

Documentation for the program computing the asymptotic expansion coefficients of the heat-trace for a nonminimal Laplace type operator

Thierry Masson, Bruno Iochum

Centre de Physique Théorique *

Aix Marseille Univ, Université de Toulon, CNRS, CPT, Marseille, France

January 6, 2019

Typographical conventions:

object, **method**, *property*, element (object instance), **function**, MODULE, file, folder, command line in terminal.

Contents

1	Introduction	2
2	Installation and dependencies	2
3	Documentation	2
3.1	Objects defined in the code	2
3.1.1	Object Rational defined in <code>MathRational.js</code>	2
3.1.2	Object Integer defined in <code>MathRational.js</code>	2
3.1.3	Object Polynomials defined in <code>MathRationalExpression.js</code>	2
3.1.4	Object RationalExpression defined in <code>MathRationalExpression.js</code>	3
3.1.5	Object MathElement defined in <code>MathElement.js</code>	3
3.1.6	Object MathOperator defined in <code>MathElement.js</code>	4
3.1.7	Object MathExpression defined in <code>MathExpression.js</code>	4
3.1.8	Object MathProduct defined in <code>MathExpression.js</code>	4
3.1.9	Object MathTensorProduct defined in <code>MathExpression.js</code>	4
3.1.10	Object MathArgument defined in <code>MathExpression.js</code>	5
3.1.11	Object MathSum defined in <code>MathExpression.js</code>	5
3.2	Functions and initialization data	5
3.2.1	The folder <code>FUNCTIONS-FILES</code>	5
3.2.2	The folder <code>INIT-FILES</code>	6
3.3	The computation and the script files	6
3.3.1	The folder <code>COMPUTATIONS-0-SUBSTITUTIONS</code>	6
3.3.2	The folder <code>COMPUTATIONS-1-MAIN-R2-R4</code>	6
3.3.3	The folder <code>COMPUTATIONS-2-u-PARALLEL-R4</code>	6
3.3.4	The folder <code>COMPUTATIONS-3-NCT-R2-R4</code>	7
3.4	Other folders	7
3.4.1	The folder <code>RESULTS</code>	7
3.4.2	The folder <code>LATEX</code>	7
3.4.3	The folder <code>MATHEMATICA</code>	7
3.4.4	The folder <code>SIMPLIFICATIONS</code>	7

*thierry.masson@cpt.univ-mrs.fr, bruno.iochum@cpt.univ-mrs.fr

1 Introduction

Given a smooth hermitean vector bundle V of fiber \mathbb{C}^N over a compact Riemannian manifold and ∇ a covariant derivative on V , let $P = -(|g|^{-1/2} \nabla_\mu |g|^{1/2} g^{\mu\nu} u \nabla_\nu + p^\mu \nabla_\mu + q)$ be a nonminimal Laplace type operator acting on smooth sections of V where u, p^ν, q are $M_N(\mathbb{C})$ -valued functions with u positive and invertible. For any $a \in \Gamma(\text{End}(V))$, we consider the asymptotics $\text{Tr } a e^{-tP} \sim_{t \downarrow 0} \sum_{r=0}^{\infty} a_r(a, P) t^{(r-d)/2}$ where the coefficients $a_r(a, P)$ can be written as an integral of the functions defined on $x \in M$ by $a_r(a, P)(x) = \text{tr} [a(x) \mathcal{R}_r(x)]$.

The purpose of this computer program is to help to compute the matrix-valued functions \mathcal{R}_r , the so-called asymptotic expansion coefficients of the heat-trace of P , in terms of u -dependent operators which are universal (*i.e.* P -independent) and which act on tensor products of u, p^μ, q and their derivatives. We refer to [4] for more details on the mathematical side of this computation, which relies on our previous works [1, 2].

This computer program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

2 Installation and dependencies

This computer program is written in JavaScript (JS in the following) and requires the (free) framework `Node.js` to execute the scripts. Its current state is production, but many aspects of the program can be improved.

`Node.js` can be installed on your computer from <https://nodejs.org>. The program also depends on the `CHALK` module, which is already installed in the `node_modules` folder (<https://www.npmjs.com/package/chalk>) given with the code.

To install the code, just make a copy of the full folder at any place on your computer. Once node is installed, a script file named “`the_script.js`” can be executed using the command line “`node the_script.js`” in the terminal. The shell script `computations-and-checks.sh` executes all the computations referred to in the scientific paper.

3 Documentation

All the files have documentation written with the code. That point should nevertheless be improved. The global structure and the objects defined in the code are described below to give some understanding of the way the code is conceived. The scripts used to perform the computations rely on the objects described below, and, as far as possible, they contain their own documentation.

3.1 Objects defined in the code

Here is a description of the objects defined in the code, with their properties. Some methods are mentioned, but we refer to the documentation in the code for the complete list of methods and their characteristic. All the files mentioned are in the folder `MATH-OBJECTS`.

3.1.1 Object Rational defined in `MathRational.js`

This defines rational objects ($p/q \in \mathbb{Z}$) with methods to manipulate them: addition, multiplication, reduction...

Properties:

- *numerator* is the signed integer numerator.
- *denominator* is the positive integer denominator.

3.1.2 Object Integer defined in `MathRational.js`

Extends **Rational** as a rational with denominator = 1.

3.1.3 Object Polynomials defined in `MathRationalExpression.js`

Define polynomials objects with many methods to manipulate them: addition, multiplication...

Properties:

- *monomials* is an array of integers: the position k of the signed integers a_k in the array (starting at $k = 0$ as is usual with JS) corresponds to the coefficient a_k in front of the monomial at power k in the polynomial. Some “empty” slot in the array means that there is no monomial with this power.
- *letter* is the letter displayed, i.e. the (math) “variable” of the polynomials.

3.1.4 Object RationalExpression defined in MathRationalExpression.js

Define rational expression objects, whose numerator is a **Polynomials** and denominator is a positive integer, with many methods to manipulate them: addition, multiplication, reduction...

This object plays the role of a prefactor in more complicated objects, where the numerator polynomials is in the space dimension parameter.

Properties:

- *numerator* is a polynomials.
- *denominator* is an integer.

The *monomials* array in *numerator* is normalized so that $\text{GCD} = 1$ with the *denominator*. The *denominator* is always strictly positive.

3.1.5 Object MathElement defined in MathElement.js

This is **the base object** on which everything is constructed in a pyramidal tree way (see *list*, *prefactor*, etc below).

Properties:

- *id* is a unique identifier, it is used to compare two instances of **MathElement**, and so to simplify expressions.
- *letter* is the string used to display in the terminal.
- *latex_letter* is a tex string used to display in latex exportations.
- *math_letter* is a string used to display in Mathematica exportations.
- *productKind* is a flag to indicate the behavior in product: commutative or noncommutative.
- *sum_weight* is a weight (integer) used to order terms in sum using a computed “global” weight for each term.
- *product_weight* is a weight (integer) used to order factors in (commutative) expressions.
- *indices* is an object containing informations about indices:
 - *id* is unique to each instance, it is used to create contraction array. It is managed by the function **FH.nextIndiceID** and is based on sequence of letters in alphabetic order. The global array GLOBAL.listElementID maintains the lists of used *id*.
 - *list* is the list of integer positions of indices, 0,1,2...
 - *symmetric* is an array of indices (referred by their integer positions) that are symmetric.
 - *antisymmetric* is an array of indices (referred by their integer positions) that are antisymmetric.

These informations are used for contractions and normalization of terms.

- *contractions* is an array of pairs of strings (alphabetically sorted) of the form ‘indiceID’ = indices.id + ‘-’ + ‘integer of indice position’. This collects all the contractions in the **MathElement**.
- *copiedInfo* is used to propagate (in a bottom-up way) information when the method **makeCopy** is fired.
- *productStringWeight* is a string constructed by the method **getProductStringWeight** using *product_weight* data in order to sort **MathElement** in (commutative) products.
The global arrays GLOBAL.productWeightData and GLOBAL.productWeightOperatorData contain information to transform integer values (*product_weight*) into strings.
- *normalizedIndices* is an object containing information about the indices, their positions in a global expression and how they should appear (outer, inner, number). *normalizedIndices* is constructed by the method **fillNormalizedIndicesArray** (see the documentation of this method where it is defined) and propagated into every layers of an expression by the method **propagateNormalizedIndicesArray**: then all elements in an expression get this information, which is used to display expressions in a “normalized” way (and export to \LaTeX).

MathElement_one is a shared instance of **MathElement** which represents “1”, the unit in the algebra when products are taken into account.

3.1.6 Object **MathOperator** defined in `MathElement.js`

Extends **MathElement**. This represents a differential operator that can be applied on **MathElement**.

Properties:

- *behavior* is the behavior as a derivation (Leibniz rule is the only one used until now).
- *status* is a flag to inform if the operator has an argument or not.
- *argument* is a **MathElement** or **MathOperator** on which it acts. There can be a “chain” of arguments of **MathOperator** acting on **MathOperator**...

3.1.7 Object **MathExpression** defined in `MathExpression.js`

This is a base object which should not be used *per se*. It should be looked at as an abstract object from which useful objects are extensions.

A lot of methods defined for **MathExpression** are aware of subclasses (see below) in order to factor the code.

Properties:

- *list* is an array of **MathElement** or **MathOperator** or **MathExpression** (or subclasses).
- *prefactor* is a **RationalExpression**, a prefactor in “front of” the *list*.
- *contractions* is, as in **MathElement**, an array of pairs representing contractions of indices of **MathElement** instances inside *list*.
- *copiedInfo* is used to propagate (in a bottom-up way) information when the method **makeCopy** is fired.
- *sumStringWeight* is a string constructed by the method **getSumStringWeight** using *sum_weight* data (from **MathElement** instances inside *list*) in order to sort **MathExpression** in sums.
The global arrays `GLOBAL.sumWeightData` and `GLOBAL.sumWeightOperatorData` contain information to transform integer values (*sum_weight*) into strings.
- *normalizedIndices* is an array containing information about the indices, their positions in a global expression and how they should appear (outer, inner, number). *normalizedIndices* is constructed by the method **fillNormalizedIndicesArray** (see the documentation of this method where it is defined) and propagated into every layers of an expression by the method **propagateNormalizedIndicesArray**: then all elements in an expression get this information, which is used to display expressions in a “normalized” way (and export to \LaTeX).
- *stringRepresentation* is a normalized string representation produced by the method **getStringRepresentation** that is used to compare two expressions. When two expressions are the same, their strings should be equal. This uses *normalizedIndices* data.
- *display_symbols* is an object in which are defined various properties used to display a **MathExpression**. It varies according to subclasses of **MathExpression**.

3.1.8 Object **MathProduct** defined in `MathExpression.js`

Extends **MathExpression**. It represents a product of **MathElement** and **MathOperator**.

Properties:

- *list* contains **MathElement** and **MathOperator** only.
- *prefactor* contains information (it is not normalized to 1 contrary to other extensions of **MathExpression**, see below).

Normalization: when there are multiple **MathElement_one** in the *list* (see 3.1.5), they are removed (since it represents the unit in the algebra): at the end, there are only “non” **MathElement_one** elements remaining or only one **MathElement_one** element in *list* if this latter contains only one element.

3.1.9 Object **MathTensorProduct** defined in `MathExpression.js`

Extends **MathExpression**. It represents a tensor product of **MathProduct**.

Properties:

- *list* contains **MathProduct** only.
- *prefactor* contains information.

Normalization: all the *prefactor* of **MathProducts** in *list* are normalized to 1, so that only the main *prefactor* is meaningful (it collects all the *prefactor* of **MathProducts** in *list*).

3.1.10 Object **MathArgument** defined in `MathExpression.js`

Extends **MathTensorProduct**. It represents the structure of the “arguments” in the computation (see [4]): it contains a prefactor (**RationalExpression**), a pre-element (**MathProduct** of commutative **MathElement**), a main part represented by the *list* (**MathTensorProduct**) and a post-element (**MathProduct**).

Properties:

- *list* contains only **MathProduct** (since it extends **MathTensorProduct**).
- *preElement* is a **MathProduct**, which collects all the commutative elements in **MathProduct** in its *list* when the method `collectCommutativeElementsInPreElement` is fired: metric, inverse metric, Riemann tensor, Ricci, scalar curvature...
- *postElement* is a **MathProduct**, an object multiplying on the right in the final computation of arguments (gauge curvature, etc). It represents $Q[A]$ in [4].
- *prefactor* contains no information (normalized to 1).

Normalization: the main *prefactor*, the *prefactor* of the **MathProduct** in *list* (as in **MathTensorProduct**) and the *prefactor* of *postElement* are all normalized to 1. Only *prefactor* of *preElement* contains information (collected from the other *prefactor*). So, *preElement* plays the role of a “super prefactor” which collects what can be put outside the spectral operators “X” in the computation (by linearity).

3.1.11 Object **MathSum** defined in `MathExpression.js`

Extends **MathExpression**. It represents the sum of **MathExpression** objects.

Its *list* contain terms which are all of the same type: **MathProduct**, or **MathTensorProduct**, or (for the computation) **MathArgument**.

Normalization: the main *prefactor* is normalized to 1 (since a “global” factor for the sum is meaningless). So, only *prefactor* of terms contain information (see other classes to get where it is located in these terms).

3.2 Functions and initialization data

The scripts written to perform the computations need functions that implement the mathematical results described in [4], as well as functions to help the coding. These functions are defined in files in the folder `FUNCTIONS-FILES`. Moreover, initial data are managed in a central place in files in the folder `INIT-FILES`.

3.2.1 The folder `FUNCTIONS-FILES`

This folder contains files defining collections of functions used for the computations. The most important files, because they rely on mathematical results, are `functions-computation.js` and `functions-rules.js`.

The file `functions-computation.js` defines functions to perform computations, for instance the propagation of the derivations in arguments. All functions are grouped in the JS object `FC`: so all functions beginning with “FC.” are defined in this file.

The file `functions-rules.js` defines a procedure to apply mathematical rules, like raising of indices of tensors with the inverse metric or contractions of tensors. All functions are grouped in the JS object `FR`. The set of rules applied consists of a collection of functions that share some code but that rest unique, except for a limited number of rules that rely on “generic code” factored out in “generic functions”. This collection of rules need to be enlarged if other computations are performed (\mathcal{R}_r for $r \geq 6$, Noncommutative Torus...).

The file `functions-substitutions.js` defines functions (JS object `FS`) specifically used to compute substitutions. These substitution procedures help to decompose the computations into successive steps, using generic mathematical objects at some point, which are later replaced by more specific mathematical objects.

The file `functions-input.js` defines functions (JS object `FI`) to interpret expressions written in a simplified syntax, for instance the hand-written files of the simplified expressions obtained at the end of the computation (see below).

The file `functions-factorize.js` defines functions (JS object `FF`) to help factorize polynomials over integers (using the “Rational Root Theorem”).

The file `functions-helpers.js` is a collection of many functions (JS object `FH`) used in the code: interface to save and read files, management of arrays and strings, management of unique ID’s...

3.2.2 The folder INIT-FILES

The various scripts written to compute \mathcal{R}_r need initial data that are shared by all of them. These data, called “initialization data”, are collected in the unique file `initialization-data.js`. It contains instances of objects used to perform the computation ($P, K^\mu, H^{\mu\nu}, u, N^\mu, \nabla_\mu, \widehat{\nabla}_\mu \dots$) collected in the JS object `PROTO` (“PROTO” for “prototype” since only copies of these instances are used); initial substitutions, collected in the JS object `SUBST`; rules data passed as arguments of rules functions, collected in the JS object `RULE_DATA`; and selection data used to separate terms in the computation of \mathcal{R}_4 , collected in the JS object `SELECTION`.

The file `options.js` contains more general data: the JS object `GLOBAL` which collects various “global” data; a collection of constant parameters in the JS object `CONST`; symbols used to display the results in the JS object `DISPLAY_SYMBOLS`; options passed to display functions in the JS object `OPT`; and a list of file paths used by the scripts in the JS object `FILES`.

The file `options-substitutions.js` contains data used to compute substitutions, in the JS object `FS_DATA`.

All these data are passed to the scripts that need them at the beginning of files.

3.3 The computation and the script files

The computations are performed in script files that use the objects described in Section 3.1. These files are gathered in folders described below. The shell file `computations-and-checks.sh` executes, in the correct order, all the computations mentioned below.

3.3.1 The folder COMPUTATIONS-0-SUBSTITUTIONS

This folder contains scripts (with self-explained file names) used for the computations of the substitutions data into normal coordinates and from ∇ to $\widehat{\nabla}$. As explained in [4], these substitutions are computed starting from the substitutions of the metric $g_{\mu\nu}$ up to 4th derivation: they are given “by hand” in the file `initialization-data.js` in the object `SUBST.g_powNC`. An important improvement to the code would be to compute these “initial” substitutions, so that they could be generated to any order of derivations. Computed substitutions are saved (JSON file format) in the folder `RESULTS` in a way the code can directly read and use them.

The script `show-substitutions.js` exports to human readable \LaTeX files these computed substitutions: the master file `preliminary-computations.tex` in the folder `LATEX/MASTER-FILES` displays these results. Since substitutions take care of contractions of indices, they are presented fully contracted with a “placeholder” tensor to show how contractions behave.

3.3.2 The folder COMPUTATIONS-1-MAIN-R2-R4

This folder contains scripts used for the computations of \mathcal{R}_2 and \mathcal{R}_4 in the generic situation. The computation of \mathcal{R}_2 requires only one file, `compute-R2.js`, and the results are saved in JSON and \LaTeX files.

Concerning \mathcal{R}_4 , the computation is decomposed into several steps. The first step is the propagation of the derivations in the arguments: the result is split according to the “post element” $Q[A]v = \nabla^k v$. For each values of k , the results are saved. For $k = 1, 2, 3, 4$, the next step of the computation is done in the scripts `compute-R4-1.js` to `compute-R4-4.js` and the results are saved in JSON and \LaTeX files. The script `compute-R4-234-split.js` splits these results according to the patterns of the terms for further simplifications (see folders with “SPLIT” root in `RESULTS`). For $k = 0$, the computation is performed in several steps: substitutions to normal coordinates and to full covariant derivatives, simplifications, exportation to \LaTeX , split of the result according to the patterns of the terms. Terms with double derivations on u get a special treatment (split according to the symmetric and antisymmetric parts) in the script `compute-R4-double-derivations-of-u.js`.

Terms with same patterns are saved into (text) files (folder `MATHEMATICA`) that can be inserted (cut and paste) into `Mathematica`. The simplifications of collections of terms with same patterns is not unique and require to make (human) decisions from propositions computed using `Mathematica`.

These simplified expressions are then hand-written in files using a simplified syntax, see folder `SIMPLIFICATIONS`, that are used in the script `check-simplifications-R4.js` to check them against the saved results. The script also checks that all the “split files” (collected in files with suffixes `-list.json` in `RESULTS/LISTS`) are taken into account in the final result. This script exports all the (hand-written) simplified expressions in \LaTeX files that are used to display the results (as they appear in [4]).

3.3.3 The folder COMPUTATIONS-2-u-PARALLEL-R4

The script `compute-R4-u-parallel.js` uses the saved results (generated by scripts in the folder `COMPUTATIONS-1-MAIN-R2-R4`) and substitutes all derivations of u to 0. The result is then simplified, saved (JSON and \LaTeX files), and split

according to the patterns of the terms. Using `Mathematica`, simplifications are then computed and hand-written in files using a simplified syntax.

The script `check-simplifications-R4-u-parallel.js` checks these simplifications against the saved results and exports the simplified expressions in \LaTeX files.

3.3.4 The folder `COMPUTATIONS-3-NCT-R2-R4`

This folder is devoted to the computation of \mathcal{R}_2 for the NonCommutative Torus (NCT), see [3]. The computation of \mathcal{R}_4 is a work in progress.

The script `compute-substitutions-NCT.js` generates substitutions for the special case of the NCT and the script `compute-R2-NCT.js` explicitly computes \mathcal{R}_2 . Since in [3] the results were presented in terms of spectral functions (instead of universal operators in the present situation), `Mathematica` compatible files are generated to check the results against the previous one (folder `MATHEMATICA/SPECTRAL-FUNCTIONS`).

3.4 Other folders

Besides the files defining the objects, the initialization data, and the scripts, some folders collect computed data. In these folders, “`SPLIT`” sub folders (already mentioned above) collect data according to patterns, with file names explicitly referring to these patterns.

3.4.1 The folder `RESULTS`

In this folder and its sub folders (with more or less explicit names), results in `JSON` file format are saved. Many of these results are used by scripts as input to perform further computations.

A priori no human intervention is required in these files. Since the `JSON` file format can be read by a lot of coding language, these data could be used (may be after “translation”) by other computer algebraic systems.

3.4.2 The folder `LATEX`

This folder collects exportations of results in \LaTeX format, in small pieces of files containing only equations. The `align*` environment has been chosen to encapsulate equations. The sub folder `MASTER-FILES` contains files that compile these results using `\input` commands. These master files can be edited and used to look at results. Some lengthy results displayed in `results.tex` are not presented in [4].

The master file `split-results.tex` displays the split results, and the numeration of sections in this file is used in the code to refer to particular groups of terms.

The master file `simplified-split-results.tex` displays the simplified results (generated by the checking scripts `check-simplifications-R4.js` and `check-simplifications-R4-u-parallel.js` in the sub folder `SIMPLIFICATIONS`) obtained after studying results displayed in `split-results.tex`. These simplified results are the ones presented in [4].

The master file `preliminary-computations.tex` displays the substitutions given by hand in the code and those which are computed. This information can be used to check these computations with known results, in particular concerning normal coordinates.

3.4.3 The folder `MATHEMATICA`

Since `Mathematica` has been used (essentially to simplify the final results) as an auxiliary computer algebraic system, some results have been generated in text files whose content can be cut and paste directly in `Mathematica`. These text files are saved in sub folder in the `MATHEMATICA` folder.

The folders with root names “`SPLIT-NORMALIZED-`” are versions where the patterns are simplified to contain successive letters a, b, c, \dots instead of $\widehat{v}_{v_1}u, \widehat{v}_{v_1 v_2}^2u, \widehat{v}_{v_1 v_2 v_3}^3u, \dots$ for instance. This permits to work in `Mathematica` without worrying about the exact meaning of the constitutive elements in the patterns.

3.4.4 The folder `SIMPLIFICATIONS`

This folder contains (in sub folders) hand-written files for simplified expressions of final results. These files (`JSON` file format) use a simplified syntax to code terms, that can be read by scripts using functions coded in `functions-input.js`.

References

- [1] Bruno Iochum and Thierry Masson. Heat trace for Laplace type operators with non-scalar symbols. *Journal of Geometry and Physics*, 116:90–118, 2017.
- [2] Bruno Iochum and Thierry Masson. Heat asymptotics for nonminimal Laplace type operators and application to non-commutative tori. Mathematica Notebook added as ancillary file on arXiv, 2017.
- [3] Bruno Iochum and Thierry Masson. Heat asymptotics for nonminimal laplace type operators and application to noncommutative tori. *Journal of Geometry and Physics*, 129:1–24, 2018.
- [4] Bruno Iochum and Thierry Masson. Heat coefficient a_4 for nonminimal laplace type operators. Preprint, 2019.