Documentation for `Hawaii Hybrid` v.0.1

A. Finenko and D. Chistikov

April 15, 2025

Contents

1 Problems addressed by library

1. Compute statistical averages over ensembles of classical trajectories restricted by chosen phase-space domains (e.g., bound or unbound states), enabling the calculation of dipole autocorrelation functions.
2. Calculate static phase-space averages using rejection sampling and adaptive Monte Carlo algorithms to estimate zeroth and second spectral moments.
3. Transform correlation functions into spectral functions, apply smoothing, and convert them into spectral profiles.

To ensure clarity, we will outline the key objects and their corresponding units of measurement. In the context of this document, the correlation function of the dipole moment, $C(t)$, is defined as:

$$C(t) = V\langle\mu(0)\mu(t)\rangle. \tag{1}$$

Note that this definition differs from others, such as the one in Ref. [**Chistikov2021**], as it does not include the $1/(4\pi\varepsilon_0)$ factor. Instead, this factor is incorporated into the definition of the spectral function. The correlation function values are produced by `calculate_correlation_and_save` and `calculate_correlation_array_and_save` in units of ($m^3\cdot$ atomic unit of dipole). The time is treated internally in atomic time units, and the produced `CFnc` object stores the time in the atomic time units. However, when the correlation function is to be processed in order to be converted into spectral function (by `fit_baseline` function), time axis needs to be in seconds.

The spectral function is defined as follows:

$$G(\nu) = \frac{1}{2\pi}\frac{1}{4\pi\varepsilon_0}\int_{-\infty}^{+\infty} C(t)e^{-2\pi ic\nu t}\,\mathrm{d}t, \tag{2}$$

where ν is the wavenumber (cm^{-1}). The values of spectral function are $\mathrm{J}\cdot\mathrm{m}^6\cdot\mathrm{s} = \mathrm{kg}\cdot\mathrm{m}^8\cdot\mathrm{s}$.

The binary absorption coefficient is then related to the spectral function according to:

$$\alpha(\nu) = \frac{\tau(\nu)}{\rho_1\rho_2} = \frac{(2\pi)^4 N_L^2}{3h}\nu\left[1-\exp\left(-\frac{hc\nu}{k_\mathrm{B}T}\right)\right]G(\nu), \tag{3}$$

where the absorption coefficient is

$$\tau(\nu) = L^{-1}\ln(I_0/I), \tag{4}$$

and ρ_1 and ρ_2 are gas densities in mixture. We will express the binary absorption coefficient as per convention in $\mathrm{cm}^{-1}\mathrm{Amagat}^{-2}$.

2 Examples

2.1 Propagating trajectory for CO_2-Ar

As an initial illustration of the use of `Hawaii Hybrid` package, we give a sample program called `trajectory_co2_ar.cpp` to propagate a trajectory of collisional dynamics of CO_2-Ar system. The problem consists of solving the following Hamilton dynamical equations:

	trajectory	*phase-space moments*	*single correlation function*	*array of correlation functions*
He-Ar	✓	✓	✓	×
CO_2-Ar	✓	✓	✓	✓
H_2-Ar	×	×	×	×
CH_4-CO_2	✓	✓	✓	✓

$$
\begin{aligned}
\dot{R} &= \frac{p_R}{m}, \\
\dot{p}_R &= \frac{p_\Theta^2}{mR^3} + \frac{p_\Phi^2}{mR^3 \sin^2\Theta} - \frac{\partial U}{\partial R}, \\
\dot{\Phi} &= \frac{p_\Phi}{mR^2 \sin^2\Theta}, \\
\dot{p}_\Phi &= -\frac{\partial U}{\partial \Phi}, \\
\dot{\Theta} &= \frac{p_\Theta}{mR^2}, \\
\dot{p}_\Theta &= \frac{p_\Phi^2 \cos\Theta}{mR^2 \sin^3\Theta} - \frac{\partial U}{\partial \Theta}, \\
\dot{\varphi}_1 &= \frac{p_1^\varphi}{I_1 \sin^2\vartheta_1}, \\
\dot{p}_1^\varphi &= -\frac{\partial U}{\partial \varphi_1}, \\
\dot{\vartheta}_1 &= \frac{p_1^\vartheta}{I_1}, \\
\dot{p}_1^\theta &= \frac{\left(p_1^\varphi\right)^2 \cos\theta_1}{I_1 \sin^3\theta_1} - \frac{\partial U}{\partial \theta_1}.
\end{aligned}
$$

The `hawaii.h` header files provides definition of the struct `MoleculeSystem` that is used to evaluate the right-hand side of the dynamical equations. The `trajectory.h` header provides struct `Trajectory` that provides interface with `CVode` library that performs the integration of dynamical equations (see Section **??**). Next, we provide implementations for functions that calculate potential energy `pes` and its derivatives with respect to coordinates `dpes`. Each user program that instantiates the `MoleculeSystem` is expected to implement these functions. To implement these functions `hawaii` provides functions to transform angles between different reference frames (see Section 3.9).

2.2 Propagating trajectory for H_2-Ar while requantizing the angular momentum of H_2

2.3 Calculating the zeroth and second spectral moments of CO_2-Ar as phase-space averages using rejection-based sampler

This example demonstrates how to use interface with `hep` to calculate spectral moments. First, the zeroth moment over unbound states is calculated as follows:

$$M_0 = \frac{(2\pi)^4 N_L^2}{3(Q/V)h} \frac{1}{4\pi\epsilon_0} \frac{1}{2\pi c} \int\limits_{\Omega: H>0} \mu^2 \exp\left(-\frac{H}{kT}\right) \mathrm{d}\mathbf{q}\, \mathrm{d}\mathbf{p}.$$

The integral is evaluated using `mpi_perform_integration`, whereas the partition can be calculated using `analytic_full_partition_function_by_V`.

2.4 Calculating a single correlation function for CO_2-Ar

2.5 Calculating a spectral function using p_r/μ-representation for CO_2-Ar

2.6 Calculating an array of correlation functions for CO_2-Ar

2.7 Processing correlation function for CO_2-Ar

3 Complete list of function and code organization

3.1 Setting up a molecule pair

`enum MonomerType`

Values
```
ATOM = 0
LINEAR_MOLECULE = 4
LINEAR_MOLECULE_REQUANTIZED_ROTATION = MODULO_BASE + 4
/* LINEAR_VIBRATING_MOLECULE = MODULO_BASE + 6 */
ROTOR = 6
ROTOR_REQUANTIZED_ROTATION = 2*MODULO_BASE + 6
```

Description: This enum is used to distinguish between systems of different types and store the size of the phase point: `size(phase_point) = MonomerType % MODULO_BASE`, where `MODULO_BASE` is `#defined` to 100 by default.

`struct Monomer`

Fields:
`MonomerType t`
`double I[3]` – values of tensor of inertia
`double *qp` – dynamic variables (currently, Euler angles and conjugated momenta) at the current step of simulation
`double *dVdq` – the derivatives of potential energy with respect to coordinates pertaining to this monomer (the order of coordinates is the same as for `qp`)
`bool apply_requantization`

Description: The `apply_requantization` will be set to `true` in `rhs` to signal that the requantization of the monomer's angular momentum is required during trajectory propagation. The order of variables in the `qp` array is specified by the following indices:

```
#define IPHI 0
#define IPPHI 1
#define ITHETA 2
#define IPTHETA 3
#define IPSI 4
#define IPPSI 5
```

`enum PairState`

Values `FREE_AND_METASTABLE` = 0
`BOUND` = 1

Description

`struct MoleculeSystem`

Fields `double intermolecular_qp[6]` – Coordinates and conjugated momenta that correspond to the intermolecular motion: (Φ, p_Φ, Θ, p_Θ, R, p_R).
`Monomer m1`
`Monomer m2`
`double mu` – reduced mass of molecule pair
`size_t Q_SIZE` – total number of coordinates for molecule pair
`size_t QP_SIZE` – total number of coordinates and momenta for molecule pair (a.k.a. `size(phase_point)`)
`double *intermediate_q` – contiguous vector of coordinates
`double *dVdq` – contiguous vector of potential energy derivatives

Description Keep in mind that angular variables and momenta are stored in the same order as for `qp` in `Monomer`. These variables' locations are `#defined` as follows:

```
#define IPHI 0
#define IPPHI 1
#define ITHETA 2
#define IPTHETA 3
#define IR 4
#define IPR 5
```

Keep in mind that intermolecular coordinates and monomer's coordinates are not stored contiguously. The contiguous vector of coordinates can be assembled by calling `extract_q_and_write_into_ms` function, which stores the coordinates in memory pointed at by `intermediate_q`. These coordinates are passed to external functions that compute the values of intermolecular energy, its derivatives with respect to coordinates and induced dipole (see section **??**).There is no guarantee that coordinates stored in `Monomer`'s and coordinates in memory at `intermediate_q` are always in sync. The function `extract_q_and_write_into_ms` must be invoked if the contiguous vector of coordinates is desired at a certain point of the program execution.

Function `init_ms`

Call `MoleculeSystem *ms = MoleculeSystem init_ms(mu, t1, t2, I1, I2, seed)`

Arguments `double mu` – the reduced mass of the molecule pair
`MonomerType t1` specifies the type of first monomer
`MonomerType t2` specifies the type of second monomer
`double* I1` contains inertia tensor values for first monomer. If the monomer is atom, no values will be read from the pointer, so `NULL` can be passed. Two and three values are expected for the rotor and linear molecule, respectively.
`double* I2` contains inertia tensor values for second monomer.
`size_t seed` is the seed for random number generator. A unique seed will be produced if `0` is passed.

Description The function prepares the `MoleculeSystem` struct based on the specified monomer types, allocates the memory using `malloc` (probably should be avoided due to cache locality) and initializes the random number generator.

Function `kinetic_energy`

Call `double kinetic_energy(*ms)`

Arguments `MoleculeSystem* ms`

Description The kinetic energy function is calculated at the phase-point stored in `MoleculeSystem`. Currently, implemented for intermolecular degrees of freedom, linear molecule (tested) and rotor (tested). Parts of the kinetic energy corresponding to intermolecular degrees of freedom:

$$T_{\text{int}} = \frac{p_R^2}{2\mu} + \frac{p_\Theta^2}{2\mu R^2} + \frac{p_\Phi^2}{2\mu R^2 \sin^2\Theta};$$

to linear molecule:

$$T_{\text{lin}} = \frac{p_\theta^2}{2I} + \frac{p_\varphi^2}{2I \sin^2\theta};$$

to rotor:

$$T_{\text{rotor}} = \frac{1}{2I_1 \sin^2\theta_2}\Big[(p_\varphi - p_\psi\cos\theta)\sin\psi + p_\theta \sin\theta\cos\psi\Big]^2 + \\ + \frac{1}{2I_2 \sin^2\theta}\Big[(p_\varphi - p_\psi\cos\theta)\cos\psi - p_\theta\sin\theta\sin\psi\Big]^2 + \frac{1}{2I_3}(p_\psi)^2.$$

Function `Hamiltonian`

Call	`double Hamiltonian(*ms)`
Arguments	`MoleculeSystem* ms`
Description	Calls to `kinetic energy`, assembles a contiguous vector of coordinates via `extract_q_and_write_into_ms` and passes it to external `pes`.

The potential energy and its derivatives are expected to be provided as functions with the following signatures:

Function `pes`

Call	`double pes(*q);`
Arguments	`double *q`
Description	An array of `Q_SIZE` variables is passed to this function; `MoleculeSystem` describes their order. Intermolecular distance measured in Bohrs, while angles are measured in radians. The function is expected to return energy in Hartree.

Function `dpes`

Call	`void dpes(*q, *dpesdq);`
Arguments	`double *q` `double *dpesdq`
Description	An array of `Q_SIZE` variables is passed to this function; `MoleculeSystem` describes their order. Intermolecular distance measured in Bohrs, while angles are measured in radians. The function is expected to return an array of derivatives of energy in units of Hartree/Bohr of length `Q_SIZE`.

Function `dipole`

Call	`void dpes(*q, *dpesdq);`
Arguments	`double *q` `double *dpesdq`
Description	

`pes` and `dpes` are defined as external functions in the header `hawaii.h`, so they have to be implemented in the user program, whereas the dipole function needs to be set through function pointer named `dipole`.

3.2 Sampling the phase-space point

Function `q_generator`

Call	`void q_generator(*ms, *params)`
Arguments	`MoleculeSystem* ms` `CalcParams *params`
Description	Generates R with density $\rho \sim R^2$ in the range [`params.sampler_Rmin`, `params.sampler_Rmax`]. The distributions of φ, ψ are $\varphi, \psi \sim U[0, 2\pi]$ and for θ is $\cos\theta \sim U[0, 1]$. Implemented for intermolecular degrees of freedom, linear molecule and rotor.

Function `p_generator`

Call	`void p_generator(*ms, T)`
Arguments	`MoleculeSystem* ms` `double T`
Description	Samples momenta p from distribution $\rho \sim e^{-K/kT}$ at given temperature. Switches between `p_generator_linear_molecule` and `p_generator_rotor` to sample momenta for monomers.

Function `reject`

Call	`bool reject(*ms, Temperature, pesmin)`

Arguments	`MoleculeSystem* ms`
	`double Temperature`
	`double pesmin` – the minimum value of PES

Description	Applies the rejection step to the phase-point that is stored in the `MoleculeSystem`. It presupposes that the provided phase-point is sampled from $\rho \sim e^{-K/kT}$ using `q_generator` and `p_generator` functions. The random variable $u \sim U[0, 1]$ is chosen, to determine whether the current phase-point is to be accepted with probability $\rho \sim \exp(-H/kT)$.

3.3 Calculating a trajectory: interfacing with `CVode` library

`struct Trajectory`

Fields	`size_t DIM`
	`size_t mxsteps`
	`N_Vector y`
	`N_Vector abstol`
	`N_Vector reltol`
	`SUNMatrix A`
	`SUNLinearSolver LS`
	`void *cvode_mem`

Description	

Function `init_trajectory`

Call	`Trajectory init_trajectory(*ms, *transformed, reltol);`
Arguments	`MoleculeSystem *ms` `double reltol`
Description	

Function `free_trajectory`

Call	`void free_trajectory(*traj);`
Arguments	`Trajectory *traj`
Description	

Function `reinit_trajectory`

Call `void reinit_trajectory(*traj, t);`
Arguments `Trajectory *traj`
`double t`
Description

Function `make_step`

Call `int make_step(*traj, tout, *t);`
Arguments `Trajectory *traj`
`double tout`
`double *t`
Description

Function `set_initial_condition`

Call `void set_initial_condition(*traj, qp);`
Arguments `Trajectory *traj`
`Array qp`
Description

Function `make_vector`

Call `N_Vector make_vector(size);`
Arguments `int size`
Description

Function `set_tolerance`

Call `void set_tolerance(*traj, tolerance);`
Arguments `Trajectory *traj`
* `double tolerance`
Description

Function `rhs`

Call `void rhs(t, y, ydot, *user_data);`
Arguments `UNUSED(realtype t)`
`N_Vector y` stores coordinates and conjugated momenta
`N_Vector ydot` is filled by function with numerical values of right-hand side of Hamilton's equations of motion at provided phase point
`void *user_data` is employed to pass `MoleculeSystem*` inside the function (see section **??**)
Description This function is passed to `CVode` library to propagate the trajectory (see section **??**).First, the phase-point coordinates are stored into `MoleculeSystem` struct. A contiguous vector of coordinates is assembled via `extract_q_and_write_into_ms`. Next, by calling the external function `dpes`, the derivatives of potential energy are computed and stored into `MoleculeSystem.dVdq`. The components of derivative vector are then copied into the field `Monomer.dVdq` of the corresponding monomer via the call to `extract_dVdq_and_write_into_monomers`. The right-hand side of Hamilton's equations with respect to intermolecular degrees of freedom are readily obtained and filled into `ydot`, while the derivatives with respect to monomer's coordinates are handled by `rhsMonomer` function.

Function `rhsMonomer`

Call `void rhsMonomer(*m, *deriv);`

Arguments `Monomer *m`
`double *deriv` stores the right-hand side of Hamilton's equations of motion with respect to coordinates and momenta that correspond to the passed-in monomer

Description In addition to differentiating the kinetic energy, the derivatives of potential energy, which are taken from `Monomer.dVdq`, are also added to compute the right-hand side. When `apply_requantization` flag is set, then the momenta in `qp` are rescaled so that angular momentum is brought to the closest half-integer. Implemented for cases of LINEAR_MOLECULE (tested), LINEAR_MOLECULE_REQUANTIZED_ROTATION (not tested enough) and ROTOR (tested).

Function `compute_numerical_rhs`

Call `Array compute_numerical_rhs(*ms, order);`
Arguments `MoleculeSystem *ms`
`size_t order` – the order of the central finite-difference formula (implemented for 2, 4 or 6)
Description Computes numerically the right-hand side of the Hamiltonian equations of motion. The order corresponds to the order of variables employed in MoleculeSystem struct.

3.4 Conducting angular momentum requantization

Function `j_monomer`

Call `double j_monomer(m);`
Arguments `Monomer m`
Description Computes the magnitude of angular momentum of passed-in monomer. Currently, implemented only for linear molecules.

$$[\text{linear molecule}] : j = \begin{bmatrix} -p_\theta \sin\varphi - p_\varphi \cos\varphi / \tan\theta \\ p_\theta \cos\varphi - p_\varphi \sin\varphi / \tan\theta \\ p_\varphi \end{bmatrix}$$

Function `torque_monomer`

Call `double torque_monomer(m);`
Arguments `Monomer m`
Description Computes the magnitude of torque of passed-in monomer. Needs the derivative of potential `m.dVdq` to be set within the monomer. Currently, implemented only for linear molecules.

$$[\text{linear molecule}] : \tau = \begin{bmatrix} \sin\varphi \dfrac{\mathrm{d}V}{\mathrm{d}\theta} + \cos\varphi / \tan\theta \dfrac{\mathrm{d}V}{\mathrm{d}\varphi} \\ -\cos\varphi \dfrac{\mathrm{d}V}{\mathrm{d}\theta} + \sin\varphi / \tan\theta \dfrac{\mathrm{d}V}{\mathrm{d}\varphi} \\ -\dfrac{\mathrm{d}V}{\mathrm{d}\varphi} \end{bmatrix}$$

3.5 Computing averages over phase-space

Function `analytic_full_partition_function_by_V`

Call `double analytic_full_partition_function_by_V(*ms, *params, Temperature)`
Arguments `MoleculeSystem *ms`
`double Temperature`

Description Returns analytically calculated value for the ratio of the partition function for the totality of states to the volume. Currently, implemented only for atom-atom, linear molecule-atom and rotor-linear molecule.

$$[\text{linear molecule - atom}] : Q = 4\pi \times (2\pi\mu kT)^{3/2} \times (2\pi IkT)$$

$$[\text{rotor - linear molecule}] : Q = 32\pi^3 \times (2\pi\mu kT)^{3/2} \times (2\pi kT)^{3/2} \sqrt{I_1^{\text{rotor}} I_2^{\text{rotor}} I_3^{\text{rotor}}} \times (2\pi I_1^{\text{lin}} kT)$$

Function `calculate_M0`

Call `void calculate_M0(*ms, *params, Temperature, *m, *q);`
Arguments `MoleculeSystem *ms`
`CalcParams *params`
`double Temperature`
`double *m` – the estimate of M_0
`double *q` – the error of the estimate
Description By sampling from $\rho \sim e^{-K/kT}$ and rejecting some of the points using the `reject` function, `params.initialM0_npoints` phase-points are produced to estimate M_0 and its error. The average over the sampled phase-points is multiplied by the `params.partial_partition_function_ratio`, which is supposed to be a ratio of the partition function over the part of the phase space that is pertinent to the select pair state to the total partition function and multiplied by volume. This ratio can be calculated using `mpi_perform_integration` function that invokes hep library.

Function `calculate_M2`

Call `void calculate_M0(*ms, *params, Temperature, *m, *q);`
Arguments `MoleculeSystem *ms`
`CalcParams *params`
`double Temperature`
`double *m` – the estimate of M_2
`double *q` – the error of the estimate
Description By sampling from $\rho \sim e^{-K/kT}$ and rejecting some of the points using the `reject` function, `params.initialM2_npoints` phase-points are produced to estimate M_2 and its error. The derivative of dipole with respect to coordinates is done using cental 2-point finite-difference formula. The energy is differentiated with respect to momenta using `compute_dHdp`. Here we use GSL's wrappers over BLAS to conduct matrix-by-vector multiplication. The average over the sampled phase-points is multiplied by the `params.partial_partition_function_ratio`, which is supposed to be a ratio of the partition function over the part of the phase space that is pertinent to the select pair state to the total partition function and multiplied by volume. This ratio can be calculated using `mpi_perform_integration` function that invokes hep library.

MPI Function `mpi_calculate_M0`

Call `void calculate_M0(*ms, *params, Temperature, *m, *q);`
Arguments `MoleculeSystem *ms`
`CalcParams *params`
`double Temperature`
`double *m`
`double *q`
Description The task of iterating `params.initialM0_npoints` points is split equally between processes of communicator. The behavior is the same as in `calculate_M0`.

MPI Function `mpi_calculate_M2`

Call `void calculate_M2(*ms, *params, Temperature, *m, *q);`

Arguments `MoleculeSystem *ms`
`CalcParams *params`
`double Temperature`
`double *m`
`double *q`

Description The task of iterating `params.initialM2_npoints` points is split equally between processes of communicator. The behavior is the same as in `calculate_M2`.

Function `compute_dHdp`

Call `void compute_dHdp(*ms, *dHdp);`

Arguments `MoleculeSystem *ms`
`gsl_matrix *dHdp`

Description Calls to `rhsMonomer` to fill in derivatives of Hamiltonian with respect to momenta pertaining to monomers.

Function `transform_variables`

Call `void transform_variables(x, *transformed, *Jac);`

Arguments `hep::mc_point<double> const& x`
`double *transformed`
`double *Jac`

Description Transforms a point x from n-dimensional hypercube $[0,1]^n$ to a point in phase-space of the `MoleculeSystem` (which is passed in using global pointer). The following transformations are implemented

$$R \leftarrow 1/x,$$
$$\theta \leftarrow \pi x,$$
$$\phi \leftarrow 2\pi x,$$
$$p \leftarrow \tan(\pi(x - 1/2)).$$

The jacobian of the transformation is accumulated along the steps. Implemented for linear molecule-atom and rotor-linear molecule pairs.

Function `integrand_pf`

Call `double integrand_pf(x);`

Arguments `hep::mc_point<double> const& x`

Description Based on the point in hypercube returns the value of the following integrand:

$$\text{jac} \cdot \exp\left(-\frac{E}{kT}\right).$$

(The energy is computed for `MoleculeSystem` that is passed in using global pointer.)

Function `integrand_M0`

Call `double integrand_M0(x);`

Arguments `hep::mc_point<double> const& x`

Description Based on the point in hypercube returns the value of the following integrand:

$$\text{jac} \cdot \mu^2 \cdot \exp\left(-\frac{E}{kT}\right).$$

(The energy is computed for `MoleculeSystem` that is passed in using global pointer. The pointer to dipole function is also assumed to be set globally.)

Function `integrand_M2`

Call	`double integrand_M2(x);`
Arguments	`hep::mc_point<double> const& x`
Description	Based on the point in hypercube returns the value of the following integrand:

$$\text{jac} \cdot \Big[\mu, H\Big] \cdot \exp\left(-\frac{E}{kT}\right).$$

The derivative of dipole with respect to coordinates is computed using 2-point central finite-difference formula. (The energy is computed for `MoleculeSystem` that is passed in using global pointer. The pointer to dipole function is also assumed to be set globally.)

Function `mpi_perform_integration`

Call	`double mpi_perform_integration(*ms, integrand, *params, Temperature, niterations, npoints, *m, *q);`

Arguments	`MoleculeSystem *ms`
	`Integrand integrand` – `typedef double (*Integrand)(hep::mc_point<double> const&)`
	`CalcParams *params`
	`double Temperature`
	`size_t niterations`
	`size_t npoints`
	`double *m`
	`double *q`

Description	Evaluates the integral of requested `integrand`

3.6 Performing averaging over trajectory ensembles

`struct CalcParams`

Fields

```
PairState ps
/* sampling */
double sampler_Rmin // a.u.
double sampler_Rmax // a.u.
double pesmin // Hartree
/* initial spectral moments check */
size_t initialM0_npoints
size_t initialM2_npoints
double partial_partition_function_ratio
/* requantization */
size_t torque_cache_len
double torque_bound
/* trajectory */
double sampling_time
size_t MaxTrajectoryLength
double cvode_tolerance
/* applicable to both correlation function AND spectral function calculations */
size_t niterations
size_t total_trajectories
/*correlation function calculation ONLY */
const char *cf_filename
double Rcut // distance at which the trajectory is forcefully stopped, a.u.
/* pr/mu calculation ONLY */
const char *sf_filename
double ApproximateFrequencyMax // cm-1
double R0 // initial distance, a.u.
```

```
/*correlation function array ONLY */
double* partial_partition_function_ratios
double *satellite_temperatures
size_t num_satellite_temperatures
const char **cf_filenames
```

`struct CFnc`

Fields
`double *t`
`double *data`
`size_t len` – number of samples in `*t` and `*data`
`size_t capacity` – capacity of `*t` and `*data`
`size_t ntraj` – number of trajectories used for averaging
`double Temperature`
`bool normalized` – flag that indicates whether the data samples are normalized by # of trajectories

`struct CFncArray`

Fields
`double *t`
`double **data`
`size_t ntemp`
`size_t len` – number of samples in `*t` and elements of `*data`
`double* nstar` – number of effective trajectories at each temperature
`size_t ntraj` - number of calculated trajectories at base temperature

Description
An array of correlation functions for a fixed `base temperature` (sampling temperature) and a `satellite temperature` (target temperature for re-weighting).

3.6.1 Calculating correlation functions

Function `correlation_eval`

Call
`int correlation_eval(*ms, *traj, *params, *crln, *tps);`

Arguments
`MoleculeSystem *ms`
`Trajectory *traj`
`CalcParams *params`
`double *crln`
`int *tps`

Description
Evaluates the mean between forward and backward correlation functions from the initial condition set in the `*ms`. It also tracks the number of turning points and returns it using the output parameter `*tps`.

MPI Function `calculate_correlation_and_save`

Call
`CFnc calculate_correlation_and_save(*ms, *params, Temperature);`

Arguments
`MoleculeSystem *ms`
`CalcParams *params`
`double Temperature`

Description First, estimates of the zeroth and second specral moments are obtained using `params.initialM0_npoints` and `params.initialM2_npoints` points, respectively. The accumulation of `params.total_trajectories` individual correlation functions is divided into `params.niterations` iterations. The individual correlation functions are obtained using `correlation_eval` function. The current aggregate estimate of the correlation function is saved to `params.cf_filename` at the end of each iteration. The zeroth moment based on the current estimate of the correlation function is made and compared to the value obtained during static phase-space sampling. NEED to correct the calculation of the estimate of the second moment based on the current correlation function. If `FREE_AND_METASTABLE` pair state is requested, then we SHOULD divide the contributions depending on the number of turning points provided by `correlation_eval`. The tracking of the number of turning points is already implemented. The communication between processes is realized using `MPI_Allreduce` function.

MPI Function `calculate_correlation_array_and_save`

Call `CFncArray calculate_correlation_array_and_save(*ms, *params, Temperature);`
Arguments `MoleculeSystem *ms`
`CalcParams *params`
`double base_temperature`
Description

Function `save_correlation_function`

Call `void save_correlation_function(*fp, cf, *params);`
Arguments `FILE *fp`
`CFnc cf`
`CalcParams *params`
Description The correlation function values `cf.data` assumed to be unnormalized by the number of trajectories `cf.ntraj`. The file stream `fp` is truncated using `ftruncate` before writing into it.

Function `invert_momenta`

Call `void invert_momenta(*ms);`
Arguments `MoleculeSystem *ms`
Description Inverts the momenta stored inside MoleculeSystem.

3.6.2 Calculating spectral functions

`struct SFnc`

Fields `double *nu`
`double *data`
`size_t len` – number of samples in `*nu` and `*data`
`size_t capacity` – capacity of `*nu` and `*data`
`size_t ntraj` – number of trajectories used for averaging
`double Temperature`

MPI Function `calculate_spectral_function_using_prmu_representation`

Call `SFnc calculate_spectral_function_using_prmu_representation_and_save(*ms, *params, Temperature)`
Arguments `MoleculeSystem *ms`
`CalcParams *params`
`double Temperature`

Description First, we check that `params.ApproximateFrequencyMax` is less than Nyquist frequency of the signal that will be sampled with requested `params.sampling_time`. Then, based on the calculated frequency step, a length of the frequency array is calculated. Since the frequency step depends on the `params.sampling_time` and `params.MaxTrajectoryLength`, the maximum frequency of the calculated spectral function will be somewhat close to the requested `params.ApproximateFrequencyMax` (as close as possible using the frequency step). The estimates of the zeroth and second spectral moments are obtained using `params.initialM0_npoints` and `params.initialM2_npoints`, respectively. The accumulation of `params.total_trajectories` is divided into `params.niterations`. For each trajectory the intermolecular distance is set to `params.R0`. The trajectory is cut at the same distance `params.R0`. Connes apodization is applied to time dependencies of Cartesian components of dipole throughout collisional trajectory before applying Fourier transform to them. The length of the dipole array which is equal to `params.MaxTrajectoryLength` needs to be a power of 2. The current estimate of the spectral function is saved to `params.sf_filename` at the end of each iteration. The zeroth and second spectral moments are obtained from the current estimate of the spectral function and compared to the values obtained through static phase-space sampling.

Function `save_spectral_function`

Call `void save_spectral_function(*fp, sf, *params);`
Arguments
```
FILE *fp
SFnc sf
CalcParams *params
```
Description The spectral function values `sf.data` assumed to be unnormalized by the number of trajectories `sf.ntraj`. The file stream `fp` is truncated using `ftruncate` before writing into it.

Function `connes_apodization`

Call `void connes_apodization(a, sampling_time);`
Arguments
```
Array a
double sampling_time
```
Description Multiplies the provided array by Connes apodization with time-normalization factor (a) set to `sampling_time`: $A(t) = (1 - t^2/a^2)^2$. See link.

3.7 Processing results

Function `read_correlation_function`

Call `bool read_correlation_function(*filename, *sb, *cf);`
Arguments
```
const char *filename
String_Builder *sb
CFnc *cf
```
Description Reads the correlation function from the file `filename` expecting a header containing metainformation (the lines should begin with `'#'`) and data presented in the two-column format. The header is read into `String_Builder` without parsing. The numerical values are parsed using `fscanf` function and checked to be non-NaN. Returns `false` when either the file could no be correctly opened to a parser error was encountered.

Function `read_spectral_function`

Call `bool read_spectral_function(*filename, *sb, *sf);`
Arguments
```
const char *filename
String_Builder *sb
SFnc *sf
```

Description Reads the spectral function from the file `filename` expecting a header containing metainformation (the lines should begin with `'#'`) and data presented in the two-column format. The header is read into `String_Builder` without parsing. The numerical values are parsed using `fscanf` function and checked to be non-NaN. Returns `false` when either the file could no be correctly opened to a parser error was encountered.

Function `writetxt`

Call `bool writetxt(*filename, *x, *y, len, *header);`
Arguments `const char *filename`
`double *x`
`double *y`
`size_t len`
`const char *header`
Description

Function `integrate_composite_simpson`

Call `double integrate_composite_simpson(*x, *y, len);`
Arguments `double *x`
`double *y`
`size_t len`
Description Performs numerical integration using composite Simpson's 3/8 rule. See link.

Function `compute_M0_from_sf`

Call `double compute_M0_from_sf(sf);`
Arguments `SFnc sf`
Description Computes the zeroth moment of the spectral function using `integrate_composite_simpson`. The dimensions of the frequency and spectral function values are expected to be cm^{-1} and $\mathrm{J}\cdot\mathrm{m}^6\cdot\mathrm{s}^{-1}$, respectively. The zeroth moment is returned in units of $\mathrm{cm}^{-1}\cdot\mathrm{Amagat}^{-2}$.

Function `compute_M2_from_sf`

Call `double compute_M2_from_sf(sf);`
Arguments `SFnc sf`
Description Computes the second moment of the spectral function using `integrate_composite_simpson`. The dimensions of the frequency and spectral function values are expected to be cm^{-1} and $\mathrm{J}\cdot\mathrm{m}^6\cdot\mathrm{s}^{-1}$, respectively. The zeroth moment is returned in units of $\mathrm{cm}^{-3}\cdot\mathrm{Amagat}^{-2}$.

`struct WingParams`

Fields `double A`
`double B`
`double C`
Description Stores the parameters for the Lorentzian function shifted vertically by constant value:

$$y = C + A/(1 + B^2x^2).$$

`struct WingData`

Fields `size_t n`
`double *t`
`double *y`

Description

Function `wingmodel`

Call `double wingmodel(*wp, t);`
Arguments `WingParams *wp`
`double t`
Description

Graphical interface is needed to visualize this step...

Function `fit_baseline`

Call `WingParams fit_baseline(*cf, EXT_RANGE_MIN);`
Arguments `CFnc *cf`
`size_t EXT_RANGE_MIN`
Description

Function `dct`

Call `double* dct(*v, len);`
Arguments `double *v`
`size_t len`
Description Implements the Fast Discrete Cosine Transform using the trick of Makhoul (2N + half-sample shift) and FFT, also known as DCT-II. For details, see original paper. A new vector with result is allocated and returned. Note that the output vector is not scaled by any time/ frequency units, making the implementation unit-agnostic.

Function `idct`

Call `double* idct(*v, len);`
Arguments `double *v`
`size_t len`
Description Implements the Inverse Fast Discrete Cosine Transform using the trick of Makhoul (2N + half-sample shift) and IFFT. For details, see original paper. A new vector with result is allocated and returned. Note that the output vector is not scaled by any time/ frequency units, making the implementation unit-agnostic.

Function `dct_sf_to_cf(SFnc sf)`

Call `CFnc dct_sf_to_cf(cf);`
Arguments `SFnc sf`
Description

Function `dct_cf_to_sf(CFnc cf)`

Call `CFnc dct_cf_to_sf(cf);`
Arguments `CFnc cf`
Description

Function `dct_numeric_sf`

Call `SFnc dct_numeric_sf(cf, *wp);`
Arguments `CFnc cf`
`WingParams *wp`
Description

Function `desymmetrize_d2`

Call `SFnc desymmetrize_d2(sf);`
Arguments `SFnc sf`
Description needs at least visual check

Function `desymmetrize_sch`

Call `SFnc desymmetrize_sch(sf);`
Arguments `SFnc sf`
Description

Function `desymmetrize_egf`

Call `SFnc desymmetrize_egf(sf);`
Arguments `SFnc sf`
Description

Function `desymmetrize_frm`

Call `SFnc desymmetrize_frm(sf);`
Arguments `SFnc sf`
Description

`struct Spectrum`

Fields `double *nu`
`double *data`
`size_t len` – number of samples in `*nu` and `*data`
`size_t capacity` – capacity of `*nu` and `*data`

Function `compute_alpha`

Call `Spectrum compute_alpha(sf);`
Arguments `SFnc sf`
Description

3.8 LOESS: locally weighted polynomial regression

LOESS blends the simplicity of linear least squares regression with the adaptability of nonlinear regression. It achieves this by fitting simple model to localized subset of the data, gradually constructing a function that captures the deterministic pattern of the variation in the data – effectively filtering out the random component that follows some probability distribution.

Degree of local polynomials. The local polynomials fitted to each subset of the data are typically of either first or second degree. Employing a zero-degree polynomial reduces LOESS to a weighted moving average. While higher-degree polynomials could theoretically be used, they are not aligned with the spirit of LOESS. Such polynomials are prone to overfitting within each subset and thus often lead to numerical instability.

Weight function. The weight function, gives the most weight to the data points nearest the point of estimation and the least weight to the data points that are furthest away. The use of the weights is based on the idea that points near each other are more likely to be related to each other in a simple way than points that are further apart. The traditional weight function used for LOESS is the tricube weight function: $w(x) = (1 - |x|^3)^3$ for $|x| < 1$ and 0 otherwise. The main criteria for the weight function are the following (Cleveland, 1979):

- $w(x) > 0$ for $|x| < 1$ since negative weights do not make sense
- $w(-x) = w(x)$: there is no reason to treat points to the left of x differently from those to the right

- $w(x)$ is a nonincreasing function for $x \geqslant 0$: it seems unreasonable to allow a point that is closer to x to have less weight than the one that is further away
- $w(x) = 0$ for $|x| \geqslant 1$

In addition it seems desirable that $w(x)$ decrease smoothly to 0 as x goes from 0 to 1. Such a weight function is more likely to produce a smoothed result. The tricube has been chosen since it enhances a χ^2-distributional approximation of an estimate of the error variance. So it should provide an adequate smooth in many situations. The weight for a specific point in any localized subset of data is obtained by evaluating the weight function at the distance between that point and the point of estimation, after scaling the distance so that the maximum absolute distance over all of the points in the subset of data is exactly one.

Let us consider the case of local second-degree polynomials:

$$\hat{y} = \beta_0 + \beta_1 x + \beta_2 x^2$$

In this model, β_0, β_1, β_2 are coefficients needed to be estimated using the data pairs (x_i, y_i) within a specified window. The predicted value $\hat{y}$ is derived from the this local polynomial model.

The matrix X is built using the x-values within the window. Each row of X corresponds to a separate x-value, while the columns represent the constant term (β_0), the linear term (β_1), and the quadratic term (β_2).

$$X = \begin{bmatrix} 1 & x_1 & x_1^2 \\ 1 & x_2 & x_2^2 \\ \dots & \dots & \dots \\ 1 & x_n & x_n^2 \end{bmatrix}$$

In weighted regression, each observation is assigned a weight w_i. These weights are organized into a digonal weight matrix W, defined as:

$$W = \begin{bmatrix} w_1 & 0 & 0 & \dots \\ 0 & w_2 & 0 & \dots \\ \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & w_n \end{bmatrix}$$

The normal equations for weighted regression problem are expressed as:

$$(X^T W X)\beta = X^T W y$$

To solve these equations we explore the following methods:

- complete orthogonal decomposition of $X^T W X$ to compute its pseudo-inverse
- QR decomposition of $X^T W X$ (no pivoting) to compute its pseudo-inverse
- robust Cholesky decomposition-based solver

For smoothing CH_4−CO_2 spectra we hit windows consisting 10,000 points in the tail.

enum `WEIGHT_FUNC`

Values	`WEIGHT_TRICUBE` – defined as $(1 - \|x\|^3)^3$ for $\|x\| < 1$ `WEIGHT_BISQUARE` – defined as $(1 - \|x\|^2)^2$ for $\|x\| < 1$
Description	Enumeration defines the types of weight functions available for assigning weights to data points. The choice of weight function determines how influence is assigned to observations based on their proximity to the central point at which smoothing is performed.

enum `LS_METHOD`

Values	`LS_COMPLETE_ORTHOGONAL_DECOMPOSITION` – uses complete orthogonal decomposition to compute the pseudo-inverse of the matrix $(X^T W X)$ `LS_QR_NO_PIVOTING` – fastest out of variants of QR decompositions, but maybe unstable if the matrix is not rull rank

`LS_CHOLESKY_SOLVER` – is usually the fastest. However, if the matrix is even mildly ill-conditioned, this is not a good method. It loses roughly twice as many digits of accuracy using the normal equation, compared to the more stable methods mentioned above.

Description Enumeration of numerical methods for solving linear least squares problem: $(X^T W X)beta = X^T W Y$

Function `loess_init`

Call `void loess_init(*x, *y, len);`
Arguments `double *x`
`double *y`
`size_t len`
Description This function prepares the input data for LOESS by storing the predictor 'x' and response 'y' values, and setting up the necessary values for subsequent LOESS calculations. The function assumes that the input arrays 'x' and 'y' are of equal length and contain valid numerical data.

`struct Smoothing_Config`

Fields `size_t degree` – degree of local polynomial [recommended: 2-3]
`size_t ws_min` – minimum window size
`size_t ws_step` – window size step
`size_t ws_delay` – optional: the index at which the window is starting to increase
`size_t ws_cap` – optional: cap on window size

Function `loess_estimate`

Call `double loess_estimate(x, window_size, degree);`
Arguments `double x`
`size_t window_size`
`size_t degree`
Description This function computes a local polynomial fit of provided degree within a specified window around the predictor value 'x' and uses it to estimate the response value 'y'.

Function `loess_create_grid`

Call `double *loess_create_grid(xmin, xmax, npoints);`
Arguments `double xmin`
`double xmax`
`size_t npoints`
Description Generates a uniformly spaced grid of predictor values within a specified range. This function creates a grid of equally spaced 'grid_npoints' points between 'grid_xmin' and 'grid_xmax' (inclusive). The function checks the validity of the provided grid bounds with respect to the range of the previously provided input data for smoothing.

Function `loess_apply_smoothing`

Call `double* loess_apply_smoothing(*config);`
Arguments `Smoothing_Config *config`
Description This function performs LOESS smoothing on the input data by fitting local polynomials to subsets of the data and combining the results to produce a smoothed curve. The behavior of the smoothing process is controlled by the `Smoothing_Config` struct, which specifies parameters such as the polynomial degree, window size, and parameters of window extension

Function `loess_free`

Call `void loess_free();`

Description Releases the memory allocated for smoothing.

3.9 Functions to transform angles between frames of reference

Function `linear_molecule_atom_lab_to_mol`

Call `void linear_molecule_atom_lab_to_mol(*qlab, *qmol);`
Arguments `double *qlab`
`double *qmol`
Description Put here a description from dissertation

Function `linear_molecule_atom_Jacobi_mol_by_lab`

Call `void linear_molecule_atom_Jacobi_mol_by_lab(jac, *qlab, *qmol);`
Arguments `Eigen::Ref<Eigen::MatrixXd> jac`
`double *qlab`
`double *qmol`
Description Put here a description from dissertation

Function `CH4_linear_molecule_lab_to_kal`

Call `void CH4_linear_molecule_lab_to_kal(*qlab, *qkal);`
Arguments `double *qlab`
`double *qkal`
Description Put here a description from dissertation

Function `CH4_linear_molecule_Jacobi_kal_by_lab`

Call `void CH4_linear_molecule_Jacobi_kal_by_lab(jac, *qlab, *qkal);`
Arguments `Eigen::Ref<Eigen::MatrixXd> jac`
`double *qlab`
`double *qkal`
Description Put here a description from dissertation

3.10 Utility structs & functions

`struct Array`

Fields `double *data`
`size_t n`
Description Sized array in dynamic memory.

Function `create_array`

Call `Array create_array(n);`
Arguments `size_t n`
Description

Function `init_array`

Call `void init_array(n);`
Arguments `Array *a`
`double *data`
`size_t n`
Description

Function `free_array`

Call `void free_array(n);`
Arguments `Array *a`
Description

`struct String_Builder`

Fields `char *items` – dynamically allocated buffer holding the string data
`size_t count` – the current number of characters in the buffer
`size_t capacity` – the total allocated capacity of the buffer in bytes
Description This struct represents a resizable buffer designed to build strings dynamically. It stores characters in a contiguous block of memory, allowing for manipulation of strings.

Function `sb_append`

Call `void sb_append(*sb, *line, n);`
Arguments `String_Builder *sb`
`const char *line`
`size_t n`
Description Appends a sequence of characters to the `String_Builder`. This function appends the first ‘n‘ characters from the provided ‘line‘ to the `String_Builder`. If ‘n‘ exceeds the length of ‘line‘, the behavior is undefined. The `String_Builder` automatically resizes its buffer if necessary to accommodate the new characters.

Function `sb_append_cstring`

Call `void sb_append(*sb, *line, n);`
Arguments `String_Builder *sb`
`const char *line`
Description This function appends a C-style (null-terminated) string `line` to the `String_Builder`'s buffer. If the `String_Builder` does not have sufficient capacity, its storage is automatically extended to accommodate the new content. If the `String_Builder`'s capacity is zero, it is first resized to `INIT_SB_CAPACITY` bytes before any extension occurs.

Function `sb_append_format`

Call `void sb_append_format(*sb, *format, ...);`
Arguments `String_Builder *sb`
`const char *format`
`varargs(...)`
Description This function takes a format string and a variable number of arguments, formats them according to the specified format, and appends the resulting string to the provided `String_Builder`. If the `String_Builder` lacks sufficient capacity, its storage is automatically extended to accommodate the new content. If the `String_Builder`'s capacity is zero, it is first resized to `INIT_SB_CAPACITY` bytes before any extension occurs.

Function `sb_reset`

Call `void sb_reset(*sb);`
Arguments `String_Builder *sb`
Description This function effectively clears the content of the `String_Builder` by setting its length to zero. The underlying buffer is not deallocated, allowing it to be reused for subsequent operations.

Function `sb_free`

Call `void sb_free(*sb);`
Arguments `String_Builder *sb`

Description This function releases the memory held by the internal buffer of the `String_Builder` and resets the fields.

Supplied routines:

1. spherical decomposition for *ab initio* PES for CO_2-Ar (Kalugina/Lokshtanov)
2. spherical decomposition for *ab initio* IDS for CO_2-Ar (Kalugina/Lokshtanov)
3. spherical decomposition for *ab initio* PES for CH_4-CO_2 (Finenko)
4. spherical decomposition for *ab initio* IDS for CH_4-CO_2 (Finenko)
5. spherical decomposition for *ab initio* PES for H_2-Ar (LeRoy/Chistikov)
6. spherical decomposition for *ab initio* PES for $CO-Ar$ (Pederson)
7. spherical decomposition for *ab initio* IDS for $CO-Ar$ (Rizzo)

Routines to be added in the future:

1. spherical decomposition for full-dimensional *ab initio* PES for N_2-Ar (Finenko)
2. PIP-NN representation for *ab initio* PES surface for N_2-Ar (Finenko)
3. PIP-NN representation for *ab initio* IDS surface for N_2-Ar (Finenko)
4. spherical decomposition for long-range IDS for N_2-Ar (Wang)
5. spherical decomposition for long-range $\mathrm{d}\mu/\mathrm{d}r$ surface for N_2-Ar (Wang)
6. spherical decomposition for long-range IDS for H_2-Ar (Kalugina)
7. spherical decomposition for *ab initio* induced dipole for H_2-Ar (Meyer)
8. PIP-NN representation for *ab initio* IDS for H_2-Ar (Meyer/Finenko)
9. spherical decomposition for *ab initio* PES for CO_2-CO_2 (Kalugina/Lokshtanov)
10. spherical decomposition for *ab initio* IDS for CO_2-CO_2 (Kalugina/Lokshtanov)
11. spherical decomposition for *ab initio* PES for N_2-N_2 (Karman/Chistikov)
12. spherical decomposition for *ab initio* IDS for N_2-N_2 (Karman/Chistikov)
13. spherical decomposition for *ab initio* PES for N_2-H_2 (Kalugina)
14. spherical decomposition for long-range IDS for N_2-H_2 (Kalugina)
15. spherical decomposition for *ab initio* PES for CH_4-N_2 (Finenko)
16. spherical decomposition for *ab initio* IDS for CH_4-N_2 (Finenko)
17. PIP-NN representation for full-dimensional *ab initio* PES for CH_4-N_2 (Finenko)

4 A skeleton of the user's program

For now, let us assume that the user is supposed to use `Hawaii Hybrid` as a library, not via the configuration file to a driver program (which could be arranged in the future). Then the following structure is expected:

1. **Initialize parallel environment**
   Call `MPI_Init` to initialize MPI if desired.
2. **Initialize** `MoleculeSystem`
   Call `init_ms` specifying types of monomers, their tensors of inertia, the reduced mass of the molecule pair and the generator seed.
3. ...

5 Changelog

24.12.2024 `rhsMonomer`: accepts pointer so that monomer's `qp` can be changed if `apply_requantization` flag is set.

06.01.2025 `Makefile`: switched to `Makefile` from build script.

08.01.2025 added MPI library which is hidden by the guard macro, implemented `mpi_calculate_M0`. Now the MPI and non-MPI versions of `hawaii.c` are compiled into two object files.

09.01.2025 implemented `calculate_correlation_and_save`.

09.01.2025 Porting some of `hep`-functionality in `hep_hawaii.cpp`. Zeroth moment can be calculated using adaptive Monte Carlo integration over phase space.

10.01.2025 Tracking the number of turning points is added to `correlation_eval`.

10.01.2025 PES and its derivatives for CH_4-CO_2 are adapted.

11.01.2025 fixing error inverting momenta in `correlation_eval`. Calculating correlation function for CO_2-Ar seems to be working correctly. Need to run a calculation with larger number of trajectories.

12.01.2025 added higher-order finite-difference formulas for differentiating energy: `compute_numerical_rhs`

12.01.2025 trajectory for CH_4-CO_2 using kinetic energy only works correctly. Angles transformation and its jacobian needs to be adapted from FUNCHAL to account for different order of coordinates.

15.01.2025 Run calculation of correlation function for CO_2-Ar for 10.000.000 trajectories. The correlation function is in close agreement with the previous results. Now moving on to adapting code for processing correlation functions in `hawaii`.

17.01.2025 Adapted functions related to processing correlation function from FUNCHAL. Code implementing `loess` algorithm is left as-is because of the use of Eigen3.

19.01.2025 Added checks for NaN values of dipole in `correlation_eval` to avoid the corruption of the correlation function estimate. At least one corrupted value of correlation function for CO_2-Ar has occurred during the calculation of 10.000.000 trajectories. The reason for the occurence of the NaN is unknown. Should investigate when initial condition leading to NaN values is found. Could this NaN be the consequence of overflow in `generate_normal`?

19.01.2025 Implemented `calculate_spectral_function_using_prmu_representation_and_save` function and tested that it produces correct result for CO_2-Ar at 300 K.

20.01.2025 Enable `switch-enum` option for compiler to invoke warning in the switch-cases where one of the cases of the enum is not explicitly handled even though default case is present.

20.01.2025 Caught a possible (but really rare) overflow error in the implementation of Box-Mueller algorithm for sampling normally distributed variable: `generate_normal`.

20.01.2025 Estimate M_0 based on the spectral function obtained during each iteration of the pr-mu calculation

20.01.2025 Estimate M_2 using rejection sampling and using `hep`. Works for CO_2-Ar

22.01.2025 Trying out calculating correlation function for $CO-Ar$. Why is the error in M_2 is 13% for 20.000.000 points? $M_0 = 1.847 \cdot 10^{-4}$, $M_2 = 4.148$. There was error in passing the arguments to dipole function (fixed). The correlation function "breaks" at some point during the calculation. Extremely large values (1e194) occur at approximately the same time intervals in the correlation function rendering it useless. Maybe it happens because of the PES, not because of IDS?

25.01.2025 jacobian is working for CH_4-CO_2. `hawaii` is extended with function to compute numerical jacobian; analytical and numerical jacobians are in close agreement.

25.01.2025 fixed a bug in `extract_q_and_write_into_ms` where coordinates for second monomer actually were overwriting the coordinates for the first monomer

26.01.2025 adapted dipole function for CH_4-CO_2

26.01.2025 `examples/mpi_phase_space_integration_ch4_co2` produces M_0 and M_2 spectral moments at 300 K which are consistent with previous estimations: $M_0 = 8.29 \cdot 10^{-4}$, $M_2 = 5.37$.

26.01.2025 differentiate between debug/release build in Makefile: apply separate compilation flags

27.01.2025 `examples/correlation_ch4_co2` produces what seems to be a correct spectrum, the spectral moments are in agreement with their phase-space counterparts.

27.01.2025 bug fix: desymmetrization procedure didn't propagate temperature to the output structure

31.01.2025 Implementing `calculate_correlation_array_and_save` that employs individual trajectory reweighting to produce results at satellite temperatures: testing on CO_2-Ar in `examples/correlation_array_co2_ar`

01.02.2025 During the correlation and pr-mu calculations for CO_2-Ar on cluster, the file with temporary result is not written (at least for several iterations at the end of which the file is supposed to be written). Turns out that if stream is flushed using `fflush` and filesystem caches for a given file descriptor are forced to be committed to disk using `syncfs` the problem is resolved.

01.02.2025 `calculate_correlation_array_and_save` produces spectral profiles for CO_2-Ar (unbound states) that are in close agreement with the results obtained in 2021. Ensemble of approximately 8 million trajectories was used. Need to check for bound states as well.

06.02.2025 CO-Ar 300K: $M_0 = 1.850 \cdot 10^{-4}$, $M_2 = 1.884$.

08.02.2025 the factor by ALU^3 is moved from `save_correlation_function` to `correlation_eval`. correctness: check that after moving the factor ALU^3 here, the produced correlation functions are correct for both single-correlation function and correlation-array calculations

09.02.2025 added `normalized` flag to CFnc, changed reading/saving correlation function from/to file (not documented)

09.02.2025 added parsing Temperature and ntraj from header of the file using regex and added averaging of correlation functions

09.02.2025 rewrite of LOESS code to better understand what's going on there: prepare for OpenMP parallelization, check out different approaches to solve the weighted linear squares problem

09.02.2025 loess: added OpenMP parallelization of the main loop

11.02.2025 loess: refactored creating window & added stubs for running outside OpenMP environment

12.02.2025 added checking for energy conservation inside trajectory. maybe need to check this info in `correlation_eval` and discard the outlying trajectories?

12.02.2025 implemented `average_correlation_functions` using vararg

12.03.2025 calculated correlation functions for He-Ar at 50K and 300K using 500mln and 300mln trajectories, respectively. At 300K we achieved a decent looking spectrum, however at 50K it still looks not converged at large frequencies. Moreover, the correlation function at 50K exhibits pulsing (beating) oscillations which result in some spectral feature at 3.5 cm-1. What is it? Is it reproducible in quantum calculation?

16.03.2025 `hawaii`: use appropriate partial partition function for each temperature when calculating M_0 before proceeding to propagate trajectories

18.03.2025 Caught this mysterious error during bound states calculation for CH_4-CO_2: `[n01p012:2488401:0:2488401] Caught signal 7 (Bus error: nonexistent physical address)`. No stack trace, location of error unknown.

23.03.2025 Added functions to perform D4 and D4a desymmetrization

6 Todo's

- `mpi_calculate_M0`/`mpi_calculate_M2`: the temporary result is not accumulated over communicator, it is printed only for values accumulated only for zeroth process.
- think about adding more information to output file, at least need to add timestamps
- get rid of OpenMP magic in LOESS and instead use explicitly `pthread` library
- Allow providing the value (values) of zeroth and second spectral moment to `calculate_correlation_and_save` and `calculate_correlation_array_and_save`. For CH_4-CO_2 the calculation of second moments is time-consuming, so the values should probably be cached somehow. Honestly, calculating zeroth moments for 10+ temperatures is also no joke. It should be possible to cache these values as well.
- CO_2-CO_2 in 200-300K range (communicate with Wishnow about the possibility to conduct an experiment)
- How to go about computing cross-term? We need to allow to have 2 dipole functions
- How to organize and store the calculated results for examples? The results include CFs, SFs, spectra, spectral moments and partial partition functions.
- We should store the results of individual iterations of `calculate_correlation_and_save` and `calculate_correlation_a` Maybe use SQL database for this?
- Arena Allocator for storing some temporary strings and small arrays. At the moment, there are waaaay too many mallocs/frees.
- Saving the contributions of free and metastable states in correlation function separately. How can we pass in the information about the desired filenames? How to organize the storage for these contributions?
- The information about the number of turning points could be saved into gsl_histogram, and then partially displayed in the output file and saved to file system
- We are using MPI_Allreduce mechanism for broadcasting. This should be changed to direct MPI_Send / MPI_Recv calls.
- Can the `hawaii.h` file be used without constructing the MoleculeSystem? I moved code for processing correlation function there, but it can't be used without creating stubs for PES/dPES functions. Should `hawaii` create stubs when certain macro is in place?
- Maybe we need a check that processes in the communicator get different seeds each?
- In case when correlation function is calculated for bound states, we can separate the contributions for bound-bound and bound-free transitions (Fakhardji trick)
- In case when correlation function is calculated for bound states, we should use the Zimmermann trick (use any point of trajectory as a 'starting point' of the correlation function). This should be the default approach for this correlation.
- Adapt transforming angles between frames of reference for linear molecule-linear molecule
- How to organize Makefiles for use in different environments (several clusters)? If we push the changes to remote git repo on the remote machine, the local changes get overriden, thus removing the changes made to local Makefile. Annoying...
- I would like to make a graphical shell in which it will be possible to set parameters for calculation (for example, in the form of a list with available options for each parameter). The program is given paths to files (or drag-and-dropped) with potential and dipole functions and it assembles an executable file. Here I make an assumption that it is better to have the potential to be statically compiled with the rest of the code rather than dynamically loading it from library. It would probably be nice to be able to run the trajectory program from graphical shell locally and display the calculation result. If the correlation function is calculated, then there is no need to add its processing within the same "stage". Let the

correlation function be calculated separately, then the result should be loaded and converted into a spectrum. In the case of calculating the spectral function in the pr-mu representation, smoothing is also required, it is also carried out in a separate "stage". This would enable us to have a starting point for a graphical shell that can demonstrate some of the features of the library without overcomplicating from the very beginning. Later on, we could think about establishing TCP connection between shell and the main program running remotely enabling the user to setup the calculation with the desired parameters and monitor the calculation.

7 Notes

Usage of GNU profiler (gprof):

1. Compile the program with the `-pg` flag: `g++ -pg -o test test.cpp`
2. Run the program: `./test`
3. Generate the profiling report: `gprof test gmon.out > analysis.txt`