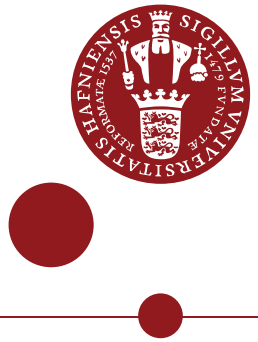
UNIVERSITY OF COPENHAGEN



Massively Lockstep-Parallel Algorithms for Full-Isomer Space Quantum Chemistry

Masters

Anton M. Nielsen & Asmus Tørsleff

Supervised by Assoc. Professor, Ph.D. James Avery
Co-supervised by Professor, Ph.D., Dr. Scient. Kurt V. Mikkelsen

Department of Computer Science
Quantum Information Science

August 15, 2025

Masters

Massively Lockstep-Parallel Algorithms for Full-Isomer Space Quantum Chemistry

By Anton M. Nielsen & Asmus Tørsleff

Supervised by Assoc. Professor, Ph.D. James Avery
Co-supervised by Professor, Ph.D., Dr. Scient. Kurt V. Mikkelsen

Date of submission: August 15, 2025

University of Copenhagen
Faculty of Science
Department of Computer Science
Quantum Information Science
Universitetsparken 5
2100 Copenhagen Ø

Acknowledgements

We would like to thank our supervisor James E. Avery for guidance and advise during this project, especially in recommending articles and frequently checking in on the project.

We would also like to thank our co-supervisor Kurt V. Mikkelsen sharing his encyclopedic chemistry expertise and for his expedient and intuitive introduction to the chemistry.

We would like to thanks Jonas D. de la Cour for technical help, answering questions and advocating for us in the first few meetings regarding the project.

"I would like to thank my family, friends, and colleagues for their support and for being understanding of my absence during the last few weeks of the project." - Anton

"Thank you to my family, friends and especially room-mate for caring for and tolerating a very spent co-inhabitant in the final days before hand-in." - Asmus

Abstract

We take a step towards faster screening for large isomer spaces of fullerenes by introducing both GPU and quantum algorithms for approximating their chemical energies. To this end we implement a prototype for large parts of a semi-empirical method obtaining energies matching the reference implementation. We show that this method is lockstep-parallel for isomer spaces with fixed element ordering and how to reduce computations by only considering fullerenes. The quantum algorithms provided allow sampling of isomers with low anisotropic energy from the entire isomer space, with better than classical probability, using qubits logarithmic in the size of the isomer space and having circuit depth quadratic in the number of atoms in an isomer.

Individual Contributions to the Project

Due to the type of thesis this is handed in as, we will disclose which sections are individual contributions and responsibilities of each author and which are common. Regardless, every section has been proof read and discussed in common.

Anton has contributed and taken responsibility for section 4 and section 7.2 as well as the related code, including the Nix setup.

Asmus has contributed and taken responsibility for section 5 and any paragraph mentioning quantum computation.

Table of Contents

1 Introduction **1**

2 Code Structure **3**

3 Introduction to GFN2-xTB **5**

3.1 Energy Terms 5
3.2 Constructing the Initial Density Matrix 8
3.3 Constructing the Overlap, Dipol and Quadrupol Tensors 9
3.4 Extended Hückel Energy 16
3.5 Repulsion Energy 18
3.6 Isotropic Energy 19
3.7 Anisotropic Electrostatic Energy 20
3.8 Anisotropic Exchange Correlation 22
3.9 Electronic Entropy 22
3.10 Dispersion Energy 23

4 High Performance Parallel Computing **27**

4.1 Case Study 29
4.2 Data Coalescing 32
4.3 Introduction to a Parallel Lockstep Algorithm 34
4.3.1 Orbitals as a Constant 34
4.3.2 Making a Carbon-specific Initial Density Guess 37
4.3.3 Making a Carbon-specific Overlap Computation 39

5 Introduction to quantum algorithmic approaches **41**

5.1 Calculating E^{Γ} using Quantum Digital Arithmetic 42
5.2 Sampling using Quantum Amplitude Arithmetic 44
5.3 Calculating E^{Γ} with Quantum Amplitude Arithmetic. 45
5.4 Calculating E^{γ} using Quantum Digital Arithmetic 53
5.5 Complexity 54

6 Related Work **57**

6.1 xtb version 6.4.0 . . . 57
6.2 dxtb . . . 57

7 Methodology **58**
7.1 Porting the Reference Implementation . . . 58
7.2 Testing . . . 60
7.2.1 Towards Reproducability with Nix . . . 61
7.2.2 Patching xtb . . . 65
7.2.3 Implementing the Tests . . . 66

8 Results **68**
8.1 Our Contributions . . . 68
8.2 Validation . . . 69
8.3 Benchmarks . . . 70

9 Reflections **71**

10 Conclusion **73**

11 GAI Declaration **74**

Bibliography **75**

1 Introduction

Calculating the properties of molecules is of major interest to any chemist. Computational approaches have become astonishingly precise and are of great use. A challenge is that the more accurate methods usually take weeks or months to run on large molecules. If we want results in a more timely manner we either have to spend more money on hardware, make faster software and algorithms or find a way to do more targeted calculations.

In this thesis we will contribute to James E. Avery's efforts to develop an efficient screening pipeline for fullerenes. The concept of a screening pipeline applies when you want to calculate the properties of a large amount of molecules e.g. large isomer spaces in search of the best molecules for your purposes. In such a pipeline progressively slower techniques are used in each step while discarding poor candidate molecules in each.

For our purpose we work with the isomer spaces C_n where their size is on the order of $O(n^9)$. These quickly become very large and so the first step in the screening pipeline must be very efficient. This motivates picking a fast approximate algorithm and implementing a highly optimised version. We landed on GFN2-xTB for this role. It is part of a family of algorithms for computing the geometries, frequencies and noncovalent bonds using extended tight binding methods. It is semi-emperical and already decently fast.

Calculating the properties of a molecule in C_200 takes about 30 seconds on a good desktop using the GFN2-xTB reference implementation. A quick back of the envelope calculation then puts the time to calculate all the molecules in the C_200 isomer space at around 200 years. To bring this down we now only have two levers to pull, acquiring more hardware or faster software.

Thus this thesis will provide a blueprint for accelerating the calculation of entire isomer spaces of highly similar molecules in GFN2-xTB, as well as an in-depth look at the GFN2-xTB algorithm. We will exploit the fact that fullerenes are entirely made up of carbon atoms to provide lockstep-parallel algorithms suitable for running on GPUs and taking advantage of their high floating point operation throughput.

We will also evaluate the opportunities for using quantum computing to speed up the search for good candidates. We will design quantum circuits for computing the GFN2-xTB isotropic energy terms and discuss how this can be applied to superposition's of molecules from an isomer space.

In chapter 2 we will introduce the repository structure, and in chapter 3 we will explain the GFN2-xTB method including code snippets. In chapter 4 we will go over the practicalities of designing and implementing a lockstep-parallel algorithm on the GPU. In chapter 5 we will Design and analyse quantum circuits as an alternative approach to calculating the anisotropic energy terms of GFN2-xTB. In chapter 6 we will briefly discuss related projects. Chapter 7 introduces our implementation approach and how we ensure correctness in relation to the reference implementation. In chapters 8, 9 and 10 we reflect on and conclude the project while reiterating our results.

Code Structure

2

The code repository can be found at: `https://github.com/AceMouse/xTB-math`.

```
xTB-math/
├── bin2xyz/
│   ├── bin2xyz.cpp ..... converter from float64 coords into xyz format
│   └── C200_10000_fullerenes.float64 ..... 3d coords for 10k fullerenes
├── flake.lock
├── flake.nix ..... conventional structure for Nix inputs and outputs
├── nix/ ..... Nix package definitions for xtb and its dependencies
│   ├── cpx.nix ..... CPCM-X - a solvation model
│   ├── numsa.nix ..... solvent accessible surface area calculation
│   ├── patches/ ..... patches for extracting data for validation
│   │   ├── dftd4/ ..... dispersion correction
│   │   │   ├── log_args_and_outputs.patch
│   │   │   └── use_gfn2.patch
│   │   └── xtb/
│   │       ├── log_args_and_outputs.patch
│   │       ├── log_electro.patch
│   │       └── log_utils.patch
│   └── xtb.nix ..... extended tight-binding program
├── README.md
├── report/
└── xtb-python/ ..... Python port of xTB paper and Fortran impl
    ├── basisset.py
    ├── blas.py
    ├── cmp_impls.py ..... validation tests against Fortran impl
    ├── data/
    │   ├── C200.xyz
    │   └── caffeine.xyz
    ├── dftd4.py ..... computation for dispersion correction
    ├── dftd4_reference.py ..... constants for dftd4
    ├── energy.py ..... various energy computations
    ├── fock.py ..... Fock matrix computation
    ├── gfn2.py ..... GFN2-xTB specific constants
    ├── lapack.py ..... Facade for LAPACK functions
    ├── scc.py ..... computation for self-consistent charges
    ├── slater.py ..... computation for slater determinants
    ├── util.py
    └── xyz_reader.py
```

A seperate code repository for the SYCL GPU implementation can be found at: `https://github.com/Maroka-chan/xtb-gpu`.

```
xtb-gpu/
├── flake.lock
├── flake.nix
├── nix/
│   ├── nvhpc.nix ..... patching nvhpc to make nvfortran work on Nix
│   └── xtb.nix ..... xtb version 6.4.0 compiled with nvfortran
├── README.md
└── sycl/
    ├── build_SDQH0.cpp ..... incomplete SYCL impl for build_SDQH0
    ├── data_caffeine/ ..... test data for electro.cpp computed from caffeine
    │   ├── atomicGam.txt
    │   ├── dqsh.txt
    │   ├── dq.txt
    │   ├── H0.txt
    │   ├── jmat.txt
    │   ├── P.txt
    │   ├── shellGam.txt
    │   └── shift.txt
    ├── data_c200/ ..... test data for electro.cpp computed from a C200 fullerene
    │   ├── atomicGam.txt
    │   └── ...
    └── electro.cpp ..... SYCL impl for computing electrostatic energy
```

A staging repository for the code that ended up in this report can be found at `https://github.com/AceMouse/Simple-GFNn-xTB`

```
Simple-GFNn-xTB/
├── chemistry_constants.py ..... well known constants, not specific to any method
├── cmp_impls.py ..... validation tests against Fortran impl
├── D4_constants.py ..... constants referenced in D4'
├── data/
├── file_formats.py ..... for reading xyz and coord files
├── flake.lock
├── flake.nix
├── GFN2_constants.py ..... constants referenced in GFN2-xTB
├── GFN2_xTB.py ..... Implementation of GFN2-xTB
├── README.md
└── slater_constants.py ..... constants related the approximation of STOs in GFN2-xTB
```

Introduction to GFN2-xTB

3

In this section we will go over the GFN2-xTB energy terms put forth in the related papers[1–3]. GFN2-xTB is a part of the GFNn-xTB (Geometry, Frequency, Non covalent, eXtended Tight Binding) family of semi-empirical methods for computational chemistry.

The method gives good approximations for molecular geometries, vibrational frequencies, and non-covalent interaction energies and also does well on a variety other properties. Overall it strives to hit a balance between being accurate, close to the physics, general over a wide range of elements and not too computationally expensive.

This is achieved by approximating a true quantum mechanical simulation, using carefully chosen approximations and parameters. In contrast to forcefield methods that often operate on the level of atoms or even functional groups interacting, GFN2-xTB still treats the calculations at the level of individual orbitals in many places.

GFN2-xTB uses a Self-Consistent Charges (SCC) approach i.e. it makes an initial guess at a density matrix and an energy which it then iteratively refines until both have converged. We are mostly interested in the final energy.

3.1 Energy Terms

The terms in GFN2-xTB that determine the energy are as follows

$$E = E_{rep} + E_{disp}^{D4'} + E_{EHT} + E_{\gamma} + E_{AES} + E_{AXC} + E_{\Gamma} + G_{Fermi} \tag{3.1}$$

We then have to also factor in the SCC. Let us consider the following code sketch in figure 3.1 which we will fill out in this chapter. The chapter will go over each term and add code snippets illustrating the mathematics in a more concrete way.

```python
def get_GFN2_energy(atoms: list[int], positions : list[list[float]],
↪  free_electrons : int, charge : int) -> float:
    density = density_initial_guess(atoms)
    overlap, dipol_dipol, charge_quadrupol = overlap_dipol_quadrupol(atoms,
    ↪  positions)
    huckel_matrix = huckel_matrix(atoms, positions, overlap)
    charges = mulliken_population_analysis(density, atoms)
    E_repulsion = repulsion_energy(atoms, positions)
    E_dispersion = D4Prime_energy(charges, atoms, positions)
    E_huckel = huckel_energy(density, extended_huckel_theory_matrix)
    E_anisotropic = anisotropic_energy(charges, overlap, dipol_dipol,
    ↪  charge_quadrupol, positions)
    E_isotropic = isotropic_energy(charges, positions)
    initial_hamiltonian = fock_matrix(...)
    eigen_values = diagonalise(initial_hamiltonian, overlap) # HC=SCe
    E_fermi = fermi_energy(eigen_values, atoms, free_electrons, charge)
    E = E_repulsion + E_dispersion + E_huckel + E_anisotropic + E_isotropic +
    ↪  E_Fermi
    while not (energy_converged and densities_converged):
        # change densities along the gradients and
        # update everything using the new density
        # ...
        energy_converged = (E-E_new)**2 < tolerance
        densities_converged = error_squared(density, new_density) < tolerance
        E = E_new
        density = new_density
```

Figure 3.1: Python code illustrating the main loop of a GFN2-xTB implementation. Line 2-14 computes the energy in a non SCC manner, line 15-22 iteratively improves the energy using SCC. Another way to describe this is that the zeroth iteration of SCC happens outside the loop as there is some setup required.

Let us define a small helper function to decrease the amount of indentation in the later code examples. This will allow us to easily loop over the orbitals and know which atom and shell each belongs to.

```python
def get_orbitals(atoms: list[int]) -> list[tuple[int]]:
    orbitals = []
    for atom_idx,atom in enumerate(atoms):
        for subshell in range(number_of_subshells[atom]):
            l = angular_momentum_of_subshell[atom][subshell]
            for orbital in range(l*2+1):
                orbitals.append((atom_idx,atom,subshell,orbital))
    return orbitals
```

Figure 3.2: Python code for generating a convenient list of orbitals to iterate though.

And some functions used when we want to signal to the reader that the python list we just created will stay a fixed size. Additionally we get to contain the python list comprehension syntax here.

```python
def vector(n: int) -> list[float]:
    return [0.0 for _ in range(n)]

def square_matrix(n: int) -> list[list[float]]:
    return [[0.0 for _ in range(n)] for _ in range(n)]

def square_matrix_of_vectors(n: int, v:int) -> list[list[list[float]]]:
    return [[[0.0 for _ in range(v)] for _ in range(n)] for _ in range(n)]
```

Figure 3.3: Python code for creating a n long vector, $n \times n$ matrix and $n \times n \times v$ tensor.

For many of the upcoming calculations we will need to work with the distance between atoms. We will say the position of atom A is:

$$\boldsymbol{R}_A = \begin{pmatrix} R_{Ax} \\ R_{Ay} \\ R_{Az} \end{pmatrix} \tag{3.2}$$

Where R_u with $u \in x, y, z$ is the coresponding component in $\boldsymbol{R}$.

Then we note the euclidean distance as follows:

$$\boldsymbol{R}_{AB} = \boldsymbol{R}_A - \boldsymbol{R}_B = \begin{pmatrix} R_{Ax} - R_{Bx} \\ R_{Ay} - R_{By} \\ R_{Az} - R_{Bz} \end{pmatrix} \tag{3.3}$$

$$R_{AB} = \sqrt{(R_{ABx})^2 + (R_{ABy})^2 + (R_{ABx})^2} \tag{3.4}$$

Now let us dig into the calculations.

3.2 Constructing the Initial Density Matrix

Many of the terms use the density matrix as part of their calculation and it is central to the SCC method. In the GFN2 paper the initial density matrix guess is formulated as a superposition of neutral atomic reference densities $P_0 = \sum_A P_{A_0}$. In simpler terms this means that we let P_0 be a diagonal matrix that is n by n where n is the total number of orbitals across the whole molecule. The values on the diagonal are the fractional number of electrons in the orbitals, the fractional occupations. They are reference densities so we get them in a table we can index into per subshell, thus we just have to divide the electrons evenly between the orbitals in the subshell. The number of orbitals in a subshell is $2l+1$ where l is the quantum number corresponding to the angular momentum of the subshell.

```
def density_initial_guess(atoms: list[int]) -> list[list[float]]:
    orbitals = get_orbitals(atoms)
    fractional_occupations = square_matrix(len(orbitals))
    for orbital_idx,(_,atom,subshell,_) in enumerate(orbitals):
        l = angular_momentum_of_subshell[atom][subshell]
        orbitals_in_subshell = l*2+1
        electrons_in_subshell = reference_occupations[atom][subshell]
        electrons_per_orbital = electrons_in_subshell/orbitals_in_subshell
        fractional_occupations[orbital_idx][orbital_idx] = electrons_per_orbital
    return fractional_occupations
```

Figure 3.4: Python code for computation of the initial density matrix P_0

For fullerenes the guess is simply all ones on the diagonal as `number_of_subshells[C] = 2`, `angular_momentum_of_subshell[C] = [0,1,0]` and `reference_occupations[C] = [1.0,3.0,0.0]`. Here `C=5` is the index for carbon. Thus `fractional_occupations` will contain a repeating series of $\frac{1}{0\cdot2+1}, \frac{3}{1\cdot2+1}, \frac{3}{1\cdot2+1}, \frac{3}{1\cdot2+1}$ on the diagonal.

3.3 Constructing the Overlap, Dipol and Quadrupol Tensors

We need the overlap matrix, S, for several of the terms. The related dipol and quadrupol tensors denoted D and Q are also needed and due to their very similar construction we will go over the implementation and mathematics in one fell swoop. An element in each tensor is computed in the following way.

$$S_{\nu\mu} = \langle\psi_\nu|\psi_\mu\rangle \qquad \forall\nu \in l \in A, \quad \forall\mu \in l' \in B, \tag{3.5}$$

$$D^u_{\nu\mu} = \langle\psi_\nu|(r_u - R_{Pu})|\psi_\mu\rangle \qquad \forall u \in x, y, z \tag{3.6}$$

$$Q^{uv}_{\nu\mu} = \langle\psi_\nu|(r_u - R_{Pu})(r_v - R_{Pv})|\psi_\mu\rangle \qquad \forall v \in x, y, z \tag{3.7}$$

These are integrals between two Slater type orbitals (STOs) over the positions r. When we write $l \in A$ we mean to iterate though the subshells in every element A and take their angular momentum. $\nu \in l$ means that we iterate though the $2l+1$ orbitals in the subshell with quantum number $m \in \{-l, .., 0, .., l\}$. See figure 3.4 for a code example. R_P is the x, y, z position of the product of the orbitals, we will define this later.

For D there are 3 unique values, corresponding to the 3 dimensions, to compute for each pair of orbitals which we will pack into an array. Since $(r_u - R_{Pu})(r_v - R_{Pv}) = (r_v - R_{Pv})(r_u - R_{Pu})$ there are 6, rather than 9, unique combinations of dimensions to compute for each element in the Q tensor which we will also pack into an array. For the sake of clarity we will define the indices corresponding to the combinations of directions, these are however arbitrary and only used for array indexing.

```
x = 0
y = 1
z = 2
xx = 0
yy = 1
zz = 2
xy = 3
xz = 4
yz = 5
```

Figure 3.5: Constants for the combinations of directions.

```python
def overlap_dipol_quadrupol(atoms: list[int], positions: list[list[float]])->
↪  tuple[list[list[float]],list[list[list[float]]],list[list[list[float]]]]:
    orbitals = get_orbitals(atoms)
    S = square_matrix(len(orbitals))
    D = square_matrix_of_vectors(len(orbitals),3)
    Q = square_matrix_of_vectors(len(orbitals),6)
    for idx_A, (atom_idx_A,atom_A,subshell_A,orbital_A) in enumerate(orbitals):
        for idx_B, (atom_idx_B,atom_B,subshell_B,orbital_B) in
        ↪  enumerate(orbitals):
            R_A = positions[atom_idx_A]
            R_B = positions[atom_idx_B]
            l_A = angular_momentum_of_subshell[atom_A][subshell_A]
            l_B = angular_momentum_of_subshell[atom_B][subshell_B]
            s,d,q = compute_STO_integrals(atom_A, subshell_A, orbital_A, R_A,
            ↪  l_A, atom_B, subshell_B, orbital_B, R_B, l_B)
            S[idx_A][idx_B] = s
            for dir in [x,y,z]:
                D[idx_A][idx_B][dir] = d[dir]
            for dir in [xx,yy,zz,xy,xz,yz]:
                Q[idx_A][idx_B][dir] = q[dir]
    return S,D,Q
```

Figure 3.6: Python snippet illustrating construction of the overlap, dipol and quadrupol tensors; S, D and Q.

To compute the integrals we need to remember that in GFN2-xTB an STO is approximated as a linear combination of gaussian type orbitals:

$$\begin{aligned}
|\psi_\nu\rangle =& \psi(\zeta_{A,l}, \boldsymbol{r} - \boldsymbol{R}_A) \\
=& N_{STO,l} |\boldsymbol{r} - \boldsymbol{R}_A|^{n-1} e^{-\zeta_{A,l}|\boldsymbol{r}-\boldsymbol{R}_A|} Y_l^m(\boldsymbol{r} - \boldsymbol{R}_A) \\
\approx& \sum_i^{N_{A,l}} c_{i,\nu} N_{GTO,l} (r_x - R_{Ax})^{l_x} (r_y - R_{Ay})^{l_y} (r_z - R_{Az})^{l_z} e^{-\alpha_{i,\nu}|\boldsymbol{r}-\boldsymbol{R}_A|^2} \\
=& \sum_i^{N_{A,l}} c_{i,\nu} \phi(\alpha_{i,\nu}, \boldsymbol{r} - \boldsymbol{R}_A) \\
=& \sum_i^{N_{A,l}} c_{i,\nu} |\phi_i\rangle
\end{aligned} \tag{3.8}$$

Where $N_{A,l}$ is an element and shell dependant constant, it is the number of GTOs used to approximate the STO. ζ and α are the slater and gaussian exponents. The contraction coefficient c is a fitted value, it is fitted with the assumption that $\zeta = 1$ and to get the value we will actually be using, we need to scale the fitted value by ζ^2. $N_{STO,l}$ and $N_{GTO,l}$ are normalisation terms. We will refer to

the terms $(r_u - R_{Au})^{l_u} \quad \forall u \in x, y, z$ as polynomial prefactors. The approximation in terms of GTOs means we can express our overlap in terms of GTO integrals instead:

$$S_{\nu\mu} = \sum_{i}^{N_{A,l}} \sum_{j}^{N_{B,l'}} c_{i,\nu} c_{j,\mu} \langle \phi_i | \phi_j \rangle \tag{3.9}$$

$$D^{u}_{\nu\mu} = \sum_{i}^{N_{A,l}} \sum_{j}^{N_{B,l'}} c_{i,\nu} c_{j,\mu} \langle \phi_i | (r_u - R_{Pu}) | \phi_j \rangle \tag{3.10}$$

$$Q^{uv}_{\nu\mu} = \sum_{i}^{N_{A,l}} \sum_{j}^{N_{B,l'}} c_{i,\nu} c_{j,\mu} \langle \phi_i | (r_u - R_{Pu})(r_v - R_{Pv}) | \phi_j \rangle \tag{3.11}$$

```
def compute_STO_integrals(atom_A: int, subshell_A: int, orbital_A: int, R_A:
↪  float, l_A: int, atom_B: int, subshell_B: int, orbital_B: int, R_B: float,
↪  l_B: int) -> tuple[float,list[float],list[float]]:
    number_of_gaussians_A = number_of_gaussians[atom_A][subshell_A]
    number_of_gaussians_B = number_of_gaussians[atom_B][subshell_B]
    slater_exponent_A = slater_exponents[atom_A][subshell_A]
    slater_exponent_B = slater_exponents[atom_B][subshell_B]
    overlap = 0
    dipol = vector(3)
    quadrupol = vector(6)
    for gaussian_i in range(number_of_gaussians_A):
        for gaussian_j in range(number_of_gaussians_B):
            exponent_i = normalised_gaussian_exponent(atom_A, subshell_A,
            ↪  gaussian_i, slater_exponent_A)
            exponent_j = normalised_gaussian_exponent(atom_B, subshell_B,
            ↪  gaussian_j, slater_exponent_B)
            contraction_i = normalised_contraction_coeficient(atom_A, subshell_A,
            ↪  gaussian_i, exponent_i)
            contraction_j = normalised_contraction_coeficient(atom_B, subshell_B,
            ↪  gaussian_j, exponent_j)
            distance_between_A_B = euclidean_distance(R_A, R_B)
            s,d,q = compute_GTO_integrals(R_A, l_A, orbital_A, R_B, l_B,
            ↪  orbital_B, exponent_i, exponent_j, distance_between_A_B)
            overlap += contraction_i*contraction_j*s
            for dir in [x,y,z]:
                dipol[dir] += contraction_i*contraction_j*d[dir]
            for dir in [xx,yy,zz,xy,xz,yz]:
                quadrupol[dir] += contraction_i*contraction_j*q[dir]
    return overlap,dipol,quadrupol
```

Figure 3.7: Python snippet illustrating construction of the STO integrals $\langle \psi_\nu | \cdots | \psi_\mu \rangle$.

The gaussian normalisation term[**daudel**] is:

$$1 = N_{GTO,l}^2 (2\alpha_{i,\nu})^{-(l+3/2)} \frac{(l_{Ax}-1)!!(l_y-1)!!(l_z-1)!!}{2^{l/2}} \pi^{3/2} \tag{3.12}$$

$$N_{GTO,l}^2 = \left((2\alpha_{i,\nu})^{-(l+3/2)} \frac{(l_x-1)!!(l_y-1)!!(l_z-1)!!}{2^{l/2}} \pi^{3/2} \right)^{-1} \tag{3.13}$$

$$N_{GTO,l} = \left((2\alpha_{i,\nu})^{-(l+3/2)} \frac{(l_x-1)!!(l_y-1)!!(l_z-1)!!}{2^{l/2}} \pi^{3/2} \right)^{-1/2} \tag{3.14}$$

Here l_u for $u \in x, y, z$ is the angular momentum in that dimension for the orbital, summing over u gives l. Which for fullerenes would be

$$N_{GTO,l} = \left((2\alpha_{i,\nu})^{-(l+3/2)} 2^{-l/2} \pi^{3/2} \right)^{-1/2} \tag{3.15}$$

We only need the GTO ones and can compute them in the following way:

```python
def normalised_gaussian_exponent(atom: int, subshell: int, gaussian: int,
↪  slater_exponent: float) -> float:
    normalisation_factor = slater_exponent**2
    return gaussian_exponents[atom][subshell][gaussian]*normalisation_factor

def normalised_contraction_coefficient(atom: int, subshell: int, gaussian: int,
↪  gaussian_exponent: float) -> float:
    l = angular_momentum_of_subshell[atom][subshell]
    l_x = angular_momentum_of_subshell_in_dimension[atom][subshell][x]
    l_y = angular_momentum_of_subshell_in_dimension[atom][subshell][y]
    l_z = angular_momentum_of_subshell_in_dimension[atom][subshell][z]
    df_x = double_factorial(l_x-1)
    df_y = double_factorial(l_y-1)
    df_z = double_factorial(l_z-1)
    normalization_factor = (2*alpha)**(-(l+3/2)) * ((df_x*df_y*df_z)/(2**(l/2)))
    ↪  * pi**(3/2)
    return contraction_coeficients[atom][shell][gaussian]*normalisation_factor

def double_factorial(n: int) -> int:
    if n <= 1:
        return 1
    return n*double_factorial(n-2)
```

Figure 3.8: Python snippet illustrating normalisation of the gaussian exponents and contraction coefficients.

As the GTOs in the integral are real valued functions we can drop the conjugation when we write them out. We see that we have to compute the products of GTOs. The product of two Gaussians is a new Gaussian centred at a point between the two, R_P, which we will finally introduce properly.

$$\alpha = \alpha_{i,\nu} + \alpha_{j,\mu} \tag{3.16}$$

$$K_{AB} = \left(\frac{2\alpha_{i,\nu}\alpha_{j,\mu}}{\alpha\pi}\right)^{\frac{3}{4}} e^{-\frac{\alpha_{i,\nu}\alpha_{j,\mu}}{\alpha}R_{AB}^2} \tag{3.17}$$

$$\boldsymbol{R}_P = \frac{\alpha_{i,\nu}\boldsymbol{R}_A + \alpha_{j,\mu}\boldsymbol{R}_B}{\alpha} \tag{3.18}$$

$$\phi(\alpha_{i,\mu}, \boldsymbol{r} - \boldsymbol{R}_A)\phi(\alpha_{j,\nu}, \boldsymbol{r} - \boldsymbol{R}_B) = K_{AB}\phi(\alpha, \boldsymbol{r} - \boldsymbol{R}_P) \tag{3.19}$$

This rewrite only works if we also shift the polynomial prefactors to be relative to the product centre R_P, for all $u \in x, y, z$[4, eq. 5.46]:

$$\begin{aligned}(r_u - R_{Au})^{l_{Au}} &= \sum_{m_i=0}^{l_{Au}} \binom{l_{Au}}{m_i} (R_{Pu} - R_{Au})^{(l_{Au}-m_i)} (r_u - R_{Pu})^{l_{Au}} \\ &= \sum_{m_i=0}^{l_{Au}} v_{m_i} (r_u - R_{Pu})^{l_{Au}}\end{aligned} \tag{3.20}$$

```
def shift_polynomial(l_dim: int, difference_to_P_dim: float) -> list[float]:
    poly_coefficients = vector(l_dim+1)
    for m in range(l_dim+1):
        poly_coefficients[m] = comb(l_dim, m)*difference_to_P_dim**(l_dim-m)
    return poly_coefficients
```

Figure 3.9: Python code illustrating computation of the shifted polynomial coefficients.

If we multiply these shifted prefactors we get the prefactors for the product.

$$\begin{aligned}(r_u - R_{Au})^{l_{Au}}(r_u - R_{Bu})^{l_{Bu}} &= \sum_{m_i=0}^{l_{Au}} \sum_{m_j=0}^{l_{Bu}} v_{m_i} v_{m_j} (r_u - R_{Pu})^{l_{Au}+l_{Bu}} \\ &= \sum_{t}^{l_{Au}+l_{Bu}} v_t (r_u - R_{Pu})^t\end{aligned} \tag{3.21}$$

In the code we compute the values of v_t as a convolution:

```
def convolute(coef_A: list[float], coef_B: list[float]) -> list[float]:
    max_t = len(coef_A)+len(coef_B)
    poly_coefficients = vector(max_t+1)
    for i,ci in enumerate(coef_A):
        for j,cj in enumerate(coef_B):
            poly_coefficients[i+j] += ci*cj
    return poly_coefficients
```

Figure 3.10: Python code illustrating computation of the R_P polynomial coefficients via a convolution.

To compute the integrals over the GTOs we can split them in terms of the 3 dimensions we are integrating over, x, y and z. We can also pull the GTO normalisation terms out front. Each of these integrals can now be expanded using the analytical solution to Gaussian integrals of this form. All the integrals get solutions of the following form for even t[4, eq. 5.53]:

$$i_t = \alpha^{-(t+1)/2} \frac{(t-1)!!}{2^{t/2}} \sqrt{\pi} \tag{3.22}$$

And 0 for odd t. The analytical solution to the integrals is independent of the direction so we can precompute them:

```
def compute_gaussian_integral_factors(alpha:float, l_A:int, l_B:int)->
↪   list[float]:
    factors = vector(l_A+l_B+3)
    for t in range(l_A+l_B+3):
        if t % 2 == 0:
            df = double_factorial(t-1)
            factors[t] = alpha**(-(t+1)/2) * (df/(2**(t/2))) * sqrt(pi)
        else:
            factors[t] = 0
    return factors
```

Figure 3.11: Python code illustrating computation of the integral factors for even and odd powers

We compute an extra 2 factors to accommodate for the second moment. We then multiply each with the appropriate v_t and sum over t[4, eq. 5.49]. We also compute the higher moments[4, sec. 5.2.2.5] in the same code block in accordance with the sum:

$$\sum_{t=0}^{l+M} \sum_{m=0}^{M} \binom{M}{m} {R_P}_u^{M-m} v_t i_t \tag{3.23}$$

```
def compute_GTO_integrals(R_A: float, l_A: int, orbital_A: int, R_B: float, l_B:
↪  int, orbital_B: int, exponent_i: float, exponent_j: float,
↪  distance_between_A_B: float) -> tuple[float,list[float],list[float]]:
    alpha = exponent_i + exponent_j
    exponents = exponent_i * exponent_j
    K_AB = ((2*exponents)/(alpha*pi))**(3/4) * e**(-(exponents/alpha)*distance)
    integral_factors = compute_gaussian_integral_factors(alpha, l_A, l_B)
    zeroth_moment = vector(3)
    first_moment = vector(3)
    second_moment = vector(3)
    for dim in [x,y,z]:
        gaussian_product_center_in_dim =
        ↪  (exponent_i*R_A[dim]+exponent_j*R_B[dim])/alpha
        centre_relative_to_A = gaussian_product_center_in_dim-R_A[dim]
        centre_relative_to_B = gaussian_product_center_in_dim-R_B[dim]
        l_A_dim = angular_momentum_in_dimension[l_A][orbital_A][dim]
        l_B_dim = angular_momentum_in_dimension[l_B][orbital_B][dim]
        l_max_dim = max(l_A_dim, l_B_dim)
        vmis = shift_polynomial(l_A_dim, centre_relative_to_A)
        vmjs = shift_polynomial(l_B_dim, centre_relative_to_B)
        vts = convolute(vmis, vmjs)
        for t, vt in enumerate(vts):
            zeroth_moment[dim] += vt * integral_factors[t+m]
            for m in range(1+1):
                first_moment[dim] += comb(1,m) *
                ↪  gaussian_product_center_in_dim**(1-m) * vt *
                ↪  integral_factors[t+m]
            for m in range(2+1):
                second_moment[dim] += comb(2,m) *
                ↪  gaussian_product_center_in_dim**(2-m) * vt *
                ↪  integral_factors[t+m]

    overlap = zeroth_moment[x]*zeroth_moment[y]*zeroth_moment[z]
    dipol = vector(3)
    dipol[x] = first_moment[x]*zeroth_moment[y]*zeroth_moment[z]
    dipol[y] = zeroth_moment[x]*first_moment[y]*zeroth_moment[z]
    dipol[z] = zeroth_moment[x]*zeroth_moment[y]*first_moment[z]

    quadrupol = vector(6)
    quadrupol[xx] = second_moment[x]*zeroth_moment[y]*zeroth_moment[z]
    quadrupol[yy] = zeroth_moment[x]*second_moment[y]*zeroth_moment[z]
    quadrupol[zz] = zeroth_moment[x]*zeroth_moment[y]*second_moment[z]
    quadrupol[xy] = first_moment[x]*first_moment[y]*zeroth_moment[z]
    quadrupol[xz] = first_moment[x]*zeroth_moment[y]*first_moment[z]
    quadrupol[yz] = zeroth_moment[x]*first_moment[y]*first_moment[z]
    return overlap, dipol, quadrupol
```

Figure 3.12: Python code illustrating computation of the integral $\langle\phi_i|\cdots|\phi_j\rangle$ using the dimension-wise decomposition.

3.4 Extended Hückel Energy

As a part of computing the Hückel energy we need to construct the Hückel matrix. The Hückel matrix comes from extended Hückel theory. Before we can construct it we need a few definitions. First the GFN2 type coordination number for atom A is

$$CN'_A = \sum_{B \neq A} \left(1 + e^{-10\left(\frac{4\left(R_{A,cov}+R_{B,cov}\right)}{3R_{AB}}-1\right)}\right)^{-1} \left(1 + e^{-20\left(\frac{4\left(R_{A,cov}+R_{B,cov}+2\right)}{3R_{AB}}-1\right)}\right)^{-1} \tag{3.24}$$

Where $R_{A,cov}$ is the covalent atomic radius of A specified for the GFN2 method. Now with EN_A being the Pauling electronegativity and $k_EN = 0.02$ as well as $k^{\text{poly}}_{A,l}$ being GFN2 specific constants we define the following 3 functions:

$$X(EN_A, EN_B) = 1 + k_{EN}(EN_A - EN_B)^2 \tag{3.25}$$

$$\Pi(R_{AB}, l, l') = \left(1 + k^{\text{poly}}_{A,l}\left(\frac{R_{AB}}{R_{A,cov}+R_{B,cov}}\right)^{\frac{1}{2}}\right)\left(1 + k^{\text{poly}}_{B,l'}\left(\frac{R_{AB}}{R_{A,cov}+R_{B,cov}}\right)^{\frac{1}{2}}\right) \tag{3.26}$$

$$Y(\zeta_{A,l}, \zeta_{B,l'}) = \left(\frac{2\sqrt{\zeta_{A,l}\zeta_{B,l'}}}{\zeta_{A,l}+\zeta_{B,l'}}\right)^{\frac{1}{2}} \tag{3.27}$$

The Hückel matrix is then calculated as follows:

$$H^{EHT}_{\nu\nu} = H^l_A - H_{CN_A} CN'_A \tag{3.28}$$

$$\begin{aligned} H^{EHT}_{\mu\nu} = {} & \frac{1}{2} K^{ll'}_{AB} S_{\mu\nu}(H^{EHT}_{\mu\mu} + H^{EHT}_{\nu\nu}) \\ & \cdot X(EN_A, EN_B) \\ & \cdot \Pi(R_{AB}, l, l') \\ & \cdot Y(\zeta^A_l, \zeta^B_{l'}), \qquad \forall \mu \in l \in A, \nu \in l' \in B \end{aligned} \tag{3.29}$$

$K^{ll'}_{AB}$ is an element and shell specific fitted constant however, in GFN2 it only depends on the shells. $S_{\mu\nu} = \langle \phi_\mu | \phi_\nu \rangle$ is the just introduced overlap of the orbitals. H^l_A and $H^l_{CN'_A}$ are both fitted constants. EN_A is the electronegativity of the element of atom A, given in the original `xtb` code.

```python
def huckel_matrix(atoms: list[int], positions: list[list[float]], overlap:
↪  list[list[float]]) -> list[list[float]]:
    orbitals = get_orbitals(atoms)
    H_EHT = square_matrix(len(orbitals))
    CN = get_coordination_numbers(atoms,positions)
    for orbital_idx, (atom_idx,atom,subshell,orbital) in enumerate(orbitals):
        CN_A = CN[atom_idx]
        H_A = self_energy[atom][subshell] # constant
        H_CN_A = GFN2_H_CN_A[atom][subshell] # constant
        H_EHT[orbital_idx][orbital_idx] = H_A - H_CN_A*CN_A

    for idx_A,(atom_A_idx,atom_A,subshell_A,_) in enumerate(orbitals):
        l_A = angular_momentum_of_subshell[atom_A][subshell_A]
        EN_A = electro_negativity[atom_A]
        R_A = positions[atom_A_idx]
        Rcov_A = covalent_radii[atom_A]
        k_poly_A = k_poly[atom_A][l_A]
        for idx_B,(atom_B_idx,atom_B,subshell_B,_) in enumerate(orbitals):
            if idx_A == idx_B:
                continue
            l_B = angular_momentum_of_subshell[atom_B][subshell_B]
            EN_B = electro_negativity[atom_B]
            R_B = positions[atom_B_idx]
            Rcov_B = covalent_radii[atom_B]
            k_poly_B = k_poly[atom_B][l_B]
            K_ll = GFN2_K_AB[l_A][l_B]
            delta_EN_squared = (EN_A-EN_B)**2
            k_EN = 0.02
            X = 1+k_EN*delta_EN_squared
            R_AB = euclidean_distance(R_A,R_B)
            Rcov_AB = Rcov_A + Rcov_B
            PI = (1+k_poly_A*sqrt(R_AB/Rcov_AB)) *
            ↪  (1+k_poly_B*sqrt(R_AB/Rcov_AB))
            slater_exp_A = slater_exponent[atom_A][l_A]
            slater_exp_B = slater_exponent[atom_B][l_B]
            Y = sqrt((2*sqrt(slater_exp_A*slater_exp_B)) /
            ↪  (slater_exp_A+slater_exp_B))
            H_nn = H_EHT[idx_A][idx_A]
            H_mm = H_EHT[idx_B][idx_B]
            S_nm = overlap[idx_B][idx_B]
            H_EHT[idx_A][idx_B] = k_ll*(1/2)*(H_nn+H_mm)*S_nm*Y*X*PI
    return H_EHT
```

Figure 3.13: Python code illustrating construction of the Hückel matrix

Using this we can now compute the Hückel energy where P is the density matrix at this point in the SCC procedure:

$$E_{EHT} = \sum_{\mu\nu} P_{\mu\nu} H^{EHT}_{\mu\nu} \tag{3.30}$$

```python
def huckel_energy(density_matrix:list[list[int]], huckel_matrix:list[list[int]])
↪ -> float:
    E_EHT = 0
    orbitals = len(density_matrix)
    for shell_mu in range(orbitals):
        for shell_nu in range(orbitals):
            E_EHT += density_matrix[shell_nu][shell_mu] *
            ↪ huckel_matrix[shell_mu][shell_nu]
    return E_EHT
```

Figure 3.14: Python code illustrating computation of the Hückel energy

3.5 Repulsion Energy

The repulsion energy in GFN2-xTB is computed classically using the following formula:

$$E_{rep} = \frac{1}{2} \sum_{A} \sum_{B \neq A} \frac{Z^{eff}_A Z^{eff}_B}{R_{AB}} e^{-\sqrt{a_A a_B}(R_{AB})^{(k_f)}} \tag{3.31}$$

$$k_f = \begin{cases} 1 & if A, B \in \{\text{H}, \text{He}\} \\ \frac{3}{2} & otherwise \end{cases} \tag{3.32}$$

The effective nuclear charge Z^{eff} and a are variables fitted for each element. R_{AB} is the distance between the A and B atoms. This can be implemented in the following way:

```python
H = 0
He = 1
def repulsion_energy(atoms: list[int], positions : list[list[float]]) -> float:
    sum = 0
    for idx_A, (atom_A, position_A) in enumerate(zip(atoms, positions)):
        for idx_B, (atom_B, position_B) in enumerate(zip(atoms, positions)):
            if idx_A == idx_B:
                continue
            k_f = 3/2
            if atom_A in [H, He] and atom_B in [H, He]:
                k_f = 1
            R_AB = euclidean_distance(position_A, position_B)
            alpha_A = repulsion_alpha[atom_A]
            alpha_B = repulsion_alpha[atom_B]
            Y_A = effective_nuclear_charge[atom_A]
            Y_B = effective_nuclear_charge[atom_B]
            sum += ((Y_A*Y_B)/R_AB) * e ** ( - sqrt(alpha_A*alpha_B) * R_AB**k_f
            ↪  )
    E_rep = (1/2)*sum
    return E_rep
```

Figure 3.15: Python code illustrating computation of the repulsion energy

3.6 Isotropic Energy

The isotropic energy terms E_γ and E_Γ require us to first do a Mulliken population analysis to get the subshell-wise partial Mulliken charges $q_{A,l}$.

$$q_{A,l} = \sum_{\mu \in l} \sum_{B} \sum_{l' \in B} \sum_{\nu \mathcal{V}} P_{\mu\nu} S_{\mu\nu} \tag{3.33}$$

With η_A being an element specific constant and $k_{A,l}$ being a shell specific constant we define, both specific to the parametrisation of GFN2:

$$\eta = \frac{(1 + k_{A,l})\eta_A + (1 + k_{B,l'})\eta_B}{2} \tag{3.34}$$

$$\gamma_{AB,ll'} = \sqrt{\frac{1}{R_{AB}^2 + \eta^{-2}}} \tag{3.35}$$

We can then define one of the two parts of the isotropic energy, E_γ

$$E_\gamma = \frac{1}{2} \sum_{A,B}^{N_{atoms}} \sum_{l \in A} \sum_{l' \in B} q_{A,l} q_{B,l'} \gamma_{AB,ll'} \tag{3.36}$$

With the element and shell specific GFN2 constant $\Gamma_{A,l}$ we also get the definition for the other part:

$$E_\Gamma = \frac{1}{3} \sum_{A}^{N_{atoms}} \sum_{l \in A} (q_{A,l})^3 \Gamma_{A,l} \tag{3.37}$$

3.7 Anisotropic Electrostatic Energy

The anisotropic electrostatic energy term E_{AES} is supposed to capture changes in the energy for electrostatic interactions that arise from anisotropic density distributions (polarization). It represents the energy caused by charge-dipole interactions and charge-quadrupole interactions.

Below we introduce all the necessary definitions for the energy term. $\boldsymbol{\mu}_A$ is the cumulative atomic dipole moment of atom A, and $\boldsymbol{\Theta}_A$ as the corresponding traceless quadrupole moment. The superscript T is the vector transpose operation.

$$\boldsymbol{\mu}_A = \begin{pmatrix} \mu_A^x & \mu_A^y & \mu_A^z \end{pmatrix}^T \tag{3.38}$$

$$\boldsymbol{\Theta}_A = \begin{pmatrix} \Theta_A^{xx} & \Theta_A^{xy} & \Theta_A^{xz} \\ \Theta_A^{yx} & \Theta_A^{yy} & \Theta_A^{yz} \\ \Theta_A^{zx} & \Theta_A^{zy} & \Theta_A^{zz} \end{pmatrix} \tag{3.39}$$

Θ_A^{uv} is a specific element in the traceless quadrupole moment matrix for A.

$$\Theta_A^{uv} = \frac{3}{2}\theta_A^{uv} - \frac{\delta_{uv}}{2}(\theta_A^{xx} + \theta_A^{yy} + \theta_A^{zz}) \tag{3.40}$$

Below we define the zeroth, first, and second order cumulative atomic multipole moments (CAMM) q_A, μ_A^u, and θ_A^{uv}.

$$q_A = Z_A - \sum_{l \in A} q_{A,l} \tag{3.41}$$

q_A is the Mulliken charge for an atom A. Z_A is the number of protons in the atom A. $q_{A,l}$ is the previously defined partial Mulliken charge.

$$\mu_A^u = \sum_{l \in A} \sum_{\kappa \in l} \sum_{B} \sum_{l' \in B} \sum_{\lambda \in l'} P_{\kappa\lambda}(R_{A_u} S_{\lambda\kappa} - D^u_{\lambda\kappa}) \tag{3.42}$$

μ_A^u is the cumulative atomic dipole moment for a specific dimension u of atom A. $D^u_{\lambda\kappa}$ is an element in the dipole tensor.

$$\theta_A^{uv} = \sum_{l \in A} \sum_{\kappa \in l} \sum_{B} \sum_{l' \in B} \sum_{\lambda \in l'} P_{\kappa\lambda}(R_{A_u} D^v_{\lambda v} + R_{A_v} D^u_{\lambda\kappa} - R_{A_u} R_{A_v} S_{\lambda\kappa} - Q^{uv}_{\lambda\kappa}) \tag{3.43}$$

The Cartesian components u and v are the dimensions for which we compute the second order CAMM.

At short distances the terms are dampened to avoid divergence for the AES energy. The distance dependent damping function is given by:

$$f_n(R_{AB}) = \frac{1}{R_{AB}^n} \times \frac{1}{1 + 6\left(\frac{R_0^{AB}}{R_{AB}}\right)^{a_n}} \tag{3.44}$$

a_n are adjusted global parameters and R_{AB}^n is the distance to the power of n.

$$a_3 = 3.0 \qquad a5 = 4.0$$

Now we have all the definitions needed for the anisotropic electrostatic energy, which is given by:

$$\begin{aligned} E_{AES} &= E_{q\mu} + E_{q\Theta} + E_{\mu\mu} \\ &= \frac{1}{2}\sum_{A,B}\{f_3(R_{AB})[q_A(\boldsymbol{\mu}_B^T\boldsymbol{R}_{BA}) + q_B(\boldsymbol{\mu}_A^T\boldsymbol{R}_{AB})] \\ &\quad + f_5(R_{AB})[q_A\boldsymbol{R}_{AB}^T\boldsymbol{\Theta}_B\boldsymbol{R}_{AB} + q_B\boldsymbol{R}_{AB}^T\boldsymbol{\Theta}_A\boldsymbol{R}_{AB} \\ &\quad - 3(\boldsymbol{\mu}_A^T\boldsymbol{R}_{AB})(\boldsymbol{\mu}_B^T\boldsymbol{R}_{AB}) + (\boldsymbol{\mu}_A^T\boldsymbol{\mu}_B)R_{AB}^2]\} \end{aligned} \tag{3.45}$$

3.8 Anisotropic Exchange Correlation

The anisotropic exchange correlation term E_{AXC} is supposed to capture changes in the atomic exchange (XC) energy that arise from anisotropic density distributions (polarization). It represents the energy caused by charge-dipole interactions and charge-quadrupole interactions.

$$E_{AXC} = \sum_A (f_{XC}^{\mu_A}|\boldsymbol{\mu}_A|^2 + f_{XC}^{\Theta_A}||\boldsymbol{\Theta}_A||^2) \tag{3.46}$$

In the equation we have the constants $f_{XC}^{\mu_A}$ and $f_{XC}^{\Theta_A}$, which are fitted element-specific parameters. The values for each element can be found in the supporting information of the GFN2-xTB paper[2]. We already defined both $\boldsymbol{\mu}_A$ and $\boldsymbol{\Theta}_A$ in section 3.7.

3.9 Electronic Entropy

In GFNn-xTB they use the following term to assign the (spin restricted) fractional orbital occupations. It is based on the eigenvalues from diagonalising $FC = SC\varepsilon$ where F is the GFN2-xTB equivalent of a Fock matrix, and C is the matrix of GTO contraction coefficients. S is the overlap matrix and ε is the diagonal matrix of eigen values. The electronic temperature $T_{el} = 300$ kelvin is non-zero and thus we have fractional occupations $n_{i\sigma}$ for the i'th molecular spin orbital with spin σ chosen based on the Fermi-distribution::

$$n_{i\sigma} = \frac{1}{e^{(\varepsilon_{ii} - \varepsilon_F^\sigma)/(K_B T_{el})} + 1} \tag{3.47}$$

with the Fermi level ε_F^σ chosen such that the following equation holds $\sum_i n_{i\sigma} = N_\sigma$. Here K_B is the Boltzmann constant. $N_\alpha = \frac{N-N_f}{2} + N_f$ and $N_\beta = \frac{N-N_f}{2}$ is based on the total number of electrons $N = C + \sum_A Z_A$ and the requested number of free electrons N_f with the restriction that $0 \leq N - N_f$ is even. C is the requested total charge. The Fermi energy term is then:

$$G_{Fermi} = K_B T_{el} \sum_{\sigma\in\alpha,\beta} \sum_i n_{i\sigma} \ln(n_{i\sigma}) + (1 - n_{i\sigma}) \ln(1 - n_{i\sigma}) \tag{3.48}$$

This repeated diagonalising is one of the more computationally demanding tasks for the GFNn-xTB methods.

3.10 Dispersion Energy

The dispersion energy term is used to include the energy caused by London dispersion forces.

A London dispersion force is a temporary attractive force that occurs when electrons in two adjacent atoms occupy positions that causes the atoms to form temporary dipoles[5].

This happens from temporary fluctuations in the electron distribution around a molecule. Expansions and contractions of the atomic density will therefore respectively incerase and decrease the magnitude or the interation[2].

The two-body London dispersion energy is given by

$$-\sum_{A>B} \sum_{n=6,8} s_n \frac{C_n^{AB}(q_A, CN_{cov}^A, q_B, CN_{cov}^B)}{R_{AB}^n} f_{damp,BJ}^{(n)}(R_{AB}) \tag{3.49}$$

The values q_A and q_B are the Mulliken charges for the atoms, and the scaling parameters s_n are defined as

$$s6 = 1.0 \qquad s8 = 2.7$$

The Becke–Johnson (BJ) damping function is used because it has shown to provide better results for nonbonded distances[6], which are the only type of distances we consider in xTB methods. With it, no pair-specific cut-off radii are required and no artificial repulsive interatomic forces at short distances are present. The BJ damping function is given by

$$f_{damp,BJ}^{(n)}(R_{AB}) = \frac{R_{AB}^n}{R_{AB}^n + (a_1 R_{AB}^0 + a_2)^n} \tag{3.50}$$

Equation 3.50 is taken from [3] where they raise the parenthesised term in the denominator to the power of n, whereas the constant 6 is used in the GFN2 paper[2]. We do not have an explanation for this, so it is worth taking note of this discrepancy.

The global damping parameters are

$$a1 = 0.52 \qquad a2 = 5.0$$

and the damping constant is given by

$$R_{AB}^0 = \sqrt{C_8^{AB}/C_6^{AB}} \tag{3.51}$$

C_6^{AB} is the pairwise dipole-dipole dispersion coefficients calculated by numerical integration as given by

$$C_6^{AB} = \frac{3}{\pi}\sum_j w_j \overline{\alpha}_A(i\omega_j, q_A, CN_{cov}^A)\overline{\alpha}_B(i\omega_j, q_B, CN_{cov}^B) \tag{3.52}$$

C_9^{ABC} is the triple-dipole constant and is given by

$$C_9^{ABC} = \frac{3}{\pi}\int_0^\infty \alpha^A(i\omega)\alpha^B(i\omega)\alpha^C(i\omega)d\omega \tag{3.53}$$

This three-body contribution is typically $< 5 - 10\%$ of E_{disp}, so it is small enough that we can reasonably approximate the coefficients by a geometric mean as[7]

$$C_9^{ABC} \approx -\sqrt{C_6^{AB} C_6^{AC} C_6^{BC}} \tag{3.54}$$

C_8^{AB} is calculated recursively from the lowest order C_6^{AB} coefficients.

$$C_8^{AB} = 3C_6^{AB}\sqrt{\mathcal{Q}^A\mathcal{Q}^B} \tag{3.55}$$

$$\mathcal{Q}^A = s_{42}\sqrt{Z^A}\frac{\langle r^4\rangle^A}{\langle r^2\rangle^A} \tag{3.56}$$

We have not been able to find s_{42} nor the fraction in any related research papers or supporting material, but we were able to find both in the code for the dftd4 project[1]. The whole term $\mathcal{Q}^A$ is stored as an array of element-specific constants, named `sqrt_z_r4_over_r2`, with s_{42} set to 0.5. Lastly we have Z^A, which is the atomic number of A.

Equation 3.57 is the Axilrod–Teller–Muto (ATM) term, which describes the three-body interaction energy between atoms, and Equation 3.58 is the corresponding zero-damping function for the term.

$$-s_9 \sum_{A>B>C} \frac{(3cos(\theta_{ABC})cos(\theta_{BCA})cos(\theta_{CAB})+1)C_9^{ABC}(CN_{cov}^A, CN_{cov}^B, CN_{cov}^C)}{(R_{AB}R_{AC}R_{BC})^3} \tag{3.57}$$

$$f_{damp,zero}^{(9)}(R_{AB}, R_{AC}, R_{BC}) \tag{3.58}$$

The scaling parameter s_9 is 5.0, and the values θ_{ABC}, θ_{BCA}, and θ_{CAB} are the angles between the edges going from B, C, and A to the other two atoms respectively.

The final D4' energy term is given by

[1] `https://github.com/dftd4/dftd4`

$$
\begin{aligned}
E_{disp}^{D4'} = & -\sum_{A>B}\sum_{n=6,8} s_n \frac{C_n^{AB}(q_A, CN_{cov}^A, q_B, CN_{cov}^B)}{R_{AB}^n} f_{damp,BJ}^{(n)}(R_{AB}) \\
& - s_9 \sum_{A>B>C} \frac{(3cos(\theta_{ABC})cos(\theta_{BCA})cos(\theta_{CAB})+1)C_9^{ABC}(CN_{cov}^A, CN_{cov}^B, CN_{cov}^C)}{(R_{AB}R_{AC}R_{BC})^3} \\
& \times f_{damp,zero}^{(9)}(R_{AB}, R_{AC}, R_{BC})
\end{aligned}
\tag{3.59}
$$

The dispersion energy was the last of the energy terms in GFN2-xTB that we needed to cover. To reiterate, we have now walked through all the energy terms for GFN2-xTB, which are shown in full in Equation 3.1.

4 High Performance Parallel Computing

Now that we have taken a look at the computations performed by the xTB method, we can begin to see that the steps are the same for all molecules, and that the exact number of steps are identical between isomers. Isomers are molecules that are identical in their composition of atoms, that is, they have the same molecular formula, and differ only in the relative positions between atoms or the bonds between them. This project focuses on large isomer spaces of fullerenes in the range $C_{20}, \cdots, C_{200}$ which is a total of $2,157,751,423$ molecules, or in other words in the order of 10^9. A fullerene is a molecule consisting only of carbon atoms where each atom has three neighbors and is arranged such that the molecule produces a mesh of exactly 12 pentagonal faces and the rest being hexagonal faces.

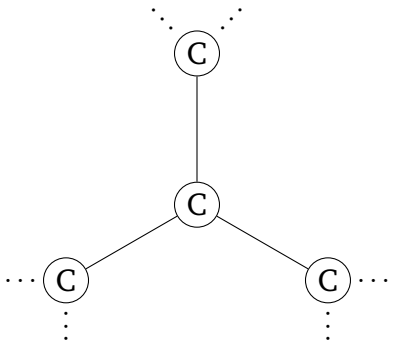


Figure 4.1: Each carbon atom in a fullerene has exactly 3 neighboring carbon atoms.

For each isomerspace we essentially have two matrices, one for the coordinates of each carbon atom, and one containing the three neighbors for each of the atoms. The first matrix as seen in Figure 4.2, is of size $N \times 3 \times M$ where N is the amount of atoms for each of the fullerenes within the same isomerspace, 3 is the three coordinates for each atom, and M is the number of fullerenes within the isomerspace. The second matrix as seen in Figure 4.3, is the exact same shape, but instead of coordinates it holds the indices for the three neighboring carbon atoms. The xTB method only needs the positions for each atom, so we are not concerned with the second matrix.

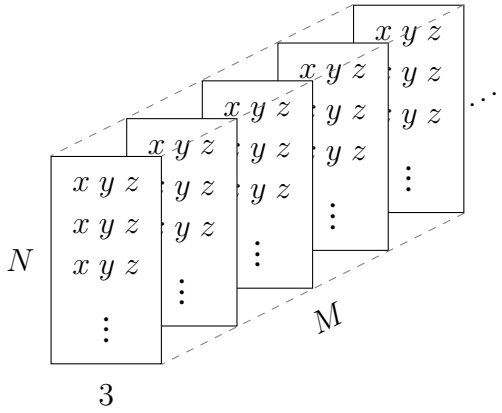


Figure 4.2: Matrix with the atom coordinates for each fullerene within an isomerspace. The 3-dimensional matrix holds the 3 coordinates for each of the N carbon atoms for all M fullerenes in a given isomerspace.

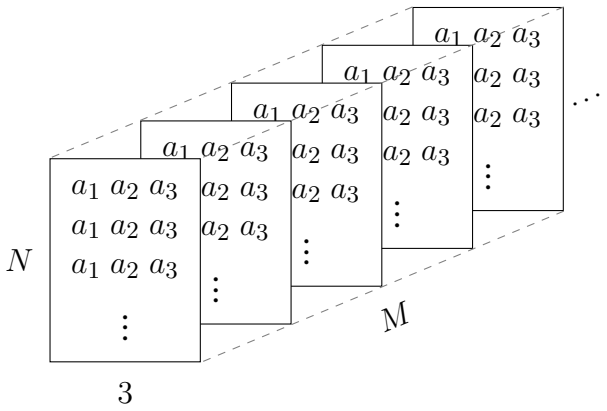


Figure 4.3: Matrix holding an adjacency list of 3 neighbors for each carbon atom for all fullerenes in a given isomerspace.

The xtb program takes a single molecule as input, but the fact that fullerenes in the same isomerspace executes the same series of instructions, makes a great case for lockstep-parallelism. Running computations in lockstep means doing the exact same instructions in parallel, and if we can do this, then we can efficiently utilize the single instruction, multiple threads (SIMT) execution model. The SIMT model is used when a warp scheduler dispatches a single instruction to all threads within a warp. All of the threads in the warp will then execute the instruction simultaniously, which means that they are synchronized at a hardware level. This is perfect as it means that we can use warps to execute instructions for multiple fullerenes in lockstep. With this in mind, instead of feeding the program a single fullerene at a time, we want to give it a complete isomerspace so that we can utilize all threads on one or even multiple GPUs. The largest isomerspace we consider is C_{200}, which is $214,127,742$ fullerenes, or in other words in the order of 10^8. Parallelism scales with the isomerspace, so this means that we can use on the order of 10^8 GPU cores if available.

It takes around 30 seconds for a 12th generation Intel mobile processor to run xtb on a single C_{200} fullerene. Going through the whole isomerspace on this CPU would take about 204 years to complete. This is a bit long to wait for a failed run, and by that time we will probably have achieved more efficient ways of computation, or the world might simply have more pressing matters than keeping this computer alive. The degree of lockstep-parallelism scales with the isomerspace, and since there are no interdependencies between the molecules, this makes it a perfect case for horizontal scaling by using hundreds or even thousands or GPUs. The CPU used has 12 cores and 24 threads, 12 of which can run in parallel. An NVIDIA GPU has hundredthousands of threads and tenthousands of CUDA cores, though these cores are not equivalent to CPU cores, and we can in general not compare the relative performance between the two by looking at this. Something that

is typically compared though, is the theoretical throughput of floating point operations per second (FLOPS). The top of the line AMD EPYC 9965 CPU has a theoritical throughput of 27.6 TeraFLOPS (TFLOPS) for 32-bit floating point values[8], while the NVIDIA A100 has 156 TFLOPS[9]. From these observations we can say that the GPU is much more capable of massive parallelism, which is what we seek for these massive isomerspaces.

4.1 Case Study

To get an idea of the scale of parallelism we can achieve on a single GPU, we will take a look at the memory specifications of the NVIDIA A100 GPU. We hope to use this to get an idea of how many fullerenes can fit into memory. The equations we present are generally applicable to other GPUs, though be aware that terminology vary between manufacturers.

Using all the levels of memory on a GPU efficiently is crucial to achieve high performance in parallel computing tasks. Let us therefore take a moment to familiarize ourselves with the different types typically found in the memory hiearchy on a GPU. Below you will find an overview of the main memory levels with short explanations for each of them:

- Global - Accessible from all threads on the GPU. This is the largest but also the slowest pool of memory in terms of latency.
- Shared - Tied to a thread block (or workgroup), so it can be accessed by the threads in the same block. This pool of memory is smaller but faster to access than global memory.
- Register Memory - Each thread on a GPU has private access to a number of registers. This is the fastest type of memory used to store local variables and intermediate results.

We will specifically look at the NVIDIA A100 GPU [9], which has 40 GB of memory and 108 SMs each with 64 FP32 CUDA cores and 4 tensor cores resulting in a total of 6912 CUDA cores and 432 tensor cores.

To find out how many fullerenes worth of coordinate data can fit into global memory, we can use the following equation:

$$GlobalMemFullereneCapacity = \left\lfloor \frac{GlobalMemInBits}{FullereneAtomCount \times 3 \times 32} \right\rfloor \tag{4.1}$$

Figure 4.4: The amount of 3-dimensional fullerene coordinate data that fits in global memory. The 3×32 comes from the three 32-bit floating point numbers that represents the 3-dimensional coordinates. This equation does not consider other data such as results or intermediate values.

$$MinimumBatchesToFitIsomerspace = \left\lceil \frac{IsomerspaceSize}{GlobalMemFullereneCapacity} \right\rceil \tag{4.2}$$

Figure 4.5: The minimum number of batches required to process a complete isomerspace in global memory.

Be aware that this equation does not consider any space needed for the results or any intermediate values, but we can use this equation to calculate the minimum amount of batches needed to process the whole isomerspace. If we want to process the whole isomerspace at once, then we might need to add additional GPUs. For example, the C_{200} isomerspace needs a minimum of 13 GPUs with 40 GB of global memory each.

We are also interested in knowing how much non-global memory is available to each xTB computation. Knowing this will tell us how many of the threads, warps, and blocks we can utilize on the GPU. To figure this out, we first need to know how much of the various memory levels are available to a single thread if distributed over all threads.

Each SM has up to 164 KB of shared memory and is divided into four partitions, each containing a 64 KB register file, an one warp scheduler, 16 CUDA Cores dedicated for processing FP32 operations, 16 CUDA Cores for processing INT32 operations, 8 CUDA Cores for processing FP64 operations, and a tensor core, which is specialized for matrix operations. See Figure 4.6 for a figure of the SM architecture.

Figure 4.6: The NVIDIA A100 Architecture for a streaming-multiprocessor. [9, Figure 5]

There are a maximum of 64 warps per SM, so each warp scheduler has 16 warps allocated to them. Each warp has 32 threads, so if we distribute a register file over the 512 threads for an SM, then each of them have 320 bytes (see Equation 4.3). If we need to allocate more memory per thread, then we can utilize fewer of the threads.

To get the amount of shared memory available to each thread when distributed evenly, we can use Equation 4.4. There is 164 KB of shared memory per SM, so each thread has 80 bytes to work with.

$$RegisterMemoryPerThread = \frac{RegisterFile}{WarpsPerPartition \times ThreadsPerWarp} \tag{4.3}$$

$$SharedMemoryPerThread = \frac{SharedMemoryPerSM}{WarpsPerSM \times ThreadsPerWarp} \tag{4.4}$$

Depending on the upperbound on intermediate data required by the GFN2-xTB algorithm, we can tweak the amount of non-global memory available to each fullerene by using fewer threads, warp, and blocks. For example, using only 16 of the 32 threads in each warp will double the available register and shared memory for each thread. There is a point of diminishing returns as a single thread is limited to a maximum of 255 32-bit registers. In the case that the non-global memory alone is not enough to hold the intermediate data of a molecule, then we will need to also utilize global memory. Shared memory has a latency of 20 to 30 cycles while global memory has a latency of 200 to 1000 cycles [10]. Global memory is slow to access, so when used it becomes especially important to place data strategically to allow for coalesced access.

4.2 Data Coalescing

Our goal is an xTB implementation capable of working with data for several molecules in parallel. This opens the opportunity for coalesced data access if we can structure the data in a way that achieves spacial locality. Spacial locality is achieved when related data is close together in memory such that it can be accessed as a contiguous block. This minimizes cache misses by eliminating random access across cache lines. One way to achieve spacial locality is to store data as structures of arrays (SOA) instead of arrays of structures (AOS) (see Figure 4.7). In order to allow coalesced access to data in device memory, we can essentially realign the data to match the access patterns of our kernels.

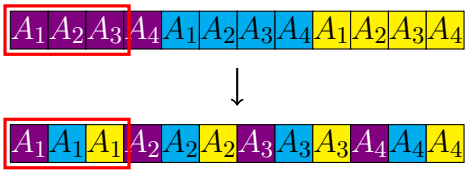


Figure 4.7: Transforms arrays of structures where the memory for each structure is laid out contiguously, into a flattened structure of arrays where data relevant for a computation is close together. This way of grouping related data close together achieves spatial locality and allows for coalesced access of the data.

This project is part of a larger pipeline, so the fullerene data that is fed to the GFN2-xTB algorithm is already in device memory when we get it. The GFN2-xTB implementation therefore never moves any data between the host and device. As such, data coalescing is something we consider when data is moved from higher to lower levels of memory on a device.

In contrast to a task-parallel model where different instructions are executed in parallel, the threads in a warp is optimized around the single instruction multiple threads model (SIMT). This means that all 32 threads in a warp can execute the exact same instructions in lockstep, thus being synchronized on a hardware level. To run in lockstep, data access has to complete for all the threads before

the warp can advance, so to best utilize the SIMT model we need to fetch the data efficiently in a coalesced manner to minimize latency and maximize throughput.

No matter how efficiently we store our data, the throughput will still be bottlenecked by the memory bandwidth if our cores are constantly waiting for new data. To maximize throughput each core has to execute multiple instructions per byte of data. This will ensure that new data can arrive in time for when the cores need it. We will now show how to calculate the required amount of instructions per byte with the A100 GPU as an example. To keep things simple, we will assume that each CUDA core can execute one instruction per clock cycle, and we will only consider the FP32 Cores.

The A100 uses HBM2 DRAM memory, which is organized as five active HBM2 stacks with eight memory dies per stack. Each channel is 128 bits wide, so a whole die is 1024 bits wide, which makes a stack 5120 bits wide. The memory has a double data rate (DDR) of 1215 MHz. DDR gives us two transfers per clock cycle, so it is effectively 2430 MHz, or 2.43 GT/s.

$$StackMemoryInBytes = \frac{ChannelWidth \times ChannelsPerDie \times DiesPerStack}{8} \tag{4.5}$$

$$MemoryBandwidth = StackMemoryInBytes \times DataRateInMHz \times GigaTransfersPerSecond \tag{4.6}$$

Using Equation 4.6 with the specifications of the A100 GPU, we get a total memory bandwidth of 1555 GB/s, which matches the A100 whitepaper from NVIDIA[9].

With this bandwidth we can fetch 1555 bytes per 1GHz clock cycle, so if we use the 6912 FP32 CUDA Cores on the A100, then each core should execute 5 instructions per byte fetched.

$$InstructionsPerByte = \left\lceil \frac{CoreCount}{ByteTransfersPer1GHz} \right\rceil \tag{4.7}$$

We did not consider that a single core can execute more than one instruction per cycle, but if we look at the peak FP32 rate of 19.5 TFLOPS, then we can calculate that we actually need 13 instructions per byte to reach maximum throughput (see Equation 4.8).

$$InstructionsPerByteGivenTFLOPS = \left\lceil \frac{TFLOPSInBytes}{ByteTransfersPer1GHz} \right\rceil \tag{4.8}$$

4.3 Introduction to a Parallel Lockstep Algorithm

We want to ensure that all threads within a warp can work on different fullerenes in parallel. To achieve this, normally we would need to remove branching in the code and place upper bounds on all loops to make sure the exact same instructions are executed by all threads in a warp. This is not necessary in our case since we are working with sorted isomerspaces (by atom number), which means that the only difference between atoms in a warp, is their positions.

This being the only difference between atoms means that we only need to look out for conditionals and loops that depend on the positions. Our computations do not have such conditionals or loops, so the code is already lockstep-parallel. We cannot guarantee that such cases do not appear in the reference Fortran implementation, but this observation makes it clear what to look out for.

Our problem domain is even more specialized since we only consider carbon atoms, and this allows us to simplify a lot of code by unrolling loops and removing conditionals by replacing it with the case that applies for carbon. This section will show how we can simplify some of the functions presented in section 3 by virtue of this fact.

4.3.1 Orbitals as a Constant

This following helper function would never be called in the actual implementation, but instead be computed on the fly. This section will show however, that we never need to compute this at all and can instead store it as a compact constant array.

```python
def get_orbitals(atoms: list[int]) -> list[tuple[int]]:
    orbitals = []
    for atom_idx,atom in enumerate(atoms):
        for subshell in range(number_of_subshells[atom]):
            l = angular_momentum_of_subshell[atom][subshell]
            for orbital in range(l*2+1):
                orbitals.append((atom_idx,atom,subshell,orbital))
    return orbitals
```

Figure 4.8: The original helper function for computing the orbitals for each atom in a molecule.

The orbitals for each atom will always be the same since we are only working with carbon atoms, so we can simply compute the orbitals for the first atom and then use the count of atoms as a bound for how many times we can read the array. This means that we only return an array with the orbitals for the two subshells in the first carbon atom, and we can thus remove the outermost loop.

```python
def get_orbitals() -> list[int]:
    orbitals = []

    # We only work with carbon atoms
    carbon = 6-1 # -1 because we index from 0
    # There are always 2 subshells in a carbon atom
    subshell_count = number_of_subshells[carbon]

    for subshell in range(subshell_count):
        # There are always two subshells and l will always be 0 for the first
        ↪ and 1 for the second
        l = angular_momentum_of_subshell[carbon][subshell]
        for orbital in range(l*2+1):
            # We do not need atom_idx or atom because all are carbon
            orbitals.append((subshell, orbital))

    return orbitals
```

Figure 4.9: Helper function for computing the amount of orbitals for a single carbon atom. This function only considers carbon atoms, so only the first atom is computed and we can therefore remove the outer loop over atoms.

We can simplify this even further by unrolling the loops, since they have a constant number of iterations. Also, since we only access carbon, we can remove values for other atoms from

`number_of_subshells` and `angular_momentum_of_subshell`. These changes gives us the following:

```
def get_orbitals() -> list[int]:
    orbitals = []

    angular_momentum_of_subshell = [0, 1]
    l1 = angular_momentum_of_subshell[0] # Always 0
    l2 = angular_momentum_of_subshell[1] # Always 1

    orbitals_in_subshell1 = l1*2+1 # Always 1
    orbitals_in_subshell2 = l2*2+1 # Always 3

    # If we unroll the loops then we get:

    # Iteration 1
    # [subshell_0, orbital_0]
    orbitals.append_all([0, 0])

    # Iteration 2
    # [subshell_0, orbital_0, orbital_1, orbital_2]
    orbitals.append_all([1, 0, 1, 2])

    return orbitals # Always [0,0,1,0,1,2]
```

Figure 4.10: Helper function for computing the amount of orbitals a single carbon atom. A carbon atom always has 4 orbitals in total, so this function unrolls the loop over orbtials and show that we always get the same constant array as result.

As noted in the code, this always results in the array `[0,0,1,0,1,2]`, so we can just store this as a constant instead of doing the computation. This array is only for a single carbon atom, but instead of replicating this by the number of atoms, we can save the space by indexing into this array for all atoms and bound the number of reads to the count of atoms in the fullerene.

4.3.2 Making a Carbon-specific Initial Density Guess

```
def density_initial_guess(atoms: list[int]) -> list[list[float]]:
    orbitals = get_orbitals(atoms)
    fractional_occupations = square_matrix(len(orbitals))
    for orbital_idx,(_,atom,subshell,_) in enumerate(orbitals):
        l = angular_momentum_of_subshell[atom][subshell]
        orbitals_in_subshell = l*2+1
        electrons_in_subshell = reference_occupations[atom][subshell]
        electrons_per_orbital = electrons_in_subshell/orbitals_in_subshell
        fractional_occupations[orbital_idx][orbital_idx] = electrons_per_orbital
    return fractional_occupations
```

Figure 4.11: The original simplified code for the initial density matrix guess.

We just simplified the orbitals to a constant, but we are only using the length of the array in this function, which is no longer representative of the total number of orbitals. Instead we can use `atom_count*4` as the range, since each atom has four orbitals. Below we show the iteration for the first atom to show that there is always 1 electron per orbital when distributing them evenly, which means we can just create a matrix with 1's along the main diagonal.

```python
def density_initial_guess(atom_count: int) -> list[list[float]]:
    orbital_count = atom_count*4 # There are 4 orbitals for each atom

    fractional_occupations = [0] * orbital_count**2

    angular_momentum_of_subshell = [0, 1]
    l1 = angular_momentum_of_subshell[0] # Always 0
    l2 = angular_momentum_of_subshell[1] # Always 1

    orbitals_in_subshell1 = l1*2+1 # Always 1
    orbitals_in_subshell2 = l2*2+1 # Always 3

    reference_occupations = [1.0, 3.0] # Occupations for carbon
    electrons_in_subshell1 = reference_occupations[0] # Always 1.0
    electrons_in_subshell2 = reference_occupations[1] # Always 3.0

    electrons_per_orbital_in_subshell1 =
    ↪  electrons_in_subshell1/orbitals_in_subshell1 # 1/1 = 1
    electrons_per_orbital_in_subshell2 =
    ↪  electrons_in_subshell2/orbitals_in_subshell2 # 3/3 = 1

    # Subshell 1
    fractional_occupations[0] = electrons_per_orbital_in_subshell1 # Always 1
    # Subshell 2
    for i in range(1, orbitals_in_subshell2 + 1):
        fractional_occupations[i * orbital_count + i] =
        ↪  electrons_per_orbital_in_subshell2 # Always 1

    return fractional_occupations

    # This should be repeated atom_count amount of times so we get the full
    ↪  (atom_count*4)**2 density matrix.
    # Orbital_occupations are always 1 we can just return the (atom_count*4)**2
    ↪  with 1s along the main diagonal.

    # The above computations can therefore be ignored and we can instead just
    ↪  do the following instead:

    fractional_occupations = [0] * row_length**2
    for i in range(row_length):
        fractional_occupations[i * row_length + i] = 1

    return fractional_occupations
```

Figure 4.12: Function for computing the initial density matrix guess. This code only shows an unrolled version of the first iteration to show that we can simply make an array with 1's along the main diagonal, since there is always 1 electron per orbital when distributing them evenly.

4.3.3 Making a Carbon-specific Overlap Computation

```
def overlap_dipol_quadrupol(atoms: list[int], positions: list[list[float]])->
↪  tuple[list[list[float]],list[list[list[float]]],list[list[list[float]]]]:
    orbitals = get_orbitals(atoms)
    S = square_matrix(len(orbitals))
    D = square_matrix_of_vectors(len(orbitals),3)
    Q = square_matrix_of_vectors(len(orbitals),6)
    for idx_A, (atom_idx_A,atom_A,subshell_A,orbital_A) in enumerate(orbitals):
        for idx_B, (atom_idx_B,atom_B,subshell_B,orbital_B) in
        ↪  enumerate(orbitals):
            R_A = positions[atom_idx_A]
            R_B = positions[atom_idx_B]
            l_A = angular_momentum_of_subshell[atom_A][subshell_A]
            l_B = angular_momentum_of_subshell[atom_B][subshell_B]
            s,d,q = compute_STO_integrals(subshell_A, orbital_A, R_A, l_A,
            ↪  subshell_B, orbital_B, R_B, l_B)
            S[idx_A][idx_B] = s
            for dir in [x,y,z]:
                D[idx_A][idx_B][dir] = d[dir]
            for dir in [xx,yy,zz,xy,xz,yz]:
                Q[idx_A][idx_B][dir] = q[dir]
    return S,D,Q
```

Figure 4.13: The original simplified overlap function.

We can again remove the atoms argument and replace it with the number of atoms in a fullerene. This is all we need to compute the subshells and orbitals for each atom, so we do not need to call `get_orbitals`. The angular momentum of subshells for each atom is reduced to only contain the values for carbon, so we only index into it with the subshell. Lastly we have also flattened the first dimension of S, D, and Q. All of these changes can be seen below:

```python
def overlap_dipol_quadrupol(atom_count: int, positions: list[list[float]])->
↪  tuple[list[float],list[list[float]],list[list[float]]]:
    orbital_count = atom_count*4;
    S = [0] * orbital_count**2
    D = [[0,0,0]] * orbital_count**2
    Q = [[0,0,0,0,0,0]] * orbital_count**2

    angular_momentum_of_subshell = [0, 1]

    for idx_A in range(orbital_count):
        for idx_B in range(orbital_count):
            R_A = positions[idx_A]
            R_B = positions[idx_B]

            orbital_A = idx_A \% 4
            orbital_B = idx_B \% 4

            subshell_A = int(orbital_A != 0)
            subshell_B = int(orbital_B != 0)

            l_A = angular_momentum_of_subshell[subshell_A]
            l_B = angular_momentum_of_subshell[subshell_B]

            s,d,q = compute_STO_integrals(subshell_A, orbital_A, R_A, l_A,
            ↪  subshell_B, orbital_B, R_B, l_B)

            S[idx_A * orbital_count + idx_B] = s
            for dir in [x,y,z]:
                D[idx_A * orbital_count + idx_B][dir] = d[dir]
            for dir in [xx,yy,zz,xy,xz,yz]:
                Q[idx_A * orbital_count + idx_B][dir] = q[dir]

    return S,D,Q
```

Figure 4.14: This version of the overlap function flattens the first dimension of S, D, and Q. It also uses the reduced angular momentum array, and it computes the subshells on the fly by using the fact that we only have carbon atoms.

5 Introduction to quantum algorithmic approaches

In this section we will construct quantum algorithms for calculating the isotropic energy terms of GFN2-xTB[2]: E^{Γ} from equation 3.37 and E^{γ} from equation 3.36. We will showcase two very different approaches to doing a calculation as a building block of a larger circuit.

The conceptually simplest approach is to directly translate classical logic circuits to the quantum world using ancillary qubits to ensure reversibility. Here most of the computation happens in the state, and the result is readable in the bits of the measurement output. This approach has seen some development beyond this simple translation resulting in some very qubit efficient primitives for multiplication and addition in particular[11, 12]. This approach will be applied to both the E^{Γ} and E^{γ} terms, and be referred to as Quantum Digital Arithmetic in this report.

Our second approach is Quantum Amplitude Arithmetic[13]. In this approach we try to prepare the desired result not as a easily read measurement result, but as part of the amplitude of a state, which determines the probabilities. This is useful when we want to rank solutions for further processing or sampling of good candidates with high probability. We will use this approach for the E^{Γ} term to prepare a qubit in the state $w|0\rangle + \alpha|0\rangle$ where we can choose α to be proportional to the E^{Γ} of the molecule. Alternatively we can choose α to be proportional to $E^{\Gamma} - E^{\Gamma}_{H}$ where E^{Γ}_{H} is the E^{Γ} energy for some known high energy isomer. This is not something we imagine being a common thing to want, however it is something which we want for the total energy. The issue that is solved by subtracting a known high energy is the following: We want large relative differences between stable isomers and unstable ones. This is to ensure that if we run a superposition of isomers though the circuits required for the complete GFN2-xTB method we are able to efficiently sample the low energy isomers in the isomer space. Say we know all the energies; if all the energies lie between -100 and -101 (units not important), which may make a large difference, the relative difference is not large. If we subtract the energy of a known high energy isomer of say -100.1 we get much larger relative differences where the low energy isomers will have a much larger amplitude on 1, α, than the high energy isomers.

For both of these approaches we will assume that we have access to some intricately prepared states. We will not go into how the are prepared other than the fact that classically we can generate the approximate geometries for entire isomer spaces without any other information. As any classical computation in theory also can be applied to a quantum computer after modifications to make it reversible it is a possibility to prepare these states. A sketch of the preparation could be to generate all the IDs of the isomers, create a superposition and then run the algorithms for determining the geometry on this superposition of IDs.

As a final note leading in to the implementations Quantum Amplitude Amplification is not to be confused with Quantum Amplitude Amplification, both shortened to QAA some times. We will be very explicit in which we are discussing in this thesis.

5.1 Calculating E^{Γ} using Quantum Digital Arithmetic

The GFN2-xTB E^{Γ} term has the following form[3]

$$E^{\Gamma} = \frac{1}{3}\sum_{A}\sum_{\mu\in A}(q_{A,\mu})^3\Gamma_{A,\mu}, \tag{5.1}$$

where $q_{A,\nu} = \sum_B \sum_{\nu\in B} P_{\mu\nu}S_{\mu\nu}$ is the Mulliken partial charge of shell μ associated with atom A. P, S are the density and overlap matrices. $\Gamma_{A,\mu} = \Gamma_A K_\mu$ is just the product of an element specific constant and a shell specific constant, for our purposes the element is always carbon and the shell is either the first or second in GFN2 thus we have 2 numbers $\Gamma_{\text{Carbon},0(1)}$ henceforth referred to as $\mathit{\Gamma}_{0(1)}$.

Let us first rewrite the inner expression a bit given our new definition and knowledge of the atoms we are working with.

$$\sum_{\mu\in A}(q_{A,\mu})^3\Gamma_{A,\mu} = \sum_{\mu\in\{0,1\}}(q_{A,\mu})^3\mathit{\Gamma}_{\mu} \tag{5.2}$$

To implement this equation as a circuit we want to be able to add the terms in the inner expression to an accumulator. Thus we need a unitary which computes this function on a given state $|\mathit{\Gamma}_\mu\rangle_\Gamma |q_{A,\mu}\rangle_Q |acc\rangle_E \to |\mathit{\Gamma}_\mu\rangle_\Gamma |q_{A,\mu}\rangle_Q |acc + (q_{A,\mu})^3\mathit{\Gamma}_\mu\rangle_E$. The subscripts on the kets refer to the quantum register they represent. Consider having access to the following fused multiply add unitary

$|A\rangle|B\rangle|acc\rangle \to |A\rangle|B\rangle|acc + A * B\rangle$, call it $\text{MADD}_{A,B,C}$. Let us to our initial Γ, Q, E registers add two ancillary registers, W_1, W_2. We can now apply the following unitary

$$E_{i\ (\Gamma,Q,W_1,W_2,E)}^{\Gamma} = \text{MADD}_{\Gamma,Q,W_1}^{\dagger}\text{MADD}_{Q,W_1,W_2}^{\dagger}\text{MADD}_{Q,W_2,E}\text{MADD}_{Q,W_1,W_2}\text{MADD}_{\Gamma,Q,W_1} \tag{5.3}$$

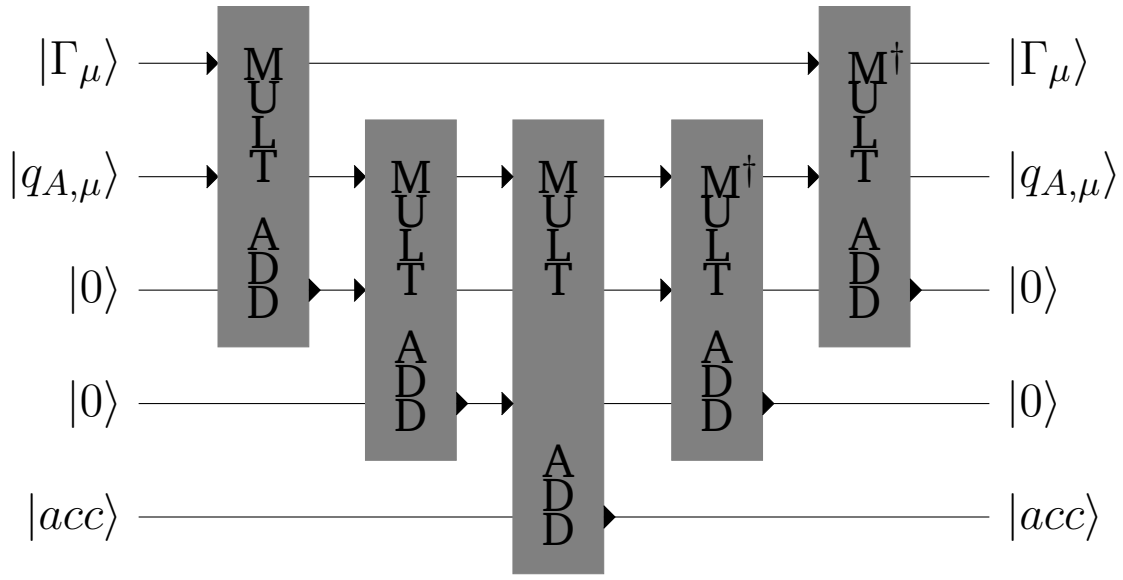


Figure 5.1: Visualization of the circuit presented in equation 5.3. The arrows going into a block symbolise that qubit being used for the calculation happening in a given block. The arrows leaving a block symbolise the block changing something about that qubit. † is the complex conjugate used for doing an operation in reverse.

Let us follow the process:

$$E_{i\ (\Gamma,Q,W_1,W_2,E)}^{\Gamma} |\Gamma_\mu\rangle_\Gamma |q_{A,\mu}\rangle_Q |0\rangle_{W_1} |0\rangle_{W_2} |acc\rangle_E \tag{5.4}$$

$$= \text{MADD}_{\Gamma,Q,W_1}^{\dagger}\text{MADD}_{Q,W_1,W_2}^{\dagger}\text{MADD}_{Q,W_2,E}\text{MADD}_{Q,W_1,W_2} |\Gamma_\mu\rangle_\Gamma |q_{A,\mu}\rangle_Q |\Gamma_\mu q_{A,\mu}\rangle_{W_1} |0\rangle_{W_2} |acc\rangle_E \tag{5.5}$$

$$= \text{MADD}_{\Gamma,Q,W_1}^{\dagger}\text{MADD}_{Q,W_1,W_2}^{\dagger}\text{MADD}_{Q,W_2,E} |\Gamma_\mu\rangle_\Gamma |q_{A,\mu}\rangle_Q |\Gamma_\mu q_{A,\mu}\rangle_{W_1} |\Gamma_\mu (q_{A,\mu})^2\rangle_{W_2} |acc\rangle_E \tag{5.6}$$

$$= \text{MADD}_{\Gamma,Q,W_1}^{\dagger}\text{MADD}_{Q,W_1,W_2}^{\dagger} |\Gamma_\mu\rangle_\Gamma |q_{A,\mu}\rangle_Q |\Gamma_\mu q_{A,\mu}\rangle_{W_1} |\Gamma_\mu (q_{A,\mu})^2\rangle_{W_2} |acc + \Gamma_\mu (q_{A,\mu})^3\rangle_E \tag{5.7}$$

$$= \text{MADD}_{\Gamma,Q,W_1}^{\dagger} |\Gamma_\mu\rangle_\Gamma |q_{A,\mu}\rangle_Q |\Gamma_\mu q_{A,\mu}\rangle_{W_1} |0\rangle_{W_2} |acc + \Gamma_\mu (q_{A,\mu})^3\rangle_E \tag{5.8}$$

$$= |\Gamma_\mu\rangle_\Gamma |q_{A,\mu}\rangle_Q |0\rangle_{W_1} |0\rangle_{W_2} |acc + \Gamma_\mu (q_{A,\mu})^3\rangle_E \tag{5.9}$$

$$\tag{5.10}$$

We see that already in eq. 5.7 we have the result we want in the accumulation register. We continue with the uncomputation of the W_1, W_2 registers purely to be able to reuse them in the remaining

calculations. This saves the qubits required for having 2 ancillary registers for every calculation. The 'fused multiply add' gates here could be implementing using QFT multipliers[12] in which case we wouldn't need any ancillaries beyond the two mentioned. If we decompose our QFT multiplier into its components it is essentially a chain of QFT additions[11, 12] and multiplications by a constant power of two. These additions are built up of two components: (inverse) Fourier transforms and conditional rotations. When we chain them together like this however many of the transforms can be taken out as they are always followed or preceded by their inverse except for in the beginning and end.

Let us say we are given a circuit, SDA, for encoding a molecule from its ID in the following manner, and a circuit $\text{DA} = \prod_A \prod_{\mu\in\{0,1\}} E_i^{\Gamma}{}_{(\Gamma_\mu, Q_{A,\mu}, W_1, W_2, E)}$. Then

$$\begin{aligned}
&\text{DA SDA}\,|id\rangle_{ID}\,|0\rangle = \\
&\quad \text{DA}\,|id\rangle_{ID}\left(\bigotimes_{\mu\in\{0,1\}}|\Gamma_\mu\rangle_{\Gamma_\mu}\bigotimes_{A}|q_{A,\mu}\rangle_{Q_{A,\mu}}\right)|0\rangle_{W_1}\,|0\rangle_{W_2}\,|0\rangle_{E} = \\
&\quad |id\rangle_{ID}\left(\bigotimes_{\mu\in\{0,1\}}|\Gamma_\mu\rangle_{\Gamma_\mu}\bigotimes_{A}|q_{A,\mu}\rangle_{Q_{A,\mu}}\right)|0\rangle_{W_1}\,|0\rangle_{W_2}\,|E_{id}^{\Gamma}\rangle_{E}
\end{aligned} \tag{5.11}$$

will give us the E^Γ energy term in the E register corresponding to the ID in the ID register. Let us look at what happens as we are given a superposition of molecules as input instead. We will use a method introduced in the same paper[13] as Quantum Amplitude Arithmetic for sampling the low energy candidates. This will lead us gently into the later section where we use Quantum Amplitude Arithmetic to construct a circuit for calculating E^Γ.

5.2 Sampling using Quantum Amplitude Arithmetic

Assume that we are given an equal superposition of all the canonical IDs of the fullerenes in an isomer-space. We can apply SDA to set up the encoding and then apply DA. We now have computed the E^Γ energies for every isomer. However we can only sample once. Let us say that we are interested in the isomers with the lowest energies. We then would like the probability of sampling an isomer to be proportional to the gap between the highest energy and the energy of the isomer.

Wang et al. use their introduced addition and multiplication primitives to construct a probabilistic circuit which transforms the state $|x\rangle_D\,|0\rangle_C\,|0\rangle_W \to \left[\frac{1}{2}\frac{1}{2^n}\right]x\,|x\rangle_D\,|0\rangle_C\,|1\rangle_W + \alpha\,|\omega\rangle_{D\otimes C\otimes W}$ where α is some normalization factor, and $|\omega\rangle$ is some state with no overlap with the state containing all 0's in

the control register, C, and 1 in the work register, W. This property makes it very easy to check if the circuit succeeded, we can just use the measurement output that collapses the state to see if we measured the correct pattern of all zeroes and a one. The term in the brackets is the probability of success with n being the number of bits used to describe x. α is then related to the probability of failure as the result should be normalised.

When using this circuit we can treat the E register containing our resulting E^Γ term as the data register, D. We can reuse the W_1, W_2 registers as the control and work registers. Let us take a look at that.

$$\begin{aligned}&\sum_{id\in\text{isomers}} |id\rangle_{ID} \left(\bigotimes_{\mu\in\{0,1\}} |\Gamma_\mu\rangle_{\Gamma_\mu} \bigotimes_A |q_{A,\mu}\rangle_{Q_{A,\mu}} \right) |0\rangle_{W_1} |0\rangle_{W_2} |E^\Gamma\rangle_E = \\ &\sum_{id\in\text{isomers}} |id\rangle_{ID} \left(\bigotimes_{\mu\in\{0,1\}} |\Gamma_\mu\rangle_{\Gamma_\mu} \bigotimes_A |q_{A,\mu}\rangle_{Q_{A,\mu}} \right) \left(\frac{1}{2} \frac{E^\Gamma_{id}}{2^n} |0\rangle_{W_1} |1\rangle_{W_2} |E^\Gamma_{id}\rangle_E + \alpha_{id} |\omega_{id}\rangle \right)\end{aligned} \tag{5.12}$$

If we now sample from this superposition and postselect for $W_1 = 0$ and $W_2 = 1$ we are more likely to sample a low energy fullerene. The likelihood of sampling a given canonical fullerene ID is now proportional with E^Γ_{id} for that fullerene. Let us now look at what it would take to prepare the E^Γ_{id} energies directly in the amplitude using Quantum Amplitude Arithmetic.

5.3 Calculating E^Γ with Quantum Amplitude Arithmetic.

In this section we will encode a molecule as follows

$$\begin{aligned}&\text{SAA} |id\rangle_{ID} |0\rangle_{E,E_w,C_E,q_{o(1,2,3)},\Gamma_w,\Gamma_{1,\ldots,n},q_{1,\ldots,n},C_{1,\ldots,\lceil log(n+1)\rceil}} \\ &= |id\rangle_{ID} |0\rangle_{E,E_w,C_E,q_{o(1,2,3)},\Gamma_w} \bigotimes_{\mu\in\{0,1\}} |\Gamma_\mu\rangle_{\Gamma_{1,\ldots,n}} \bigotimes_A |q_{A,\mu}\rangle_{q_{1,\ldots,n}} |0\rangle_{C_{1,\ldots,\lceil log(n+1)\rceil}}\end{aligned} \tag{5.13}$$

Again we will not go into detail with the SAA circuit.

We now present a circuit for computing $E^\Gamma - E^\Gamma_H$ on a one atom molecule, with 4 bits of precision, with a single shell and therefore Γ value. It is trivial to increase precision as it only involves straight forwards extension of the crimson region to use more qubits for q, Γ and C.

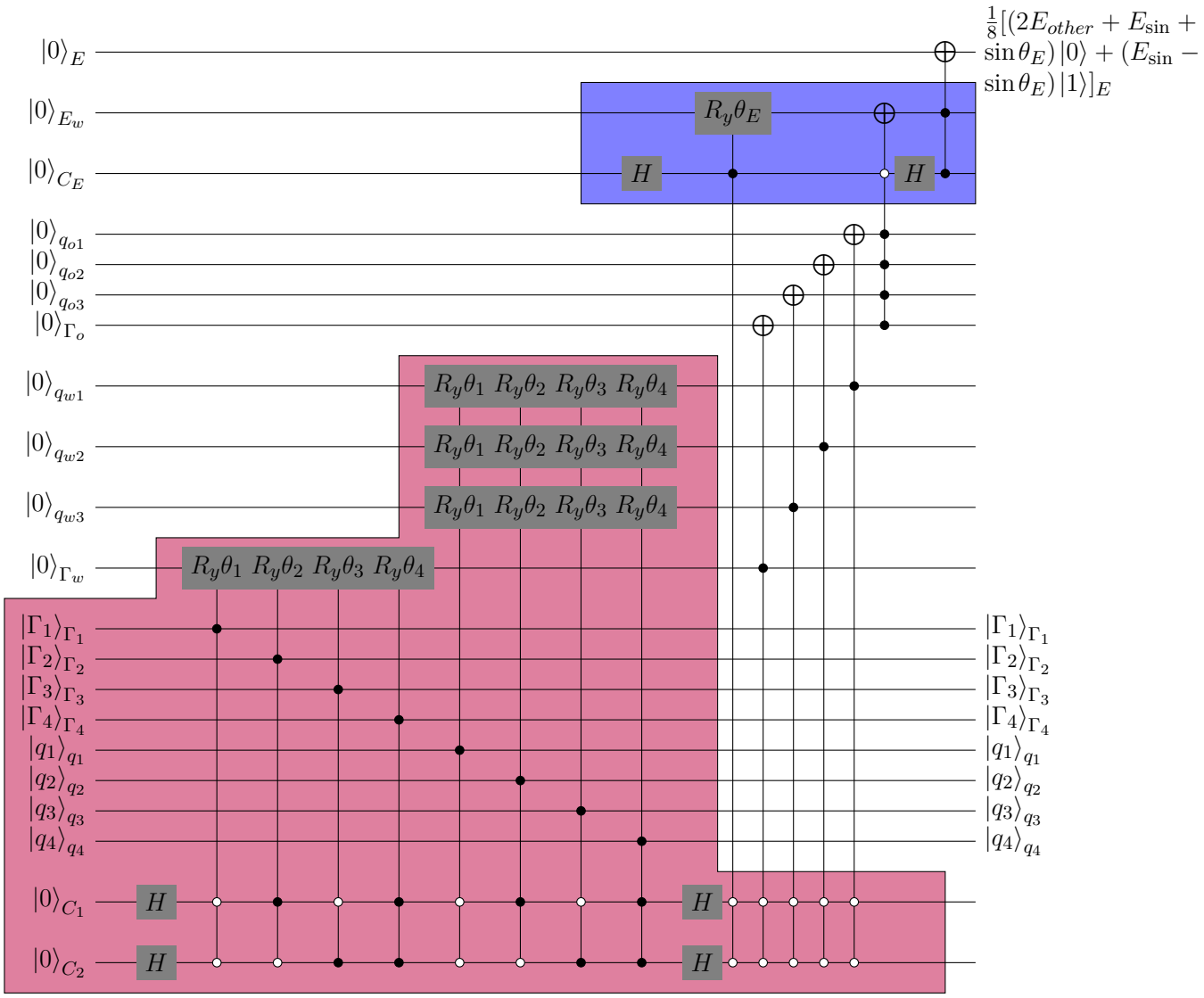


Figure 5.2: Circuit for computing $E^{\Gamma} - E^{\Gamma}_H$ on a one atom molecule where the charge q and Γ are described using 4 bites. The crimson shaded area scales with the number of bits. The blue shaded area is for subtracting E^{Γ}_H and is completely optional.

This circuit is built from the addition and multiplication primitives introduced in the QA-Arithmetic paper[13]. We also do a slight modification to get subtraction.

If using more than one atom and shells the additional q's and Γ's can be added below and the result added to the final E_w register using an extra addition. C_E should be scaled appropriately as the base 2 logarithm of the number of q, Γ pairs plus 1.

Let us go though the mathematics of our 4 bit example to show that A.) it is sound and B.) make the areas that will be extended more obvious.

The controlled gate notation here is the following, t is the target register and $c1, c2, c3, \ldots$ are the control registers. $a, b, c, \ldots$ are all 1 except if there is a bar over the corresponding $c1, c2, c3, \ldots$ in which case it is 0.

$$CU_t^{c1,c2,c3,\cdots} = (U_t - I_t) \otimes |a\rangle \langle a|_{c1} \otimes |b\rangle \langle b|_{c2} \otimes |c\rangle \langle c|_{c3} \otimes \cdots + \sum_{\alpha,\beta,\zeta,\cdots \in \{0,1\}} I_t \otimes |\alpha\rangle \langle \alpha|_{c1} \otimes |\beta\rangle \langle \beta|_{c2} \otimes |\zeta\rangle \langle \zeta|_{c3} \otimes \ldots. \tag{5.14}$$

As we do the calculation we neglect writing out the $q_{1,2,3,4}, \Gamma_{1,2,3,4}$ as they never change throughout, we also neglect the registers outside of the crimson region for now. We begin by applying the Hadamard gates.

$$\begin{aligned} H_{C1} H_{C2} |0000\rangle_{q_{w(1,2,3)},\Gamma_w} |00\rangle_{C_{(1,2)}} = \\ \frac{1}{2}(|0000\rangle_{q_{w(1,2,3)},\Gamma_w} |00\rangle_{C_{(1,2)}} + \\ |0000\rangle_{q_{w(1,2,3)},\Gamma_w} |01\rangle_{C_{(1,2)}} + \\ |0000\rangle_{q_{w(1,2,3)},\Gamma_w} |10\rangle_{C_{(1,2)}} + \\ |0000\rangle_{q_{w(1,2,3)},\Gamma_w} |11\rangle_{C_{(1,2)}}) \end{aligned} \tag{5.15}$$

When we apply the conditional rotation gates to a register such as Γ_w we do the following

$$\begin{aligned} CRy_{\Gamma_w}^{\Gamma_4,C_1,C_2}(2\theta_4) CRy_{\Gamma_w}^{\Gamma_3,\bar{C_1},C_2}(2\theta_3) CRy_{\Gamma_w}^{\Gamma_2,C_1,\bar{C_2}}(2\theta_2) CRy_{\Gamma_w}^{\Gamma_1,\bar{C_1},\bar{C_2}}(2\theta_1) \\ \frac{1}{2} \sum_{x_1,x_2 \in \{0,1\}} |0000\rangle_{q_{w(1,2,3)},\Gamma_w} |x_1 x_2\rangle_{C_{(1,2)}} = \\ \frac{1}{2}(CRy_{\Gamma_w}^{\Gamma_1}(2\theta_1) |0000\rangle_{q_{w(1,2,3)},\Gamma_w} |00\rangle_{C_{(1,2)}} + \\ CRy_{\Gamma_w}^{\Gamma_2}(2\theta_2) |0000\rangle_{q_{w(1,2,3)},\Gamma_w} |01\rangle_{C_{(1,2)}} + \\ CRy_{\Gamma_w}^{\Gamma_3}(2\theta_3) |0000\rangle_{q_{w(1,2,3)},\Gamma_w} |10\rangle_{C_{(1,2)}} + \\ CRy_{\Gamma_w}^{\Gamma_4}(2\theta_4) |0000\rangle_{q_{w(1,2,3)},\Gamma_w} |11\rangle_{C_{(1,2)}}) = \\ \frac{1}{2}(|000\rangle [\Gamma 1(\cos\theta_1 |0\rangle + \sin\theta_1 |1\rangle) + (1 - \Gamma_1) |0\rangle]_{q_{w(1,2,3)},\Gamma_w} |00\rangle + \\ |000\rangle [\Gamma 2(\cos\theta_2 |0\rangle + \sin\theta_2 |1\rangle) + (1 - \Gamma_2) |0\rangle]_{q_{w(1,2,3)},\Gamma_w} |01\rangle + \\ |000\rangle [\Gamma 3(\cos\theta_3 |0\rangle + \sin\theta_3 |1\rangle) + (1 - \Gamma_3) |0\rangle]_{q_{w(1,2,3)},\Gamma_w} |10\rangle + \\ |000\rangle [\Gamma 4(\cos\theta_4 |0\rangle + \sin\theta_4 |1\rangle) + (1 - \Gamma_4) |0\rangle]_{q_{w(1,2,3)},\Gamma_w} |11\rangle) \end{aligned} \tag{5.16}$$

To make our computations fit more easily on the page let us adopt the notation $|\Psi_i^t\rangle = t(\cos\theta_i|0\rangle + \sin\theta_i|1\rangle) + (1-t)|0\rangle$ before redoing the application using our new notation. We also apply the rotation gates for the q_w registers:

$$
\begin{aligned}
&CRy_{\Gamma_w}^{\Gamma_4,C_1,C_2}(2\theta_4)CRy_{\Gamma_w}^{\Gamma_3,\bar{C_1},C_2}(2\theta_3)CRy_{\Gamma_w}^{\Gamma_2,C_1,\bar{C_2}}(2\theta_2)CRy_{\Gamma_w}^{\Gamma_1,\bar{C_1},\bar{C_2}}(2\theta_1)\\
&CRy_{q_{w1}}^{q_4,C_1,C_2}(2\theta_4)CRy_{q_{w1}}^{q_3,\bar{C_1},C_2}(2\theta_3)CRy_{q_{w1}}^{q_2,C_1,\bar{C_2}}(2\theta_2)CRy_{q_{w1}}^{q_1,\bar{C_1},\bar{C_2}}(2\theta_1)\\
&CRy_{q_{w2}}^{q_4,C_1,C_2}(2\theta_4)CRy_{q_{w2}}^{q_3,\bar{C_1},C_2}(2\theta_3)CRy_{q_{w2}}^{q_2,C_1,\bar{C_2}}(2\theta_2)CRy_{q_{w2}}^{q_1,\bar{C_1},\bar{C_2}}(2\theta_1)\\
&CRy_{q_{w3}}^{q_4,C_1,C_2}(2\theta_4)CRy_{q_{w3}}^{q_3,\bar{C_1},C_2}(2\theta_3)CRy_{q_{w3}}^{q_2,C_1,\bar{C_2}}(2\theta_2)CRy_{q_{w3}}^{q_1,\bar{C_1},\bar{C_2}}(2\theta_1)\\
&\qquad\qquad\frac{1}{2}\sum_{x_1,x_2\in\{0,1\}}|0000\rangle_{q_{w(1,2,3)},\Gamma_w}|x_1x_2\rangle =\\
&\frac{1}{2}(CRy_{q_{w1}}^{q_1}(2\theta_1)CRy_{q_{w2}}^{q_1}(2\theta_1)CRy_{q_{w3}}^{q_1}(2\theta_1)CRy_{\Gamma_w}^{\Gamma_1}(2\theta_1)|0000\rangle_{q_{w(1,2,3)},\Gamma_w}|00\rangle +\\
&\quad CRy_{q_{w1}}^{q_2}(2\theta_2)CRy_{q_{w2}}^{q_2}(2\theta_2)CRy_{q_{w3}}^{q_2}(2\theta_2)CRy_{\Gamma_w}^{\Gamma_2}(2\theta_2)|0000\rangle_{q_{w(1,2,3)},\Gamma_w}|01\rangle +\\
&\quad CRy_{q_{w1}}^{q_3}(2\theta_3)CRy_{q_{w2}}^{q_3}(2\theta_3)CRy_{q_{w3}}^{q_3}(2\theta_3)CRy_{\Gamma_w}^{\Gamma_3}(2\theta_3)|0000\rangle_{q_{w(1,2,3)},\Gamma_w}|10\rangle +\\
&\quad CRy_{q_{w1}}^{q_4}(2\theta_4)CRy_{q_{w2}}^{q_4}(2\theta_4)CRy_{q_{w3}}^{q_4}(2\theta_4)CRy_{\Gamma_w}^{\Gamma_4}(2\theta_4)|0000\rangle_{q_{w(1,2,3)},\Gamma_w}|11\rangle) =\\
&\qquad\frac{1}{2}(|\Psi_1^{q_1}\Psi_1^{q_1}\Psi_1^{q_1}\Psi_1^{\Gamma_1}\rangle_{q_{w(1,2,3)},\Gamma_w}|00\rangle +\\
&\qquad\quad|\Psi_2^{q_2}\Psi_2^{q_2}\Psi_2^{q_2}\Psi_2^{\Gamma_2}\rangle_{q_{w(1,2,3)},\Gamma_w}|01\rangle +\\
&\qquad\quad|\Psi_3^{q_3}\Psi_3^{q_3}\Psi_3^{q_3}\Psi_3^{\Gamma_3}\rangle_{q_{w(1,2,3)},\Gamma_w}|10\rangle +\\
&\qquad\quad|\Psi_4^{q_4}\Psi_4^{q_4}\Psi_4^{q_4}\Psi_4^{\Gamma_4}\rangle_{q_{w(1,2,3)},\Gamma_w}|11\rangle)
\end{aligned}
\tag{5.17}
$$

Let us now apply the second set of Hadamard gates:

$$
\begin{aligned}
H_{C_1}H_{C_2}\frac{1}{2}(&|\Psi_1^{q_1}\Psi_1^{q_1}\Psi_1^{q_1}\Psi_1^{\Gamma_1}\rangle|00\rangle+\\
&|\Psi_2^{q_2}\Psi_2^{q_2}\Psi_2^{q_2}\Psi_2^{\Gamma_2}\rangle|01\rangle+\\
&|\Psi_3^{q_3}\Psi_3^{q_3}\Psi_3^{q_3}\Psi_3^{\Gamma_3}\rangle|10\rangle+\\
&|\Psi_4^{q_4}\Psi_4^{q_4}\Psi_4^{q_4}\Psi_4^{\Gamma_4}\rangle|11\rangle)=\\
\frac{1}{4}(&|\Psi_1^{q_1}\Psi_1^{q_1}\Psi_1^{q_1}\Psi_1^{\Gamma_1}\rangle[|00\rangle+|01\rangle+|10\rangle+|11\rangle]+\\
&|\Psi_2^{q_2}\Psi_2^{q_2}\Psi_2^{q_2}\Psi_2^{\Gamma_2}\rangle[|00\rangle-|01\rangle+|10\rangle-|11\rangle]+\\
&|\Psi_3^{q_3}\Psi_3^{q_3}\Psi_3^{q_3}\Psi_3^{\Gamma_3}\rangle[|00\rangle+|01\rangle-|10\rangle-|11\rangle]+\\
&|\Psi_4^{q_4}\Psi_4^{q_4}\Psi_4^{q_4}\Psi_4^{\Gamma_4}\rangle[|00\rangle-|01\rangle-|10\rangle+|11\rangle])=\\
\frac{1}{4}&\left[|\Psi_1^{q_1}\Psi_1^{q_1}\Psi_1^{q_1}\Psi_1^{\Gamma_1}\rangle+|\Psi_2^{q_2}\Psi_2^{q_2}\Psi_2^{q_2}\Psi_2^{\Gamma_2}\rangle+|\Psi_3^{q_3}\Psi_3^{q_3}\Psi_3^{q_3}\Psi_3^{\Gamma_3}\rangle+|\Psi_4^{q_4}\Psi_4^{q_4}\Psi_4^{q_4}\Psi_4^{\Gamma_4}\rangle\right]|00\rangle+\\
\frac{1}{4}\Bigg(&\left[|\Psi_1^{q_1}\Psi_1^{q_1}\Psi_1^{q_1}\Psi_1^{\Gamma_1}\rangle-|\Psi_2^{q_2}\Psi_2^{q_2}\Psi_2^{q_2}\Psi_2^{\Gamma_2}\rangle+|\Psi_3^{q_3}\Psi_3^{q_3}\Psi_3^{q_3}\Psi_3^{\Gamma_3}\rangle-|\Psi_4^{q_4}\Psi_4^{q_4}\Psi_4^{q_4}\Psi_4^{\Gamma_4}\rangle\right]|01\rangle+\\
&\left[|\Psi_1^{q_1}\Psi_1^{q_1}\Psi_1^{q_1}\Psi_1^{\Gamma_1}\rangle+|\Psi_2^{q_2}\Psi_2^{q_2}\Psi_2^{q_2}\Psi_2^{\Gamma_2}\rangle-|\Psi_3^{q_3}\Psi_3^{q_3}\Psi_3^{q_3}\Psi_3^{\Gamma_3}\rangle-|\Psi_4^{q_4}\Psi_4^{q_4}\Psi_4^{q_4}\Psi_4^{\Gamma_4}\rangle\right]|10\rangle+\\
&\left[|\Psi_1^{q_1}\Psi_1^{q_1}\Psi_1^{q_1}\Psi_1^{\Gamma_1}\rangle-|\Psi_2^{q_2}\Psi_2^{q_2}\Psi_2^{q_2}\Psi_2^{\Gamma_2}\rangle-|\Psi_3^{q_3}\Psi_3^{q_3}\Psi_3^{q_3}\Psi_3^{\Gamma_3}\rangle+|\Psi_4^{q_4}\Psi_4^{q_4}\Psi_4^{q_4}\Psi_4^{\Gamma_4}\rangle\right]|11\rangle\Bigg)\\
=\frac{1}{4}&\left[|\Psi_1^{q_1}\Psi_1^{q_1}\Psi_1^{q_1}\Psi_1^{\Gamma_1}\rangle+|\Psi_2^{q_2}\Psi_2^{q_2}\Psi_2^{q_2}\Psi_2^{\Gamma_2}\rangle+|\Psi_3^{q_3}\Psi_3^{q_3}\Psi_3^{q_3}\Psi_3^{\Gamma_3}\rangle+|\Psi_4^{q_4}\Psi_4^{q_4}\Psi_4^{q_4}\Psi_4^{\Gamma_4}\rangle\right]|00\rangle+|M\rangle\\
=&|N\rangle+|M\rangle
\end{aligned}
\tag{5.18}
$$

Before the next step let us define:

$$q_{\sin}=q_1\sin\theta_1+q_2\sin\theta_2+q_3\sin\theta_3+q_4\sin\theta_4 \tag{5.19}$$

$$q_{other}=q_1\cos\theta_1+q_2\cos\theta_2+q_3\cos\theta_3+q_4\cos\theta_4+4-q_1-q_2-q_3-q_4 \tag{5.20}$$

$$\Gamma_{\sin}=\Gamma_1\sin\theta_1+\Gamma_2\sin\theta_2+\Gamma_3\sin\theta_3+\Gamma_4\sin\theta_4 \tag{5.21}$$

$$\Gamma_{other}=\Gamma_1\cos\theta_1+\Gamma_2\cos\theta_2+\Gamma_3\cos\theta_3+\Gamma_4\cos\theta_4+4-\Gamma_1-\Gamma_2-\Gamma_3-\Gamma_4 \tag{5.22}$$

$$E_{\sin}=\Gamma_{\sin}(q_{\sin})^3 \tag{5.23}$$

$$
\begin{aligned}
E_{other}=&\Gamma_{other}(q_{other}^3+3q_{other}^2q_{\sin}+3q_{other}q_{\sin}^2+q_{\sin}^3)\\
&+\Gamma_{\sin}(q_{other}^3+3q_{other}^2q_{\sin}+3q_{other}q_{\sin}^2)
\end{aligned}
\tag{5.24}
$$

$$|\sigma_t\rangle=t_{other}|0\rangle+t_{\sin}|1\rangle \tag{5.25}$$

$$\tag{5.26}$$

Let us now add in the $_o$ registers and apply the first 4 conditional not gates:

$$
\begin{aligned}
CX_{qo1}^{qw1,\bar{C}_1,\bar{C}_2} CX_{qo2}^{qw2,\bar{C}_1,\bar{C}_2} CX_{qo3}^{qw3,\bar{C}_1,\bar{C}_2} CX_{\Gamma_o}^{\Gamma_w,\bar{C}_1,\bar{C}_2} |0000\rangle_{E,q_{o(1,2,3)},\Gamma_o} (|N\rangle + |M\rangle) = \\
\left(CX_{qo1}^{qw1} CX_{qo2}^{qw2} CX_{qo3}^{qw3} CX_{\Gamma_o}^{\Gamma_w} |0000\rangle |N\rangle \right) + |0000\rangle |M\rangle = \\
|\sigma_q\sigma_q\sigma_q\sigma_\Gamma\rangle_{q_{o(1,2,3)},\Gamma_o} |N\rangle + |0000\rangle |M\rangle
\end{aligned}
\tag{5.27}
$$

Now let us disregard everything in the crimson region except the C_1, C_2 control registers and do the final gates involving the $_o$ registers:

$$
\begin{aligned}
& H_{C_E} CX_{E_w}^{\bar{C}_E,q_{o1},q_{o2},q_{o3},\Gamma_o} CRy_{E_w}^{C_E,\bar{C}_1,\bar{C}_2}(\theta_E) H_{C_E} |000\rangle_{E,E_w,C_E} \frac{1}{4} \Bigg(|\sigma_q\sigma_q\sigma_q\sigma_\Gamma\rangle_{q_{o(1,2,3)},\Gamma_o} |00\rangle_{C_1,C_2} \\
& \quad + |0000\rangle_{q_{o(1,2,3)},\Gamma_o} [|01\rangle + |10\rangle + |11\rangle]_{C_1,C_2} \Bigg) = \\
& H_{C_E} \frac{1}{4} \Bigg(|0\rangle \frac{1}{\sqrt{2}} \left[CX_{E_w}^{q_{o1},q_{o2},q_{o3},\Gamma_o} |0\rangle |0\rangle + Ry_{E_w}(\theta_E) |0\rangle |1\rangle \right] |\sigma_q\sigma_q\sigma_q\sigma_\Gamma\rangle_{q_{o(1,2,3)},\Gamma_o} |00\rangle_{C_1,C_2} \\
& \quad + |00+0000\rangle_{E,E_w,C_E,q_{o(1,2,3)},\Gamma_o} [|01\rangle + |10\rangle + |11\rangle]_{C_1,C_2} \Bigg) = \\
& H_{C_E} \frac{1}{4} \Bigg(|0\rangle \frac{1}{\sqrt{2}} \left[|\sigma_E\rangle_{E_w} |0\rangle_{C_E} + (\cos\theta_E |0\rangle + \sin\theta_E |1\rangle)_{E_w} |1\rangle_{C_E} \right] |\sigma_q\sigma_q\sigma_q\sigma_\Gamma\rangle_{q_{o(1,2,3)},\Gamma_o} |00\rangle_{C_1,C_2} \\
& \quad + |00+0000\rangle_{E,E_w,C_E,q_{o(1,2,3)},\Gamma_o} [|01\rangle + |10\rangle + |11\rangle]_{C_1,C_2} \Bigg) = \\
& \frac{1}{4} \Bigg(|0\rangle \frac{1}{\sqrt{2}} \left[|\sigma_E\rangle_{E_w} |+\rangle_{C_E} + (\cos\theta_E |0\rangle + \sin\theta_E |1\rangle)_{E_w} |-\rangle_{C_E} \right] |\sigma_q\sigma_q\sigma_q\sigma_\Gamma\rangle_{q_{o(1,2,3)},\Gamma_o} |00\rangle_{C_1,C_2} \\
& \quad + |000000\rangle_{E,E_w,C_E,q_{o(1,2,3)},\Gamma_o} [|01\rangle + |10\rangle + |11\rangle]_{C_1,C_2} \Bigg)
\end{aligned}
\tag{5.28}
$$

Now we can neglect the $q_{o(1,2,3),\Gamma_o,C_1,C_2}$ registers too and do some preparatory manipulations before applying the final conditional not gate.

$$
\begin{aligned}
&\frac{1}{4}\Bigg(|0\rangle\frac{1}{\sqrt{2}}\Big[|\sigma_E\rangle_{E_w}|+\rangle_{C_E}+(\cos\theta_E|0\rangle+\sin\theta_E|1\rangle)_{E_w}|-\rangle_{C_E}\Big]+3|000\rangle\Bigg)=\\
&\quad\frac{1}{4}\Bigg(|0\rangle\frac{1}{2}\Big[|\sigma_E\rangle_{E_w}|0\rangle_{C_E}+|\sigma_E\rangle_{E_w}|1\rangle_{C_E}+(\cos\theta_E|0\rangle+\sin\theta_E|1\rangle)_{E_w}|0\rangle_{C_E}\\
&\quad-(\cos\theta_E|0\rangle+\sin\theta_E|1\rangle)_{E_w}|1\rangle_{C_E}\Big]+3|000\rangle\Bigg)=\\
&\quad\frac{1}{4}\Bigg(|0\rangle\frac{1}{2}\Big[(|\sigma_E\rangle+\cos\theta_E|0\rangle+\sin\theta_E|1\rangle)_{E_w}|0\rangle_{C_E}\\
&\qquad+(|\sigma_E\rangle-\cos\theta_E|0\rangle-\sin\theta_E|1\rangle)_{E_w}|1\rangle_{C_E}\Big]+3|000\rangle\Bigg)=\\
&\qquad\frac{1}{8}\Bigg(|0\rangle[|\sigma_E\rangle+\cos\theta_E|0\rangle+\sin\theta_E|1\rangle]_{E_w}|0\rangle_{C_E}\\
&\qquad+|0\rangle[|\sigma_E\rangle-\cos\theta_E|0\rangle-\sin\theta_E|1\rangle]_{E_w}|1\rangle_{C_E}+6|000\rangle\Bigg)=\\
&\qquad\frac{1}{8}\Bigg(|0\rangle[E_{other}|0\rangle+E_{\sin}|1\rangle+\cos\theta_E|0\rangle+\sin\theta_E|1\rangle]_{E_w}|0\rangle_{C_E}\\
&\qquad+|0\rangle[E_{other}|0\rangle+E_{\sin}|1\rangle-\cos\theta_E|0\rangle-\sin\theta_E|1\rangle]_{E_w}|1\rangle_{C_E}+6|000\rangle\Bigg)=\\
&\qquad\frac{1}{8}\Bigg(|0\rangle[(E_{other}+\cos\theta_E)|0\rangle+(E_{\sin}+\sin\theta_E)|1\rangle]_{E_w}|0\rangle_{C_E}\\
&\qquad+|0\rangle[(E_{other}-\cos\theta_E)|0\rangle+(E_{\sin}-\sin\theta_E)|1\rangle]_{E_w}|1\rangle_{C_E}+6|000\rangle\Bigg)=\\
&\qquad\quad\frac{1}{8}\Bigg((E_{other}+\cos\theta_E)|000\rangle+(E_{\sin}+\sin\theta_E)|010\rangle\\
&\qquad\quad+(E_{other}-\cos\theta_E)|001\rangle+(E_{\sin}-\sin\theta_E)|011\rangle+6|000\rangle\Bigg)
\end{aligned}
\tag{5.29}
$$

In the last step we just did, see that we can 'select' either addition or subtraction of $\sin\theta_E$ just by controlling our next gate on 010 or 011. We now apply the final conditional not gate:

$$
\begin{aligned}
CX_E^{E_w,C_E}\frac{1}{8}\Big(&(E_{other}+\cos\theta_E)\,|000\rangle+(E_{\sin}+\sin\theta_E)\,|010\rangle\\
&+(E_{other}-\cos\theta_E)\,|001\rangle+(E_{\sin}-\sin\theta_E)\,|011\rangle+6\,|000\rangle\Big)=\\
\frac{1}{8}\Big(&(E_{other}+\cos\theta_E)\,|000\rangle+(E_{\sin}+\sin\theta_E)\,|010\rangle\\
&+(E_{other}-\cos\theta_E)\,|001\rangle+X_E(E_{\sin}-\sin\theta_E)\,|011\rangle+6\,|000\rangle\Big)=\\
\frac{1}{8}\Big(&(E_{other}+\cos\theta_E)\,|000\rangle+(E_{\sin}+\sin\theta_E)\,|010\rangle\\
&+(E_{other}-\cos\theta_E)\,|001\rangle+(E_{\sin}-\sin\theta_E)\,|111\rangle+6\,|000\rangle\Big)
\end{aligned}
\tag{5.30}
$$

After applying those gates we see that the amplitude on $|1\rangle_E$ across the whole state is

$$\tfrac{1}{8}(E_{\sin}-\sin\theta_E)=(\Gamma_1\sin\theta_1+\Gamma_2\sin\theta_2+\Gamma_3\sin\theta_3+\Gamma_4\sin\theta_4)(q_1\sin\theta_1+q_2\sin\theta_2+q_3\sin\theta_3+q_4\sin\theta_4)^3-\sin\theta_E.$$

Let us say we know the E^Γ energy of some high energy molecule in the isomer space

$$E_H^\Gamma=(0b0.\Gamma_H)(0b0.q_H)^3$$

If we specify

$$\theta_i=arcsin\frac{1}{2^i},\quad \theta_E=arcsin[(0b0.\Gamma_H)(0b0.q_H)^3]$$

we get that

$$E_{\sin}=(\frac{\Gamma_1}{2}+\frac{\Gamma_2}{2^2}+\frac{\Gamma_3}{2^3}+\frac{\Gamma_4}{2^8})(\frac{q_1}{2}+\frac{q_2}{2^2}+\frac{q_3}{2^3}+\frac{q_4}{2^8})^3=(0b0.\Gamma_1\Gamma_2\Gamma_3\Gamma_4)(0b0.q_1q_2q_3q_4)^3$$

and that

$$\frac{1}{8}(E_{\sin}-\sin\theta_E)=\frac{1}{8}[(0b0.\Gamma_1\Gamma_2\Gamma_3\Gamma_4)(0b0.q_1q_2q_3q_4)^3-(0b0.\Gamma_H)(0b0.q_H)^3]$$

. This is proportional to $E^\Gamma-E_H^\Gamma$ and thus the circuit is sound.

We will now take a look at the E^γ term.

5.4 Calculating E^{γ} using Quantum Digital Arithmetic

The E_{γ} term is formulated as follows:

$$\eta_{AB,ll'} = \frac{1}{2}\left[\eta_A(1+K_A^l)+\eta_B(1+K_B^{l'})\right] \tag{5.31}$$

$$R_{AB}^2 = (A_x-B_x)^2+(A_y-B_y)^2+(A_z-B_z)^2 \tag{5.32}$$

$$\gamma_{AB,ll'} = \frac{1}{\sqrt{R_{AB}^2+\eta_{AB,ll'}^{-2}}} \tag{5.33}$$

$$E_{\gamma} = \frac{1}{2}\sum_{A,B}\sum_{l\in A}\sum_{l'\in B} q_l q_{l'}\gamma_{AB,ll'} \tag{5.34}$$

$$\tag{5.35}$$

For fullerenes we can view $\eta_{AB,ll'}^{-2}$ as only dependant on the angular momenta l and l', so there are 4 configurations as there are only 2 shells for carbon in GFN2-xTB and the rest of the terms are constants. Additionally $l=0,l'=1$ and $l=1,l'=0$ are equivalent. Thus we don't actually have to compute η during the quantum circuit and can just bake in those 3 configurations as constants. A small circuit for calculating R_{AB}^2 could be made up of 3 repetitions of the DIFF^2 circuit described bellow which for 2 numbers a,b and an accumulator acc computes $acc+(a-b)^2$:

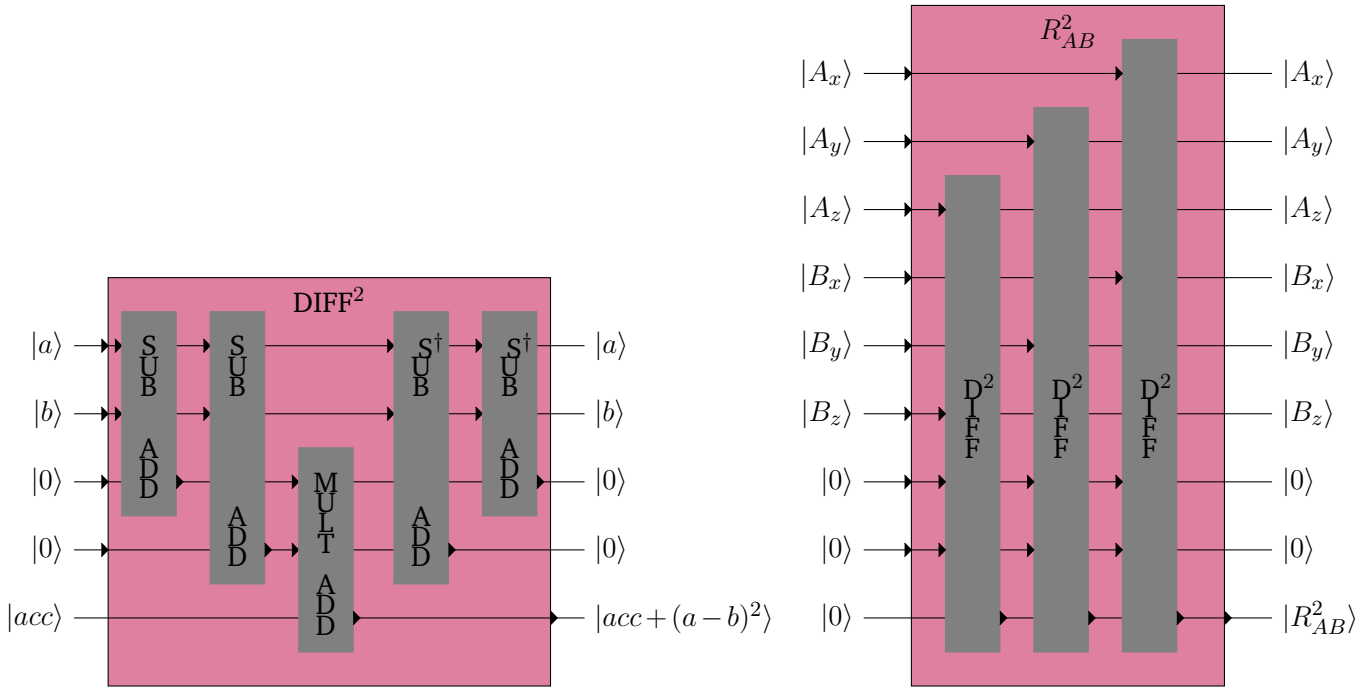


Figure 5.3: Circuits for calculating $|a\rangle|b\rangle|acc\rangle \to |a\rangle|b\rangle|acc+(a-b)^2\rangle$ and $|A\rangle|B\rangle|0\rangle \to |A\rangle|B\rangle|R_{AB}^2\rangle$

Using this circuit along with a constant addition of $\eta_{AB,ll'}$ and the use of an inverse square root circuit[14] we get a fixed point approximation of $\gamma_{AB,ll'}$. We can use this with a QFT fused multiply add operation to get the inner sum of the E^{γ} term. There could potentially be a lot of pairs, however we can skip almost half with the following rewrite:

$$\begin{aligned} E^{\gamma} &= \frac{1}{2}\sum_{A,B}\sum_{l\in A}\sum_{l'\in B} q_l q_{l'}\gamma_{AB,ll'} \\ &= \sum_{A}\sum_{l\in A}\left[\sum_{l'\in A} q_l q_{l'}\gamma_{AA,ll'} + \sum_{B>A}\sum_{l'\in B} q_l q_{l'}\gamma_{AB,ll'}\right] \end{aligned} \tag{5.36}$$

This rearrangement is also something they do in the xtb software. Let us take take a brief look at the complexity of these quantum algorithms.

5.5 Complexity

All these algorithms use $i = \lceil log_2(f) \rceil$ qubits to keep the fullerene IDs, where f is the number of fullerenes in the isomer space C_c that we are considering, thus $O(f^9)$.

The first circuit uses 5 QFT fused multiplication addition circuits. Each with $O(b^3)$[12] gates and no additional qubits, where b is the number of bits used to encode the numbers. Thus if we have encoded an isomer space we need the two Γ_0, Γ_1 as well as the partial Mulliken charges for each atom $q_{A,0}, q_{A,1}$ each using b bits we will need to perform $10c$ multiplications, resulting in $10c \cdot O(b^3) = O(cb^3)$ gates, on $i + 2cb + 2b + 3b = O(cb + i)$ qubits. We can do trade-offs in circuit depth and amount of ancillary qubits by having multiple running sums e.g. in a binary tree pattern instead of immediately summing them all in the same register, resulting in a depth of $O(log_2(c)b^3)$.

If we add in the sampling step we then have to run the state preparation circuit which adds $O(\log b)$ qubits and $O(b \log b)$ gates[13], however that is not enough to change the asymptotic runtime further given $b < c$ in most cases we care about.

If we want to say something about the number of samplings needed to have found a good candidate we have to assume a distribution for the energies in the isomer space. As a best guess we will assume they are evenly distributed between the highest and lowest energies E_H^{Γ} and E_L^{Γ}. Then for a successful run of the algorithm the probability of sampling a given id is the amplitude squared. The

amplitude is proportional to the energy E_{id}^{Γ}. The probability of the sample having energy within δ of E_L^{Γ} is then

$$\frac{\int_{E_L^{\Gamma}}^{E_L^{\Gamma}-\delta} |e|^2 \, de}{\int_{E_L^{\Gamma}}^{E_H^{\Gamma}} |e|^2 \, de} = \frac{|E_L^{\Gamma}|^3 - |E_L^{\Gamma} - \delta|^3}{|E_L^{\Gamma}|^3 - |E_H^{\Gamma}|^3} \tag{5.37}$$

This is only indirectly related to the size of the isomer space, as we might expect the gap between the highest and lowest energy in the isomer space scale with its size.

For the second circuit we use $log_2(b) + log_2(c) + 1$ control qubits and $2cb + 2b + 8cb + i$ for the rest giving a total on the order of $O(cb+i)$. In terms of gates we use $c(2b+1)$ multiple controlled rotation gates, at most $log_2(b) + log_2(c) + 1$ Hadamard gates, and $5c+1$ multiple controlled not gates. This gives a circuit depth on the order of $O(cb)$.

The algorithm for calculating E^{γ} uses $i + 3cb + 2cb + 9b$ to hold the ids, positions, partial Mulliken charges and ancillaries. The 9 ancillaries are for the inverse square root approximation assuming 5 newton iterations. They are cleaned up afterwards, so the few we need for the rest of the algorithm can be taken from here. Thus we use $O(cb+i)$ qubits.

In terms of gates we use 8 subtraction circuits and 3 multiplication circuits for each pair of atoms. For each pair of charges we do an addition by a constant the inverse square root and use 3 fused multiply add operations to add it to the running sum. For the inverse square root we will assume 5 iterations and the number of bits before the decimal point is equal to b. These assumptions make the complexity for the inverse square root $O(b^2)$ which is slightly worse than the actual complexity for a more reasonable number of bits before the decimal point, however it is still not the dominating factor. The subtraction circuits have similar complexity to the addition circuits already mentioned. This gives a total gate complexity of $c^2(8 \cdot O(b^2) + 3 \cdot O(b^3)) + (2c)^2(O(b^2) + O(b^2) + 3 \cdot O(b^3)) = O(c^2 b^3)$

These complexities are not very impressive for computing a single isomer, in which case we would get better results just running it classically. However the probability of getting a good candidate isomer after n classical calculations is

$$1 - \left(1 - \frac{\delta}{|E_L^{\Gamma}| - |E_H^{\Gamma}|}\right)^n \tag{5.38}$$

Using the algorithm analysed in equation 5.37, the probability of having found a good candidate after n samples is:

$$1 - \left(1 - \frac{|E_L^{\Gamma}|^3 - |E_L^{\Gamma} - \delta|^3}{|E_L^{\Gamma}|^3 - |E_H^{\Gamma}|^3}\right)^n \tag{5.39}$$

This enables us to potentially run this on larger isomer spaces and still have a high probability of getting some of the lowest energy candidates. This works only because we took care to sample in such a way that the probability of getting a specific candidate is proportional to its energy or, if we want to improve this result, how different it is energetically to a known high energy isomer.

Assuming a fixed b the number of qubits used for each of these quantum algorithms scales with $O(c+i)$ and the circuit depth scales with $O(c)$ or $O(c^2)$. This is modest compared to the size of the isomer spaces we are feeding into them as a super position.

6 Related Work

6.1 xtb version 6.4.0

The xtb project has basic GPU support through the Nvidia Fortran compiler 'nvfortran', but this compiler fails on newer versions of the project. To get a version that has been officially tested with nvfortran, we have to go all the way back to version 6.4.0 from Feburary 2021. We have successfully made a reproducable build for this version and managed to run it, but the output seems much different from newer versions. There has been released 7 versions since, so doing any benchmarking comparisons with this older version likely would not be fair or representative.

6.2 dxtb

The dxtb project is a re-implementation of the xTB methods written in Python. This implementation has much better comments that actually explain the functions, but we discovered the project late into the process, and as such it has not been hugely beneficial for us. The project is no substitute for a GPU implementation, but it could have helped us with our own Python re-implementation.

7 Methodology

7.1 Porting the Reference Implementation

With all the equations from the xTB paper now in place, the next step is to implement them in code. Writing another Python re-implementation might seem rendundant, but the purpose of doing so is to start with a sequential version that is hopefully easier to reason about, and which is structured in a way that can more easily be translated to parallel GPU code.

We started implementing the equations from the paper directly, but found that it gave results different from the Fortran implementation. It is not that the Fortran code does it differently from what is described in the paper, but rather that there are details that might appear obvious to a chemist, which is not explicitly described in the paper. In [2] it is not specified in equation 9 for repulsion energy (7.1), that A and B cannot be the same atom index. They cannot be the same because R_{AB} is the distance between the two atoms, so if A and B is the same, then the distance is 0, which will result in division by zero. This makes sense intuitively because an atom does not repulse itself, but this is not always immediately apparent to someone outside the field.

$$E_{rep} = \frac{1}{2} \sum_{A,B} \frac{Z_A^{eff} Z_B^{eff}}{R_{AB}} e^{-\sqrt{a_A a_B}(R_{AB})^{(k_f)}} \tag{7.1}$$

In [1] they write that they use SCC, but they do not mention which method they use. It turns out that they use Broyden's method. The unit convertions between angström and bohr that are applied to the input geometry data, does also not seem to be mentioned anywhere.

Suffice to say, at the time we were not aware of all these details, so for the sake of correctness it was safer to follow the code rather than the papers. This approach came with its own challenges as we had to become acquainted with Fortran as a programming language while deciphering all its unintuitive quirks. Here is an overview of different challenges, inconsistencies, and confusing language quirks we have encountered while working with the xtb project.

- The project is written in object-oriented Fortran, which makes the execution flow difficult to follow.

- Arrays in Fortran start at 1 by default, but it is possible to specify that it should start at any other index. We have encountered a wide mix of start indices including 0, 1, 5, and 11.

- Variable names in Fortran are case-insensitive, and we encountered a variable that was defined with uppercase but referenced with lowercase.

- Fortran uses column-major indexing, so we have swapped the indices in Python.

- The goto statement exists in Fortran, and has been used inconsistently for something that should have been a simple for loop.

- Interfaces in Fortran are a way to overload a method, and it can be difficult to know which function it calls without digging into the subtle type differences of the arguments.

- In Fortran 'pure elemental' functions are pure functions that can be given an array in place of any single value argument of the same type. The function will essentially unpack the array and call the function with each element.

- Fortran uses parenthesis to index into an array. A subroutine is called by using the 'call' keyword, but pure functions does not. This makes it unclear whether we are calling a pure function or indexing into an array.

In addition to these confusing language specifics, there are also numerous structural oddities within the code, some of which we present below.

- The code authors often use symmetric pairing functions which are functions that map some multi-dimensional index into a flat 1-dimensional index. Specifically, in this code it maps pairs of indices (x, y) and (y, x) to the same index. This would be easier to understand with a 2-dimensional matrix, but the reason they do this is to save memory on symmetric matrices where only one of the triangular halves are needed. This understanding is difficult to come up with from the code, because the pairing function is named '`lin`', which says little about its effect, and there are no comments explaining what it is.

- The non-descriptive variable name leads us to the next point, being that this is a reoccuring problem caused by the fact that most of the names are abbreviated with three letters.

This makes it impossible to know what the values actually are and makes the codebase unapproachable for new contributors.

Some of the command-line arguments have also caused some confusion. We noticed that the program caches previous computations for a molecule, so running it twice on say a C200 molecule, will make the second run much faster. This is undesirable when we need to know what the total computational time is, and so to disable this behavior, the program has the flag seen in Figure 7.1.

```
1  --[no]restart
2      restarts calculation from xtbrestart (default = true)
```

Figure 7.1: The xtb command-line flag that controls whether to continue the computation from the last run.

The description is ambiguous as it can both mean that we recompute the data stored in `xtbrestart`, or that we reuse the value and continue the computation from there. At first we thought that we would specify `--restart` because we want to restart the whole computation without any caching. In reality what we need is `--norestart` because it should not use `xtbrestart` which is the cache from the previous run. We believe that all of these points are worth taking note of for anyone new to the codebase.

When porting code from an implementation and a language with so many quirks and pitfalls, it becomes exceedingly important to have tests in place to catch these errors. Having a test suite that enables us to compare our implementation to the original has been indispensible to us for a fast iteration loop that gives insight into the mismatches and any potential patterns in the discrepancies.

7.2 Testing

The xtb codebase has a large footprint, and there is a lot of overlap between the xTB methods it implements, so even focusing on just one of them still requires a great amount of code. The large amount of computations we have to deal with when porting the code makes proper validation especially important, as it becomes increasingly easy to make mistakes.With the discrepancies between the paper and the code, in addition to details that seem to go unexplained, we realized that the obvious approach forward was to lean towards the existing implementation rather than the paper.

One of the hurdles from testing against an existing program becomes the lack of transparency regarding the logic that takes place between the initial input and the final output presented to the user. Thankfully, the source code is publicly available, which allows for easy manipulation of the original flow of execution, thus avoiding the hassle of testing against a black box or the need to resort to methods of reverse engineering.

On behalf of these considerations we decided to write patches that allow us to intercept the arguments and results of arbitrary functions just by running the program as normal. This gave us the ability to implement and test abritrary functions in an isolated manner, by eliminating the need to compute their arguments.

7.2.1 Towards Reproducability with Nix

An important part of any software is reproducability, and applying certain patches in certain scenarios is something that should preferably be automated, reproducable, and optimally also portable. This is especially important for this approach to validation as it requires a way to reproduce a specific version of xtb linked to the same versions of dependencies. Essentially an exact copy of the original shell environment to ensure that patches work, results are the same, and no new bugs appear in the program itself or its dependencies. All of this should be achievable without having to add, remove, downgrade or upgrade system packages on your system.

The well-known contenders for this is any of the numerous containerization solutions on Linux, such as Docker, Podman, LXC etc. There are some problems with these options though, one being that it can be difficult to truly reproduce package versions without saving the resulting container image, another being that it does not solve the problem of having multiple versions of the same package installed. Some other notable limitations are that it limits the process to run within the container and passing in a GPU or other hardware is not as straightforward. A container also does not have access to the X or Wayland session needed to run GUI applications, though that is not currently relevant in this case.

Another approach which has been growing in popularity in recent years are tools that take unique approaches to package management in order to make not only packages reproducable, but also shell environments, system configurations and other forms of "outputs". Two such popular package managers are Nix[1] from the Nix team and Guix[2] from the GNU foundation. Nix is arguably the more popular option and it is also the solution that has been chosen for this project.

[1] `https://nixos.org/`
[2] `https://guix.gnu.org/`

Nix is an umbrella term that can refer to either the Nix functional programming language, the package manager, or the Nix based Linux distribution NixOS. The language and package manager go hand in hand and can be used on any Linux distribution. As such, NixOS is not required for the needs of this project and will not be mentioned going forward.

Nix does not follow the Unix Filesystem Hierarchy Standard (FHS), which brings with it some challenges, but this fundamental difference from other package managers is a major part of what makes Nix so powerful. Rather than installing packages into the usual system paths like '/bin', '/lib' etc. Nix installs everything into a read-only path called the Nix store under '/nix/store'. Everything in the Nix store is a result of a core concept in Nix called a derivation, which is essentially a build task to produce some output of files into the Nix store. All outputs into the store is marked with a custom hash in the filename called a NAR hash. These fundamental ideas fix some common problems such as circular dependencies and allow having multiple versions of the same package installed as they will simply coincide in the Nix store with different NAR hashes.

The typical binary on Linux is dynamically linked against the FHS compliant paths and it is not uncommon to have them hardcoded either. To make use of the packages in the Nix store, it is required to either recompile the program against the store paths, or in the case of proprietary software, patching the ELF header is needed to change the path to the interpreter and to dynamically linked libraries. Thankfully the Nix package repository 'NixPkgs' is the largest and freshest out there[3], so as a typical user doing this is rarely needed. Nixpkgs is a version-controlled repository on GitHub, so using older versions of packages even alongside newer ones, is fairly trivial as it simply requires fetching multiple revisions of the repository.

This along with the previously mentioned features have allowed a greatly simplified process of not only running the newest version 6.7.1 of the xtb program, but also running the much older nvfortran compatible version 6.4.0 alongside it. NixPkgs is also a collection of library functions, and the helper functions for making derivations called 'mkDerivation' make it easy to define all the stages of packaging a program including unpacking, patching, building, checking, and installing the files. With this, the whole pipeline of patching, compiling, running, and passing the data over to the Python validation tests can be achieved with a single shell command.

```
> nix run .#cmp-impls
```

This command takes the form '`nix run <path to flake>#<output>`'. Path to flake refers to a file-tree whose root directory contains a file called 'flake.nix'. Nix flakes is an experimental but

[3] `https://repology.org/repositories/graphs`

widely adopted feature, which provides a standard way to write Nix expressions and a way to manage their dependencies through a version-pinned lock file. The 'flake.nix' file follows a uniform naming schema for declaring inputs and outputs, where inputs are the dependencies, and outputs are Nix expressions to be exposed. The new Nix command-line interface needed to interact with flakes is naturally also an experimental feature that has to be enabled explicitly. The run command instructs Nix to build and run the derivation 'cmp-impls', which is defined as an app in the flake outputs.

```
{
  inputs = {
    nixpkgs.url = "github:nixos/nixpkgs/nixos-unstable";
  };

  outputs = { self, nixpkgs, ... }: {
    apps."x86_64-linux" = let
      pkgs = nixpkgs.legacyPackages."x86_64-linux";
      ...
    in {
      "cmp-impls" = let
        python = (pkgs.python3.withPackages (python-pkgs: with python-pkgs; [
          numpy scipy cvxopt
        ]));
      in {
        type = "app";
        program = toString (pkgs.writeShellScript "cmp-impls" ''
          PYTHONPATH=${pkgs.lib.cleanSource ./xtb-python} exec
          ↪ ${python}/bin/python \
            ${./xtb-python/cmp_impls.py} ${xtb_test_data}
        '');
      };
    };
    ...
  };
}
```

Figure 7.2: This is a snippet of our Nix flake which shows the app declaration for running our test suite. The app depends on the derivation that generates the test data.

Python is declared with the required packages and is then used in the app to call the `cmp_impls.py` script. The script is called with the test data acquired from running the xtb program. This data comes from another derivation which executes patched versions of xtb and dftd4 on a C_{200} fullerene to get the relevant function arguments and results as binary files.

```
xtb_test_data = builtins.derivation {
  name = "xtb-test-data";
  system = "x86_64-linux";
  builder = "${pkgs.bash}/bin/bash";
  src = ./xtb-python/data/C200.xyz;
  args = ["-c" ''
    PATH=$PATH:${pkgs.coreutils}/bin
    mkdir -p ./calls/{build_SDQH0,coordination_number,\
      dim_basis,dtrf2,electro,form_product,get_multiints,\
      h0scal,horizontal_shift,multipole_3d,newBasisset, olapp}
    ${xtb}/bin/xtb $src
    ${dftd4}/bin/dftd4 $src
    mv calls $out
  ''];
};
```

Figure 7.3: This is the Nix derivation for generating the test data from our patched versions of xtb and dftd4.

The directories for the binary files are created in advance as checking whether they exist when writing the binary files has a large overhead. This derivation in turn uses derivations for xtb and dftd4. Luckily dftd4 is already in NixPkgs, but it still needs to be patched in order to extract the required data for validation. Thankfully the `mkDerivation` function used in NixPkgs makes overriding and patching a package very straightforward.

```
dftd4 = (pkgs.dftd4.overrideAttrs (finalAttrs: previousAttrs: {
  src = pkgs.fetchFromGitHub {
    owner = "dftd4";
    repo = "dftd4";
    rev = "502d7c59bf88beec7c90a71c4ecf80029794bd5e";
    hash = "sha256-FEABtBAZK0xQ1P/Pbj5gUuvKf8/ZLITXaXYB+btAY/8=";
  };
  buildInputs = [ multicharge ] ++ previousAttrs.buildInputs;
  doCheck = false;
  patches = previousAttrs.patches ++ [
    ./nix/patches/dftd4/use_gfn2.patch
    ./nix/patches/dftd4/log_args_and_outputs.patch
  ];
}));
```

Figure 7.4: This is the Nix expression for overriding the dftd4 package derivation. We essentially update dftd4 to a newer revision and add patches to log arguments and results to a binary file, and another patch to use GFN2.

The version is bumped by overriding the source, and the multicharge project is added from NixPkgs and also bumped as a requirement of this newer version. Some of the tests were timing out, so they have been disabled by setting `doCheck` to false. Lastly the patches are applied by providing the relevant patch files.

The xtb project and two of its dependencies, namely CPCM-X and numsa are not in NixPkgs, so we had to package them ourselves. This is one of our contributions, which makes xtb easily available to all Linux distributions. It is a reproducable package that works the same on different machines regardless of their respective system packages. We have also packaged the old version 6.4.0 of xtb, which requires various old dependencies and is the only way currently to run xtb on a GPU through nvfortran. We have shared this achievement on the xtb GitHub repository, and others have since mentioned our GitHub issue to highlight the challenges of getting the GPU version to run.

7.2.2 Patching xtb

Now we have a way to apply patches and repoduce our tests. Next we will dive into what the patches do and how they are used. All the patches follow the structure seen in Figure 7.5 where the original function is prefixed with a 'g', such that the new wrapper function will be called instead. The wrapper function writes the function arguments to a binary file before calling the actual function before finally writing the result of the function to the same binary file. Writing a file for each call to a function is a bit excessive and will produce a very large amount of files, so a threshold has been used to create an upperbound on the number of files that can be created for each function.

```
+   logical :: hit_threshold
+   integer :: u
+   character(len=200) :: path
+
+   hit_threshold = testfile_path('electro', path)
+   if (.not.hit_threshold) then
+     open(newunit=u, file=trim(path), form='unformatted', access='stream')
+     write(u) nbf
+     write(u) size(H0), H0
+     write(u) size(P, 1), size(P, 2), P
...
+     if (allocated(ies%thirdOrder%atomicGam)) then
+       write(u) size(ies%thirdorder%atomicgam), ies%thirdorder%atomicgam
+     else
+       write(u) 0
+     end if
...
+     write(u) size(ies%jmat, 1), size(ies%jmat, 2), ies%jmat
+     write(u) size(ies%shift), ies%shift
+   end if
+
+   call gelectro(n,at,nbf,nshell,ies,H0,P,dq,dqsh,es,scc)
+
+   if (.not.hit_threshold) then
+     write(u) es
+     write(u) scc
+     close(u)
+   end if
```

Figure 7.5: This is a diff file for the electro energy function. A diff file reflects the changes between two files and can be used to patch code by applying these changes. All our patches follow this structure of writing the arguments to a file, then running the original function before finally writing the result to the same binary file.

7.2.3 Implementing the Tests

With the binary files containing the arguments and results, we now have all the data necessary to compare against our Python implementation. We have made a test suite in the file `cmp-impls.py` where all tests follow these same steps:

1. Load an deserialize a binary file for the appropriate function

2. Call the corresponding Python function with the deserialized arguments

3. Compare the result against the deserialized Fortran result

4. Repeat until there are no more binary files for this function

```python
def test_electro():
    fn_name = "electro"
    for i, file_path in enumerate(glob.glob(f'{directory}/{fn_name}/*.bin')):
        with open(file_path, 'rb') as f:
            def read_ints(n=1):
                return np.fromfile(f, dtype=np.int32, count=n)

            nbf = read_ints(1)[0]
            HO1 = read_ints(1)[0]
            HO = np.fromfile(f, dtype=np.float64, count=HO1)
            m, n = read_ints(2)
            P = np.fromfile(f, dtype=np.float64, count=m * n).reshape((n, m))
            ...
            atomicGam1 = read_ints(1)[0]
            atomicGam = None  if atomicGam1 == 0
                              else np.fromfile(f, dtype=np.float64,
                              ↪  count=atomicGam1)
            ...
            es_res, scc_res = read_reals(2)
            es, scc = electro(nbf, HO, P, dq, dqsh, atomicGam, shellGam, jmat,
            ↪  shift)

            is_equal(es, es_res, "es", fn_name)
            is_equal(scc, scc_res, "scc", fn_name)

    print(f"matches! [{fn_name}]")
```

Figure 7.6: This is some of the code from the test for the electro function. It shows how we iterate through each binary file, deserialize its data, call the corresponding Python implementation, and then compare the results.

8 Results

8.1 Our Contributions

In this section we will present and reiterate the contributions we have made as a result of this project.

We have contributed with a simplified Python version of the original GFN2-xTB Fortran implementation. The implementation and validation of the dispersion term is not complete, and we are missing the anisotropic electrostatic term (AES), the anisotropic exchange correlation term (AXC), the Fermi term, and the implementation for the self-consistent charges. The most difficult part of AES and AXC is getting the S, D, and Q matrices, which we already have. Given the size of the reference implementation we are quite satisfied with this. The Python implementation should hopefully make the GFN2-xTB algorithm more approachable for our successors and give a good foundation for making the lockstep-parallel GPU implementation.

We have contributed a testing framework for comparing results against the original Fortran implementation. We utilize Nix to ensure tests that are reproducible, regardless of Linux distribution and system configuration. In connection to this, we have also contributed a reproducible and easy way to build, run, and patch xtb. This also includes its dependencies, and even the older GPU compatible version 6.4.0 of xtb known from GitHub issues to be difficult to get up and running.

This thesis contributes an in-depth walk-through of the GFN2-xTB algorithm with code snippets that directly link to the equations in a way that should be more easily digestible for a computer scientist. As an extension of this, we have presented insights and ideas on how to approach a massively lockstep-parallel implementation by optimizing code snippets for our problem domain, and transforming them such that they adhere to the requirements of a lockstep-parallel kernel.

We have also presented a case study with the NVIDIA A100 data centre GPU to show how GPU architectures are structured and how to use the technical specifications to estimate hardware requirements.

Additionally we have designed quantum circuits for computing the isotropic energy terms in GFN2-xTB, with probability

$$1 - \left(1 - \frac{|E_L^\Gamma|^3 - |E_L^\Gamma - \delta|^3}{|E_L^\Gamma|^3 - |E_H^\Gamma|^3}\right)^n \tag{8.1}$$

of finding a candidate within δ of the lowest energy using n samples, compared to the classical

$$1 - \left(1 - \frac{\delta}{|E_L^\Gamma| - |E_H^\Gamma|}\right)^n \tag{8.2}$$

. These could be used as a starting point for a fully quantum computational approach.

8.2 Validation

All except one of our tests pass indicating that our results match the reference implementation. We had to introduce a tolerance for some of the comparisons as they came extremely close. Figure 8.1 shows the maximum squared deviations of all tests for the electrostatic term. This term returns the isotropic electrostatic energy and the self-consistent charges, so there are two plots to show the deviation for both.

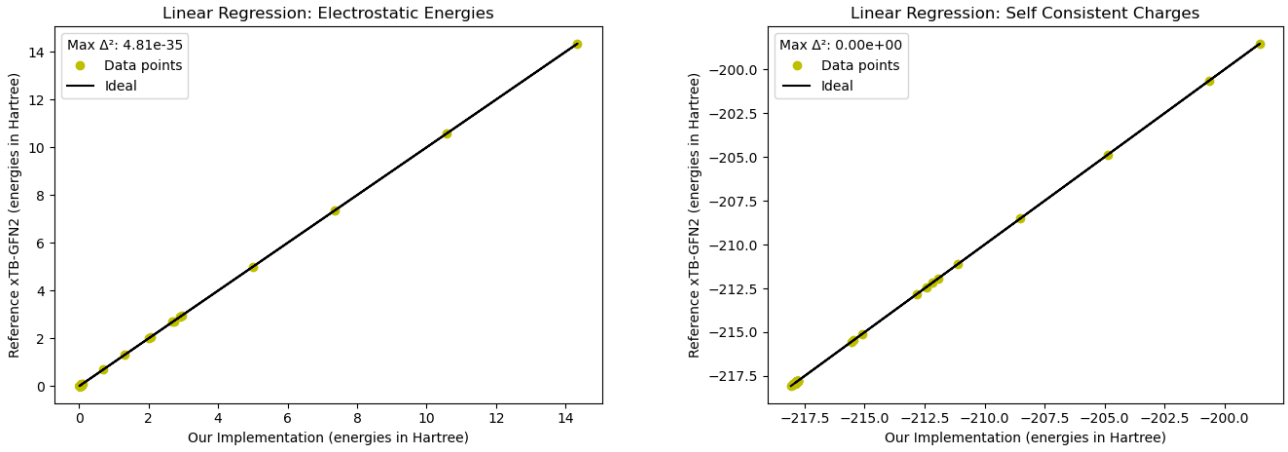


Figure 8.1: Linear regressions showing how much the Python implementation for the electrostatic term deviates from the Fortran results.

We can see that the results for the self-consistent charges match exactly, and that the maximum squared deviation for the electrostatic energies is miniscule. Figure 8.2 shows the results from our comparison tests.

```
matches! [olapp]
matches! [multipole_3d]
matches! [multipole_3d_simple]
matches! [horizontal_shift]
matches! [form_product]
matches! [horizontal_shift_simple]
matches! [form_product_simple]
matches! [dtrf2]
li and lj are 0 for all calls, so s is not actually modified ToT
AKA all dtrf2 calls are essentially skipped...
matches! [get_multiints]
[build_overlap_dipol_quadrupol] S: Is not close!
...
matches! [build_overlap_dipol_quadrupol]
matches! [build_SDQH0]
matches! [dim_basis]
matches! [atovlp]
matches! [new_basis_set]
matches! [electro]
matches! [coordination_number]
matches! [h0scal]
```

Figure 8.2: Truncated output of our comparison tests.

8.3 Benchmarks

9 Reflections

In this section we will reflect on our choices and approaches to highlight what we think worked well. We will also share possible changes worth considering for anyone pursuing to continue this project.

We have primarily made use of the Python programming language for the prototype implementation of GFN2-xTB. This has worked well due to Python's simple syntax, runtime checks, and abstractions of low-level details such as memory management. These traits have let us focus on correctness rather than specifics about the code and language.

The package manager and functional language Nix has given us similar advantages by providing us with reproducable development environments, a flexible solution for software packaging, and a way to combine this to make a reproducable testing pipeline.

Nix is known to have a rather steep learning curve, which deters many, but given our experiences, we can wholeheartedly recommend this workflow. The testing setup with Nix has been especially valuable, continuing to pay dividends by ensuring correctness as we transition from a prototype to a high performance implementation.

The xTB family of algorithms presented by Grimme et al. is still being actively developed. In fact g-xTB came out a little over half way though this project and is rather impressive regarding accuracy and breadth of application. We urge anyone continuing this project to consider if the resulting changes impact fullerenes enough to warrant switching to the new method. We were unable to make this assessment as the reference implementation has still not landed in the tblite repository, and the associated paper is still only a preprint available on chemRxiv[15]. The eventual advent of g-xTB may justify adapting our work to the new method.

Completing the Python prototype is highly relevant as it is a crucial step towards the main prize of a fully lockstep-parallel implementation of the algorithm. An interesting stand-alone part of the method is implementing the D4' dispersion model.

Slightly outside of the project scope, we would like to extend the prototype from just computing the energy terms mentioned to include also polarization and excitation energies. Implementing at least one of the solvation models available in the xtb program would also be a welcome addition. The current prototype handles everything as one 'cell', perfect for simulating individual molecules, however extending it to handle repeating cells, such as in crystals, should be a rather small change.

Regarding quantum algorithms we want to investigate using quantum singular value decomposition transforms for some of the terms that are more heavy on tensors. This would be a good thesis topic for anyone interested in quantum algorithm design. It would also be interesting to investigate if any of the approximations in GFN2-xTB are unnecessary in the domain of quantum computing.

10 Conclusion

We have shown that massive lockstep-parallelization on a GPU is a good fit for the problem domain, and for GFN2-xTB in general when considering isomer spaces. Specifically we have explained how the magnitude of parallalism scales with the size and number of isomer spaces, and that this allow us utilize GPU cores proportional to the order of fullerenes, which is 10^8 in total when considering all the isomer spaces $C_{20}, \cdots, C_{200}$. We have shared ways to evaluate the capability of hardware for our problem domain, in order to estimate the order of parallalism, and to make considerations about the choice of hardware. We have also covered considerations about input structure when generalizing lockstep algorithms for all isomer spaces.

Additionally we have given quantum algorithms for the anisotropic terms that scale with the logarithm of the size of the isomer space in terms of qubits. In terms of circuit depth we achieve quadratic scaling in the number of atoms per isomer for each sampling. This is the similar to the classical algorithm but with a much better probability of sampling a good candidate from the isomer space. The probability of sampling an isomer within δ of the lowest energy after n samples is $1-(1-\delta/(|L|-|H|))^n$ in the classical case and $1-(1+(|L|^3-|L-\delta|^3)/(|L|^3-|H|^3))^n$ in the quantum case with L and H being the lowest and highest energies in the isomer space and assuming energies evenly distributed in the interval between.

11 GAI Declaration

Generative AI has been used in the making of this thesis.

Specifically ChatGPT has been used as a patient explainer of basic chemistry and Fortran in the research phase and occasionally to give feedback on phrasing during the writing phase.

Asmus tried to get it to help with debugging, with no success, as it is underwhelming in its ability to reason about the quantum chemistry it explains in terms of correct code. It is however surprisingly good at linking the short variable names in the reference implementation of GFN2-xTB to googlable terms, which was quite needed.

Anton has used it to quickly find Python equivalents to builtin Fortran functions when porting some of the code. He has also used it to confirm what to consider when calculating memory bandwidth.

None of the output is part of the code or this thesis.

Bibliography

[1] S. Grimme, C. Bannwarth, and P. Shushkov. “A Robust and Accurate Tight-Binding Quantum Chemical Method for Structures, Vibrational Frequencies, and Noncovalent Interactions of Large Molecular Systems Parametrized for All spd-Block Elements (Z = 1–86)”. In: *Journal of Chemical Theory and Computation* 13.5 (2017). PMID: 28418654, pp. 1989–2009. DOI: 10.1021/acs.jctc.7b00118. eprint: https://doi.org/10.1021/acs.jctc.7b00118.

[2] C. Bannwarth, S. Ehlert, and S. Grimme. “GFN2-xTB—An Accurate and Broadly Parametrized Self-Consistent Tight-Binding Quantum Chemical Method with Multipole Electrostatics and Density-Dependent Dispersion Contributions”. In: *Journal of Chemical Theory and Computation* 15.3 (Mar. 2019), pp. 1652–1671. ISSN: 1549-9618. DOI: 10.1021/acs.jctc.8b01176.

[3] C. Bannwarth, E. Caldeweyher, S. Ehlert, A. Hansen, P. Pracht, J. Seibert, S. Spicher, and S. Grimme. “Extended tight-binding quantum chemistry methods”. In: *WIREs Computational Molecular Science* 11.2 (2021), e1493. DOI: https://doi.org/10.1002/wcms.1493. eprint: https://wires.onlinelibrary.wiley.com/doi/pdf/10.1002/wcms.1493.

[4] R. Daudel, G. Leroy, D. Peeters, and M. Sana. *Quantum Chemistry*. John Wiley & Sons, 1983, pp. 160, 161, 175–183. ISBN: 9780471901358.

[5] P. University. *London Dispersion Forces*. Purdue. URL: https://www.chem.purdue.edu/gchelp/liquids/disperse.html (visited on Aug. 14, 2025).

[6] S. Grimme, S. Ehrlich, and L. Goerigk. “Effect of the damping function in dispersion corrected density functional theory”. In: *Journal of Computational Chemistry* 32.7 (2011), pp. 1456–1465. DOI: https://doi.org/10.1002/jcc.21759. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/jcc.21759.

[7] S. Grimme, J. Antony, S. Ehrlich, and H. Krieg. “A Consistent and Accurate Ab Initio Parametrization of Density Functional Dispersion Correction (DFT-D) for the 94 Elements H-Pu”. In: *The Journal of chemical physics* 132 (Apr. 2010), p. 154104. DOI: 10.1063/1.3382344.

[8] R. Hormuth. *Leadership HPC Performance with 5th Generation AMD EPYC Processors*. 2025. URL: https://www.amd.com/en/blogs/2025/leadership-hpc-performance-with-5th-generation-amd.html (visited on Aug. 11, 2025).

[9] NVIDIA. *NVIDIA A100 Tensor Core GPU Architecture*. V1.0. NVIDIA.

[10] NASA. *Basics on NVIDIA GPU Hardware Architecture*. NASA Advanced Supercomputing (NAS) Division. 2025. URL: https://www.nas.nasa.gov/hecc/support/kb/basics-on-nvidia-gpu-hardware-architecture_704.html (visited on Aug. 10, 2025).

[11] T. G. Draper. *Addition on a Quantum Computer*. 2000. arXiv: quant-ph/0008033 [quant-ph].

[12] L. Ruiz-Perez and J. C. Garcia-Escartin. "Quantum arithmetic with the quantum Fourier transform". In: *Quantum Information Processing* 16.6 (Apr. 2017). ISSN: 1573-1332. DOI: 10.1007/s11128-017-1603-1.

[13] S. Wang, Z. Wang, G. Cui, L. Fan, S. Shi, R. Shang, W. Li, Z. Wei, and Y. Gu. *Quantum Amplitude Arithmetic*. 2020. arXiv: 2012.11056 [quant-ph].

[14] T. Häner, M. Roetteler, and K. M. Svore. *Optimizing Quantum Circuits for Arithmetic*. 2018. arXiv: 1805.12445 [quant-ph].

[15] T. Froitzheim, M. Müller, A. Hansen, and S. Grimme. *g-xTB: A General-Purpose Extended Tight-Binding Electronic Structure Method For the Elements H to Lr (Z=1–103)*. This content is a preprint and has not been peer-reviewed. 2025. DOI: 10.26434/chemrxiv-2025-bjxvt.

[16] A. Gilyén, Y. Su, G. H. Low, and N. Wiebe. "Quantum singular value transformation and beyond: exponential improvements for quantum matrix arithmetics". In: *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing*. STOC 2019. Phoenix, AZ, USA: Association for Computing Machinery, 2019, pp. 193–204. ISBN: 9781450367059. DOI: 10.1145/3313276.3316366.