Thesis for the degree of Master of Science in Engineering Physics

Fluid dynamics

Simon Vajedi

Fundamental Physics
Chalmers University of Technology
GÃ¶teborg, Sweden 2013

Fluid dynamics

Simon Vajedi
Fundamental Physics
Chalmers University of Technology
SE-412 96 GÃ¶teborg, Sweden

Abstract

In this thesis we outline U-duality covariant dynamics for a D_1-brane in 9-dimensional supergravity and for a D_2-brane in 8-dimensional supergravity. By introducing modified field strengths, all world-volume fields couple to the background fields in a manifestly U-duality symmetric way. Our methods produce an action involving a function Φ which can only be deduced by demanding it to fulfill certain duality relations. This is achieved (to some extent) by implementing the equations in a computer program. Furthermore, solving the field equations for some of the potentials produces integration parameters which can be identified as the brane charges. The thesis also contains an introduction to supergravity, Kaluza-Klein reduction and U-duality, as well as the complete construction of bosonic 9-, 8- and 7-dimensional supergravity.

Acknowledgments

We would like to express our gratitude to our supervisor Bengt E.W. Nilsson, for guidance and for introducing us to the fascinating world of M-theory. A special thanks for never losing your patience although, at times, it seemed like the work with this thesis could go on forever. The authors would also like to thank each other for good work and cooperation and for the complete understanding of the importance of breaks.

Contents

1

Introduction

This thesis concerns the investigation of small finite-size particles in a fluid, where the density of the particles differs from that of the fluid.

1.1 Outline

Part I

Dimensional Reduction

2

Nan

The motions of suspended particles in a turbulent fluid are complex and approximations must be made. A very convenient one would be to consider inertaless point particles, in which case the equation of motion would be given by

$$\dot{\boldsymbol{r}}(t) = \boldsymbol{u}(\boldsymbol{r}(t), t). \tag{2.1}$$

The particles would hence follow the flow and at every point take on the velocity of the fluid. Particles which satisfy this equation of motion are *advected* and are called *passive tracers*. This approximation has many applications in fluid dynamics and in the early days, Eq. (2.1) was used predominantly. However, in order to describe more complex behavior like ... and ..., this is not enough.

γ is the damping rate. The Stokes number $\mathrm{St} = 1/\gamma\tau$ is the (inverse?) intensity of the damping. When $\mathrm{St} \to 0$ the particles are completely advected and behave like point particles with no inertia, and the equation of motion is then given by Eq. (2.1).

Vorticity is the measure of how fast fluid elements rotate about themselves. The curl of the velocity field is the vorticity field and is denoted $\boldsymbol{\omega} = \nabla \times \boldsymbol{u}$. Vorticity is a local measure and does is not necessarily nonzero even if the fluid rotates at a large scale.

A vortex is a region of high vorticity.

The particle Reynolds number is given by

$$\mathrm{Re}_p = \frac{d|\boldsymbol{\nu}_p - \boldsymbol{u}|}{\nu}. \tag{2.2}$$

2.1 The Maxey-Riley equation

- For small spherical rigid particle advected by a (smooth) flow. - Valid for small particles at low particle Reynolds numbers Re_p. - Velocity difference across the

particle must be small

The Maxey-Riley equation is given by

$$m_p\dot{\boldsymbol{v}} = m_f\frac{\mathrm{D}}{\mathrm{D}t}\boldsymbol{u}(\boldsymbol{r}(t),t) - \frac{1}{2}m_f\left(\dot{\boldsymbol{v}} - \frac{\mathrm{D}}{\mathrm{D}t}\left[\boldsymbol{u}(\boldsymbol{r}(t),t) + \frac{1}{10}a^2\nabla^2\boldsymbol{u}(\boldsymbol{r}(t),t)\right]\right) - 6\pi a\rho_f\nu\boldsymbol{q}(t) + (m_p - m_f)\boldsymbol{g} - 6\pi a^2\rho_f\nu\int_0^t \frac{\mathrm{d}\tau}{\sqrt{\pi\nu(t-\tau)}}\frac{\mathrm{d}\boldsymbol{q}(\tau)}{\mathrm{d}\tau}, \tag{2.3}$$

where

$$\boldsymbol{q}(t) = \boldsymbol{v}(t) - \boldsymbol{u}(\boldsymbol{r}(t),t) - \frac{1}{6}a^2\nabla^2\boldsymbol{u} \tag{2.4}$$

m_p is the particle mass, a the particle radius, m_f the mass of the fluid displaced by the particle, ρ_f is the density of the fluid, ν the viscosity of the fluid. The first term on the right-hand side of the Maxey-Riley is the force exerted by a fluid element in position $\boldsymbol{r}(t)$ and corresponds to the force by the force of the undisturbed fluid. The second term is due to the *added-mass effect*, which results from the fact that the particle displaces a certain amount of fluid along its trajectory, which makes the particle appear to have additional mass. The third and the fourth terms result from the viscosity and the buoyancy force, respectively, of the fluid, which represent the Stokes' drag. The factor of $a^2\nabla^2\boldsymbol{u}$ is due to the spatial variation of the velocity field across the particle, and the terms containing this are called *Faxén corrections*.

The integral is the *Basset-Boussinesq history term*, which accounts for the fact that the vorticity diffuses away from the particle due to viscosity.

$$\frac{\mathrm{D}\boldsymbol{u}}{\mathrm{D}t} = \frac{\partial\boldsymbol{u}}{\partial t} + (\boldsymbol{u}\cdot\nabla)\boldsymbol{u}. \tag{2.5}$$

$$\frac{\mathrm{d}\boldsymbol{u}}{\mathrm{d}t} = \frac{\partial\boldsymbol{u}}{\partial t} + (\boldsymbol{v}\cdot\nabla)\boldsymbol{u}. \tag{2.6}$$

A

Conventions and some basic formulae

Sometimes, we suspect that the authors of physical articles ignore to state their conventions on purpose, so that any weird minus signs or misprints in the text will be virtually impossible to check. Here we are brave enough to introduce the conventions we have used in this thesis. So if you find something weird in this thesis, it is a result of our own failure and not of some brilliant convention the world has never seen before. We also introduce some "good-to-know" formulas.

A.1 Index conventions

Space	Index	Signature	Range
11-dim curved	M,N,P,...	$(-+++\cdots+)$	1 ... $\hat{D}$
11-dim flat	A,B,C,...	$(-+++\cdots+)$	1 ... $\hat{D}$
Compactified curved	m,n,p,...	$(+++\cdots+)$	1 ... D
Compactified flat	a,b,c,...	$(+++\cdots+)$	1 ... D
Uncompactified curved	μ,ν,ρ,...	$(-+++\cdots+)$	1 ... d
Uncompactified flat	i,j,k,...	$(-+++\cdots+)$	1 ... d
World volume	α,β,γ,...	$(-+++\cdots+)$	1 ... p

Table A.1: Index conventions and signature for different spaces

Uncompactified tensors are denoted with a hat, like $\hat{g}_{MN}, \hat{C}_{PMN}$ etc. When compactifying, we denote the dimension of the starting theory with $\hat{D}$, the number of compactified dimensions D and the number of uncompactified dimensions d.

Throughout this thesis we set the Newton constant in 11 dimensions to $\kappa_{11}^2 = \frac{1}{2}$.

Symmetrisation and antisymmetrisation are denoted by

$$A_{(M_1\cdots M_p)} = \frac{1}{p!}\left(A_{M_1\cdots M_p} + \text{symmetric permutations}\right),$$
$$A_{[M_1\cdots M_p]} = \frac{1}{p!}\left(A_{M_1\cdots M_p} + \text{antisymmetric permutations}\right). \tag{A.1}$$

A.2 Antisymmetric tensor and p-forms

Define the Levi-Civita symbol with lower indices as $\epsilon_{A_1A_2\cdots A_n}$, antisymmetric in all indices and $\epsilon_{12\cdots n} = +1$. Define further the Levi-Civita symbol with upper indices as $\epsilon^{A_1A_2\cdots A_n} = (-1)^s\,\epsilon_{A_1A_2\cdots A_n}$, where s is the number of timelike coordinates in the metric. In flat spacetime this symbol acts like a tensor. In curved spacetime the Levi-Civita symbol transforms as

$$\epsilon^{M_1M_2\cdots M_n} \to \epsilon^{M_1'M_2'\cdots M_n'} \tag{A.2}$$

under a general coordinate transformation $x \to x'$. Using that the determinant of some matrix A obeys the relation

$$\epsilon^{M_1M_2\cdots M_n}|A| = \epsilon^{N_1N_2\cdots N_n}A_{N_1}{}^{M_1}A_{N_2}{}^{M_2}\cdots A_{N_n}{}^{M_n}, \tag{A.3}$$

we find that (using A=$\partial x'^{M_i'}/\partial x^{M_j}$)

$$\epsilon^{M_1'M_2'\cdots M_n'} = \left|\frac{\partial x'}{\partial x}\right|^{-1}\frac{\partial x'^{M_1'}}{\partial x^{M_1}}\frac{\partial x'^{M_2'}}{\partial x^{M_2}}\cdots\frac{\partial x'^{M_n'}}{\partial x^{M_n}}\epsilon^{M_1M_2\cdots M_n}. \tag{A.4}$$

Compare this to the square root of the determinant of the metric $\sqrt{|g|}$ that transforms as (consider $\partial x^{N_i}/\partial x'^{M_j}$ as matrices)

$$\sqrt{|g|} \to \sqrt{|g|'} = \sqrt{\left|\det\left(\frac{\partial x^P}{\partial x'^M}g_{PQ}\frac{\partial x^Q}{\partial x'^N}\right)\right|} = \left|\left|\frac{\partial x}{\partial x'}\right|\right|\sqrt{|g|}. \tag{A.5}$$

Combining equations (A.4) and (A.5) gives

$$\frac{1}{\sqrt{|g|}}\epsilon^{M_1\cdots M_n} \to \frac{1}{\sqrt{|g|'}}\epsilon'^{M_1\cdots M_n} = \frac{1}{\sqrt{|g|}}\frac{\partial x'^{M_1'}}{\partial x^{M_1}}\frac{\partial x'^{M_2'}}{\partial x^{M_2}}\cdots\frac{\partial x'^{M_n'}}{\partial x^{M_n}}\epsilon^{M_1M_2\cdots M_n}, \tag{A.6}$$

which transforms as a four tensor. Thus one can form an antisymmetric covariant tensor in curved space $\varepsilon^{M_1M_2\cdots M_n}$ as

$$\varepsilon^{M_1M_2\cdots M_n} = \frac{1}{\sqrt{|g|}}\epsilon^{M_1M_2\cdots M_n}. \tag{A.7}$$

Define the antisymmetric tensor with lower indices as

$$\varepsilon_{M_1M_2\cdots M_n} = g_{M_1N_1}g_{M_2N_2}\cdots g_{M_nN_n}\varepsilon^{N_1N_2\cdots N_n} = \sqrt{|g|}\epsilon_{M_1M_2\cdots M_n}, \tag{A.8}$$

which transforms as a covariant tensor.

Contraction of two antisymmetric tensors

$$\varepsilon^{M_1\cdots M_p N_{p+1}\cdots N_n}\varepsilon_{M_1\cdots M_p P_{p+1}\cdots P_n} = (-1)^s\, p!\,(n-p)!\delta^{N_{p+1}\cdots N_n}_{[P_{p+1}\cdots P_n]}, \tag{A.9}$$

where $\delta^{N_{p+1}\cdots N_n}_{[P_{p+1}\cdots P_n]} = \delta^{N_{p+1}}_{[P_{p+1}}\cdots\delta^{N_n}_{P_n]}$ is the generalized Kronecker delta with the nice property $\delta^{N_1\cdots N_p}_{[M_1\cdots M_p]}A^{M_p\cdots M_1} = A^{N_p\cdots N_1}$ for a p-form $A^{(p)}$.

In two dimensions By using contraction of antisymmetric tensors one can show the useful identities

$$\varepsilon_{MN}g^{NP}\varepsilon_{PQ} = (-1)^{s+1}g_{MQ} \tag{A.10}$$

and

$$\varepsilon_{MN}g_{PQ} = \varepsilon_{MQ}g_{PN} + \varepsilon_{QN}g_{PM} = \varepsilon_{MP}g_{NQ} + \varepsilon_{PN}g_{MQ}. \tag{A.11}$$

In three dimensions We get the corresponding relations to (A.11)

$$\begin{aligned}\varepsilon_{MNP}g_{S[Q}g_{R]T} &= \varepsilon_{STP}g_{M[Q}g_{R]N} - \varepsilon_{SNT}g_{P[Q}g_{R]M} + \varepsilon_{MST}g_{N[Q}g_{R]P}\\ \varepsilon_{MNP}g_{QR} &= \varepsilon_{RNP}g_{MQ} + \varepsilon_{MRP}g_{QN} + \varepsilon_{MNR}g_{QP}\end{aligned} \tag{A.12}$$

Forms We use the superspace convention of differential forms:

$$\begin{aligned}A_{(p)} &= \frac{1}{p!}dx^{M_1}\wedge dx^{M_2}\cdots\wedge dx^{M_{p-1}}\wedge dx^{M_p}A_{M_pM_{p-1}\cdots M_2M_1}\\ &= \frac{1}{p!}e^{A_1}\wedge e^{A_2}\cdots\wedge e^{A_{p-1}}\wedge e^{A_p}A_{A_pA_{p-1}\cdots A_2A_1}\end{aligned} \tag{A.13}$$

where $e^A = dx^M e_M{}^A$ and $A_{M_pM_{p-1}\cdots M_2M_1}$ is antisymmetric in all indices and the wedge products have the characteristic property $dx^M\wedge dx^N = -dx^N\wedge dx^M$. Whenever we use a form without form-index, its type is given according to table A.2.

Type	p	notation
Background gauge potentials	1, 2, 3,...	A, B, C, D,...
Background field strengths	2, 3, 4,...	F, H, G, I,...
World volume gauge potentials	0, 1, 2, 3,...	ϕ, a, b, c, d,...
World volume field strengths	1, 2, 3, 4,...	ω, f, h, g, i,...

Table A.2: P-form conventions.

Exterior derivative $d = dx^M \partial_M$ acting from right:

$$dA_{(p)} = \partial_{M_{p+1}} A_{(p)} \wedge dx^{M_{p+1}} = \frac{1}{p!} dx^{M_1} \wedge \cdots \wedge dx^{M_p} \wedge dx^{M_{p+1}} \partial_{M_{p+1}} A_{M_p \cdots M_1} \quad \text{(A.14)}$$

giving the wedge product derivation law

$$d\left(A_{(p)} \wedge B_{(q)}\right) = A_{(p)} \wedge dB_{(q)} + (-1)^q dA_{(p)} \wedge B_{(q)}. \quad \text{(A.15)}$$

Note that, for well behaved functions, two derivatives commute so

$$d^2 A_{(p)} = \partial_{(M} \partial_{N)} A_{(p)} \wedge dx^{[N} \wedge dx^{M]} = 0. \quad \text{(A.16)}$$

Hodge duality of a p-form Map from a p-form to an $(n-p)$-form defined as

$$* A_{(p)} = \frac{1}{p!(n-p)!} dx^{M_{p+1}} \wedge \cdots \wedge dx^{M_n} \varepsilon_{M_{p+1} \cdots M_n N_1 \cdots N_p} A^{N_p \cdots N_1}, \quad \text{(A.17)}$$

which in component form becomes

$$(*A)_{M_n \cdots M_{p+1}} = \frac{1}{p!} \varepsilon_{M_{p+1} \cdots M_n N_1 \cdots N_p} A^{N_p \cdots N_1} \quad \text{(A.18)}$$

or looking only at the differentials

$$*\left(dx^{M_1} \wedge \cdots \wedge dx^{M_p}\right) = \frac{1}{(n-p)!} dx^{N_{p+1}} \wedge \cdots \wedge dx^{N_n} \varepsilon_{N_{p+1} \cdots N_n}{}^{M_1 \cdots M_p}. \quad \text{(A.19)}$$

Using (A.18) and swapping order of n with n-p antisymmetric indices, one sees that two hodge dualities performed after each other gives back the starting form with a possible additional minus sign

$$*\left(*A_{(p)}\right) = (-1)^{s+p(n-p)} A_{(p)}, \quad \text{(A.20)}$$

where $s = 0$ for Riemannian space and $s = 1$ for Minkowski space. One can also rewrite an arbitrary p-form in terms of its Hodge duality by

$$\begin{aligned} A_{(p)} &= (-1)^{s+p(n-p)} * (*A_{(p)}) \quad \text{(A.21)} \\ &= \frac{(-1)^{s+p(n-p)}}{(n-p)!p!} dx^{M_1} \wedge \cdots \wedge dx^{M_p} \varepsilon_{M_1 \cdots M_p N_{p+1} \cdots N_n} \left(*A_{(p)}\right)^{N_n \cdots N_{p+1}} \end{aligned}$$

or in component form

$$A_{M_p \cdots M_1} = \frac{(-1)^{s+p(n-p)}}{(n-p)!} \varepsilon_{M_1 \cdots M_p N_{p+1} \cdots N_n} (*A)^{N_n \cdots N_{p+1}}. \quad \text{(A.22)}$$

Volume form The volume form is

$$\begin{aligned}\sigma = *1 &= \frac{1}{n!} dx^{M_1} \wedge \cdots \wedge dx^{M_n} \varepsilon_{M_1 \cdots M_n} \\ &= \frac{(-1)^s}{n!} \sqrt{|g|} d^n x \varepsilon^{M_1 \cdots M_n} \varepsilon_{M_1 \cdots M_n} = + \sqrt{|g|} d^n x, \end{aligned} \tag{A.23}$$

where we have used

$$dx^{M_1} \wedge \cdots \wedge dx^{M_n} = (-1)^s d^n x \epsilon^{M_1 \cdots M_n} = (-1)^s \sqrt{|g|} d^n x \varepsilon^{M_1 \cdots M_n}. \tag{A.24}$$

Inner product If one has 2 forms A and B of equal length p one can form an inner product $< A, B >= \frac{1}{p!} A \cdot B$ from

$$\begin{aligned} *A \wedge B &= \frac{\varepsilon_{M_{p+1} \cdots M_n N_1 \cdots N_p}}{(n-p)! p!^2} A^{N_p \cdots N_1} B_{M_p \cdots M_1} dx^{M_{p+1}} \cdots dx^{M_n} dx^{M_1} \cdots dx^{M_p} \\ &= \frac{(-1)^s}{(n-p)! p!^2} \sqrt{g} \epsilon_{M_{p+1} \cdots M_n N_1 \cdots N_p} A^{N_p \cdots N_1} B_{M_p \cdots M_1} d^n x \epsilon^{M_{p+1} \cdots M_n M_1 \cdots M_p} \\ &= d^n x \sqrt{|g|} \frac{1}{(n-p)! p!^2} (n-p)! p! \delta^{M_1 \cdots M_p}_{[N_1 \cdots N_p]} A^{N_p \cdots N_1} B_{M_p \cdots M_1} \\ &= \sigma \frac{1}{p!} A^{M_p \cdots M_1} B_{M_p \cdots M_1} = \sigma < A, B >, \end{aligned} \tag{A.25}$$

where the inner product is $< A, B >= \frac{1}{p!} A^{M_p \cdots M_1} B_{M_p \cdots M_1}$. Note that we never used that the hodge star was acting on the A form. We could as well have acted on the B form, so

$$\sigma < A, B >= *A \wedge B = A \wedge *B = p! \sigma < *A, *B > . \tag{A.26}$$

Differentiation of forms The differentiation of the components of a p-form $A_{(p)}$ with respect to the components of another p-form of the same type is

$$\frac{\delta A_{M_p \cdots M_1}}{\delta A_{N_p \cdots N_1}} = \delta^{N_p}_{M_p} \cdots \delta^{N_2}_{M_2} \delta^{N_1}_{M_1} - \delta^{N_p}_{M_p} \cdots \delta^{N_1}_{M_1} \delta^{N_2}_{M_2} + \cdots = p! \delta^{N_p \cdots N_1}_{[M_p \cdots M_1]}. \tag{A.27}$$

Now suppose a p-form $A_{(p)}$ constructed by 3 different forms $A^1_{(q)}$, $A^2_{(r)}$ and $A^3_{(s)}$ of orders q, r and s, with $q + r + s = p$. I.e. $A_{(p)} = A^1_{(q)} \wedge A^2_{(r)} \wedge A^3_{(s)}$. Differentiate with respect to $A^2_{(r)}$ in component form

$$\begin{aligned} \frac{\delta A_{M_p \cdots M_1}}{\delta A^2_{N_r \cdots N_1}} &= \frac{p!}{q! r! s!} A^1_{M_q \cdots M_1} \frac{\delta A^2_{M_{q+r} \cdots M_{q+1}}}{\delta A^2_{N_r \cdots N_1}} A^3_{M_p \cdots M_{q+r+1}} \\ &= \frac{p!}{q! r! s!} A^1_{M_q \cdots M_1} r! \delta^{N_r N_{r-1} \cdots\cdots\cdots N_1}_{[M_{q+r} M_{q+r-1} \cdots M_{q+1}]} A^3_{M_p \cdots M_{q+r+1}}. \end{aligned} \tag{A.28}$$

Let $B_{(t)} = \frac{\delta A_{(p)}}{\delta A^2_{(r)}}$ be a t-form and multiply both sides from the right with $dx^{N_1} \wedge \cdots \wedge dx^{N_r}/r!$, so

$$\begin{aligned} B_{(t)} \wedge \frac{1}{r!} dx^{N_1} \wedge \cdots \wedge dx^{N_r} &= \frac{1}{r!p!} \frac{\delta A_{M_p \cdots M_1}}{\delta A^2_{N_r \cdots N_1}} dx^{M_1} \wedge \cdots \wedge dx^{M_p} \\ &= \frac{1}{q!r!s!} A^1_{M_q \cdots M_1} A^3_{M_p \cdots M_{q+r+1}} dx^{M_1} \cdots dx^{M_q} dx^{N_1} \cdots dx^{N_r} dx^{M_{q+r+1}} \cdots dx^{M_p} \\ &= \frac{1}{r!} A^1_{(q)} \wedge dx^{N_1} \wedge \cdots \wedge dx^{N_r} \wedge A^3_{(s)}. \end{aligned} \tag{A.29}$$

$$\tag{A.30}$$

We can now identify $B_{(t)}$ as

$$B_{(t)} = B_{(p-r)} = \frac{\delta A_{(p)}}{\delta A^2_{(r)}} = (-1)^{rs} A^1_{(q)} \wedge A^3_{(s)}, \tag{A.31}$$

which we use as a definition of derivation of a form with respect to a form. The reason we put $dx^{N_1} \wedge \cdots \wedge dx^{N_r}/r!$ to the right of $B_{(t)}$ and not to the left, which would have given $B_{(t)} = (-1)^{qr} A^1_{(q)} \wedge A^3_{(s)}$, is that we put all variations of forms $\delta A^2_{(r)}$ to the right.

To differentiate a scalar function $\Phi\left(A_{M_p \cdots M_1}\right)$, which depends only on the components of $A_{(p)}$, we just do a component differentiation

$$B^{N_p \cdots N_1} = \frac{\delta \Phi\left(A_{M_p \cdots M_1}\right)}{\delta A_{N_p \cdots N_1}} \tag{A.32}$$

and contract the free indices with $dx^{N_1} \wedge \cdots \wedge dx^{N_p}/p!$ to get a p-form

$$B^{(p)} = \frac{\delta \Phi\left(A_{M_p \cdots M_1}\right)}{\delta A_{(p)}} = \frac{1}{p!} \frac{\delta \Phi\left(A_{M_p \cdots M_1}\right)}{\delta A_{N_p \cdots N_1}} dx^{N_1} \wedge \cdots \wedge dx^{N_p}. \tag{A.33}$$

With the tools collected so far one easily obtains the differentiation with respect to a p-form $A_{(p)}$ of the hodge dual of a max-form $B_{(n)}$, such that $B_{(n)} = B^2_{(n-p)} \wedge A_{(p)}$ as

$$\begin{aligned} *\frac{\delta * B_{(n)}}{\delta A_{(p)}} &= \frac{1}{p!(n-p)!} \varepsilon_{M_1 \cdots M_n} B^{2M_n \cdots M_{p+1}} * \left(dx^{M_1} \wedge \cdots \wedge dx^{M_p}\right) \\ &= \frac{(-1)^{s+p(n-p)}}{(n-p)!} B^2_{M_n \cdots M_{p+1}} dx^{M_{p+1}} \wedge \cdots \wedge dx^{M_n} = (-1)^{s+p(n-p)} B^2_{(n-p)}. \end{aligned} \tag{A.34}$$

Variation of an action with respect to forms Consider a scalar Lagrangian density $\mathcal{L}\left(A_{(p)}\right)$ that is a function of some p-form $A_{(p)}$. The variation of the action

with respect to the components of $A_{(p)}$ is

$$\delta_A S = \delta_A \left[\int d^n x \sqrt{|g|} \mathcal{L}\left(A_{(p)}\right) \right] = \int d^n x \sqrt{|g|} \frac{\delta \mathcal{L}\left(A_{(p)}\right)}{\delta A_{M_p \cdots M_1}} \delta A_{M_p \cdots M_1}. \tag{A.35}$$

Consider

$$B^{M_p \cdots M_1} = \frac{\delta \mathcal{L}\left(A_{(p)}\right)}{\delta A_{M_p \cdots M_1}} \tag{A.36}$$

as the components of a p-form $B_{(p)}$ and use the definition of the inner product (A.25) between $B_{(p)}$ and $\delta A_{(p)}$ to get

$$\begin{aligned} \delta_A S &= p! \int \sigma \left\langle \frac{\delta \mathcal{L}\left(A_{(p)}\right)}{\delta A_{M_p \cdots M_1}}, \delta A_{M_p \cdots M_1} \right\rangle \\ &= p! \int * \frac{\delta \mathcal{L}\left(A_{(p)}\right)}{\delta A_{(p)}} \wedge \delta A_{(p)}. \end{aligned} \tag{A.37}$$

A.3 General relativity

Covariant divergence of a general covariant vector V^M is [6]

$$\begin{aligned} D_M V^M &= \frac{1}{\sqrt{|g|}} \partial_M \left(\sqrt{|g|} V^M \right) \\ &\Rightarrow \int d^D x \sqrt{|g|} D_M V^M = 0, \text{ if } V^M = 0 \text{ at } \infty. \end{aligned} \tag{A.38}$$

Variation of $g = \det g_{MN}$ and pure gravity with respect to g_{MN}

$$\delta g = g g^{MN} \delta g_{MN}, \tag{A.39}$$

$$\delta\left(\sqrt{g} R\right) = \left(R^{MN} - \frac{1}{2} g^{MN} R \right) \delta g_{MN}. \tag{A.40}$$

A.4 Matrix identities

Consider a matrix $\mathcal{M}$ and calculate

$$0 = \partial \mathbb{I} = \partial\left(\mathcal{M}^{-1}\mathcal{M}\right) = \partial \mathcal{M}^{-1} \mathcal{M} + \mathcal{M}^{-1} \partial \mathcal{M}, \tag{A.41}$$

giving the identity

$$\begin{aligned} \partial \mathcal{M}^{-1} &= -\mathcal{M}^{-1} \partial \mathcal{M} \mathcal{M}^{-1}, \\ \partial \mathcal{M}^{-1} \partial \mathcal{M} &= -\left(\mathcal{M}^{-1} \partial \mathcal{M}\right)^2. \end{aligned} \tag{A.42}$$

Determinant conditions For an arbitrary matrix $\mathcal{M}$ we have

$$\delta \ln \det(\mathcal{M}) = \mathrm{Tr}\left(\mathcal{M}^{-1}\delta\mathcal{M}\right) \tag{A.43}$$

from p. 106 in Weinberg[6]. Letting $\mathcal{M} = \exp(\mathcal{N})$ gives

$$\delta \ln \det \exp(\mathcal{N}) = \mathrm{Tr}\left(\exp(-\mathcal{N})\delta \exp \mathcal{N}\right) = \mathrm{Tr}\left(\delta\mathcal{N}\right) = \delta \mathrm{Tr}\left(\mathcal{N}\right), \tag{A.44}$$

giving

$$\det \mathcal{M} = \exp\left(\mathrm{Tr}\left(\ln \mathcal{M}\right)\right) \tag{A.45}$$

which will have many applications when dealing with determinants.

Determinant of block matrix Consider an $n \times n$-matrix and divide the rows and columns in two, forming a 4 piece block matrix.

$$\mathcal{M} = \begin{pmatrix} A & B \\ C & D \end{pmatrix} = \begin{pmatrix} \mathrm{A} & \mathrm{B} \\ \mathrm{C} & \mathrm{D} \end{pmatrix} \tag{A.46}$$

Note that we can decompose $\mathcal{M}$ as (proof by calculating backwards)

$$\mathcal{M} = \begin{pmatrix} \mathbb{I} & BD^{-1} \\ 0 & \mathbb{I} \end{pmatrix} \begin{pmatrix} A - BD^{-1}C & 0 \\ 0 & D \end{pmatrix} \begin{pmatrix} \mathbb{I} & D^{-1}C \\ 0 & \mathbb{I} \end{pmatrix}. \tag{A.47}$$

Take the determinant of $\mathcal{M}$ and expand along rows or columns consisting of only one 1 and zeroes to get two relations of the determinant

$$\det \mathcal{M} = \det\left(A - BD^{-1}C\right) \det D = \det A \det\left(D - CA^{-1}B\right), \tag{A.48}$$

where the second relation comes from a similar decomposition of $\mathcal{M}$ into Lower-Diagonal-Upper-form.

B

Lengthy calculations

B.1 Reduction and scaling of the Ricci scalar

We will use the tangent space formalism to dimensionally reduce the Ricci scalar and transform the metric to the Einstein frame.

B.1.1 Locally flat geometry

Locally flat frames To each point in a curved space with metric g_{MN}, we can assign a locally flat coordinate system with metric η_{AB}, tangent to the curved one. To transform between the two coordinate systems we use vielbeins $e_M{}^A$, defined to move between different metrics as

$$g_{MN} = e_M{}^A \eta_{AB} e^B{}_N \tag{B.1}$$

Since the n-dimensional covariant metric (without gauge conditions) has $n(n+1)/2$ degrees of freedom, there are $n^2 - n(n+1)/2 = n(n-1)/2$ undetermined components in the vielbeins. These are related to the $n(n-1)/2$ degrees of freedom of a local SO($\hat{D}$) rotation, or a SO($\hat{D}$-1,1) local Lorentz transformation in our case where we have one timelike coordinate. This symmetry comes from the fact that the vielbeins form a scalar in the local indices $e_M{}^A e_{AN}$ and the metric is thus invariant under local Lorentz transformations

$$\begin{aligned} &e_M{}^A \to e_M{}^B (\Lambda^{-1})_B{}^A \\ &e_A{}^M \to \Lambda_A{}^B e_B{}^M, \qquad \Lambda = \Lambda(x) \in \mathrm{SO}(\hat{D}-1,1) \end{aligned} \tag{B.2}$$

This symmetry can be used to set $n(n-1)/2$ of the components in $e_M{}^A$ to whatever you want.

Covariant derivative The exterior derivative acting on a tensor or form with m upper and n lower SO($\hat{D}$-1,1) Lorentz indices, transforms as

$$\begin{aligned} dT^{A_1\cdots A_m}_{\ \ C_1\cdots C_n} \rightarrow & d\left(\Lambda_{C_1}{}^{D_1}\cdots\Lambda_{C_n}{}^{D_n}\Lambda^{A_1}{}_{B_1}\cdots\Lambda^{A_n}{}_{B_n}T^{B_1\cdots B_m}_{\ \ D_1\cdots D_n}\right) \\ =&\Lambda\cdots\Lambda dT+\Lambda\cdots\Lambda T d\Lambda_{C_1}{}^{D_1}+\cdots+\Lambda\cdots\Lambda T d\Lambda_{C_n}{}^{D_n} \\ &+\Lambda\cdots\Lambda T d\Lambda^{A_1}{}_{B_1}+\Lambda\cdots\Lambda T d\Lambda^{A_m}{}_{B_m} \end{aligned} \tag{B.3}$$

The $d\Lambda$ terms prevents this from being a Lorentz tensor. We thus introduce a covariant derivative

$$\begin{aligned} DT^{A_1\cdots A_m}_{\ \ C_1\cdots C_n} =& dT^{A_1\cdots A_m}_{\ \ C_1\cdots C_n}+T^{B_1A_2\cdots A_m}_{\ \ \ C_1\ \ \cdots C_n}\omega^{A_1}{}_{B_1}+\cdots+T^{A_1\cdots B_m}_{\ \ C_1\cdots C_n}\omega^{A_m}{}_{B_m} \\ &+T^{A_1\ \ \cdots A_m}_{D_1C_2\cdots C_n}\omega_{C_1}{}^{D_1}+\cdots+T^{A_1\cdots B_m}_{\ \ C_1\cdots D_n}\omega_{C_m}{}^{D_m} \end{aligned} \tag{B.4}$$

and demand DT to transform as a Lorentz-tensor, i.e. $DT\rightarrow\Lambda\cdots\Lambda DT$, by assigning the correct transformation properties on the 1-form spin connection $\omega^A{}_B$. Since the transformation properties of ω are independent of n, we can look at the case m=1, n=0 with requirement $DT'^A\equiv\Lambda^A{}_B DT^B$, giving

$$\begin{aligned} DT^A\rightarrow DT'^A =& dT'^A+T^B\omega'^A{}_B=\Lambda^A{}_B dT^B+T^B d\Lambda^A{}_B+\Lambda^B{}_C T^C\omega'^A{}_B \\ \equiv&\Lambda^A{}_B DT^B=\Lambda^A{}_B\left(dT^B+T^C\omega^B{}_C\right) \end{aligned} \tag{B.5}$$

after removing T^C and multiplying by $\Lambda_B{}^C$ we get

$$\omega'^A{}_B=\Lambda^A{}_D\omega^D{}_C(\Lambda^{-1})^C{}_B-d\Lambda^A{}_C(\Lambda^{-1})^C{}_B \tag{B.6}$$

which is the transformation ω must obey. By calculating the covariant derivative in the coordinate basis, using the affine connection Γ and comparing this to the covariant derivative in a mixed basis, using the spin connection ω, we can find a relation between the two connections[27]. The relation becomes

$$\omega_M{}^A{}_B=e_N{}^A e_B{}^P\Gamma^N_{MP}-e_B{}^P\partial_M e_P{}^A \tag{B.7}$$

Using spin connections rather than the regular affine connections in the covariant derivative allows the descriptions of spinors in space time and it allows taking covariant derivatives of spinors (hence the name). Furthermore spin connection lets us describe the torsion and curvature as vector and (1,1)-tensor valued 2-forms as we shall see next.

Torsion Now consider the 2-form $T^A=De^A$, which components can, by using (B.7), be identified as the torsion tensor

$$T_{MN}{}^A e_A{}^P=T_{MN}{}^P=2\Gamma^P_{[MN]}, \tag{B.8}$$

which becomes 0 for the standard Riemannian general relativity Christoffel connection $\Gamma^P_{(MN)}$. This constraint and the metricity condition $D_M g_{NP} = 0$ (also used in Riemannian geometry) uniquely defines ω by

$$T^A = De^A = de^A + e^B \wedge \omega^A{}_B = de^A - 2\Omega^A = 0, \tag{B.9}$$

which defines the 2-form Ω^A by

$$\Omega^A = \frac{1}{2!} e^B \wedge e^C \Omega_{CB}{}^A = -\frac{1}{2} e^B \wedge \omega^A{}_B = \frac{1}{2} e^B \wedge e^C \omega_{CB}{}^A. \tag{B.10}$$

We can thus read of the components of Ω^A as

$$\Omega_{CBA} = \omega_{[CB]A} = \frac{1}{2} \left(\omega_{CBA} - \omega_{BCA} \right) \tag{B.11}$$

which gives

$$\begin{aligned} \Omega_{CBA} + \Omega_{ACB} - \Omega_{BAC} = \frac{1}{2} \big(& \omega_{CBA} - \omega_{BCA} + \omega_{ACB} - \omega_{CAB} \\ & - \omega_{BAC} + \omega_{ABC} \big) = \omega_{CBA}, \end{aligned} \tag{B.12}$$

where we have used the antisymmetry of the spin connection $\omega_{CBA} = -\omega_{CAB}$. This antisymmetry is obvious if one considers the metricity condition of the Lorentz metric $D\eta_{AB} = 0$, giving

$$D\eta_{AB} = d\eta_{AB} + \eta_{CA} \omega_B{}^C + \eta_{CB} \omega_A{}^C = \omega_{BA} + \omega_{AB} = 0 \tag{B.13}$$

Curvature Form the 2-form[1]

$$\Theta^A{}_B = d\omega^A{}_B - \omega^A{}_C \omega^C{}_B \tag{B.14}$$

which transforms as (use (A.41) to cancel the terms)

$$\begin{aligned} \Theta \to & \Theta' = d\omega' - \omega'\omega' \\ & = d\left(\Lambda\omega\Lambda^{-1} - d\Lambda\Lambda^{-1} \right) - \left(\Lambda\omega\Lambda^{-1} - d\Lambda\Lambda^{-1} \right) \left(\Lambda\omega\Lambda^{-1} - d\Lambda\Lambda^{-1} \right) \\ & = \Lambda\omega d\Lambda^{-1} + \Lambda d\omega \Lambda^{-1} - d\Lambda\omega\Lambda^{-1} - d\Lambda d\Lambda^{-1} \\ & - \Lambda\omega\omega\Lambda^{-1} + \Lambda\omega\Lambda^{-1} d\Lambda\Lambda^{-1} + d\Lambda\omega\Lambda^{-1} - d\Lambda\Lambda^{-1} d\Lambda\Lambda^{-1} \\ & = \Lambda \left(d\omega - \omega\omega \right) \Lambda^{-1} = \Lambda\Theta\Lambda^{-1} \end{aligned} \tag{B.15}$$

under Lorentz transformations i.e. $\Theta^B{}_A$ is a Lorentz tensor. The antisymmetry of ω_{AB} makes Θ_{BA} antisymmetric in $A \leftrightarrow B$. The definitions of T^A (B.9) and Θ^B_A (B.14) are usually denoted as the Maurer Cartan structure equations (c.f. Section **??**). Since Θ is a 2-form it can also be written on the form

$$\Theta^B{}_A = \frac{1}{2} e^C \wedge e^D \Theta_{DC}{}^B{}_A = \frac{1}{2} dx^P \wedge dx^Q e_N{}^B e_A{}^M R_{QP}{}^N{}_M \tag{B.16}$$

where $\Theta_{DC}{}^B{}_A$ is identified to be the Riemann tensor with flat indices (use (B.7)).

[1]Usually Θ is defined with a plus sign, but for the Bianchi identity $d\Theta$ to imply $D\Theta = 0$, we need the minus sign with our superspace convention of external derivatives acting from the right.

B.1.2 Reduction of the Ricci scalar

Here we will calculate the Ricci scalar after Kaluza-Klein compactification on T^n, for a general n. From the Kaluza-Klein Ansatz made in (**??**) we found that the vielbeins can be decomposed (**??**) as

$$\hat{e}^A = (\hat{e}^a, \hat{e}^i) = (e^a, A^{1i} + e^i) \tag{B.17}$$

and in our Kaluza-Klein analysis in Section **??** we found that, for our considerations, we can set all fields to be independent on the compactified coordinates x^m. This effectively means that all derivatives ∂_m with respect to x^m will be zero and we get the exterior derivative in $\hat{D}$ dimensions

$$\hat{d} = d = dx^\mu \partial_\mu = e^a \partial_a \tag{B.18}$$

The torsion (B.9) of pure gravity is zero, which gives

$$\hat{d}\hat{e}^A = \hat{e}^B \wedge \hat{e}^C \hat{\Omega}_{CB}{}^A \tag{B.19}$$

Note that inverting $e^i = dx^m e_m{}^i$ gives $dx^m = e^i e_i{}^m$ and calculate both left and right hand side of (B.19) (use (**??**) as definition of F^1)

$$\begin{aligned}
\text{LHS: } \hat{d}\hat{e}^A &= \partial_\mu \left(e^a, \hat{e}^i\right) \wedge dx^\mu = \left(de^a, \partial_\mu \left(A^{1i} + e^i\right) \wedge dx^\mu\right) \\
&= \left(e^b \wedge e^c \Omega_{cb}{}^a, \partial_b \left(A^{1i} + e^i\right) \wedge e^b\right) \\
&= \left(\Omega^a, \partial_b \left(e_m{}^i A_c^{1m}\right) e^c \wedge e^b + \partial_b e^i \wedge e^b\right) \\
&= \left(\Omega^a, \frac{1}{2} e_m{}^i F_{cb}^1{}^m e^b \wedge e^c + \partial_b e_m{}^i A_c^{1m} e^c \wedge e^b + \partial_b e_m{}^i dx^m \wedge e^b\right) \\
&= \left(\Omega^a, F^{1i} + A^{1m} de_m{}^i + \partial_b e_m{}^i e_j{}^m e^j \wedge e^b\right) \\
&= \left(\Omega^a, F^{1i} + A^{1m} de_m{}^i + \partial_b e_m{}^i e_j{}^m \hat{e}^j \wedge e^b - \partial_b e_m{}^i e_j{}^m A^{1j} \wedge e^b\right) \\
&= \left(\Omega^a, F^{1i} + \partial_b e_m{}^i e_j{}^m \hat{e}^j \wedge e^b\right)
\end{aligned} \tag{B.20}$$

$$\begin{aligned}
\text{RHS: } \hat{e}^B \wedge \hat{e}^C \Omega_{CB}{}^A &= \left(e^b, \hat{e}^j\right) \wedge \left(e^c, \hat{e}^k\right) \Omega_{CB}{}^A \\
&= e^b \wedge e^c \hat{\Omega}_{cb}{}^A + e^b \wedge \hat{e}^k \hat{\Omega}_{kb}{}^A + \hat{e}^j \wedge e^c \hat{\Omega}_{cj}{}^A + \hat{e}^j \wedge \hat{e}^k \hat{\Omega}_{kj}{}^A \\
&= e^b \wedge e^c \hat{\Omega}_{cb}{}^A + 2e^b \wedge \hat{e}^k \hat{\Omega}_{kb}{}^A + \hat{e}^j \wedge \hat{e}^k \hat{\Omega}_{kj}{}^A
\end{aligned} \tag{B.21}$$

Comparing the two sides, starting with A=a

$$e^b \wedge e^c \Omega_{cb}{}^a = e^b \wedge e^c \hat{\Omega}_{cb}{}^a + 2e^b \wedge \hat{e}^k \hat{\Omega}_{kb}{}^a + \hat{e}^j \wedge \hat{e}^k \hat{\Omega}_{kj}{}^a \tag{B.22}$$

and then with A=i

$$\frac{1}{2} F_{cb}^1{}^i e^b \wedge e^c - \partial_b e_m{}^i e_k{}^m e^b \wedge \hat{e}^k = e^b \wedge e^c \hat{\Omega}_{cb}{}^i + 2e^b \wedge \hat{e}^k \hat{\Omega}_{kb}{}^i + \hat{e}^j \wedge \hat{e}^k \hat{\Omega}_{kj}{}^i \tag{B.23}$$

gives the following components of $\hat{\Omega}^A$

$$\begin{aligned}
&\hat{\Omega}_{cb}{}^{a} = \Omega_{cb}{}^{a}, && \hat{\Omega}_{cb}{}^{i} = \frac{1}{2}F^{1}_{cb}{}^{i}, \\
&\hat{\Omega}_{kb}{}^{a} = -\hat{\Omega}_{bk}{}^{a} = 0, && \hat{\Omega}_{kb}{}^{i} = -\hat{\Omega}_{bk}{}^{i} = -\frac{1}{2}\partial_b e_m{}^{i} e_k{}^{m}, \\
&\hat{\Omega}_{kj}{}^{a} = 0, && \hat{\Omega}_{kj}{}^{i} = 0
\end{aligned} \tag{B.24}$$

Thus the components of $\hat{\omega}$ becomes, using (B.11)

$$\begin{aligned}
&\hat{\omega}_{cba} = \Omega_{cba} + \Omega_{acb} - \Omega_{bac} = \omega_{cba} \\
&\hat{\omega}_{cbi} = -\hat{\omega}_{cib} = \frac{1}{2}F^{1}_{cbi} + 0 + 0 = \frac{1}{2}F^{1}_{cbi} \\
&\hat{\omega}_{iba} = 0 + 0 - \frac{1}{2}F^{1}_{bai} = -\frac{1}{2}F^{1}_{bai} \\
&\hat{\omega}_{cji} = \frac{1}{2}\partial_c e_{mi} e_j{}^{m} - \frac{1}{2}\partial_c e_{mj} e_i{}^{m} - 0 = \partial_c e_{m[i} e_{j]}{}^{m} \\
&\hat{\omega}_{jbi} = -\hat{\omega}_{jib} = -\frac{1}{2}\partial_b e_{mi} e_j{}^{m} + 0 - \frac{1}{2}\partial_b e_{mj} e_i{}^{m} = -\partial_b e_{m(j} e_{i)}{}^{m} \\
&\hat{\omega}_{kji} = 0 + 0 - 0 = 0
\end{aligned} \tag{B.25}$$

where we have used that flat indices can be raised and lowered inside the partial derivatives. The components of $\hat{\omega}$ as a 1-form

$$\hat{\omega}_{BA} = \hat{e}^{C}\hat{\omega}_{CBA} = \hat{e}^{c}\hat{\omega}_{cBA} + \hat{e}^{i}\hat{\omega}_{iBA} \tag{B.26}$$

becomes

$$\begin{aligned}
&\hat{\omega}_{ba} = \hat{e}^{c}\omega_{cba} - \frac{1}{2}\hat{e}^{i}F^{1}_{bai} = \omega_{ba} - \frac{1}{2}\hat{e}^{i}F^{1}_{bai} \\
&\hat{\omega}_{ja} = -\hat{\omega}_{aj} = \frac{1}{2}\hat{e}^{c}F^{1}_{acj} + \hat{e}^{i}\partial_a e_{m(j} e_{i)}{}^{m} \\
&\hat{\omega}_{kj} = \hat{e}^{c}\partial_c e_{m[j} e_{k]}{}^{m} + 0 = \hat{e}^{c}\partial_c e_{m[j} e_{k]}{}^{m}.
\end{aligned} \tag{B.27}$$

From (B.16) we have

$$\begin{aligned}
\hat{\Theta}_{BA} &= \frac{1}{2}\hat{e}^{C} \wedge \hat{e}^{D}\hat{R}_{DCBA} \\
&= \frac{1}{2}\hat{e}^{c} \wedge \hat{e}^{d}\hat{R}_{dcBA} + \hat{e}^{c} \wedge \hat{e}^{i}\hat{R}_{icBA} + \frac{1}{2}\hat{e}^{i} \wedge \hat{e}^{j}\hat{R}_{jiBA}
\end{aligned} \tag{B.28}$$

where we have used the antisymmetry in the first two indices of the Riemann tensor $R_{DCBA} = -R_{CDBA}$ to put the cross terms together. To calculate the Ricci scalar from the Riemann tensor

$$\hat{R} = \hat{R}_{BA}{}^{BA} = \hat{R}_{ba}{}^{ba} + 2\hat{R}_{ja}{}^{ja} + \hat{R}_{ji}{}^{ji} \tag{B.29}$$

we need the $\hat{R}_{dcba}$, $\hat{R}_{lcja}$ and R_{lkji} components of $\hat{R}_{DCBA}$. These can be obtained from the definition of the 2-form $\hat{\Theta}_{BA}$, (B.14) (remember that we have already calculated $d\hat{e}^i$ in (B.20))

$$
\begin{aligned}
&\hat{\Theta}_{ba} = d\hat{\omega}_{ba} - \hat{\omega}_b{}^c \wedge \hat{\omega}_{ca} - \hat{\omega}_b{}^i \wedge \hat{\omega}_{ia} = \\
&= d\omega_{ba} - \frac{1}{2} d\left(\hat{e}^i F^1_{bai}\right) - \left(\omega_b{}^c - \frac{1}{2}\hat{e}^i F^{1c}_{b\ i}\right) \wedge \left(\omega_{ca} - \frac{1}{2}\hat{e}^j F^1_{caj}\right) \\
&- \left(\frac{1}{2}\hat{e}^c F^{1\ i}_{cb} - \hat{e}^{(k}\partial_b e_n{}^{i)} e_k{}^n\right) \wedge \left(\frac{1}{2}\hat{e}^d F^1_{adi} + \hat{e}^j \partial_a e_{m(i} e_{j)}{}^m\right) \\
&= d\omega_{ba} + \omega_b{}^c \omega_{ca} - \frac{1}{2} d\hat{e}^i F^1_{bai} - \frac{1}{2}\hat{e}^i dF^1_{bai} + \frac{1}{2}\omega_b{}^c \wedge \hat{e}^j F^1_{caj} + \frac{1}{2}\hat{e}^i F^{1c}_{b\ i} \wedge \omega_{ca} \\
&- \frac{1}{2}\hat{e}^i F^{1c}_{b\ i} \wedge \frac{1}{2}\hat{e}^j F^1_{caj} - \frac{1}{2}\hat{e}^c F^{1\ i}_{cb} \wedge \frac{1}{2}\hat{e}^d F^1_{adi} + \hat{e}^{(k}\partial_b e_n{}^{i)} e_k{}^n \wedge \frac{1}{2}\hat{e}^d F^1_{adi} \\
&- \frac{1}{2}\hat{e}^c F^{1\ i}_{cb} \wedge \hat{e}^j \partial_a e_{m(i} e_{j)}{}^m + \hat{e}^{(k}\partial_b e_n{}^{i)} e_k{}^n \wedge \hat{e}^j \partial_a e_{m(i} e_{j)}{}^m \\
&= \Theta_{ba} - \frac{1}{2}\left(\frac{1}{2} F^{1\ i}_{dc}\hat{e}^c \wedge \hat{e}^d + \partial_c(e_m{}^i) e_j{}^m \hat{e}^j \wedge \hat{e}^c\right) F^1_{bai} - \frac{1}{4}\hat{e}^i \wedge \hat{e}^j F^{1c}_{b\ i} F^1_{caj} \\
&- \frac{1}{2}\hat{e}^i \wedge \left(dF^1_{bai} + \omega_b{}^c F^1_{cai} - F^1_{bci}\omega^c{}_a\right) - \frac{1}{4}\hat{e}^c \wedge \hat{e}^d F^{1\ i}_{cb} F^1_{adi} \\
&- \frac{1}{2}\hat{e}^d \wedge \hat{e}^{(k}\partial_b e_n{}^{i)} e_k{}^n F^1_{adi} - \frac{1}{2}\hat{e}^c \wedge \hat{e}^j F^{1\ i}_{cb} \partial_a e_{m(i} e_{j)}{}^m + 0
\end{aligned}
\tag{B.30}
$$

For our purposes, we only need $\hat{\Theta}_{dcba} = \hat{R}_{dcba}$, which can be read off from what is multiplying $\frac{1}{2}\hat{e}^c \wedge \hat{e}^d$ in $\hat{\Theta}_{ba}$, i.e.

$$
\hat{R}_{dcba} = R_{dcba} - \frac{1}{2} F^{1\ i}_{dc} F^1_{bai} - \frac{1}{2} F^{1\ i}_{a[d} F^1_{c]bi}. \tag{B.31}
$$

Next we attack

$$
\begin{aligned}
\hat{\Theta}_{ja} &= d\hat{\omega}_{ja} - \hat{\omega}_j{}^b \wedge \hat{\omega}_{ba} - \hat{\omega}_j{}^i \wedge \hat{\omega}_{ia} \\
&= d\left(\frac{1}{2}\hat{e}^c F^1_{acj} + \hat{e}^i \partial_a e_{m(j} e_{i)}{}^m\right) \\
&- \left(\frac{1}{2}\hat{e}^c F^{1b}{}_{cj} + \hat{e}^i \partial^b e_{m(j} e_{i)}{}^m\right) \wedge \left(\omega_{ba} - \frac{1}{2}\hat{e}^k F^1_{bak}\right) \\
&- \left(\hat{e}^c \partial_c e_{m[k} e_{j]}{}^m \delta^{ik}\right) \wedge \left(\frac{1}{2}\hat{e}^d F^1_{adi} + \hat{e}^l \partial_a e_{n(i} e_{l)}{}^n\right)
\end{aligned}
$$

$$
\begin{aligned}
&= \frac{1}{2}\partial_d\left(\hat{e}^c F^1_{acj}\right)\wedge\hat{e}^d + \left(\frac{1}{2}F^1_{cb}{}^i\hat{e}^b\wedge\hat{e}^c + \partial_b e_n{}^i e_k{}^n\hat{e}^k\wedge\hat{e}^b\right)\partial_a e_{m(j}e_{i)}{}^m \\
&+ \partial_c\left(\partial_a e_{m(j}e_{i)}{}^m\right)\hat{e}^i\wedge\hat{e}^c - \frac{1}{2}F^{1b}{}_{cj}\hat{e}^c\wedge\omega_{ba} - \partial^b e_{m(j}e_{i)}{}^m\hat{e}^i\wedge\omega_{ba} \\
&+ \frac{1}{4}F^{1b}{}_{cj}F^1_{bak}\hat{e}^c\wedge\hat{e}^k + \frac{1}{2}\partial^b e_{m(j}e_{i)}{}^m F^1_{bak}\hat{e}^i\wedge\hat{e}^k \\
&- \frac{1}{2}\partial_c e_{m[k}e_{j]}{}^m F^1_{ad}{}^k\hat{e}^c\wedge\hat{e}^d - \partial_c e_{m[k}e_{j]}{}^m\delta^{ik}\partial_a e_{n(i}e_{l)}{}^n\hat{e}^c\wedge\hat{e}^l
\end{aligned}
\tag{B.32}
$$

Use $T_a = \partial_a e_{m(j}e_{i)}{}^m$ and note that

$$
DT_a = dT_a + T_b\wedge\omega_a{}^b = \partial_c\left(\partial_a e_{m(j}e_{i)}{}^m\right)\hat{e}^c + \partial_b e_{m(j}e_{i)}{}^m\omega_a{}^b,
\tag{B.33}
$$

so we can rewrite term 3 and term 5 as

$$
\partial_c\left(\partial_a e_{m(j}e_{i)}{}^m\right)\hat{e}^i\wedge\hat{e}^c - \partial^b e_{m(j}e_{i)}{}^m\hat{e}^i\wedge\omega_{ba} = D_c\left(\partial_a e_{m(j}e_{i)}{}^m\right)\hat{e}^i\wedge\hat{e}^c.
\tag{B.34}
$$

We only need $\hat{\Theta}_{lcja} = \hat{R}_{lcja} = -\hat{R}_{clja}$, which can be read off from what is multiplying $\hat{e}^c\wedge\hat{e}^l$ in $\hat{\Theta}_{ja}$ (no factor $\frac{1}{2}$ for the crossterm), i.e.

$$
\begin{aligned}
\hat{R}_{lcja} = &-e_l{}^n\partial_c e_n{}^i\partial_a e_{m(j}e_{i)}{}^m - D_c\left(\partial_{|a}e_{m|(l}e_{j)}{}^m\right) \\
&+ \frac{1}{4}F^1_{bal}F^{1b}{}_{cj} - \partial_a e_{m(i}e_{l)}{}^m\partial_c e_{n[k}e_{j]}{}^n\delta^{ik}.
\end{aligned}
\tag{B.35}
$$

Next

$$
\begin{aligned}
&\hat{\Theta}_{ji} = d\hat{\omega}_{ji} - \hat{\omega}_j{}^b\wedge\hat{\omega}_{bi} - \hat{\omega}_j{}^k\wedge\hat{\omega}_{ki} = \\
&= d\left(\hat{e}^c\partial_c e_{m[i}e_{j]}{}^m\right) - \hat{e}^c\partial_c e_{m[l}e_{j]}{}^m\delta^{kl}\wedge\hat{e}^d\partial_d e_{m[i}e_{k]}{}^m \\
&- \left(\frac{1}{2}\hat{e}^c F^{1b}{}_{cj} + \hat{e}^k\partial^b e_{m(j}e_{k)}{}^m\right)\wedge\left(\frac{1}{2}\hat{e}^c F^1_{cbi} - \hat{e}^l\partial_b e_{n(i}e_{l)}{}^n\right).
\end{aligned}
\tag{B.36}
$$

Now we only need $\hat{\Theta}_{lkji} = \hat{R}_{lkji}$, which can be read off from what is multiplying $\frac{1}{2}\hat{e}^k\wedge\hat{e}^l$ in $\hat{\Theta}_{ji}$, i.e.

$$
\hat{R}_{lkji} = -2\partial_b e_{n(i}e_{l)}{}^n\partial^b e_{m(j}e_{k)}{}^m
\tag{B.37}
$$

where we use our heads to remember the antisymmetry in $l\leftrightarrow k$ and $j\leftrightarrow i$.

And at last we can happily calculate the Ricci scalar as a contraction of the Riemann tensor

$$\begin{aligned}
\hat{R} &= \hat{R}_{BA}{}^{BA} = \hat{R}_{ba}{}^{ba} + 2\hat{R}_{ja}{}^{ja} + \hat{R}_{ji}{}^{ji} \\
&= R - \frac{1}{2}F^1_{bai}F^{1bai} - \frac{1}{4}\left(F^{1a}{}_{b}{}^{i}F^{1b}{}_{a}{}_{i} - F^{1a}{}_{a}{}^{i}F^{1b}{}_{b}{}_{i}\right) \\
&+ 2\Big\{ - e_j{}^n\partial_a e_n{}^{(i}\partial^{|a|}e_m{}^{j)}e_i{}^m - D_a\left(\partial^a e_{|m|(j}e_{k)}{}^m\delta^{jk}\right) \\
&+ \frac{1}{4}F^{1a}_b{}_j F^{1b}{}_a{}^j - \partial^a e_{m(i}e_{j)}{}^m\partial_a e_n{}^{[i}e^{j]n}\Big\} \\
&- 2\partial_b e_{n(i'}e_{[j)}{}^n\partial^b e_{|m|(i]}e_{j')}{}^m\delta^{j[j'}\delta^{i']i} \\
&= R + F^1_{bai}F^{1bai}\left\{-\frac{1}{2} - \frac{1}{4} + \frac{1}{2}\right\} \\
&- 2e_j{}^n\partial_a e_n{}^{(i}\partial^{|a|}e_m{}^{j)}e_i{}^m - 2D_a\left(e_i{}^m\partial^a e_{mi}\right) - 0 \\
&- \left(\partial_a e_{n(j}e_{j')}{}^n\partial^a e_{m(i'}e_{i)}{}^m - \partial_a e_{n(j}e_{i')}{}^n\partial^a e_{m(j'}e_{i)}{}^m\right)\delta^{jj'}\delta^{i'i} \\
&= R - \frac{1}{4}G_{mn}F^1_{ba}{}^m F^{1ban} - \partial_a e_n{}^{(i}e^{j)n}e_i{}^m\partial^a e_{mj} \\
&- 2D_a\left(e_i{}^m\partial^a e_m{}^i\right) - \partial_a e_n{}^j e_j{}^n\partial^a e_m{}^i e_i{}^m
\end{aligned} \tag{B.38}$$

B.1.3 The Ricci scalar in the Einstein frame

Consider the Ricci scalar (use (B.14) to get $R_{NM}{}^B{}_A$)

$$\begin{aligned}
R(e,\omega) &= e_B{}^N e^{AM}R_{NM}{}^B{}_A = \eta^{DB}\eta^{CA}\Theta_{DCBA} \\
&= \eta^{DB}\eta^{CA}\left\{2\partial_{[D}\omega_{C]BA} + 2\omega_{[D|B|}{}^E\omega_{C]EA}\right\},
\end{aligned} \tag{B.39}$$

under a metric scaling

$$\begin{aligned}
&e_M{}^A = e^{-s\varphi}\tilde{e}_M{}^A, && e_A{}^M = e^{s\varphi}\tilde{e}_A{}^M \\
&g_{MN} = e^{-2s\varphi}\tilde{g}_{MN}, && g^{MN} = e^{2s\varphi}\tilde{g}^{MN} \\
&A^2_{(P)} = g^{M_1N_1}\ldots g^{M_pN_p}A_{M_p\ldots M_1}A_{N_p\ldots N_1} = e^{2sp\varphi}\tilde{A}^2_{(P)}
\end{aligned} \tag{B.40}$$

where the indices of p-forms $A_{(p)}$ will be naturally downstairs curved and raised using the transformed metric $\tilde{g}^{MN}$. In particular we should be careful not moving around the indices on the derivatives too much. We can use (B.7) to express the spin connection $\omega_M{}^A{}_B$ in terms of the Christoffel connection. After some work we get R as

$$R = e^{2s\varphi}\left\{\tilde{R} + 2s\,(d-1)\,\Box\varphi + s^2(d-1)(d-2)\partial\varphi^2\right\} \tag{B.41}$$

Note that by using (A.38) and that s is chosen to cancel the prefactor of $\tilde{R} = R_E$ in the Einstein frame, the second term disappears when integrating

$$\int d^d x \sqrt{|g_E|} 2s(d-1) D_\mu D^\mu \varphi = 2s(d-1) \int d^d x \partial_\mu \left(\frac{1}{\sqrt{|g_E|}} D^\mu \varphi \right) = 0 \qquad \text{(B.42)}$$

C

128 bit integer arithmetics

To solve the duality equations in chapter **??** we need to expand the equations to very high orders, implying very large coefficients due to the binomial factors in the expansion of $(\det G)^{-1/2}$. The current PC:s use a 32 bit word length ($\sim$ 9 decimal digits) and Delphi, the programming language we use, has support for 64 bit integers ($\sim$ 19 decimal digits) in its system. To handle sufficiently large expansions with numbers larger than this we need to include 128 bit integer representations ($\sim$ 39 decimal digits) and some basic arithmetic operations on these. The easiest way to include such large integers would be to borrow the work of someone else, but to our astonishment we can't find such a work on the internet. Thus we have to write it ourselves and include it in this thesis so future solvers of grandiose duality equations can have the chance to find it.

To implement 128 bit integers is one of the cases where it is actually motivated to write the code directly using x86 assembler. To begin with we need the code to be fast since we will use the 128 bit arithmetic operations a lot. An even more important motivation is that we need operations not accessible by ordinary code, e.g. shift over 32-bit boundaries, bit string scans, addition with carry bit and subtraction with borrow bit. Of course we could do this in a complicated way using standard operations in Delphi, but that would give unnecessarily big performance flaws and would probably be even harder to implement than an assembler analogue. To get started it is a good idea to check how the compiler translates the 64 bit integer operations to machine code and try to generalize to 128 bits (The Intel Architecture Manual[28] will be in handy).

C.1 General assembler guidelines

One problem when writing code in assembler is that there is always a big number of choices like which instructions to use and how to order them. Each problem can be

solved in so many ways and it is hard to know which one is the fastest. We will try to follow the general guidelines in [29] to write fast integer code. The main points are

- Minimize branch count, i.e. avoid conditional jumps as far as possible. For the branches we cannot avoid, we should (if there is a choice) branch for the least probable condition result and fall through for the most probable result in an if statement and put the condition last in loops. This is because the branch predictor assumes, if there is no branch history for the current instruction, that a fall through will occur if the destination lies at a higher memory address and that a branch jump will occur when the destination is a lower memory address.

- Minimize memory accesses count, use the registers to keep variables as long as they are needed. Instructions involving only registers are in general faster executed than instructions involving memory accesses.

- Pair pairable instructions so the processor can run them simultaneously in the two integer pipelines and do not pair other instructions, since that would be unnecessary work (Use data in [29] to see which instructions are pairable and how many μops each instruction needs). Further do not use registers that has been changed the previous instruction, doing that prevents pairing.

- For fast cache memory access we should align our 32 byte coefficients to memory addresses divisible by 32 (the cache is read in blocks of 32 bytes), but that would increase either the memory size or the complexity of the data storage, so we store over cache boundaries with a slight (the data is still 4 byte aligned) performance loss.

An additional point should be to use smart algorithms, which of course can have greater effects than any possible optimizations using the points stated above.

C.2 The Int128 type

To represent the 128 bit integers we use the type Int128 which is simply defined as a structure of four succeeding 32 bit integers. Furthermore we define the type pInt128 as a pointer to an Int128.

```
type
  Int128 = record
    data1,data2,data3,data4:Int32;
   end;
  pInt128 = ^Int128;
```

We use the Little endian storage format, meaning that the least significant Int32 (= data1) is at the lowest memory address, the bytes within each Int32 are by machine default also Little endian[1]. The integers should be signed so we use two's complement to store the integers in memory where the most significant bit in data4 is used to indicate sign, i.e. the positive numbers $0, 1, \ldots, 2^{127}-1$ are represented by (in hexadecimal code) $000\ldots00, 000\ldots01, \ldots, 7FF\ldots FF$ and the negative numbers $-1, -2, \ldots, -2^{127}$ becomes $FFF\ldots FF, FFF\ldots FE, \ldots, 800\ldots00$. Note that this is not the order in which the integers are actually stored in memory. We define the following 128 bit functions to handle all operations on Int128 needed:

```
function IsZero128(x:pInt128):boolean;
function IsOne128(x:pInt128):boolean;
function IsNeg128(x:pInt128):boolean;
function IsPos128(x:pInt128):boolean;
function IsEqual128(x,y:pInt128):boolean;
procedure Neg128(x:pInt128);
procedure Abs128(x:pInt128);
procedure Add_128_128(x,y:pInt128);
procedure Add_128_128_128(res,x,y:pInt128);
procedure Mul_128_128_128(res,x,y:pInt128);
procedure ReduceFraction128(frac:pointer);
```

plus some conversion routines between Int128, Int32 and (text)strings. The first five conditional functions are easy to implement and we illustrate them all with one example

```
function IsPos128(x:pInt128):boolean;
// Result:=x > 0
// Params: eax = x; al = Result;
asm
 cmp [eax+12], 0     // Compare x.data4 - 0 to 0
 jnz @returnSign     // Return sign if x.data4 <> 0
 cmp [eax+8],0       // Compare x.data3 - 0 to 0
 ja @returnTrue      // Return true if unsigned x.data3 > 0
 cmp [eax+4],0       // Compare x.data2 - 0 to 0
 ja @returnTrue      // Return true if unsigned x.data3 > 0
 cmp [eax],0         // Compare x.data1 - 0 to 0
 ja @returnTrue      // Return true if unsigned x.data3 > 0
                     // Return false on fallthrough
 mov al, 0           // Set false result
 ret                 // Return from function
@returnSign:
 setg al             // Set result = signed positive
 ret                 // Return from function
@returnTrue:
 mov al, 1           // Set true result
end;
```

which returns 1 in register al if the argument x is larger than 0 and returns 0 otherwise. The negations in Neg128 and Abs128 and the additions in Add_128_128 (Add with result in first parameter) and Add_128_128_128 are almost as easy. We illustrate them all with one example

```
procedure Neg128(x:pInt128);
// var x:=-x;
// Params: eax = @x;
asm
 xor ecx, ecx        // ecx:=0
 neg [eax]           // x.data1:=- x.data1, sets flags
 sbb ecx, [eax+4]    // ecx:=0 - x.data2 - Borrow
 mov edx, 0          // edx:=0, preserves flags unlike xor
 mov [eax+4], ecx    // Write back the negation of x.data2
 sbb edx, [eax+8]    // edx:=0 - x.data3 - Borrow
 mov ecx, 0          // ecx:=0
 mov [eax+8], edx    // Write back the negation of x.data3
 sbb ecx, [eax+12]   // ecx:=0 - x.data4 - Borrow
{$IFOPT Q+}          // If overflow check is on
 jno @noOverflow
 call IntNegOverflow // Cannot negate $800000...000 =>error
```

[1]The bits within each byte is ordered least significant bit at highest address, but we will never see this when using machine instructions

```
@noOverflow:
{$ENDIF}
 mov [eax+12], ecx   // Write back the negation of x.data4
end;
```

that negates the value of x. The additions are done similarly with one "add" and 3 "adc" (add with carry) instructions.

Now there are only 2 functions left, but there is a reason they stand last. These functions will be more complex and we start with the multiplication function Mul_128_128_128. For the multiplication we finally, after 16 years of impatient wait, find an application for the elementary school second grade multiplication algorithm, which in the 2^{32} base between the 128 bit integers $x = x_4 : x_3 : x_2 : x_1$ and $y = y_4 : y_3 : y_2 : y_1$ becomes

$$\begin{array}{rr} & x_4\ x_3\ x_2\ x_1 \\ & \cdot\ \ y_4\ y_3\ y_2\ y_1 \\ \hline & \overline{c_6\ c_5\ c_4}\ c_3\ c_2\ c_1 \\ & \quad\quad\quad\quad z_{41} z_{31} z_{21} z_{11} \\ & \quad\quad\quad z_{42} z_{32} z_{22} z_{12} \\ & \quad\quad z_{43} z_{33} z_{23} z_{13} \\ + & \underline{z_{44}\ z_{34}\ z_{24}}\ z_{14} \\ \hline & r_7\ r_6\ r_5\ r_4\ r_3\ r_2\ r_1 \end{array} \tag{C.1}$$

where $z_{ij} = x_i y_j$, c_i the carry from the addition on the previous column and r_i is the i:th result component in the 2^{32} base given by addition of the column over it. For the product to be a valid Int128 we must have $r_7 = r_6 = r_5 = 0$, i.e. all symbols in the dashed box must be zeroes (or hexadecimal FFFFFFFF:s, indicating the sign extension of a negative number). Furthermore the non sign information of the product (bit $0 \ldots 126$) isn't allowed to change the value of the 127:th (sign) bit. Checking this product for overflow (calculating and checking all the symbols in the dashed box plus checks for overflow to the 127:th bit) would almost take as long time as calculating the valid symbols. Therefore we, like Delphi in the 64 bit case[2], ignore the overflow bits, but unlike Delphi we at least make a sign consistency test, checking if the result has the correct sign knowing the operands signs. This should detect half the overflows of multiplications between completely random Int128 operands. In assembler the algorithm (C.1) becomes

```
procedure MulPro_128_128_128(res,x,y:pInt128);
// dst:=x * y;
// Params: eax = @dst; edx = @x; ecx = @y;
asm
// Set up the stack frame (x and y might change if res=x|y)
 push esi
 push ebx
 push [ecx+12]
 push [ecx+8]
 push [ecx+4]
 push [ecx]
 push [edx+12]
 push [edx+8]
 push [edx+4]
 push [edx]

// y = (y4:y3,y2,y1) = [ESP+28]:[ESP+24]:[ESP+20]:[ESP+16]
// x = (x4:x3:x2:x1) = [ESP+12]:[ESP+8]:[ESP+4]:[ESP]
```

[2]It would have saved us from much trouble if they had actually told us in the documentation rather than in the source code that the overflow check on most 64 bit integer operations wasn't supported yet.

```
 mov esi, eax        // esi:=@res
 mov eax, [esp+16]   // eax:=y1
 mul [esp]           // edx:eax:=y1*x1
 mov ebx, edx        // ebx:=(y1*x1).hi,ebx now stores res2
 mov [esi], eax      // @res^.data1:=(y1*x1).lo

 mov eax, [esp+16]   // eax:=y1
 xor ecx, ecx        // ecx:=0
 mul [esp+4]         // edx:eax:=y1*x2
 add ebx, eax        // ebx:=(y1*x1).hi+(y1*x2).lo
 mov eax, [esp+20]   // eax:=y2
 adc ecx, edx        // ecx:=(y1*x2).hi+carry

 mul [esp]           // edx:eax:=y2*x1
 add ebx, eax        // ebx:=(y1*x1).hi+(y1*x2).lo
                     //   +(y2*x1).lo
 adc ecx, 0          // ecx:=(y1*x2).hi+carry, o.f. impos.
 mov [esi+4], ebx    // @res^.data2:=(y1*x1).hi+(y1*x2).lo
                     //   +(y2*x1).lo
 xor ebx, ebx        // ebx:=0, ebx is now storage for res4
 mov eax, [esp+16]   // eax:=y1
 add ecx, edx        // ecx:=(y1*x2).hi+(x1*y2).hi
 adc ebx, 0          // ebx:=0+carry

 mul [esp+8]         // edx:eax:=y1*x3
 add ecx, eax        // ecx:=(y1*x2).hi+(x1*y2).hi
                     //   +(y1*x3).lo
 mov eax, [esp+20]   // eax:=y2
 adc ebx, edx        // ebx:=(y1*x3).hi+carry

 mul [esp+4]         // edx:eax:=y2*x2
 add ecx, eax        // ecx:=(y1*x2).hi+(x1*y2).hi
                     //   +(y1*x3).lo+(y2*x2).lo
 mov eax, [esp+24]   // eax:=y3
 adc ebx, edx        // ebx:=(y1*x3).hi+(y2*x2).hi+carry
                     //   (Ignore overflow)
 mul [esp]           // edx:eax:=y3*x1
 add ecx, eax        // ecx:=(y1*x2).hi+(x1*y2).hi
                     //   +(y1*x3).lo+(y2*x2).lo+(y3*x1).lo
 adc ebx, edx        // ecx:=(y1*x3).hi+(y2*x2).hi
                     //   +(y3*x1).hi+Carry (Ign. overflow)
 mov eax, [esp+12]   // eax:=x4
 mov [esi+8], ecx    // @res^.data3:=(y1*x2).hi+(x1*y2).hi
                     //   +(y1*x3).lo+(y2*x2).lo+(y3*x1).lo

{$IFOPT Q+}          // If overflow check is on
 mov ecx, eax        // ecx:=x4, check sign using ecx
{$ENDIF}
```

```
 mul [esp+16]        // edx:eax:=x4*y1
 add ebx, eax        // ebx:=(y1*x3).hi+(y2*x2).hi
                     //   +(y3*x1).hi+(x4*y1).lo (Ign o.f.)
 mov eax, [esp+20]   // eax:=y2
 mul [esp+8]         // edx:eax:=y2*x3
 add ebx, eax        // ebx:=(y1*x3).hi+(y2*x2).hi
                     //   +(y3*x1).hi+(x4*y1).lo+(y2*x3).lo
                     // (Ignore overflow)
 mov eax, [esp+24]   // eax:=y3
 mul [esp+4]         // edx:eax:=y3*x2
 add ebx, eax        // ebx:=(y1*x3).hi+(y2*x2).hi
                     //   +(y3*x1).hi+(x4*y1).lo+(y2*x3).lo
                     //   +(y3*x2).lo (Ignore overflow)
 mov eax, [esp+28]   // eax:=y4
{$IFOPT Q+}          // If overflow check is on
 xor ecx, eax        // ecx:=x4 xor y4
{$ENDIF}
 mul [esp]           // edx:eax:=y4*x1
 add ebx, eax        // ebx:=(y1*x3).hi+(y2*x2).hi
                     //   +(y3*x1).hi+(x4*y1).lo+(y2*x3).lo
                     //   +(y3*x2).lo+(y4*x1).lo (Ign o.f.)
 mov [esi+12], ebx   // @res^.data4:=(y1*x3).hi+(y2*x2).hi
                     //   +(y3*x1).hi+(x4*y1).lo+(y2*x3).lo
                     //   +(y3*x2).lo+(y4*x1).lo (Ign o.f.)
{$IFOPT Q+}          // If overflow check is on
 mov eax, [esi]      // eax:=res1
 test ebx, ebx       // Check whether ebx=res4 is nonzero
 jnz @resNonZero
 mov edx, [esi+4]    // edx:=res2
 test eax, eax       // Check whether eax=res1 is nonzero
 jnz @resNonZero
 mov eax, [esi+8]    // eax:=res3
 test edx, edx       // Check whether edx=res2 is nonzero
 jnz @resNonZero
 test eax, eax       // Check whether eax is zero
 jz @maybeNoOverflow
@resNonZero:         // res is nonzero and the sign bit
                     //   is sgn(x) xor sqn(y)
 xor ecx, ebx        // esi:=res4 xor x4 xor y4
 jns @maybeNoOverflow // if and odd number of minuses in x,
                     //   y,res an error must have occured
 call IntMulSignOverflow
@maybeNoOverflow:
{$ENDIF}
 add esp, 32         // Reset the stack pointer
 pop ebx
 pop esi
end;
```

C.2.1 Quotient reduction

If you thought the multiplication was tedious you should probably rip out the remaining pages, the division will not be as easy. Since we chose to represent the coefficients as quotients between two 128 bit integers we can implement divisions as multiplication with the denominator. To avoid overflow we need to reduce the quotient coefficients by their greatest common divisor (gcd), i.e. if the coefficient is x/y we want to find the largest factor $z = \gcd(x, y)$, shared by both x and y, so $x \to x/z$ and $y \to y/z$ becomes the reduced quotient.

We base our reduction algorithm on an GCD algorithm made up by J. Stein in 1967[30], which is based on the following facts

- If x and y are even, then $\gcd(x, y) = 2 \cdot \gcd(x/2, y/2)$, meaning we can start by shifting x and shifting y N bits to the right, i.e. dividing by 2^N, where N is the least significant nonzero bit in either x or y (remember that the bits are numbered 0,1,2,...).

- If x even and y odd, then because 2 is not common to an even and an

odd number $\gcd(x,y) = \gcd(x/2,y)$, meaning we can reduce the gcd from $\gcd(x,y) = \gcd(x',y)$, where M is the least significant nonzero bit in x and $x' = x/2^M$.

- If x and y are odd, then $\gcd(x,y) = \gcd(x',y')$, where $x' = \max(x,y) - \min(x,y)$ and $y' = \min(x,y)$ (this is because any number that is a factor of both x and y must also be a factor of $x-y$ and vice versa) so we can come back to the previous step with x' even and y' odd.
- If $x = y$, then $\gcd(x,y) = x = y$.

Thus the algorithm is to first divide x and y by 2^N, using fact 2 and 3 successively until the transformed variables becomes equal, $x' = y'$. Thus $z = \gcd(x,y) = 2^N \gcd(x',y') = 2^N x'$ should be divided from both x and y. The factor 2^N can easily be shifted away already in the gcd-algorithm.

Division without remainder

Name the bits of the 128 bit integers x, y and q as

$$\begin{aligned} x &= x_{127} : x_{126} : \cdots : x_1 : x_0 \\ y &= y_{127} : y_{126} : \cdots : y_1 : y_0 \\ q &= q_{127} : q_{126} : \cdots : q_1 : q_0 \end{aligned} \tag{C.2}$$

To perform a division $q = x/y$, where we know that y is a factor in x, i.e. the remainder is zero, we use

$$\begin{aligned} 0 = qy - x = &\left(q_{127}2^{127} + \cdots + q_1 2^1 + q_0 2^0\right)\left(y_{127}2^{127} + \cdots + y_1 2^1 + y_0 2^0\right) \\ &- \left(x_{127}2^{127} + \cdots + x_1 2^1 + x_0 2^0\right) \end{aligned} \tag{C.3}$$

and our mission is to find q. We know x and y so we can scan them from the most significant bit to find the first non zero entries x_i and y_j, where $i \geq j$ at all times for a no remainder division. For the relation to hold (x can only cancel cross terms up to order i) we must then have

$$q_k = \begin{cases} 0, & k > i - j' \\ 1, & k = i - j' \end{cases} \tag{C.4}$$

where $j' \geq j$ depends on x and y. In the case $j' = j$ it is trivial to see that the first relation should hold and that the second relation can fail from carry bits in lower order multiplications between y and q and thus we introduce j' which might me greater than j. The relation (C.3) is

$$\begin{aligned} 0 &= \left(2^{i-j'} + q_{i-j'-1}2^{i-j'-1} + \ldots\right) y - \left(2^i + x_{i-1}2^{i-1} + \ldots\right) \\ &= \left(q_{i-j'-1}2^{i-j'-1} + \ldots\right) y - \left(2^i + x_{i-1}2^{i-1} + \ldots\right) + 2^{i-j'} y \\ &= \left(q_{i-j'-1}2^{i-j'-1} + \ldots\right) y - \left(x'_{i-1}2^{i-1} + \cdots + x'_0 2^0\right) \end{aligned} \tag{C.5}$$

where we have used that the first term in $2^{i-j'}y$ cancels 2^i in x and

$$x'_{i-i'} = \begin{cases} x_{i-i'} - y_{j'-i'}, & j' - i' \geq 0 \\ x_{i-i'}, & j' - i' < 0 \end{cases} \tag{C.6}$$

where $i' = 1, 2, \ldots, i$. We are now back in the same situation as in (C.3) but with x' with most significant bit of order less than i. Repeating this procedure until nothing remains completely determines all coefficients q_k and we are done. The only problem left is thus to find a method to decide j' given j, x and y. This can be done by checking bits after the most significant bit in x and y, i.e. let Δ_i be the number of nonzero bits after x_i and Δ_j the number after y_j. If $\Delta_i = \Delta_j$ we ignore the zero bit and continue the bit scan until $\Delta_i \neq \Delta_j$. We have

$$j' = \begin{cases} j, & \text{if } \Delta_i > \Delta_j \\ j+1, & \text{if } \Delta_i < \Delta_j \end{cases} \tag{C.7}$$

Since there are only two options on j' we can comfortably shift y left $i - j$ steps and check if $2^{i-j}y > x$ and if so we simply shift y right one step and set bit $i - j - 1$ in q. In assembler the quotient reduction algorithm becomes

```
procedure ReduceFraction128(frac:pointer);
// frac is a pointer to a 256 bit coefficient
//     fracUp:fracDown
// frac^
// fracDown is assured to be positive if fracUp is nonzero
// ReduceFraction128 divides fracUp and FracDown with their
//   greatest common divisor
procedure helperShr128;
// Helper function, shifts a 128 bit integer to the right
// Nonconventional parameter pass:
//   edi:esi:edx:eax = 128 bit integer to shift
//   cl = number of bits to shift
asm
// Note: Comparing cl to an 8 bit imm. value gives smaller
//   code than comparing ecx to a 32 bit immediate value.
// shrd doesnt pair
 cmp cl, 32          // Compare cl - 32 to 0
 jb  @shiftBelow32   // Do a word shift if cl < 32
 cmp cl, 64          // Compare cl - 64 to 0
 jb  @shiftBelow64   // Do a double word shift if cl < 64
 cmp cl, 96          // Compare cl - 96 to 0
 jb  @shiftBelow96   // Do a triple word shift if cl < 96
 cmp cl, 128         // Compare cl - 128 to 0
 jb  @shiftBelow128  // Do a quad word shift if cl < 128
                     // t > 128 => val = 0
                     // Should never reach here
 ud2                 // Undefined instr. => Raises an error
 jmp @afterShift
@shiftBelow128:
 mov eax, edi        // eax:=data4
 shr eax, cl         // eax:=data4 shifted right
                     //   96 + cl mod 32 bits (no sign)
 xor edx, edx        // edx:=0
 xor esi, esi        // esi:=0
 xor edi, edi        // edi:=0
 jmp @afterShift
@shiftBelow96:
 mov eax, esi        // eax:=data3
 mov edx, edi        // edx:=data4
 shrd eax, edx, cl   // eax:=data3 shifted right
                     //   64 + cl mod 32 bits into data1
 shr edx, cl         // edx:=data4 shifted right
                     //   64 + cl mod 32 bits (no sign)
 xor esi, esi        // esi:=0
 xor edi, edi        // edi:=0
 jmp @afterShift
@shiftBelow64:
 mov eax, edx        // eax:=data2
 mov edx, esi        // edx:=data3
 mov esi, edi        // esi:=data4
 shrd eax, edx, cl   // eax:=data3 shifted right
                     //   32 + cl mod 32 bits into data1
 shrd edx, esi, cl   // edx:=data4 shifted right
                     //   32 + cl mod 32 bits into data2
 shr esi, cl         // esi:=data4 shifted right
                     //   32 + cl mod 32 bits (no sign)
 xor edi, edi        // edi:=0
 jmp @afterShift
@shiftBelow32:       // Right shift less than 32 bits
 shrd eax, edx, cl   // eax:=data2 s.r. cl bits into data1
 shrd edx, esi, cl   // edx:=data3 s.r. cl bits into data2
 shrd esi, edi, cl   // esi:=data4 s.r. cl bits into data3
 shr edi, cl         // edi:=data4 s.r. cl bits (no sign)
@afterShift:         // Now edi:esi:edx:eax = a
end;
procedure helperShl128;
// Helper function, shifts a 128 bit integer to the left
// Works like helperShr128;
asm
 cmp cl, 32          // Compare cl - 32 to 0
 jb  @shiftBelow32   // Do a word shift if cl < 32
 cmp cl, 64          // Compare cl - 64 to 0
 jb  @shiftBelow64   // Do a double word shift if cl < 64
 cmp cl, 96          // Compare cl - 96 to 0
 jb  @shiftBelow96   // Do a triple word shift if cl < 96
 cmp cl, 128         // Compare cl - 128 to 0
 jb  @shiftBelow128  // Do a quad word shift if cl < 128
                     // t > 128 => val = 0
 jmp @afterShift
@shiftBelow128:
 mov edi, eax        // edi:=data1
 shr edi, cl         // edi:=data1 shifted left
                     //   96 + cl mod 32 bits
 xor esi, esi        // esi:=0
 xor edx, edx        // edx:=0
 xor eax, eax        // eax:=0
 jmp @afterShift
@shiftBelow96:
 mov edi, edx        // edi:=data2
 mov esi, eax        // esi:=data1
 shld edi, esi, cl   // edi:=data1 shifted left
                     //   64 + cl mod 32 bits into data4
 shl esi, cl         // esi:=data1 shifted left
                     //   64 + cl mod 32 bits
```

```
 xor edx, edx        // edx:=0
 xor eax, eax        // eax:=0
 jmp @afterShift
@shiftBelow64:
 mov edi, esi        // edi:=data3
 mov esi, edx        // esi:=data2
 mov edx, eax        // edx:=data1
 shld edi, esi, cl   // edi:=data2 shifted left
                     //   32 + cl mod 32 bits into data3
 shld esi, edx, cl   // esi:=data1 shifted left
                     //   32 + cl mod 32 bits into data2
 shl edx, cl         // edx:=data1 shifted left
                     //   32 + cl mod 32 bits
 xor eax, eax        // eax:=0
 jmp @afterShift
@shiftBelow32:       // Shift less than 32 bits to the left
 shld edi, esi, cl   // edi:=data3 s.l. cl bits into data4
 shld esi, edx, cl   // esi:=data2 s.l. cl bits into data3
 shld edx, eax, cl   // edx:=data1 s.l. cl bits into data2
 shl eax, cl         // edi:=data1 s.l. cl bits
@afterShift:         // Now edi:esi:edx:eax = a
end;
procedure helperDivZeroRem_128_128(x,y:pInt128);
// Helper function, divides two 128 bit integers, no rem.
// x:=x / y, no remainder, x > 0, y > 0 assured
// Params: eax = @x; edx = @y;
asm
// Check if y is less than 32 bits and push some regs
 mov ecx, [edx+12]   // ecx:=y.data4
 push ebx
 or ecx, [edx+4]     // ecx:=y.data4 or y.data2
 mov ebx, [edx+8]    // ebx:=y.data3
 or ebx, ecx         // ebx:=y.data4 or y.data3 or y.data2
 jz @denomLength32   // Jump if all y.data2-4 are 0
// Perform a full 128 bit division
 push esi
 push edi
 push ebp
 push eax
 mov ebp, edx        // ebp pointer to y
 mov edi, [eax+12]   // Push x to stack
 mov esi, [eax+8]
 push edi
 push esi
 mov edi, [eax+4]
 mov esi, [eax]
 mov [eax], 0        // Store 0 as result
 mov [eax+4], 0
 mov [eax+8], 0
 mov [eax+12], 0
 push edi
 push esi
// Find most significant bit in y and store its pos in ebx
 mov ebx, 96
 bsr ecx, [ebp+12]   // Get most significant bit in y.data4
 jnz @foundMostSignY
 bsr ecx, [ebp+8]    // Get most significant bit in y.data3
 mov ebx, 64
 jnz @foundMostSignY
 bsr ecx, [ebp+4]    // Get most significant bit in y.data2
 mov ebx, 32
 jnz @foundMostSignY
 bsr ecx, [ebp]      // Get most significant bit in y.data1
 mov ebx, 0
 jnz @foundMostSignY // Jump if bit was found
                     // Should never reach here (y = 0)
 ud2                 // Undef. instruct. => Raises an error
@foundMostSignY:
 add ebx, ecx        // ebx:=Pos of first bit = j
 push ebx            // Put ebx=j on bottom of stack
// Current mem layout:
//   regs: edi:esi:edx:eax = y, ebp = pointer to mem y
//   stack: [esp]=j, [esp+4]=x, [esp+20]=pointer to res,
//     [esp+24]=old ebp,edi,esi
// Repeat x':=x-2^(i-j)y until x'=0
@beforeRepeat1:
// Find most significant bit of x and store its pos in ecx
 mov ecx, 96
 bsr ebx, [esp+16]   // Get most significant bit in x.data4
 jnz @foundMostSignX
 bsr ebx, [esp+12]   // Get most significant bit in x.data3
 mov ecx, 64
 jnz @foundMostSignX
 bsr ebx, [esp+8]    // Get most significant bit in x.data2
 mov ecx, 32
 jnz @foundMostSignX
 bsr ebx, [esp+4]    // Get most significant bit in x.data1
 mov ecx, 0
 jnz @foundMostSignX // Jump if bit was found
                     // Should never reach here (x = 0)
 ud2                 // Undef. instruct. => Raises an error
@foundMostSignX:
 add ecx, ebx        // ecx:=Pos of first bit = i
 sub ecx, [esp]      // ecx:=i-j
// Calculate x'=x-2^(i-j)y
 mov eax, [ebp]      // edi:esi:edx:eax:=y
 mov edx, [ebp+4]
 mov esi, [ebp+8]
 mov edi, [ebp+12]
 call helperShl128   // Shift y left i-j steps
 cmp [esp+16], edi   // Check if y*2^(i-j) > x
 ja @doSubtract
 jb @useCarry
 cmp [esp+12], esi
 ja @doSubtract
 jb @useCarry
 cmp [esp+8], edx
 ja @doSubtract
 jb @useCarry
 cmp [esp+4], eax
 jae @doSubtract
@useCarry:           // y*2^(i-j) > x => use j'=j+1
 mov ebx, ecx
 mov ecx, 1
 call helperShr128   // Shift y right 1 step
 mov ecx, ebx
 dec ecx
@doSubtract:
 sub [esp+4], eax    // x.data1:=x.data1-(2^(i-j)y).data1
 sbb [esp+8], edx    // x.data2:=x.data2-(2^(i-j)y).data2
 sbb [esp+12], esi   // x.data3:=x.data3-(2^(i-j)y).data3
 sbb [esp+16], edi   // x.data4:=x.data4-(2^(i-j)y).data4
// Set bit i-j in result
 mov ebx, [esp+20]   // ebx:=pointer to res
 mov eax, 1          // eax:=1 (to be shifted)
 cmp cl, 32          // Compare cl - 32 to 0
 jb  @setBitBelow32
 cmp cl, 64          // Compare cl - 64 to 0
 jb  @setBitBelow64
 cmp cl, 96          // Compare cl - 96 to 0
 jb  @setBitBelow96
                     // Set bit in res.data4
 shl eax, cl         // eax:=1 shl (i-j)mod 32
 or [ebx+12], eax    // res.data4:=res.data4 or eax
 jmp @afterSetBit
@setBitBelow96:      // Set bit in res.data3
 shl eax, cl         // eax:=1 shl (i-j)mod 32
 or [ebx+8], eax     // res.data3:=res.data3 or eax
 jmp @afterSetBit
@setBitBelow64:      // Set bit in res.data2
 shl eax, cl         // eax:=1 shl (i-j)mod 32
 or [ebx+4], eax     // res.data2:=res.data2 or eax
 jmp @afterSetBit
@setBitBelow32:      // Set bit in res.data1
 shl eax, cl         // eax:=1 shl (i-j)mod 32
 or [ebx], eax       // res.data1:=res.data1 or eax
@afterSetBit:
 cmp [esp+4], 0      // Check if x.data1 = 0
 jnz @beforeRepeat1
 cmp [esp+8], 0      // Check if x.data2 = 0
 jnz @beforeRepeat1
 cmp [esp+12], 0     // Check if x.data3 = 0
 jnz @beforeRepeat1
 cmp [esp+16], 0     // Check if x.data4 = 0
 jnz @beforeRepeat1
 add esp, 24         // Throw i, x and pointer to res away
 pop ebp
 pop edi
 pop esi
 jmp @afterDiv
@denomLength32:
// Divide a 128 bitnumerator with a 32 bit numerator
 mov ecx, eax        // ecx:=pointer to x
 mov ebx, [edx]      // ebx:=y1
 mov eax, [ecx+12]   // eax:=x4
 xor edx, edx        // edx:=0
 div ebx             // edx:eax:=rem and quote of 0:x4/y1
 mov [ecx+12], eax
 mov eax, [ecx+8]    // eax:=x3
 div ebx             // edx:eax:=rem and quote of rem:x3/y1
 mov [ecx+8], eax
 mov eax, [ecx+4]    // eax:=x2
```

```
 div ebx             // edx:eax:=rem and quote of rem:x2/y1
 mov [ecx+4], eax
 mov eax, [ecx]      // eax:=x1
 div ebx             // edx:eax:=rem and quote of rem:x1/y1
 mov [ecx], eax
@afterDiv:
 pop ebx
end;
// Now ReduceFraction128 begins!
// Local variables in ReduceFraction128:
var sign:boolean;
    a,b:int128;
begin
// If fracUp = 0 set fracDown to 0 and return
 if IsZero128(pInt128(frac)) then
  begin
   pWord(integer(frac)+16)^:=IMM_0;
   exit;
  end;

{$IFDEF DEBUG}
Assert(IsPos128(pInt128(integer(frac)+16)));
{$ENDIF}

// Set sign = x < 0
 sign:=pInt128(frac)^.data4 < 0;
// Change fracUp to its absolute value
 Abs128(pInt128(frac));

asm
 push esi
 push edi
 push ebx
// mov fracUp and fracDown to a and b
 mov eax, [frac]     // Load address of fracUp:fracDown
 mov ecx, [eax]      // edi:esi:edx:ecx:=fracUp
 mov edx, [eax+4]    //
 mov esi, [eax+8]    //
 mov edi, [eax+12]   //
 mov a.data1, ecx    // a:=fracUp
 mov a.data2, edx
 mov a.data3, esi
 mov a.data4, edi
 mov ecx, [eax+16]   // edi:esi:edx:ecx:=fracDown
 mov edx, [eax+20]   // (edx will be kept for later use)
 mov esi, [eax+24]   // (esi will be kept for later use)
 mov edi, [eax+28]   // (edi will be kept for later use)
 mov b.data1, ecx    // b:=fracDown
 mov b.data2, edx
 mov b.data3, esi
 mov b.data4, edi
// Find least sig. bit in fracDown and store its pos in ecx
 xor ecx, ecx        // ecx:=0
 bsf eax, b.data1    // Get least significant bit in
                     //   fracDown.data1 (bsf doesnt pair)
 jnz @foundFirstBit1 // Jump if bit was found
 bsf eax, edx        // Get l.s. bit in fracDown.data2
 mov cl, 32          // ecx:=32
 jnz @foundFirstBit1 // Jump if bit was found
 bsf eax, esi        // Get l.s. bit in fracDown.data3
 mov cl, 64          // ecx:=64
 jnz @foundFirstBit1 // Jump if bit was found
 bsf eax, edi        // Get l.s. bit in fracDown.data4
 mov cl, 96          // ecx:=96
 jnz @foundFirstBit1 // Jump if bit was found
                     // Should never be here (fracDown = 0)
 ud2                 // Undef. instr. => Raises an error
@foundFirstBit1:
 add ecx, eax        // ecx:=ecx+eax is the global position
// Find least signi. bit in fracUp and store its pos in ebx
 xor ebx, ebx        // ebx:=0
 bsf eax, a.data1    // Get l.s. bit in fracUp.data1
 jnz @foundFirstBit2 // Jump if bit was found
 bsf eax, a.data2    // Get l.s. bit in fracUp.data2
 mov bl, 32          // ebx:=32
 jnz @foundFirstBit2 // Jump if bit was found
 bsf eax, a.data3    // Get l.s. bit in fracUp.data3
 mov bl, 64          // ebx:=64
 jnz @foundFirstBit2 // Jump if bit was found
 bsf eax, a.data4    // Get l.s. bit in fracUp.data4
 mov bl, 96          // ebx:=96
 jnz @foundFirstBit2 // Jump if bit was found
                     // Should never be here (fracUp = 0)
 ud2                 // Undef. instr. => Raises an error
@foundFirstBit2:
 add ebx, eax        // ebx:=ebx+eax is the global position
// Get minimum number N of bits to shift
// N=ecx<-min(ecx,ebx), ebx<-max(ecx,ebx)-min(ecx,ebx)
 cmp ecx, ebx        // Compare ecx-ebx to 0
 setbe bh            // Use bit 8 in ebx to store which of
                     //   original ecx and ebx was the lar-
                     //   gest, bit is set if ecx <= ebx
                     //   (bh=bit 8..15 in ebx)
 jbe @afterSwap      // Jump if already ecx'=min(ecx,ebx)
                     //             and ebx'=max(ecx,ebx)
 mov al, cl          // eax:=ebx
 mov cl, bl          // ecx':=Min(ebx,ecx)
 mov bl ,al          // ebx':=Max(ebx,ecx)
@afterSwap:

// Divide the actual values of fracUp and fracDown with 2^N
//   by shifting N steps to the right
// Use helper function by putting edi:esi:edx:eax:=fracDown
//   edi,esi,edx already assigned from before
 mov eax, b.data1    // eax:=fracDown.data1
 call helperShr128   // Shift fracDown ecx steps right
 sub bl, cl          // ebx':=Max(ebx,ecx)-Min(ebx,ecx)
                     //   moved to avoid possible memory stall
 push ecx            // Store ecx
 mov ecx, [frac]     // Load address of fracUp:fracDown
 mov [ecx+16], eax   // Update shifted fracDown to memory
 mov [ecx+20], edx
 mov [ecx+24], esi
 mov [ecx+28], edi
 test bh, bh         // Check if it was fracDown (bh = 0)
                     //   that should be shifted extra
 jnz @afterExtraShift1 // Jump if not
 mov cl, bl          // Get number of shifts cl from bl
 call helperShr128   // Shift fracDown ecx steps right
@afterExtraShift1:
 pop ecx             // Restore ecx
 mov b.data1, eax    // Copy possibly shifted fracDown to b
 mov b.data2, edx
 mov b.data3, esi
 mov b.data4, edi
// Use helper function by putting edi:esi:edx:eax:=fracUp
 mov eax, a.data1    // Copy fracUp to edi:esi:edx:eax
 mov edx, a.data2
 mov esi, a.data3
 mov edi, a.data4
 call helperShr128   // Shift fracUp ecx steps (=>ecx free)
 mov ecx, [frac]     // Load address of fracUp:fracDown
 mov [ecx], eax      // Update shifted fracUp to memory
 mov [ecx+4], edx
 mov [ecx+8], esi
 mov [ecx+12], edi
 test bh, bh         // Check if it was fracUp (bh = 1)
 jz @afterExtraShift2//   that should be shifted extra
 mov cl, bl
 call helperShr128   // If so shift fracUp ecx steps right
@afterExtraShift2:

// edi:esi:edx:eax contains a, don't need to update memory
// Both a and b are now odd
// Repeat: a:=a-b and divide by 2 until odd if a>b
//         b:=b-a and divide by 2 until odd if b>a
// until a = b
@beforeWhile1:       // Use edi:esi:edx:eax = a on enter
 cmp edi, b.data4    // Compare a4-b4 to 0
 jnle @aGreaterWhile1 // Jump if a4>b4 => a>b (signed comp)
 jl @bGreaterWhile1  // Jump if a4<b4 => b>a (signed comp)
                     // Fallthrough if a4 = b4
 cmp esi, b.data3    // Compare a3-b3 to 0
 jnbe @aGreaterWhile1 // Jump if a3>b3 => a>b
 jb @bGreaterWhile1  // Jump if a3<b3 => b>a
                     // Fallthrough if a3 = b3
 cmp edx, b.data2    // Compare a2-b2 to 0
 jnbe @aGreaterWhile1 // Jump if a2>b2 => a>b
 jb @bGreaterWhile1  // Jump if a2<b2 => b>a
                     // Fallthrough if a2 = b2
 cmp eax, b.data1    // Compare a1-b1 to 0
 jz @afterWhile1     // Exit loop if a1 = b1 => a = b
 jb @bGreaterWhile1  // Jump if a1<b1 => b>a
                     // Fallthrough if a.data1 < b.data1
@aGreaterWhile1:
// a:=a-b, edi:esi:edx:eax = a
 sub eax, b.data1    // eax:=a1 - b1, sets flags
 sbb edx, b.data2    // edx:=a2 - b2 - Borrow, sets flags
 sbb esi, b.data3    // esi:=a3 - b3 - Borrow, sets flags
 sbb edi, b.data4    // edi:=a4 - b4 - Borrow
// Divide a with 2 until a becomes odd
```

```
@repeat1Start:
// a:=a shr 1;       // Run loop at least once
 shrd eax, edx, 1    // eax:=a2 shif. right one bit into a1
 shrd edx, esi, 1    // edx:=a3 shif. right one bit into a2
 shrd esi, edi, 1    // esi:=a4 shif. right one bit into a3
 shr edi, 1          // edi:=a4 shif. right one bit(no sgn)
 test al, 1          // Check if a even with bitwise and
 jz @repeat1Start    // Repeat if a even
 jmp @beforeWhile1   // Repeat big loop

@bGreaterWhile1:
// b:=b-a, edi:esi:edx:eax = a
 mov a.data1, eax    // Store a so we can put b in regs
 mov a.data2, edx
 mov a.data3, esi
 mov a.data4, edi
 mov ebx, eax        // ebx:=a1
 mov eax, b.data1    // eax:=b1
 sub eax, ebx        // eax:=b1 - a1, sets flags
 mov ebx, edx        // ebx:=a2, mov dowesn't affect flags
 mov edx, b.data2    // edx:=b2
 sbb edx, ebx        // edx:=b2 - a2 - Borrow, sets flags
 mov ebx, esi        // ebx:=a3
 mov esi, b.data3    // esi:=b3
 sbb esi, ebx        // edx:=b3 - a3 - Borrow, sets flags
 mov ebx, edi        // ebx:=a4
 mov edi, b.data4    // edi:=b4
 sbb edi, ebx        // edx:=b4 - a4 - Borrow
// Divide b with 2 until b becomes odd
@repeat2Start:
// b:=b shr 1;       // Run loop at least once
 shrd eax, edx, 1    // eax:=b2 shif. right one bit into b1
 shrd edx, esi, 1    // edx:=b3 shif. right one bit into b2
 shrd esi, edi, 1    // esi:=b4 shif. right one bit into b3
 shr edi, 1          // edi:=b4 shif. right one bit(no sgn)
 test al, 1          // Check if b even with bitwise and
 jz @repeat2Start    // Repeat if b even
 mov b.data1, eax    // Store b
 mov b.data2, edx
 mov b.data3, esi
 mov b.data4, edi
 mov eax, a.data1    // Load a for next big loop iteration
 mov edx, a.data2
 mov esi, a.data3
 mov edi, a.data4
 jmp @beforeWhile1   // Repeat big loop
@afterWhile1:
 mov a.data1, eax    // update a
 mov a.data2, edx
 mov a.data3, esi
 mov a.data4, edi

 pop ebx
 pop edi
 pop esi
end;
// Perform the division with pos numbers without remainder
 helperDivZeroRem_128_128(pInt128(frac),@a);
 helperDivZeroRem_128_128(pInt128(integer(frac)+16),@a);
// Dont forget to adjust the sign of the result
 if sign then Neg128(pInt128(frac));
end;
```

Although looking suspiciously long this actually runs faster than the same algorithm compiled in Delphi with just 64 bit integers.

Bibliography

[1] V. Bengtsson, M. Cederwall, H. Larsson and B. E. W. Nilsson, *U-duality covariant membranes*, `hep-th/0406223`.

[2] D. Cerdeno and C. Munoz, *An introduction to supergravity*,.

[3] *The large hadron collider*, `http://lhc-new-homepage.web.cern.ch/lhc-new-homepage/`.

[4] M. Nilsson, *Supersymmetry*, Master's thesis, Chalmers University of Technology, 1995.

[5] E. Cremmer, B. Julia and J. Scherk, *Supergravity theory in 11 dimensions*, Phys. Lett. **B76**, 409–412 (June, 1978).

[6] S. Weinberg, *Gravitation and cosmology: Principles and applications of the general theory of relativity*. New York, USA: Wiley, 1972. 657 p.

[7] E. Antonyan, *Supergravities in diverse dimensions*, `hep-th/9811145`.

[8] M. Huq and M. Namazie, *Kaluza-klein supergravity in ten-dimensions*, Class. Quant. Grav. **2**, 293 (1985).

[9] H. Lu and C. Pope, *P-brane solitons in maximal supergravities*, Nucl. Phys. **B465**, 127–156 (1996) [`hep-th/9512012`].

[10] L. Bao, *Algebraic structures in m-theory*, Master's thesis, Chalmers University of Technology, 2004.

[11] E.Cremmer, B.Julia, H. Lu and C. Pope, *Dualisation of dualities i.*, Nucl. Phys. **B523**, 73–144 (1997) [`hep-th/9710119`].

[12] D. Roest, *M-Theory and Gauged Supergravities*. PhD thesis, Rijksuniveriteit Groningen, 2004.

[13] H. Lu and C. Pope, *p-brane solitons in maximal supergravities*, `hep-th/9512012`.

[14] I. Lavrinenko, H. Lu, C. Pope and T. A. Tran, *U duality as general coordinate transformations, and space-time geometry*, Int. J. Mod. Phys. **A14**, 4915–4942 (1999) [hep-th/9807006].

[15] E. Bergshoeff, L. A. J. London and P. K. Townsend, *Space-time scale invariance and the super p-brane*, Class. Quant. Grav. **9**, 2545–2556 (1992) [hep-th/9206026].

[16] P. K. Townsend, *Membrane tension and manifest iib s-duality*, Phys. Lett. **B409**, 131–135 (1997) [hep-th/9705160].

[17] M. Cederwall and P. K. Townsend, *The manifestly sl(2,z)-covariant superstring*, JHEP **09** (1997) [hep-th/9709002].

[18] M. Cederwall and A. Westerberg, *World-volume fields, sl(2,z) and duality: The type iib 3-brane*, JHEP **02** (1997) [hep-th/9710007].

[19] M. Cederwall, B. E. W. Nilsson and P. Sundell, *An action for the super-5-brane in d = 11 supergravity*, JHEP **04** (1997) [hep-th/9712059].

[20] J.Fuchs and C. Schweigert, *Symmetries, lie algebras and representations: A graduate course for physicists*. Cambridge, UK: Univ. Pr., 1997. 438 p.

[21] R. D'Auria and P. Fré, *Geometric supergravity in d=11 and its hidden supergroup*, Nucl. Phys. **B201**, 101–140 (1982).

[22] V. Bengtsson, *M(embrane)-theory*, Master's thesis, Chalmers University of Technology, 2003.

[23] K. S. Stelle, *Lectures on supergravity p-branes*, hep-th/9701088.

[24] A. Bilal, *Introduction to supersymmetry*, hep-th/0101055.

[25] T. Adawi, M. Cederwall, U. Gran, B. E. W. Nilsson and B. Razaznejad, *Goldstone tensor modes*, JHEP **02** (1999) [hep-th/9811145].

[26] B. Pioline and A. Waldron, *The automorphic membrane*, JHEP **06** (2004) [hep-th/0404018].

[27] S. M. Carroll, *Lecture notes on general relativity*,.

[28] Intel Corporation, *IA-32 Intel Architecture Software Developer's Manual Set, Volume 2: Instruction Set Reference*, 2004.

[29] Intel Corporation, *Intel Architecture Optimization Manual*, 1997.

[30] D. E. Knuth, *The Art of Computer Programming*. Addison-Wesley, 1998.

[31] B. D. Wit, *Supergravity*, hep-th/0212245.

[32] H. Lu, *Introduction to m-theory*,.

[33] N. Alonso-Alberca, P. Meessen and T. Ortin, *An sl(3,z) multiplet of 8-dimensional type ii supergravity theories and the gauged supergravity inside*, Nucl. Phys. **B602**, 329–345 (2001) [hep-th/0012032].

[34] E. Bergshoeff, E. Sezgin and P.K.Townsend, *Supermambranes and eleven-dimensional supergravity*, Phys. Lett. **B189**, 75–78 (April, 1987).

[35] T. Ambjörnsson, *Compactification of 11 dimensional supergravity*, Master's thesis, Linköping University of Technology, 1998.