BNL-112219-2016-INRE

BNL-112219-2016-IR

A comparison of different methods to implement higher order derivatives of density functionals

H. J. van Dam

May 2016

Energy Sciences Directorate

Brookhaven National Laboratory

U.S. Department of Energy

USDOE Office of Science (SC), Biological and Environmental Research (BER) (SC-23)

Notice: This manuscript has been authored by employees of Brookhaven Science Associates, LLC under Contract No.DE-SC0012704 with the U.S. Department of Energy. The publisher by accepting the manuscript for publication acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this manuscript, or allow others to do so, for United States Government purposes.

DISCLAIMER

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, nor any of their contractors, subcontractors, or their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or any third party's use or the results of such use of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof or its contractors or subcontractors. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

BNL-112219-2016-IR

A Comparison of different methods to implement higher order derivatives of density functionals

Hubertus J.J. van Dam

May 2016

Energy and Photon Sciences Directorate

Brookhaven National Laboratory

U.S. Department of Energy
DOE Office of Science, Office of Biological and Environmental Research

Notice: This manuscript has been authored by employees of Brookhaven Science Associates, LLC under Contract No. DE-SC0012704 with the U.S. Department of Energy. The publisher by accepting the manuscript for publication acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this manuscript, or allow others to do so, for United States Government purposes.

DISCLAIMER

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, nor any of their contractors, subcontractors, or their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or any third party's use or the results of such use of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof or its contractors or subcontractors. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

A comparison of different methods to implement higher order derivatives of density functionals

Hubertus J. J. van Dam[b]

[a]*Brookhaven National Laboratory, Upton, NY 11973*

Abstract

Density functional theory is the dominant approach in electronic structure methods today. To calculate properties higher order derivatives of the density functionals are required. These derivatives might be implemented manually, by automatic differentiation, or by symbolic algebra programs. Different authors have cited different reasons for using the particular method of their choice. This paper presents work where all three approaches were used and the strengths and weaknesses of each approach are considered. It is found that all three methods produce code that is sufficiently performant for practical applications, despite the fact that our symbolic algebra generated code and our automatic differentiation code still have scope for significant optimization. The automatic differentiation approach is the best option for producing readable and maintainable code.

Email address: hvandam@bnl.gov (Hubertus J. J. van Dam)

A comparison of different methods to implement higher order derivatives of density functionals

Hubertus J. J. van Dam[b]

[b] *Brookhaven National Laboratory, Upton, NY 11973*

1. Introduction

The calculation of properties plays an important role in electronic structure studies. Properties can be expressed as derivatives of the energy with respect to nuclear coordinates, external electric or magnetic fields, or combinations thereof. In a recent review [1] Table 1 provides an excellent overview of such derivatives of the energy and the associated physical properties. In density functional theory the given order of differentiation for a particular property translates into the requirement for derivatives of the density functional to the same order.

The motivation for this work stems from the requirements for higher order derivatives for the vibrational frequencies, NMR shieldings, the TDDFT [2], and the TDDFT gradient [3] capabilities in *NWChem* [4]. In addition *NWChem* supports both Gaussian basis set and planewave [5] DFT capabilities for which a shared functional library is desirable. Previously derivatives of functionals were handcoded. The introduction of the TDDFT grdient code with the associated requirement for third order derivatives essentially forced the decision to look at alternative implementation approaches. Although the functionals implemented for *NWChem* are not distributed separately the open source license of the program as a whole enables re-use in other packages.

As described by Perdew et al. [6] functionals can be classified into different categories: Local Spin Density (LSD); Generalized Gradient Approximation (GGA); and meta-GGA functionals. The LSD functionals depend on the local spin-densities ρ^α and ρ^β only. GGA functionals depend on the densities

Email address: hvandam@bnl.gov (Hubertus J. J. van Dam)

Table 1: The number of partial derivatives of a particular order as a function of the type of density functional and the order of differentiation for the spin unrestricted case

Order	LSD	GGA	meta-GGA
1	2	5	7
2	3	15	28
3	4	35	84
4	5	70	210

like LSD does but in addition also depend on the gradient of the density through [7, 8]

$$\gamma^{\alpha\alpha}(r) = \nabla\rho^{\alpha}(r)\cdot\nabla\rho^{\alpha}(r) \quad (1)$$

$$\gamma^{\alpha\beta}(r) = \nabla\rho^{\alpha}(r)\cdot\nabla\rho^{\beta}(r) \quad (2)$$

$$\gamma^{\beta\beta}(r) = \nabla\rho^{\beta}(r)\cdot\nabla\rho^{\beta}(r) \quad (3)$$

Meta-GGAs, futhermore, also depend on the kinetic energy density (see [9] apart from the factor 1/2)

$$\tau^{\alpha}(r) = \frac{1}{2}\sum_{i\in\{\text{occ.}\}}|\nabla\phi_i^{\alpha}(r)|^2 \quad (4)$$

$$\tau^{\beta}(r) = \frac{1}{2}\sum_{i\in\{\text{occ.}\}}|\nabla\phi_i^{\beta}(r)|^2 \quad (5)$$

One way to assess the challenge of implementing the required derivatives is to look at the number of expressions needed. In Tables 1 and 2 the number of partial derivatives and the total number of expressions to evaluate up to a given order of differentiation are listed for different classes of functionals.

These tables are relevant from three perspectives. Firstly, they give an impression of the scale of the effort involved in implementing the derivatives of the functionals themselves. Secondly, to calculate a property at least as many terms as there are derivatives of the functional up to the order involved have to be anticipated and implemented. Thirdly, the more derivatives and terms a property requires the more expensive calculating that property is going to be. This suggests that there is a limit to the highest order property to be considered for practical applications.

Other authors have also realized that implementing derivatives of density functionals is a non-trivial issue. J̈emmer et al. [10] pioneered the use of symbolic algebra packages with *Mathematica* in applications to DFT. This was

Table 2: The total number of expressions, including the energy expression, up to a given order of differentiation to evaluate as a function of density functional and the order of differentiation for the spin unrestricted case. E.g. for properties involving second order derivatives of LDA there is the energy expression, two first order derivatives wrt. the spin densities, and three second order derivatives making a total of six expressions to evaluate

Order	LSD	GGA	meta-GGA
1	3	6	8
2	6	21	36
3	10	56	120
4	15	126	330

followed by Strange et al. [11] publishing the *Dfauto* script using *Maple* [12] to generate source code for density functionals and their first order derivatives. Based on this work the author generated a collection of density functional subroutines [13] including second order derivatives.

The idea of having libraries of density functionals gained broader appeal leading to more efforts along these lines [14, 15, 16]. Interestingly, the latter three efforts all use different approaches to generating the source code they distribute. Sałek et al. [14] use a symbolic algebra approach, Ekström et al. [15] use automatic differentiation, whereas Marques et al. [16] rely on hand coding. All of these papers present their arguments for the approach chosen but some of the arguments seem peculiar. These choices may be due to limited experience with the other available options. Below we review all three approaches and complement the insight from the three developments mentioned with our own experience with each and every one of them. The recommendation arrived at is that automatic differentiation and symbolic algebra approaches complement eachother. Using both is probably the most effective way of writing efficient and tested code for higher order derivatives of density functionals. However extensive testing remains a requirement and non-trivial efforts are still needed in particular when problems with an implementation emerge.

The remainder of the paper discusses considerations of numerical precision that pertain to the evaluation of derivatives in section 2. In section 3 the different coding approaches are discussed and contrasted. Section 4 compares the performance of the codes obtained by different methods. Section 5 summarizes the main findings.

2. Numerical analysis

Issues related to numerical analysis surface in multiple forms when evaluating derivatives of density functionals. For testing implementations of derivatives using finite differences seems an obvious choice. In practice this approach has some caveats as the accuracy is non-trivial to assess which will be explained in the next subsection.

The topic of generating numerically stable code has also been raised before. In this work an apparently benign step was found to generate serious numerical problems after differentiation which will be described in detail. These numerical difficulties arise from carelessly applying the rules of differentiation in the context of finite precision arithmetic, regardless of the tools used. In section 3 we will refer to this problem to dispell the claim that "the AD library provides highly accurate implementations for both the function value and the derivatives of intrinsic mathematical functions, [hence] there is less possibility for a loss of precision compared to code generated by a symbolic algebra package." [15] Instead all methods discussed can generate equally precise results as long as one appreciates the strengths and weaknesses of each and adapts the way the method is deployed accordingly.

2.1. Finite difference evaluation of gradients

An integral part of implementing any expression is testing for any possible coding errors or numerical instabilities. Both Marques et al. [16] and Sałek et al. [14] used finite difference gradients to test their derivatives. While this is a straightforward approach to estimating derivatives assessing the accuracy is non-trivial. This problem follows from two opposing error sources. Approximating a function as a linear function leads to small errors for small step sizes, whereas using finite precision arithmetic leads to small errors for large step sizes. Following the approach by Gill et al. [17] it can be shown that given a function $f(x)$ the total absolute error in the gradient calculated using the central difference formula is

$$R = \frac{\epsilon}{2h}|f(x)| + \left|\frac{h^2}{12}f'''(a)\right|, \; a \in [x-h, x+h] \tag{6}$$

where ϵ is the machine precision which is $1.0e-15$ when using double precision. The optimal h can be found by minimizing Eq. 6 but it is clear that the result will depend on the third order derivative of the function. Therefore,

in practice the optimal value of h is not used. A reaonable alternative is to choose $h = \delta x$ where δ a small number [17].

When using the central difference formula to test implementations of derivatives the relative error $(R/f'(x))$ is more important than the absolute error as this indicates how many digits of the result are significant. Evaluating the relative error for a simple functional of the form ρ^n (e.g. if $n = 4/3$ this is essentially LSD exchange [18, 19], and if $n = 5/3$ this is the LSD kinetic energy functional [20, 21])

$$\frac{R(\rho)}{f'(\rho)} = \frac{\epsilon}{2\delta} + \frac{n(n-1)(n-2)\delta^2}{12} \tag{7}$$

is obtained with the additional approximation that $a = \rho$. Thus the relative error depends on the functional only through the value of n and is independent of the value of ρ. In this case the central difference formula is a good way to estimate the value of the gradient.

However, chosing a slightly different functional such as $\rho^{4/3} + \rho^{5/3}$ and the relative error becomes

$$\frac{R(\rho)}{f'(\rho)} = \frac{3\epsilon}{2\delta}\frac{1+\rho^{1/3}}{4+5\rho^{1/3}} - \frac{\delta^2}{54} \tag{8}$$

which is already a function of ρ and the form of the functional expression.

Therefore, although the central difference formula is easy to implement and often useful to get an estimate of a derivative assessing how accurate that estimate is in general non-trivial. This assessment depends on the functional itself and therefore needs to be redone for every functional and for every linear combination of functional terms. Moreover, it also requires the third order derivative of the functional to assess the accuracy of the first order derivative. This makes robust application of this technique non-trivial and even using it for testing purposes some care is required.

2.2. Numerical stability and the chain rule

The topic of numerical stability has been raised before both by Sałek et al. [14] and Ekström et al. [15]. In the context of this work numerical stability is considered in as far as it facilitates the accurate calculation of density functionals as well as all relevant derivatives. Important is also to distinguish problems that may be related to the particular tools used and problems related to the density functional expressions themselves. Over the course of

the history of density functional theory a number of functionals have been proposed that later turned out to have some numerical problems. Of course one could try some approximations, such as series expansions or splines to eliminate these problems. However, one can argue that such approaches do not implement the functional reported in the literature. In practice the only proper way to deal with this problem is to propose another functional that does not have the identified problem. Some examples of this kind of resolution are: the PW91 functional [22] was discovered to generate some spurious wiggles in the potential [23] and this was resolved with the PBE functional [24], the Long-range Corrected (LC) functionals [25] were found to have some instabilities and the short range expression was replaced with a more stable form [26], various meta-GGAs [27, 28, 29, 30, 31] were found to have singularity problems [32] which, for the Minnesota functionals, were addressed in the M08 functional [33]. These kinds of developments are an integral part of doing science and no code generation tool can fix the deficiencies of published functionals.

A different matter entirely are cases where a given functional can be implemented in different ways with different numerical properties. The discussion by Sałek et al. was related to the work by Tawada et al. [26] and does not fall into this category as they proposed to change the functional form. To illustrate how different implementations may have different numerical properties we consider the use of the dimensionless parameter χ introduced by Becke [34] but first formulated in its current form by Becke [35] as

$$\chi = \frac{|\nabla\rho|}{\rho^{4/3}} \tag{9}$$

. In practice χ should always appear in even powers for any analytically well behaved functional [35]. Hence there are two possible formulations to implement such a functional. If χ is represented by the variable "s" and χ^2 is represented by the variable "t" and "g" represents

$$g = (\nabla\rho)\cdot(\nabla\rho) \tag{10}$$

then one naive formulation would be

$$s = \frac{\sqrt{g}}{\rho^{4/3}} \tag{11}$$

$$t = s^2 \tag{12}$$

Table 3: Numerical examples of calculating derivatives of $(\sqrt{g})^2$ for different orders of differentiation using the chain rule, for simplicity assuming that $\rho = 1$

g	$\frac{\partial(\sqrt{g})^2}{\partial g}$	$\frac{\partial^2(\sqrt{g})^2}{\partial g^2}$	$\frac{\partial^3(\sqrt{g})^2}{\partial g^3}$
3.0e+00	1.0	-0.2776e-16	-0.5551e-16
7.0e-04	1.0	0.1137e-12	0.2328e-09
6.0e-08	1.0	-0.9313e-09	-0.9375e-01
5.0e-12	1.0	0.1526e-04	0.2097e+07
5.0e-16	1.0	-0.1250e+00	-0.5629e+15
3.0e-20	1.0	0.2048e+04	-0.7556e+23
Exact	1.0	0.0	0.0

the other formulation is

$$t = \frac{g}{\rho^{8/3}} \tag{13}$$

Obviously the second derivative of t wrt. g should be zero and by differentiating Eq. 13 this is easily obtained regardless of the technique used.

Applying the chain rule to Eqs. 11 and 12, however, obtains

$$\frac{\partial s}{\partial g} = \frac{1}{2\rho^{4/3}\sqrt{g}} \tag{14}$$

$$\frac{\partial^2 s}{\partial g^2} = -\frac{1}{4\rho^{4/3}g^{3/2}} \tag{15}$$

as well as

$$\frac{\partial^2 t}{\partial g^2} = 2s\frac{\partial^2 s}{\partial g^2} + 2\left(\frac{\partial s}{\partial g}\right)^2 \tag{16}$$

To obtain the correct result the two terms in Eq. 16 have to cancel exactly. If evaluated analytically these expressions will produce the correct results. Evaluated numerically the second derivative will typically not produce the correct value due to a catastrophic cancelation of significant digits. The error made will be of the order of ϵ/g. I.e. the error will be particularly large for small gradients. Furthermore this problem gets worse for higher order derivatives generating errors of ϵ/g^{n-1} where n is the order of differentiation.

Clearly evaluating derivatives by differentiating Eq. 13 is to be preferred over differentiating Eqs. 11 and 12. The role of the technology used is that

a symbolic algebra engine is likely to discover and eliminate the superflous square root, whereas an automatic differentiation library based on operator overloading cannot perform such a substitution. In the latter case it is the sole responsibility of the programmer to manually eliminate the superfluous square root to prevent numerical problems. Hence it is not necessarily true that automatic differentiation tends to generate derivatives with higher precision. Regardless of the tools used the programmer needs to understand the numerical implications of how an expression is formulated and the limits of tools in achieving precise derivatives. Obviously in this particular example manually removing the square roots is highly recommended to eliminate any risks regarding the precision of the derivatives, and to avoid wasting compute cycles.

3. Implementing higher order derivatives

Regardless of the numerical challenges to implement functionals the bar is raised when higher order partial derivatives are required. The increasing number of expressions to evaluate as well as the increasing complexity of the expressions is largely responsible for this. In practice there are three different ways to doing this. Either the derivatives are coded by hand, either a symbolic algebra engine is used to derive the equations and generate the code, or an automatic differentiation approach is used to replace the basic operations in a functional expression with ones that evaluate the expression and its derivatives. Below these three approaches are discussed in detail to clarify their properties.

3.1. Implementing higher order derivatives by hand

The *Libxc* library published by Marques et al. [16] has been implemented by hand. They seem to not have considered the automatic differentiation option. Instead they do contrast their approach to differentiation by finite differences and symbolic algebra generated code. Differentiation by finite differences was considered unsuitable as it introduces increasing numerical errors when applied to higher order derivatives. This in addition to the issues considered in Section 2.1.

The objection leveled against against symbolic algebra generated code was based on the verbosity of the code as well as the code being essentially unreadable to humans. This issue will be revisited in the section on symbolic algebra.

While manual coding of derivatives offers excellent flexibility in deriving and structuring the code in any way that is conducive to writing correct and readable code, it is the human effort required that is the main stumbling block. Certainly deriving the relatively large number of expressions needed at third order derivatives even for GGAs (not to mention meta-GGAs) is a daunting prospect. Above and beyond that a resulting implementation needs to be tested, and if errors are uncovered their source needs to be found so they may be corrected. This implies considerably more work in particular if an error is related to a mistake early in a derivation. All these aspects suggest that this approach is an option of last resort, but it will never become a routine approach.

This fact is reflected in the experience of developing *NWChem* where a decade after the release of the DFT Hessians still not all functionals could be used due to lacking second order derivatives. Even in the *Libxc* library only LSD functionals are differentiated up to third order. The GGA functionals only offer up to second order partial derivatives. Thereby underscoring how severely limited this approach is by the human resource implications it has.

3.2. Implementing higher order derivatives by symbolic algebra

The field of symbolic algebra was born in 1954 when for the first time a computer was used to prove a theorem [36]. The emerging efforts saw the development of a program for solving symbolic integration problems written in Lisp in 1963 [37]. Extensions on this approach led to symbolic algebra systems like *Macsyma* [38], *Maxima* [39], *Maple* [12], *Mathematica* and others. Today these packages support a broad range of functionality, but for this paper the differentiation and source code generation capabilities are of particular interest.

In the field of DFT Strange et al. [11] were the first to use symbolic algebra packages to generate implementations of density functionals for *Molpro* [40]. Sałek et al. [14] published a library of density functional implementations with a similar approach. The main objective of using symbolic algebra was to reduce the human effort needed and maximize the automation of code generation. However, they conceded that the detection of numerical instabilities is difficult, and hence an extensive framework for testing the functional and its partial derivatives is needed. For this purpose they decided to exploit finite difference methods.

Obviously as discussed above there are limitations to the extend that finite difference methods can be used to test derivatives of density functionals.

Another challenge that symbolic algebra engines present stems from the way they work. In hand written code one will break an expression up into manageable parts, derive the appropriate derivatives for the parts, and assemble the functional overall by combining the parts in a suitable way. Symbolic algebra engines by contrast take all expressions defining a functional and these expressions are substituted into oneanother to construct one large expression. For example the PKZB [28] functional uses a modified PBE [28, 24] functional which in turn relies on the PW91 [41] LSD functional. A symbolic algebra tool substitutes the modified PBE and PW91 LSD expressions into PKZB and generates one expression for all of PKZB. The latter may be then be differentiated in the desired ways. Subsequently the catalog of expressions is translated into Fortran. The resulting code has no structure to help anyone understand the expressions it is implementing. Furthermore, if a functional invokes another functional multiple times then all these instances are explicitly incorporated in the overall expression leading to a very verbose implementation.

In reality this verbosity problem can be addressed with some additional effort. In the PKZB example one can refer to the PBE component without providing an explicit specification of the PBE part. Symbolic algebra tools can still differentiate the expressions but will give a symbolic specification of the required derivative of the unspecified sub-expression (e.g. $'\mathrm{diff}(\mathrm{PBE}(\rho,\gamma),\rho,1)$ for the first derivative of PBE wrt. the density) in the generated Fortran code. These symbolic specifications contain sufficient information to establish the required subroutine call and which output variables need to be referenced. Hence a script can be written that translates the raw symbolic algebra generated Fortran into a proper Fortran implementation. This way the high level structure of the expressions can be preserved. In addition the code generation becomes more efficient as for each routine more compact expressions are processed. In this work these implementations are referred to as structure preserved generated code.

Like Sałek et al. we chose *Maxima* [39] as the symbolic algebra engine. This decision was driven by the open source license that *NWChem* is released with. It would have been contradictory to provide a code under an open source license but then force any interested developers to acquire commercial tools. Nevertheless, if anyone wants to use a commercial tool additional driver scripts for those tools can be added.

As indicated in [14] the *Funclib* library does not provide code that checks for zero densities which tend to cause singularities in second and higher

Table 4: The code skeleton filled out by the symbolic algebra code generator

```
do iq = 1, nq ! loop over grid points
      if (ipol.eq.1) then ! ipol=1: closed shell; ipol=2: open shell
            if (rhoa.gt.tol_rho) then
                  <the closed shell implementation>
            endif
      else
            if (rhoa.gt.tol_rho.and.rhob.gt.tol_rho) then
                  <the unrestricted open shell implementation>
            else if (rhoa.gt.tol_rho.and.rhob.le.tol_rho) then
                  <the alpha spin only implementation>
            else if (rhoa.le.tol_rho.and.rhob.le.tol_rho) then
                  <the beta spin only implementation>
            endif
      endif
enddo
```

order derivatives. Our approach follows the work of Strange et al. [11] in this respect. The idea is to draw up a code skeleton of relevant tests and fill each branch with code for that specific case, as shown in Table 4.

Like Sałek et al. and Strange et al. the code optimization capabilities of the symbolic algebra tool are used. To underscore the importance of doing so the code generation script was run with and without optimization to generate code for three correlation functionals, PW91LDA [41], PBE [24], and PKZB [28], of increasing complexity namely LSD, GGA, and meta-GGA. In addition these functionals build on oneanother in that PKZB uses a modified PBE, and PBE in turn relies on PW91LDA. The time to generate the code as well as the size of the resulting source file are listed in Table 5. While the code generation time without optimization is about a factor of 4 shorter the source code generated is much larger. In particular as the Fortran 2008 standard [42] allows for a maximum of 255 continuation lines we found compilers rejecting this code. By contrast having *Maxima* optimize the expressions reduced the code size by a factors of 43 for LSD, 937 for GGA, and 915 for the meta-GGA functional simply by breaking out common sub-expressions. The resulting codes were sufficiently compact for compilers

Table 5: The wallclock timings in hours, minutes and seconds and the source code file sizes in megabytes of the symbolic algebra code generation for correlation functionals of increasing complexity without expression optimization, with expression optimization, and with optimization and structure preservation. For the structure preserved case the timings and file sizes are the accumulative results for the functional subroutine and the subroutines of all its sub-expressions.

Functional	Without Optimization		With Optimization		Structure Preserved	
	Time	Source size	Time	Source size	Time	Source size
PW91LDA [41]	00:01:09	3.86	00:01:18	0.09	00:01:18	0.09
PBE [24]	01:17:59	299.98	03:58:48	0.32	00:24:25	0.34
PKZB [28]	03:33:59	1034.45	12:52:02	1.13	00:19:51	0.53

to accept them. In addition the work *Maxima* performed optimizing the expressions eliminates work the compiler has to do at compile time. As the code generation is part of a typical write once read many times scenario the additional investment in time is certainly worthwhile.

Nevertheless it must be noted that the time to generate the source code is non-trivial for more complicated functionals. Code generation times of 12 hours become a nuisance if multiple cycles of the specify-generate-test process are required. With the structure preserved approach this problem is largely alleviated. The code generation time for the PBE functional is reduced by a factor 9.8 and for PKZB by a factor 39. Note that the accumulated code generation time for the PKZB functional is shorter than that of the PBE functional as the modified PBE functional used in PKZB is faster to generate than the regular PBE functional. Hence the structure preserved approach clearly has benefits beyond maintaining the code structure.

An advantage of using a symbolic algebra engine is that this machinery can exploit all properties of the expression it is working on. In principle this means that any simplifying transformation possible could be applied. In practice the number and kinds of transformations applied are limited to keep the execution time within reasonable limits. If desired certain transformations have to be issued explicitly. This requires a user to understand which transformations are particularly beneficial and apply them. Hence a problem as discussed in section 2.2 might be resolved automatically but there is no guarantee of that.

3.3. Implementing higher order derivatives by automatic differentiation

Automatic differentiation was first introduced by Wengert [43]. In a two page paper he showed that using the chain-rule of differentiation on every elementary operation any program can be transformed into one that calculates the derivatives of the expression it evaluates. Since then automatic differentiation has developed into a discipline of its own and the community has developed a web-portal collecting papers and tools [44]. A variety of approaches is available including source-to-source compilers, scripting languages, and language specific libraries. However, it takes some time to evaluate the tools on offer. Criteria of particular importance are whether the tool can generate derivatives of the order required, and whether they can be applied to your code. Many tools only provide first order derivatives, some provide first and second order derivatives, and in only in rare cases higher order derivatives are supported. Many of the source-to-source compilers are capable of handling subsets of the target language rather than the full complexity of a given language standard. As a result codes may require non-trivial modifications before they can be processed.

However as the basic concepts are straightforward it is possible to develop an approach for the specific problem of interest. This is particularly true today now modern programming concepts of operator overloading are available in common programming languages such as C++ or Fortran. Ekström et al. [15] explored this approach using C++ templates to generate code for arbitrarily high orders of derivatives of density functionals in *XCFun*. We were motivated to investigate this approach because symbolic algebra approaches had left us with incomprehensible code without an obvious way of testing it. Instead of following the C++ path set out by Ekström et al. we explored a Fortran based approach. As Fortran lacks the recursive coding capabilities of C++ templates this meant that we are limited to a fixed order of differentiation but we gain the ability to easily migrate our existing density functional code base.

Ekström et al. provided a brief introduction into automatic differentiation. Their approach is based on Taylor series presumably for the need to generate arbitrary orders of differentiation through recursive code generation. We rely on the simpler chain-rule application. Hence we can be very brief. Any code can be regarded as a sequence of elementary operations like addition, subtraction, multiplication, division, exponentiation, and elementary functions such as sine, cosine, exponent, etc. Consider, for example a

Table 6: A simple Fortran data type for automatic differentiation up to second order derivatives

```
type :: nwad_dble
      double precision :: d0 ! for the function value
      double precision :: d1 ! for the first derivative
      double precision :: d2 ! for the second derivative
end type nwad_dble
```

simple elementary step such as

$$c = a * b \tag{17}$$

mathematically it is easy to derive the first and second order derivatives using the chain rule as

$$\frac{\partial c}{\partial x} = \frac{\partial a}{\partial x} * b + a * \frac{\partial b}{\partial x} \tag{18}$$

$$\frac{\partial^2 c}{\partial x^2} = \frac{\partial^2 a}{\partial x^2} * b + \frac{\partial a}{\partial x} * \frac{\partial b}{\partial x} + a * \frac{\partial^2 b}{\partial x^2} \tag{19}$$

$$\tag{20}$$

In order to exploit this using modern programming languages all that is needed is to define a data type that stores a function value and in addition its derivatives. For this particular example using Fortran that would be a data type as given in Table 6. In addition the multiplication operator needs to be redefined to deal with the derivatives as in Table 7. Assuming we encapsulate the datatype and the operator in a Fortran module called NWAD all that is needed to transform Eq. 17 into a piece of code that evaluates this expression and all derivatives up to second order is to state in the program that the NWAD module is to be used and to declare a, b, and c to be of the data type NWAD_DBLE. Obviously the initialization of the input data and the extraction of the final results need some care, otherwise the compiler takes care of the rest.

The example presented here is extremely simple as it assumes differentiation with respect to only one variable. For differentiation with respect to multiple variables there are two options. One can stick with the univariate implementation of the operators and functions but evaluate the function as

Table 7: A simple Fortran multiplication operator evaluating derivatives up to second order

```
interface operator (*)
      module procedure nwad_dble_mult
end interface
function nwad_dble_mult(a,b) result (c)
      type(nwad_dble) :: a, b, c
      c%d0 = a%d0 * b%d0
      c%d1 = a%d1 * b%d0 + a%d0 * b%d1
      c%d2 = a%d2 * b%d0 + a%d1 * b%d1 + a%d0 * b%d2
end function nwad_dble_mult
```

many times as there are partial derivatives of the order of interest using an approach developed by Griewank et al. [45] and supported by *Rapsodia* [46]. Alternatively one can switch to a fully multivariate differentiation approach. For density functional theory the multivariate approach is more suitable as it introduces less overheads and the memory requirements for the variables are manageable.

Ekström et al. implemented LSD and GGA functionals. These are functionals in terms of two and five variables respectively, and the code evaluating those differentiates with respect to the corresponding number of variables indiscriminately. In our implementation we explicitly keep track of the variables an intermediate result depends on. This sparse approach leads to shorter loops involving only the variables actually present. The overhead of the variable administration is insignificant as only the binary operators (i.e. addition, subtraction, multiplication, division, and exponentiation of two NWAD_DBLE variables) have to deal with merging sets of variables. All other operators and functions leave the set of variables unchanged. In addition tables are needed that direct the results to the right memory location. For example if the equation

$$h(x, y, z) = f(y) * g(x, z) \tag{21}$$

is to be evaluated two tables are needed to tell that derivatives wrt. to y coming from the first position in f have to go to the second position in h. Likewise a table is needed to redirect the variables of g. These tables

are needed only while executing a particular operator and can be discarded afterwards.

Managing the variable sets is facilitated by fast bit operations. Assigning the variables ρ_α, ρ_β, $\gamma_{\alpha\alpha}, \gamma_{\alpha\beta}, \gamma_{\beta\beta}, \tau_\alpha$, and τ_β each to a particular bit two sets of variables can be merged by a binary OR operation which requires a single instruction. Storing the resulting bit pattern in a integer allows sets of variables to be compared using a straightforward integer comparison. If in a binary operation both sets of variables are identical then a simpler loop structure can be used that does not use redirection tables. Exploiting this saved 10% on the execution time in our implementation.

One limitation of automatic differentiation approaches is that if a function is defined in terms of some input variables, then only derivatives with respect to those variables can be calculated. This implies that if a density functional for a closed shell density is defined in terms of the total electron density, the total density gradient, and the total kinetic energy density, then only derivatives with respect to those variables can be obtained. In this case it is impossible, for example, to obtain

$$\frac{\partial f}{\partial \gamma_{\alpha\alpha}} \quad \text{or} \quad \frac{\partial f}{\partial \gamma_{\alpha\beta}} \tag{22}$$

In most cases these quantities are not needed, except in cases like the TDDFT excitation energy of triplet states generated from a closed shell ground state. In that case a spin-dependent energy expression has to be evaluated at a point where ρ_β equals ρ_α. The simplest option is to resort to using the spin unrestricted open-shell functional implementation at the expense of additional overheads. For the singlet TDDFT excited states the regular closed shell functional implementations can be used.

From the discussion above it is clear that automatic differentiation operates from a very local viewpoint of an expression. It tackles the expression by considering a single operator at a time. Hence the approach does not exploit any global properties of the expressions and cannot make any simplifying substitutions. Hence automatic differentiation cannot resolve the kind of problems that were discussed in section 2.2. In this approach the implementor has to manually resolve such issues in the way the energy expression is coded.

4. Performance

Apart from practicalities of coding density functionals and their derivatives the efficiency of the resulting code has been raised as a concern [16, 15]. The first comment on this issue is that in many cases the efficiency of the density functional evaluation is of minor importance. The main reason for this is that the total cost of the functional evaluation scales linearly with the number of grid points, and therefore essentially linearly with the molecule size. The evaluation of the electron density and the Kohn-Sham matrix elements, for example, formally scale as $O(N^3)$, where N is proportional to the molecule size. Hence when studying increasingly large systems the cost of the functional evaluation typically becomes insignificant. This is true even when using screening approaches to achieve limiting $O(N)$ costs for the density and matrix elements evaluation for large system sizes. This results from the large difference in scaling behavior at small system sizes causing a significant offset and prefactor in the overall limiting linear cost relation.

In our effort we started from hand written code and ported that to our automatic differentiation framework. With a special compilation flag the automatic differentiation code can be changed to write *Maxima* expressions of the code it is evaluating. These expressions are subsequently differentiated using symbolic algebra. This approach has the advantage that pre-existing Fortran subroutines that typically provide only low order derivatives can be used as a starting point for generating higher order derivatives using automatic differentiation. The exact same energy expression can then be fed to the symbolic algebra engine. The results from the symbolic algebra generated code can be compared to those from the automatic differentiation code to check for unexpected descrepancies.

Incidentally, as a by product we have all three kinds of implementations available within the same code for a subset of the functionals. Hence we should be in an good position to compare the performance of the resulting functional implementations. Of course for such a comparison to be valid all three approaches must be highly optimized such that the performance is a reasonable reflection of the best performance attainable. We performed benchmark calculations of our implementations with unrestricted open shell calculations on benzene for 1st order, 2nd order and 3rd derivatives. We found that our symbolic algebra generated code was 1.4, 2.3 and 2.3 times slower than the hand written code for the respective derivatives. Our automatic differentiation implementations were 2.9, 6.1 and 5.8 times slower than

the hand written code. In particular the latter fact is surprising as the overhead of administrating the data dependencies should become less per floating point operation when going to increasingly high orders of differentiation.

To shed light on this surprising result the performance was investigated using the the Linux *Perf* tool [47]. This investigation revealed that the time to solution correlated particularly well with both the number of L1 cache misses and the number of conditional branching instructions. Our automatic differentiation implementation is strongly affected by both. Because we use a single data type to deal with all variables in the automatic differentiation approach every instance needs to allocate sufficient space for derivatives with respect to seven variables. In many situations significantly fewer variables are actually used. As a result the code accesses the memory in a much more fragmented way than strictly required by the form of the functionals generating relatively large numbers of cache misses. In addition our automatic differentiation implementation also generates many conditional branching instructions as a result of our sparsity exploitation. Because of the sparsity there are many short loops for which the exact iteration count is not known at compile time. This condition prevents the compiler from eliminating these loops, for example by loop unrolling.

A way to remedy both the conditional branch instruction and the cache miss problems would be to introduce multiple data types each for different numbers of variables, and at the same time disregard sparsity. This is similar to the approach taken by Ekström et al. in the *XCFun* library [15]. I.e. an LSD type for variables involving only densities, and a GGA type for variables involving densities and density gradients, etc., could be defined. Instances of these data types would involve only two and five variables respectively and therefore be more compact than the data type we have used. In addition looping over all variables in a data type would fix the loop lengths to constants known at compile time allowing for more effective code optimization. To date we have not untaken such a rewrite as the fraction of the execution time spent on the functional evaluation, even with an inefficient implementation, is too small to warrant investing the effort required.

Regarding the performance of the symbolic algebra generated code our performance analysis showed that this code performs over 30% more floating point operations than the corresponding hand written code. This result is related to the expression optimization performed by *Maxima*. As stated above the optimizer breaks out common sub-expressions. It does not however replace sub-expressions with more efficient alternatives. For example, deriva-

tives of functionals involve $\rho^{4/3}$ as well as $\rho^{1/3}$. A human programmer would calculate the former as $\rho^{4/3} = \rho * \rho^{1/3}$ as the multiplication requires less cycles than the exponentiation. *Maxima* instead generates code that performs both exponentiations. To persuade *Maxima* to generate more efficient expressions additional rules are needed that explicitly substitute $(\rho * \rho^{1/3})$ for $\rho^{4/3}$ so that the optimizer recognizes the opportunity to reuse $\rho^{1/3}$. Similar substitutions are required in other cases such as square roots, polynomials and divisions. Again we have not implemented these improvements as the effort required outweighs the expected performance improvements of the application overall.

Summarizing the outcomes of the performance analysis the first point to concede is that the performance of both the symbolic algebra as well as the automatic differentiation approach presented here is a worst case scenario in that both approaches can be optimized further. In particular the automatic differentiation approach as we have implemented it suffers from cache misses as well as loop overheads that when addressed should bring the performance much closer to that of the hand written code. Hence the timings given here cannot be interpreted as a fair assessment of the technique. Regardless of the short comings of our implementation it is clear however that the performance is sufficiently good for the capability to be useful in real applications. Even when making the worst case assessment the time spend on the functional evaluation is below 10% and this fraction will reduce further with increasing molecule size.

Hence overall the approach to take depends on a number of factors. If performance is critical to your application handwritten code still performs the best, but symbolic algebra generated and automatic differentiation based code is obtainable with much less human effort. The latter implementations also perform at least well enough to be practically useful and very likely would be competitive if one is prepared to invest enough optimization work on the code generation and the automatic differentiation library. If readability and maintainability of the code is a prime concern then automatic differentiation is the best option. The operator overloading technique allows the functional expression to be formulated in an easy to read form. At the same time the fact that the operators are given in specific self contained functions in principle allows each operator to be tested and verified individually. In combination this allows for a verifiable implementation of functionals and there derivatives. Neither hand written code nor symbolic algebra generated code offer this to the same extent.

5. Conclusions

Three different approaches to creating code for higher order derivatives of density functionals have been discussed. Writing the code by hand leads to readable and maintainable code. When written well the code also shows high performance. However, this approach is severely limited by the human resource implications. This latter reason renders the approach impractical in particular for going to increasing orders of differentiation.

Generating the source code through a symbolic algebra engine leads to code that shows slightly degraded performance compared to hand written code. Part of the performance degradation is, however, due to lacking optimizations in regular implementations of symbolic algebra engines. Suitable explicit substitutions aplied to the expressions just before the expression optimization should be able to improve the performance to one that is closer to the hand written code. Problematic is that the code has little structure to facilitate human comprehension. This is compounded by the fact that very large subroutines are typically generated. Also the elapse time required to generate the source code may be significant as examplified by the 12 hours it took to generate code for the PKZB correlation functional. The latter two issues can largely be addressed by the structure preserved approach whereby a call structure similar to hand written code is maintained while still generating the code completely automatically. Nevertheless, even in the latter approach the lack of mnemonic variable names still leaves extremely hard to read code.

Automatic differentiation, in our implementation, simply requires a Fortran implementation of the energy expression. This code can be transformed into code that evaluates derivatives in addition to the original expression simply be declaring the appropriate variables to be of a derived type. The derived type operators overload the traditional Fortran operators with code the computes derivatives in addition to the energy expression. As the original code structure is maintained human comprehension is greatly facilitated. As a result automatic differentiation is the best of all three approaches for readability and maintainability. Our implementation, while sufficiently performant to be practically usable, still performs significantly worse for higher order derivatives than the hand written code, i.e. by a factor of about six. This performance degradation is the result of our simple approach of just using a single data type for all possible situations while still exploiting sparsity. This approach requires a datatype that has a large memory footprint leading

to large numbers of cache misses. In addition the code introduces many short do-loops that the compiler cannot optimize effectively. These issues can be addressed by creating different data types for different subsets of variables. This will reduce the memory footprint of the data types and will generates do-loops with explicit loop limits that can be optimized more effectively.

Regardless of the approach used to implement density functionals and their derivatives the raw performance of the functional evaluation is typically not a prime concern. The reason is that the functional evaluation cost depends only on the number of grid points and is therefore essentially linear with respect to the number of atoms. Other factors such as the density and Kohn-Sham matrix evaluation formally scale as the number of atoms cubed.

6. Acknowledgement

The research was performed using EMSL, a DOE Office of Science User Facility sponsored by the Office of Biological and Environmental Research and located at Pacific Northwest National Laboratory.

Appendix A. Location and structure of the density functional library

The implementations of the hand written, the automatic differentiation, and the symbolic algebra generated described here resided in the *NWChem* source tree in the locations shown in Table A.8. The source code pertaining to this paper can be obtained with the command

```
svn checkout -r27732 https://svn.pnl.gov/svn/nwchem/trunk nwchem
```

Table A.8: The directory structure of NWChem associated with the density functional implementations

nwchem				
	src			
		nwdft		
			xc	the hand written functional implementations
		nwxc		the automatic differentiation and symbolic algebra generated code
			nwad	the automatic differentiation library
			maxima	the symbolic algebra code
				bin executable scripts to drive the code generation
				input inputs for the functional expression printing
				max *Maxima* expressions for the functionals
				f77 *Maxima* generated Fortran code

References

[1] P. Pulay, Analytical derivatives, forces, force constants, molecular geometries, and related response properties in electronic structure theory, Wiley Interdisciplinary Reviews: Computational Molecular Science 4 (3) (2013) 169. doi:10.1002/wcms.1171.
URL http://dx.doi.org/10.1002/wcms.1171

[2] S. Hirata, M. Head-Gordon, Time-dependent density functional theory for radicals, Chemical Physics Letters 302 (5-6) (1999) 375. doi:10.1016/s0009-2614(99)00137-2.
URL http://dx.doi.org/10.1016/S0009-2614(99)00137-2

[3] D. W. Silverstein, N. Govind, H. J. J. van Dam, L. Jensen, Simulating one-photon absorption and resonance raman scattering spectra using analytical excited state energy gradients within time-dependent density functional theory, Journal of Chemical Theory and Computation 9 (12) (2013) 5490. doi:10.1021/ct4007772.
URL http://dx.doi.org/10.1021/ct4007772

[4] M. Valiev, E. Bylaska, N. Govind, K. Kowalski, T. Straatsma, H. Van Dam, D. Wang, J. Nieplocha, E. Apra, T. Windus, et al., Nwchem: A comprehensive and scalable open-source solution for large scale molecular simulations, Computer Physics Communications 181 (9) (2010) 1477. doi:10.1016/j.cpc.2010.04.018.
URL http://dx.doi.org/10.1016/j.cpc.2010.04.018

[5] E. Bylaska, K. Tsemekhman, N. Govind, M. Valiev, Large-scale plane-wave-based density functional theory: Formalism, parallelization, and applications, Computational Methods for Large Systems (2011) 77doi:10.1002/9780470930779.ch3.
URL http://dx.doi.org/10.1002/9780470930779.ch3

[6] J. P. Perdew, A. Ruzsinszky, J. Tao, V. N. Staroverov, G. E. Scuseria, G. I. Csonka, Prescription for the design and selection of density functional approximations: More constraint satisfaction with fewer fits, The Journal of Chemical Physics 123 (6) (2005) 062201. doi:10.1063/1.1904565.
URL http://dx.doi.org/10.1063/1.1904565

[7] F. Herman, J. P. Van Dyke, I. B. Ortenburger, Improved statistical exchange approximation for inhomogeneous many-electron systems, Physical Review Letters 22 (16) (1969) 807. doi:10.1103/physrevlett.22.807.
URL http://dx.doi.org/10.1103/PhysRevLett.22.807

[8] J. A. Pople, P. M. Gill, B. G. Johnson, Kohnsham density-functional theory within a finite basis set, Chemical Physics Letters 199 (6) (1992) 557. doi:10.1016/0009-2614(92)85009-y.
URL http://dx.doi.org/10.1016/0009-2614(92)85009-Y

[9] A. D. Becke, Density-functional thermochemistry. iv. a new dynamical correlation functional and implications for exact-exchange mixing, The Journal of Chemical Physics 104 (3) (1996) 1040. doi:10.1063/1.470829.
URL http://dx.doi.org/10.1063/1.470829

[10] P. Jemmer, P. J. Knowles, Generation of functional derivatives in kohn-sham density-functional theory, Computer Physics Communications 100 (1-2) (1997) 93. doi:10.1016/s0010-4655(96)00161-0.
URL http://dx.doi.org/10.1016/S0010-4655(96)00161-0

[11] R. Strange, F. Manby, P. Knowles, Automatic code generation in density functional theory, Computer Physics Communications 136 (3) (2001) 310. doi:10.1016/s0010-4655(01)00148-5.
URL http://dx.doi.org/10.1016/S0010-4655(01)00148-5

[12] B. W. Char, K. O. Geddes, G. H. Gonnet, B. Leong, M. B. Monagan, S. M. Watt, Maple v language reference manualdoi:10.1007/978-1-4615-7386-9.
URL http://dx.doi.org/10.1007/978-1-4615-7386-9

[13] H. J. J. van Dam, P. Sherwood, Density functional repository, [Accessed: 21 Jan. 2015] (2006).
URL http://www.cse.scitech.ac.uk/ccg/dft/

[14] P. Sałek, A. Hesselmann, A self-contained and portable density functional theory library for use in ab initio quantum chemistry programs, Journal of Computational Chemistry 28 (16) (2007) 2569. doi:10.1002/jcc.20758.
URL http://dx.doi.org/10.1002/jcc.20758

[15] U. Ekström, L. Visscher, R. Bast, A. J. Thorvaldsen, K. Ruud, Arbitrary-order density functional response theory from automatic differentiation, Journal of Chemical Theory and Computation 6 (7) (2010) 1971. doi:10.1021/ct100117s.
URL http://dx.doi.org/10.1021/ct100117s

[16] M. A. Marques, M. J. Oliveira, T. Burnus, Libxc: A library of exchange and correlation functionals for density functional theory, Computer Physics Communications 183 (10) (2012) 2272. doi:10.1016/j.cpc.2012.05.007.
URL http://dx.doi.org/10.1016/j.cpc.2012.05.007

[17] P. E. Gill, W. Murray, M. A. Saunders, M. H. Wright, Computing forward-difference intervals for numerical optimization, SIAM Journal on Scientific and Statistical Computing 4 (2) (1983) 310. doi:10.1137/0904025.
URL http://dx.doi.org/10.1137/0904025

[18] P. A. M. Dirac, Note on exchange phenomena in the thomas atom, Math. Proc. Camb. Phil. Soc. 26 (03) (1930) 376.

doi:10.1017/s0305004100016108.
URL http://dx.doi.org/10.1017/S0305004100016108

[19] J. C. Slater, A simplification of the hartree-fock method, Phys. Rev. 81 (3) (1951) 385. doi:10.1103/physrev.81.385.
URL http://dx.doi.org/10.1103/PhysRev.81.385

[20] L. H. Thomas, The calculation of atomic fields, Math. Proc. Camb. Phil. Soc. 23 (05) (1927) 542. doi:10.1017/s0305004100011683.
URL http://dx.doi.org/10.1017/S0305004100011683

[21] E. Fermi, Eine statistische methode zur bestimmung einiger eigenschaften des atoms und ihre anwendung auf die theorie des periodischen systems der elemente, Zeitschrift fr Physik 48 (1-2) (1928) 73. doi:10.1007/bf01351576.
URL http://dx.doi.org/10.1007/BF01351576

[22] J. P. Perdew, J. A. Chevary, S. H. Vosko, K. A. Jackson, M. R. Pederson, D. J. Singh, C. Fiolhais, Atoms, molecules, solids, and surfaces: Applications of the generalized gradient approximation for exchange and correlation, Phys. Rev. B 46 (11) (1992) 6671. doi:10.1103/physrevb.46.6671.
URL http://dx.doi.org/10.1103/PhysRevB.46.6671

[23] C. Filippi, C. J. Umrigar, M. Taut, Comparison of exact and approximate density functionals for an exactly soluble model, The Journal of Chemical Physics 100 (2) (1994) 1290. doi:10.1063/1.466658.
URL http://dx.doi.org/10.1063/1.466658

[24] J. P. Perdew, K. Burke, M. Ernzerhof, Generalized gradient approximation made simple, Physical Review Letters 77 (18) (1996) 3865. doi:10.1103/physrevlett.77.3865.
URL http://dx.doi.org/10.1103/PhysRevLett.77.3865

[25] H. Iikura, T. Tsuneda, T. Yanai, K. Hirao, A long-range correction scheme for generalized-gradient-approximation exchange functionals, The Journal of Chemical Physics 115 (8) (2001) 3540. doi:10.1063/1.1383587.
URL http://dx.doi.org/10.1063/1.1383587

[26] Y. Tawada, T. Tsuneda, S. Yanagisawa, T. Yanai, K. Hirao, A long-range-corrected time-dependent density functional theory, The Journal

of Chemical Physics 120 (18) (2004) 8425. doi:10.1063/1.1688752.
URL http://dx.doi.org/10.1063/1.1688752

[27] T. Van Voorhis, G. E. Scuseria, A novel form for the exchange-correlation energy functional, The Journal of Chemical Physics 109 (2) (1998) 400. doi:10.1063/1.476577.
URL http://dx.doi.org/10.1063/1.476577

[28] J. P. Perdew, S. Kurth, A. Zupan, P. Blaha, Accurate density functional with correct formal properties: A step beyond the generalized gradient approximation, Physical Review Letters 82 (12) (1999) 2544. doi:10.1103/physrevlett.82.2544.
URL http://dx.doi.org/10.1103/PhysRevLett.82.2544

[29] J. Tao, J. P. Perdew, V. N. Staroverov, G. E. Scuseria, Climbing the density functional ladder: Nonempirical metageneralized gradient approximation designed for molecules and solids, Physical Review Letters 91 (14). doi:10.1103/physrevlett.91.146401.
URL http://dx.doi.org/10.1103/PhysRevLett.91.146401

[30] Y. Zhao, N. E. Schultz, D. G. Truhlar, Exchange-correlation functional with broad accuracy for metallic and nonmetallic compounds, kinetics, and noncovalent interactions, The Journal of Chemical Physics 123 (16) (2005) 161103. doi:10.1063/1.2126975.
URL http://dx.doi.org/10.1063/1.2126975

[31] Y. Zhao, D. G. Truhlar, A new local density functional for main-group thermochemistry, transition metal bonding, thermochemical kinetics, and noncovalent interactions, The Journal of Chemical Physics 125 (19) (2006) 194101. doi:10.1063/1.2370993.
URL http://dx.doi.org/10.1063/1.2370993

[32] J. Grafenstein, D. Izotov, D. Cremer, Avoiding singularity problems associated with meta-gga (generalized gradient approximation) exchange and correlation functionals containing the kinetic energy density, The Journal of Chemical Physics 127 (21) (2007) 214103. doi:10.1063/1.2800011.
URL http://dx.doi.org/10.1063/1.2800011

[33] Y. Zhao, D. G. Truhlar, Exploring the limit of accuracy of the global hybrid meta density functional for main-group thermochemistry, kinetics, and noncovalent interactions, Journal of Chemical Theory and Computation 4 (11) (2008) 1849. doi:10.1021/ct800246v.
URL http://dx.doi.org/10.1021/ct800246v

[34] A. D. Becke, Density functional calculations of molecular bond energies, The Journal of Chemical Physics 84 (8) (1986) 4524. doi:10.1063/1.450025.
URL http://dx.doi.org/10.1063/1.450025

[35] A. D. Becke, Density-functional exchange-energy approximation with correct asymptotic behavior, Phys. Rev. A 38 (6) (1988) 3098. doi:10.1103/physreva.38.3098.
URL http://dx.doi.org/10.1103/PhysRevA.38.3098

[36] J. Siekmann, G. Wrightson (Eds.), Automation of reasoning: Classical papers on computational logic 1957-1966, Springer, Berlin Heidelberg, 1983.

[37] J. R. Slagle, A heuristic program that solves symbolic integration problems in freshman calculus, Journal of the ACM 10 (4) (1963) 507. doi:10.1145/321186.321193.
URL http://dx.doi.org/10.1145/321186.321193

[38] J. Moses, Macsyma: A personal history, Journal of Symbolic Computation 47 (2) (2012) 123. doi:10.1016/j.jsc.2010.08.018.
URL http://dx.doi.org/10.1016/j.jsc.2010.08.018

[39] Maxima, a computer algebra system. version 5.34.0, [Accessed: 14 Jan. 2015] (2014).
URL http://maxima.sourceforge.net/

[40] H.-J. Werner, P. J. Knowles, G. Knizia, F. R. Manby, M. Schtz, Molpro: a general-purpose quantum chemistry program package, Wiley Interdisciplinary Reviews: Computational Molecular Science 2 (2) (2011) 242. doi:10.1002/wcms.82.
URL http://dx.doi.org/10.1002/wcms.82

[41] J. P. Perdew, Y. Wang, Accurate and simple analytic representation of the electron-gas correlation energy, Phys. Rev. B 45 (23) (1992) 13244.

doi:10.1103/physrevb.45.13244.
URL http://dx.doi.org/10.1103/PhysRevB.45.13244

[42] JTC1/SC22/WG5, Information technology – programming languages – fortran – part 1: Base language, [Accessed: 3 July 2015] (2010).
URL http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=50459

[43] R. E. Wengert, A simple automatic derivative evaluation program, Communications of the ACM 7 (8) (1964) 463. doi:10.1145/355586.364791.
URL http://dx.doi.org/10.1145/355586.364791

[44] Autodiff, Autodiff – community portal for automatic differentiation, [Accessed: 14 Jan. 2015] (2015).
URL http://www.autodiff.org

[45] A. Griewank, J. Utke, A. Walther, Evaluating higher derivative tensors by forward propagation of univariate taylor series, Mathematics of Computation 69 (231) (2000) 1117. doi:10.1090/s0025-5718-00-01120-0.
URL http://dx.doi.org/10.1090/S0025-5718-00-01120-0

[46] I. Charpentier, J. Utke, Rapsodia: User manual, [Accessed: 25 Jan. 2015] (2014).
URL http://www.mcs.anl.gov/Rapsodia/userManual.pdf"

[47] Perf, performance counters for linux. version 2.6.32, [Accessed: 28 Oct. 2015] (2010).
URL https://perf.wiki.kernel.org/