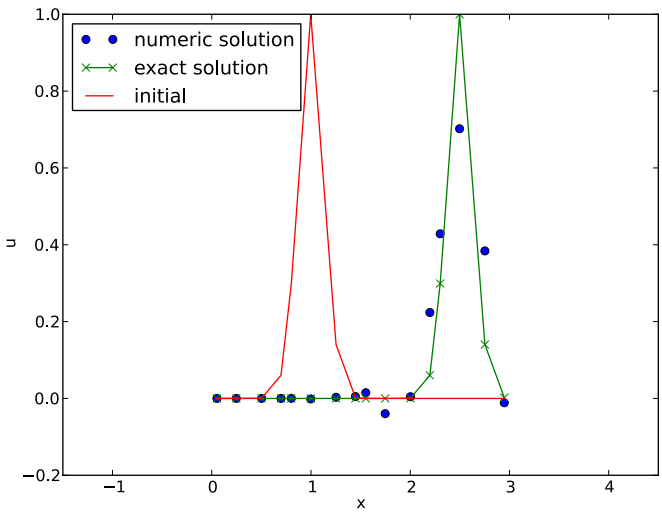

Figure 3.2: Results for $N = 3$, $K = 5$

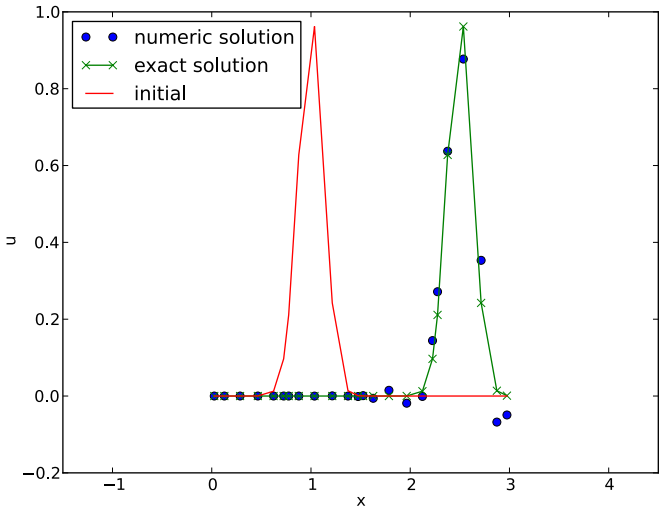

Figure 3.3: Results for $N = 5$, $K = 5$

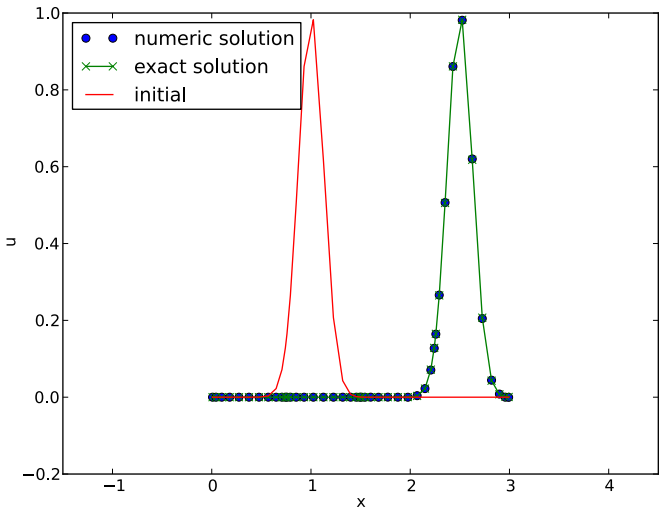

Figure 3.4: Results for $N = 10$, $K = 5$

Figures 3.2–3.4 show the behaviour of the DG method for increasing the polynomial degree N for the transport equation with the following solution

$$u(x) = 2^{-\frac{(x-t-1.0)^2}{0.0225}}. \tag{39}$$

The first and second Figure show insufficient agreement with the exact solution. The maximum error is ~ 0.2 for a polynomial degree of three, and for a polynomial degree of five the maximum error is ~ 0.1. In contrast, the third figure shows no discernible difference between the exact and numerical solution, which can also be seen from the maximum error given by ~ 0.002 (see also Figure 3.5).
The data shown in Figure 3.5 was produced from the same setup as used in the Figures 3.2–3.4, while the maximal error was calculated from the exact solution u and the approximate solution $\tilde{u}$ as

$$\epsilon_{\max} = \max |u_{k,n} - \tilde{u}_{k,n}| \tag{40}$$

with k being the element index and n the polynomial degree. For $K = 2$ the maximal error first increases before it goes down (see Figure 3.5), which is caused by the low number of elements that for

$N = 1, 2$ happen to lie closer to the exact solution.

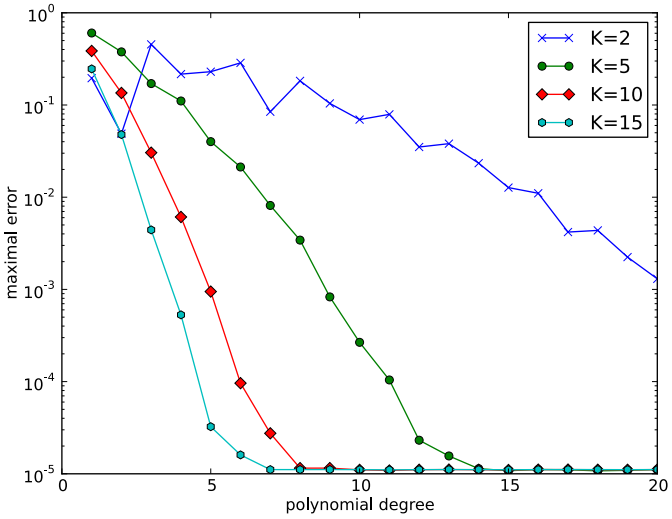

Figure 3.5: Absolute error versus each combination of polynominal order and number of elements

Figure 3.6 shows the time until the simulation was finished for the same setup used as in the Figures 3.2–3.4. From this Figure it can be inferred that using a polynomial degree tends to be computationally more efficient than using a larger number of elements. This can be seen in Figure 3.6. For example, it takes less than a second to reach an error of 10^{-4} when using $N = 10$ and $K = 5$, whereas it takes more then a second when using $N = 5$ and $K = 10$. This is due to the fact that the error decreases exponentially with the order which can be seen in Figure 3.5.

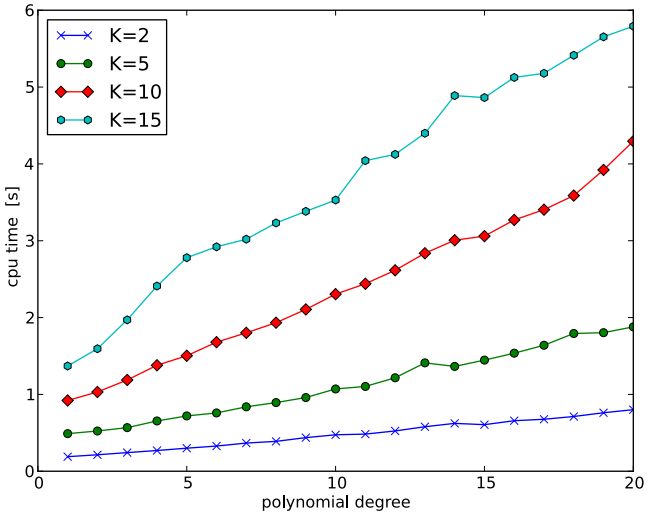

Figure 3.6: Required CPU time required for each shown combination of polynominal order and number of elements until the solution time of 2.5 is reached with a time step of 0.0025

Figure 3.7 shows the time required to reach a maximal error below 10^{-2}. The time required to solve the transport equation increases linearly by the polynomial degree until for a high number of elements

and polynomial degree for which the implemented methods stability decreases.

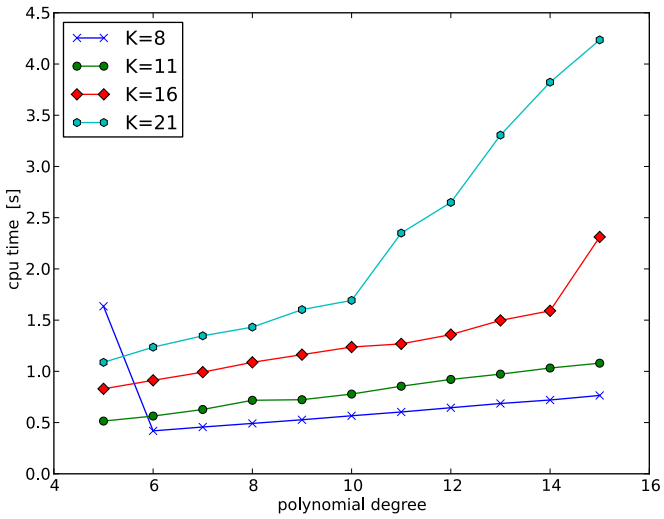

Figure 3.7: Required CPU time required for each shown combination of polynominal order and number of elements until the maximum error is below 10^{-2}

4 Conclusions

This introduction gives a concise overview of the discontinous Galerkin method from a pratical point of view, for which a simple partial differential equation, the 1D transport equation, is chosen in order to demonstrate how to derive and implement a DG method.
First the theoretical background of the DG method is introduced and the DG method is then derived mathematically (see Section 2), after which details of the implementation are given as well as results of the implemented method (see Section 3).
Because of its many beneficial features, such as effective parallelisation and refinement, local limiters and high order for instance, DG will be a relevant method for the solution of many problems in the future.

References

1. Williamson, Low storage Runge-Kutta schemes. Journal of Computational Physics 35, 48–56 (1980)

2. Reed and Hill, Triangular mesh methods for the neutron transport equation, 1973, Los Alamos Research Report LA-UR-73-479, Proceedings of the American Nuclear Society

3. Kopriva, Implementing Spectral Methods for Partial Differential Equations, 2009, Springer

4. Persson and Peraire, Sub-Cell Shock Capturing for Discontinuous Galerkin Methods, 2006, Aerospace Science Meeting and Exhibit, 44th AIAA

5. Hesthaven and Warburton, Nodal Discontinuous Galerkin Methods, 2007, Springer

6. Teukolsky, Vetterling and Flannery, Numerical Recipes: The Art of Scientific Computing (3rd ed.), 2007, Cambridge University Press

7. Dahmen and Reusken, Numerik für Ingenieure und Naturwissenschaftler, 2008, Springer

Appendix

A DG Solver

```
import numpy as np
import dg as DG
import matplotlib.pyplot as plt
import scipy as sp

##############################################################################
#  Helper function that runs the simulation a number of iterations either  #
#  using the default parameters or by either defining timeStep, startTime, #
#  endTime or the number of iterations                                     #
#                                                                          #
#  parameters:                                                             #
#  * mesh --- reference to mesh object for the simulation                  #
#                                                                          #
#  optional parameters                                                     #
#  * timeStep --- size of the time step (default=0.0025)                   #
#  * startTime --- time at which to start the simulation (default=0)       #
#  * endTime --- time at which to end the simulation (default=1.5)         #
#  * iterations --- number of iterations                                   #
#                                                                          #
##############################################################################

def run(mesh, timeStep=0.0025, startTime=0.0, endTime=1.5, iterations=None):
  if startTime>=endTime:
    raise ValueError("startTime larger than endTime")
  if not iterations:
    iterations=int((endTime-startTime)/timeStep)
  t=startTime
  for i in range(iterations):
    DGStepByRK3(t, timeStep, mesh)
    t+=timeStep

##############################################################################
#  Time integration by using a low storage RK3 variant                     #
#                                                                          #
#  parameters:                                                             #
#  * mesh --- reference to mesh object for the simulation                  #
#  * tn --- time at which to integrate                                     #
#  * dt --- time step size                                                 #
#                                                                          #
##############################################################################

def DGStepByRK3(tn, dt, mesh):
  #coefficients of the RK3 low storage variant
  a=[ 0.0, -0.55555555555, -1.1953125]
  b=[ 0.0,  0.33333333333,  0.75]
  g=[ 0.3333333333, 0.9375, 0.533333333]
  for m in range(3):
    t=tn+b[m]*dt
    mesh.GlobalTimeDerivative(t)
    for k in range(mesh.K):
      for j in range(mesh.N+1):
        mesh.ek[k].G[j]=a[m]*mesh.ek[k].G[j]+mesh.ek[k].dtQ[j]
        mesh.ek[k].Q[j]=mesh.ek[k].Q[j]+g[m]*dt*mesh.ek[k].G[j]


##############################################################################
#  Class represents a single DG element                                    #
#                                                                          #
#  public functions:                                                       #
#  * LocalTimeDerivative --- calculates the time derivative for the element#
#                                                                          #
##############################################################################
```

```
class element:

  def __init__(self, _dg, _xL, _xR):
    #element size
    self.dx=abs(_xL-_xR)
    #element left boundary
    self.xL=_xL
    #element right boundary
    self.xR=_xR
    #Nodal discontinuous galerkin
    self.dg=_dg
    #degree
    self.N=_dg.N
    #solution matrix
    self.Q=np.zeros(self.N+1)
    #time derivative matrix
    self.dtQ=np.zeros(self.N+1)
    #intermediate matrix for RK
    self.G=np.zeros(self.N+1)
    #solution on left element boundary
    self.Ql=0
    #solution on right element boundary
    self.Qr=0
    #flux through left element boundary
    self.Fl=0
    #flux through right element boundary
    self.Fr=0

  def InterpolateToBoundaries(self):
    self.Ql=np.dot(self.Q, self.dg.lm1)
    self.Qr=np.dot(self.Q, self.dg.lp1)

  def SystemDGDerivative(self, F):
    dF=np.zeros(self.N+1)
    dF=np.dot(self.dg.D, F)
    for j in range(self.N+1):
      dF[j]+=(self.Fr*self.dg.lp1[j]+self.Fl*self.dg.lm1[j])/self.dg.w[j]
    return dF

  def Flux(self, Q):
    return Q

  def LocalTimeDerivative(self):
    F=np.zeros(self.N+1)
    for j in range(self.N+1):
      F[j]=self.Flux(self.Q[j])
    dF=self.SystemDGDerivative(F)
    for j in range(self.N+1):
      self.dtQ[j]=-2.0*dF[j]/self.dx

##########################################################################
#  Class represents the mesh used for the simulation                     #
#                                                                        #
#  public functions:                                                     #
#  * GlobalTimeDerivative --- calculates the time derivative for the whole #
#    mesh                                                                #
#                                                                        #
##########################################################################

class mesh1D:
  def __init__(self, N, xk):
    #number of elements
    self.K=len(xk)-1
    #degree
```

```
    self.N=N
    #element boundary points
    self.xk=xk
    #elements
    self.ek=[]
    self.pk=[sharedNodePtrs() for i in range(self.K+1)]
    dG=DG.NodalDiscontinuousGalerkin(N)
    for i in range(self.K):
      self.ek+=[element(dG, xk[i], xk[i+1])]
    for i in range(self.K):
      self.pk[i].eLeft=i-1
      self.pk[i].eRight=i
    self.pk[0].eLeft=None
    self.pk[0].eRight=0
    self.pk[self.K].eLeft=self.K-1
    self.pk[self.K].eRight=None

  def externalState(self, qL, t, Left=False, Right=False):
    if Left:
      R=0
    if Right:
      R=2**(-((self.ek[0].xL+self.ek[0].dx*(1.0+self.ek[0].xL)/2.0)-t-1.0)**2/0.0225)
    return R


  def GlobalTimeDerivative(self, t):

    def riemannSolver(qL, qR, hn):
      #wave speed
      c=1.0
      return c*hn*qL

    for el in self.ek:
      el.InterpolateToBoundaries()
    k=self.pk[0].eRight
    QextL=self.externalState(self.ek[k].Ql, t, Left=True)
    k=self.pk[self.K].eLeft
    QextR=self.externalState(self.ek[k].Qr, t, Right=True)
    for pkEL in self.pk:
      idL=pkEL.eLeft
      idR=pkEL.eRight
      if idL==None:
        self.ek[idR].Fl=riemannSolver(QextL, self.ek[idR].Ql, -1.0)
      elif idR==None:
        self.ek[idL].Fr=riemannSolver(self.ek[idL].Qr, QextR, 1.0)
      else:
        fL=riemannSolver(self.ek[idL].Qr, self.ek[idR].Ql, 1.0)
        self.ek[idR].Fl=-fL
        self.ek[idL].Fr=fL
    for el in self.ek:
      el.LocalTimeDerivative()

class sharedNodePtrs:

  def __init__(self):
    #element to the left
    self.eLeft=None
    #element to the right
    self.eRight=None
```

B Plotting examples

```
"""
This file contains all plots that are used in the CES-seminar presentation and paper.
"""
import numpy as np
```

```
from datetime import datetime
import matplotlib.pyplot as plt
import DGSpec1D as DG
import scipy as sp

class timer:
  def __init__(self):
    self.startTime=datetime.now()

  def start(self):
    self.startTime=datetime.now()
    self.endTime=datetime.now()

  def stop(self):
    self.endTime=datetime.now()
    self.time=self.endTime-self.startTime
    return self.time.total_seconds()

def avgSumV(vec):
  result=0.0
  for num in vec:
    result+=num/len(vec)
  return result

##############################################################################
#exact solution of test function 1                                           #
##############################################################################

def exactSolutionT1(t, x):
  return 2**(-(x-t-1.0)**2/0.0225)

##############################################################################
#Plot numerical, initial and exact solution                                  #
##############################################################################

def plotNumInitEx(x, numSol, exSol, init, name):
  plt.plot(x, numSol, 'o', label="numeric solution")
  plt.plot(x, exSol, '-x', label="exact solution")
  plt.plot(x, init, '-', label="initial")
  plt.xlim([-1.5,4.5])
  plt.xlabel("x")
  plt.ylabel("u")
  plt.legend(loc=2)
  plt.savefig("./img/"+name+".svg")
  plt.close()

##############################################################################
#Plot max. and avg. error over order                                         #
##############################################################################

def plotErrorOrder(maxError, avgError, order, num):
  plt.plot(order, avgError, label="avg. error")
  plt.plot(order, maxError, label="max. error")
  plt.xlabel("polynomial degree")
  plt.ylabel("error")
  plt.yscale('log')
  plt.grid(True, ls="-")
  plt.legend(loc=0)
  plt.savefig("./img/errorOrderK"+str(num)+".svg")
  plt.close()

##############################################################################
#calculates the absolute difference between vectors and determines from#
#the result the maximal and average difference                               #
##############################################################################
```

```
def calcError(exSol, numSol):
  error=[]
  for i in range(len(exSol)):
    error+=[abs(exSol[i]-numSol[i])]
  print "maximal error", max(error), "avg. error", avgSumV(error)
  return {'error':error, 'maxError':max(error), 'avgError':avgSumV(error)}



##############################################################################
#plots with constant number of elements for different orders (3-20)    #
##############################################################################

#start time of simulation
startT=0
#end time of simulation
endT=1.5
#time stepsize
dt=0.0025
#number of elements
numElements=5

clock=timer()
avgError=[]
maxError=[]

#iterate over the order of the approximation
for N in np.arange(3,21):

  print "started calculation on", numElements,"elements and order", N

  init=[]
  x=[]
  exSol=[]
  numSol=[]
  xk=sp.linspace(0,3,num=numElements)

  mesh=DG.mesh1D(N, xk)

  #set initial values to exact results
  for el in mesh.ek:
    i=0
    for posx in el.dg.x:
      x+=[el.xL+el.dx*(1.0+posx)/2.0]
      el.Q[i]=exactSolutionT1(startT, x[-1])
      init+=[el.Q[i]]
      i+=1

  #run simulation
  clock.start()
  DG.run(mesh, timeStep=dt, endTime=endT, startTime=startT)
  print "time for calculation:", clock.stop(), "s"

  #determine the exact solution at each point to evaluate the simulation results
  for i in range(len(init)):
    exSol+=[exactSolutionT1(endT, x[i])]

  #extract the numerical solution so that it can be plotted
  for el in mesh.ek:
    temp=el.Q
    for sol in temp:
      numSol+=[sol]

  dicError=calcError(exSol, numSol)

  avgError+=[dicError['avgError']]
```

```
    maxError+=[dicError['maxError']]

    plotNumInitEx(x, numSol, exSol, init, "result_dt"+str(dt)+"N"+str(N)+"K5")

plotErrorOrder(maxError, avgError, np.arange(3,21), 5)

#plots of cputime/order/element and error/order/element

maxerrL=np.zeros((21,21))
avgerrL=np.zeros((21,21))
timing=np.zeros((21,21))
NL=np.zeros((21,21))

for K in range(3,19,3):
  N=1
  while N<=20:
    #initialize mesh
    xk=sp.linspace(0,3,num=K)
    mesh=DG.mesh1D(N, xk)

    #set initial condition
    exSol=[]
    numSol=[]
    x=[]
    for el in mesh.ek:
      i=0
      for posx in el.dg.x:
        x+=[el.xL+el.dx*(1.0+posx)/2.0]
        el.Q[i]=exactSolutionT1(startT, x[-1])
        i+=1
    t=0.0

    #run simulation
    clock.start()
    DG.run(mesh, timeStep=dt, endTime=endT, startTime=startT)
    timing[K][N-1]=clock.stop()
    print "time for calculation:", timing[K][N-1], "s"


    for i in range(len(x)):
      exSol+=[exactSolutionT1(endT, x[i])]

    for el in mesh.ek:
      temp=el.Q
      for sol in temp:
        numSol+=[sol]

    NL[K][N-1]=N

    dicError=calcError(exSol, numSol)

    maxerrL[K][N-1]=dicError['maxError']
    avgerrL[K][N-1]=dicError['avgError']

    print "for order of", N, "and", K-1, "elements with dt", dt, "maximal error",
        dicError['maxError'], "avg. error", dicError['avgError']

    N+=1

plt.plot(NL[3][0:20], timing[3][0:20], '-x', label="K=2")
plt.plot(NL[6][0:20], timing[6][0:20], '-o', label="K=5")
plt.plot(NL[12][0:20], timing[12][0:20], '-D', label="K=10")
plt.plot(NL[18][0:20], timing[18][0:20], '-h', label="K=15")
plt.xlabel("polynomial degree")
plt.ylabel("cpu time  [s]")
plt.legend(loc=0)
```

```
plt.savefig("./img/cpu-time-(order-vs-numElements).svg")
plt.close()

plt.plot(NL[3][0:20], maxerrL[3][0:20], '-x', label="K=2")
plt.plot(NL[6][0:20], maxerrL[6][0:20], '-o', label="K=5")
plt.plot(NL[12][0:20], maxerrL[12][0:20], '-D', label="K=10")
plt.plot(NL[18][0:20], maxerrL[18][0:20], '-h', label="K=15")
plt.xlabel("polynomial degree")
plt.ylabel("maximal error")
plt.yscale('log')
plt.legend(loc=1)
plt.savefig("./img/maximal-error-(order-vs-numElements).svg")
plt.close()

#plots of cputime to reach error at N, K combination

maxerrL=np.zeros((21,21))
avgerrL=np.zeros((21,21))
timing=np.zeros((21,21))
NL=np.zeros((21,21))

for K in range(3,19,3):
  N=5
  while N<=16:
    #initialize mesh
    xk=sp.linspace(0,3,num=K+6)
    mesh=DG.mesh1D(N, xk)

    #set initial condition
    exSol=[]
    numSol=[]
    x=[]
    for el in mesh.ek:
      i=0
      for posx in el.dg.x:
        x+=[el.xL+el.dx*(1.0+posx)/2.0]
        el.Q[i]=exactSolutionT1(startT, x[-1])
        i+=1
    t=0.0

    #run simulation
    clock.start()
    DG.run(mesh, timeStep=dt, endTime=endT, startTime=startT)
    timing[K][N-1]=clock.stop()
    print "time for calculation:", timing[K][N-1], "s"


    for i in range(len(x)):
      exSol+=[exactSolutionT1(endT, x[i])]

    for el in mesh.ek:
      temp=el.Q
      for sol in temp:
        numSol+=[sol]

    NL[K][N-1]=N

    dicError=calcError(exSol, numSol)

    maxerrL[K][N-1]=dicError['maxError']
    avgerrL[K][N-1]=dicError['avgError']

    print "for order of", N, "and", K-1, "elements with dt", dt, "maximal error",
        dicError['maxError'], "avg. error", dicError['avgError']

    if maxerrL[K][N-1]<0.008:
```

```
        print "convergence reached"
        dt=1.0
      else:
        dt=dt*0.9
        print "solution didnot converge decreased dt to", dt
        continue

      N+=1

plt.plot(NL[3][4:15], timing[3][4:15], '-x', label="K=8")
plt.plot(NL[6][4:15], timing[6][4:15], '-o', label="K=11")
plt.plot(NL[12][4:15], timing[12][4:15], '-D', label="K=16")
plt.plot(NL[18][4:15], timing[18][4:15], '-h', label="K=21")
plt.xlabel("polynomial degree")
plt.ylabel("cpu time  [s]")
plt.legend(loc=0)
plt.savefig("./img/cpu-time-(order-vs-numElements)-const-error.svg")
plt.close()

#plot of minimal time stepsize convergent for a given combination of elements and
    orders

maxerrL=np.zeros((21,21))
avgerrL=np.zeros((21,21))
timing=np.zeros((21,21))
NL=np.zeros((21,21))
dtL=np.zeros((21,21))
dt=1.0

for K in range(3,19,3):
  N=3
  while N<=20:
    #initialize mesh
    xk=sp.linspace(0,3,num=K)
    mesh=DG.mesh1D(N, xk)

    #set initial condition
    exSol=[]
    numSol=[]
    x=[]
    for el in mesh.ek:
      i=0
      for posx in el.dg.x:
        x+=[el.xL+el.dx*(1.0+posx)/2.0]
        el.Q[i]=exactSolutionT1(startT, x[-1])
        i+=1
    t=0.0

    #run simulation
    clock.start()
    DG.run(mesh, timeStep=dt, endTime=endT, startTime=startT)
    timing[K][N-1]=clock.stop()
    print "time for calculation:", timing[K][N-1], "s"


    for i in range(len(x)):
      exSol+=[exactSolutionT1(endT, x[i])]

    for el in mesh.ek:
      temp=el.Q
      for sol in temp:
        numSol+=[sol]

    NL[K][N-1]=N

    dicError=calcError(exSol, numSol)
```

```
        diff=abs(avgerrL[K][N-1]-dicError['avgError'])
        maxerrL[K][N-1]=dicError['maxError']
        avgerrL[K][N-1]=dicError['avgError']
        dtL[K][N-1]=dt

        print "for order of", N, "and", K-1, "elements with dt", dt, "maximal error",
            dicError['maxError'], "avg. error", dicError['avgError']

        if avgerrL[K][N-1]<=0.0001 or diff<=0.00001:
          print "convergence reached"
          dt=1.0
        else:
          dt=dt*0.9
          print "solution didnot converge decreased dt to", dt
          continue

        N+=1

plt.plot(NL[3][0:20], dtL[3][0:20], label="2 elements")
plt.plot(NL[6][0:20], dtL[6][0:20], label="5 elements")
plt.plot(NL[12][0:20], dtL[12][0:20], label="10 elements")
plt.plot(NL[18][0:20], dtL[18][0:20], label="15 elements")
plt.xlabel("polynomial degree")
plt.ylabel("convergent dt")
plt.yscale('log')
plt.legend(loc=0)
plt.savefig("./img/convdt-(order-vs-numElements).svg")
plt.close()
```