Python Framework for Higher-Order Perturbation Theory

Meret Preuß

March 2025

Contents

1 Introduction

This document shall cover the theoretical basics of time-independent (non-degenerate) perturbation theory as formulated by Tosio Kato [1], along with a guide of how to algorithmically calculate the perturbation series. Besides, this text accompanies a Python implementation of this algorithmical computation.

Perturbation theory is a method aimed at approximating solutions for systems that are slightly different from those that are already solved exactly. Therefore, the system at interest's Hamiltonian is expressed by some (known) Hamiltonian $\hat{H}_0$ and a perturbation $\hat{V}$:

$$\widehat{H} = \hat{H}_0 + \hat{V} \tag{1}$$

Furthermore, with $\lambda \in \mathbb{R} \cap [0, 1]$, $\widehat{H}$ is parametrized in a way that by increasing λ from 0 to 1, $\widehat{H}$ changes from describing the unperturbed and known system to the perturbed

one:

$$\widehat{H}(\lambda) = \hat{H}_0 + \lambda \hat{V} \tag{2}$$

Moreover, it is assumed that by increasing λ continuously, the eigenfunctions $|a\rangle$ of $\widehat{H}(0)$ with eigenvalues ε_a are continuously converted into the eigenfunctions $|A\rangle$ of $\widehat{H}(1)$ with eigenvalues E_a.
In summary, with (time-independent) perturbation theory, one aims to find the solutions to

$$\widehat{H}|A\rangle = E_a|A\rangle \tag{3}$$

by using the knowledge about

$$\widehat{H}_0|a\rangle = \varepsilon_a|a\rangle. \tag{4}$$

[2]

2 Rayleigh-Schrödinger Formalism

First, one writes the perturbed system's eigenfunctions $|A\rangle$ and their eigenvalues E_a as a power series in λ. The superscripts $^{(n)}$ refer to the order of correction, i. e. $E_a^{(1)}$ is the first order correction to the a-th eigenvalue.

$$|A\rangle = |a\rangle + \lambda|A^{(1)}\rangle + \lambda^2|A^{(2)}\rangle + ... \tag{5}$$

$$E_a = \varepsilon_a + \lambda E_a^{(1)} + \lambda^2 E_a^{(2)} + ... \tag{6}$$

In the following, all further calculations are done under the assumption that $\langle a|A\rangle = 1$ and $\langle a|A^{(n)}\rangle = 0 \ \forall n \in \{\mathbb{N} \setminus 0\}$ – implying that all corrections due to the perturbations are orthogonal to their respective unperturbed state. Inserting $|A\rangle$ and E_a into the time-independent Schrödinger equation (3) and taking the inner product with $|a\rangle$, one eventually obtains an expression for the k-th energy correction: [1]

$$E_a^{(n)} = \langle a|\widehat{V}|A^{(n-1)}\rangle \tag{7}$$

[2]

Hence, knowing the state correction $|A^{(n)}\rangle$ in a particular order n, one can directly determine the energy correction in the following order.

[1] In this section, detailed derivations are omitted for the sake of brevity. They can be found in standard textbooks on undergraduate quantum mechanics, such as [2], [3], and [4].

Next, the state corrections can be expanded in the eigenbasis of the unperturbed system:

$$|A^{(n)}\rangle = \sum_{m \neq a} |m\rangle\langle m|A^{(n)}\rangle \tag{8}$$

with the recursive expansion coefficients

$$\langle m|A^{(n)}\rangle = \frac{1}{\varepsilon_a - \varepsilon_m}\left[\langle m|\widehat{V}|A^{(a-1)}\rangle - \sum_{j=1}^{n-1} E_a^{(j)}\langle m|A^{(n-j)}\rangle\right] \tag{9}$$

[3]

Thus, with Rayleigh-Schrödinger perturbation theory, one can recursively calculate energy and state corrections from the unperturbed energy eigenstates. As long as no degenerate energy levels are considered, the so-called "energy denominators" remain unproblematic. In the case of degeneracies, the formalism of degenerate perturbation theory is needed. More information on this can be found in [4].
With equation (7), the first energy correction $E_a^{(1)}$ becomes:

$$E_a^{(1)} = \langle a|\widehat{V}|a\rangle \tag{10}$$

Thus, the first-order energy correction is the expectation value of the perturbation, taken with respect to the unperturbed state. For the first order state correction $|A^{(1)}\rangle$, one finds:

$$|A^{(1)}\rangle = \sum_{m \neq a} \frac{\langle m|\widehat{V}|a\rangle}{(\varepsilon_a - \varepsilon_m)}|m\rangle \tag{11}$$

[2]

Second order corrections for $|A\rangle$ and E_a are found to be [3]:

$$E_a^{(2)} = \sum_{m \neq a} \frac{|\langle m|\widehat{V}|a\rangle|^2}{\varepsilon_a - \varepsilon_m} \tag{12}$$

$$|A^{(2)}\rangle = \sum_{m \neq a}\sum_{l \neq a} |m\rangle \frac{\langle m|\widehat{V}|l\rangle\langle l|\hat{V}|a\rangle}{(\varepsilon_a - \varepsilon_m)(\varepsilon_a - \varepsilon_l)} - \sum_{m \neq a} |m\rangle \frac{\langle m|\widehat{V}|a\rangle\langle a|\widehat{V}|a\rangle}{(\varepsilon_a - \varepsilon_m)^2} \tag{13}$$

[3]

3 Kato Formalism

3.1 Resolvent

Let $\mathcal{H}$ be the Hilbert space over the complex numbers with a scalar product and associated norm and let $\widehat{H}$ be a closed operator on a domain D. The resolvent set then is defined as:

$$\rho(\widehat{H}) := \left\{ z \in \mathbb{C} | z \ \mathrm{id}_{\mathcal{H}} - \widehat{H} : D(\widehat{H}) \to \mathcal{H} \text{ bijective} \right\} \tag{14}$$

With that, one can define the resolvent (with $L(\mathcal{H})$ being the set of all continuous linear operators within $\mathcal{H}$) as :

$$\widehat{G} : \rho(\widehat{H}) \to L(\mathcal{H}), \ z \to \left(z \ \mathrm{id}_{\mathcal{H}} - \widehat{H} \right)^{-1} \tag{15}$$

Furthermore, the so-called point spectrum of $\widehat{H}$ is defined as:

$$\sigma_p(\widehat{H}) := \left\{ z \in \mathbb{C} | z \ \mathrm{id}_{\mathcal{H}} - \widehat{H} : D(\widehat{H}) \to \mathcal{H} \text{ not injective} \right\} \tag{16}$$

[5]

In the following, the mathematical notation will be simplified as follows: $z\mathrm{id}_{\mathcal{H}}$ shall be abbreviated by z. Besides, operator symbols $\widehat{\ }$ will be used implicitly on the respective operators, for instance, H shall now describe $\widehat{H}$. Given the definitions above, the resolvent can now be considered the operator inverse of $z - H$:

$$G(z) = (z - H)^{-1} \equiv \frac{1}{z - H} \tag{17}$$

From now on, z is a complex variable carrying the dimension of an energy while H describes a hermitian Hamiltonian operator with a partly discrete and partly continuous spectrum. $G(z)$ is analytic within the complex plane with singularities given by the spectrum of H. As H is hermitian, its eigenvalues and therefore also the singularities of $G(z)$ lie on the real axis. In this documentation, focus is solely laid on Hamiltonian operators with a purely discrete spectrum.
One can now consider the projections P_i onto the eigenspaces of H:

$$P_0 = |0\rangle\langle 0| \tag{18}$$

$$P_1 = |1\rangle\langle 1| \tag{19}$$

$$P_i = |i\rangle\langle i| \tag{20}$$

with $HP_i = E_i P_i$ and E_i being the i-th eigenvalue. Properties of these projectors include

mutual orthonormality (with idempotency):

$$P_i P_j = |i\rangle\langle i|j\rangle\langle j| = |i\rangle\langle j|\delta_{ij} = P_i\delta_{ij} \tag{21}$$

and completeness:

$$\sum_i P_i = 1 \ . \tag{22}$$

One can find an expression of $G(z)$ that solely and visibly depends on the projectors P_i by considering the Taylor expansion of $\frac{1}{a-x} = \sum_{n=0}^{\infty} a^{-(n+1)}x^n$ which converges for $\left|\frac{x}{a}\right| < 1$.

$$G(z) = \frac{1}{z-H} \tag{23}$$

$$G(z)P_i = \frac{1}{z-H}\ P_i = \sum_{n=0}^{\infty} \frac{1}{z^{n+1}} E_i^n P_i = \frac{1}{z-E_i} P_i \tag{24}$$

$$\implies \qquad G(z) = \sum_i \frac{P_i}{z-E_i} \tag{25}$$

As each discrete eigenvalue E_j of H is a simple pole of $G(z)$, each projector P_j is expressable through

$$P_j = \frac{1}{2\pi i} \oint_{\Gamma_j} G(z)\mathrm{d}z \tag{26}$$

Here, the residue theorem is used. More generally, for a curve Γ that encircles several eigenvalues E_j with $P_\Gamma = \sum_j P_j$, one can say that

$$P_\Gamma = \frac{1}{2\pi i} \oint_{\Gamma} G(z)\mathrm{d}z \tag{27}$$

Next, the application of H will yield a useful expression.

$$HP_\Gamma = \frac{1}{2\pi i} \oint_{\Gamma} HG(z)\mathrm{d}z \tag{28}$$

For this, some auxiliary calculations are applied:

$$HG(z) = HG(z) + zG(z) - zG(z) \tag{29}$$

$$= (H-z)G(z) + zG(z) \tag{30}$$

$$= -1 + zG(z) \tag{31}$$

Now, one can use that $\oint_\gamma (-1)\mathrm{d}z = 0 \quad \forall\gamma$, which reduces equation (28) to

$$HP_\Gamma = \frac{1}{2\pi i}\oint_\Gamma zG(z)\mathrm{d}z \tag{32}$$

[4]

3.2 Perturbational Approach to Resolvent

Next, one can apply a perturbational approach and split H into an easy diagonalizable part and a perturbation V, with analogous considerations as in Section 2.

$$H(\lambda) = H_0 + \lambda V \qquad H(0) = H_0,\ H(1) = H_0 + V \tag{33}$$

The full resolvent then reads

$$G(z) = \frac{1}{z - H_0 - \lambda V} \tag{34}$$

In order to express the results of the perturbational approach only using the unperturbed spectrum of H and the control parameter λ, again some auxiliary calculations are applied[2]:

$$\begin{aligned}
G(z) &= \frac{z - H_0 - \lambda V + \lambda V}{z - H_0} G(z) && (35)\\
&= \frac{1}{z - H_0}\left[(z - H_0 - \lambda V) + \lambda V\right]\frac{1}{z - H_0 - \lambda V} && (36)\\
&= \frac{z - H_0 - \lambda V}{(z - H_0)(z - H_0 - \lambda V)} + \frac{1}{z - H_0}\lambda V\frac{1}{z - H_0 - \lambda V} && (37)\\
&= \frac{1}{z - H_0} + \frac{1}{z - H_0}\lambda V\frac{1}{z - H_0 - \lambda V} && (38)\\
&= G_0 + G_0\lambda V G && (39)
\end{aligned}$$

This is called the *resolvent identity*. Reiteration then yields the following expression, which is assumed to converge.[3]

$$G = \sum_{n=0}^{\infty} \lambda^n G_0 \left(VG_0\right)^n \tag{40}$$

Let Γ_a now be a curve that encircles an unperturbed eigenvalue E_a^0 of H_0 and all the perturbed eigenvalues belonging to E_a^0, but no other eigenvalues of H_0 or H. Furthermore, let P now be the sum of projectors onto the perturbed eigenspaces belonging to E_a. Now,

[2] In the end, the dependence on z will be implicitly assumed. With this, $G(z)$ will simply be written as G.

[3] A more precise description of the convergence criteria for this series is given in [6, p.67], where it is called the *second Neumann series for the resolvent.*

the resolvent identity is inserted from equation (40) into equation (27) and – assuming sufficient convergence behavior – the order of integration and summation is switched. λ^n can be put outside of the integral. Kato calls this the *total projection*

$$P = \frac{1}{2\pi i} \sum_{n=0}^{\infty} \lambda^n \oint_{\Gamma_a} G_0 \left(VG_0\right)^n \mathrm{d}z \tag{41}$$

$$= \frac{1}{2\pi i} \oint_{\Gamma_a} G_0 \, \mathrm{d}z + \frac{1}{2\pi i} \sum_{n=1}^{\infty} \lambda^n \oint_{\Gamma_a} G_0 \left(VG_0\right)^n \mathrm{d}z \tag{42}$$

$$= P_0 + \sum_{n=1}^{\infty} \lambda^n C^{(n)} \tag{43}$$

Here, $C^{(n)}$ is an abbreviation for the circulation integral over the n-th summand of G:

$$C^{(n)} = \frac{1}{2\pi i} \oint_{\Gamma_a} G_0 \left(VG_0\right)^n \mathrm{d}z \tag{44}$$

In the following, ε will again be used to describe unperturbed eigenvalues, whilst the subscript labels the respective eigenvalue's index. The subscript a describes the index of the eigenvalue whose perturbed value shall be calculated. Following this, $|a\rangle$ shall describe the respective unperturbed state. Since G_0 has a simple pole at $z = \varepsilon_a$, $G_0 \left(VG_0\right)^n$ has a pole with order $n+1$ with $C^{(n)}$ as its residue. The next steps now aim at determining $C^{(n)}$. For this, one can develop G_0 into a Laurent-Series about ε_a and start with its decomposition into the individual projectors

$$G_0 = \sum_i \frac{P_i}{z - \varepsilon_i} \tag{45}$$

$$= \frac{P_0}{z - \varepsilon_a} + \sum_{i \neq a} \frac{P_i}{z - \varepsilon_a + \varepsilon_a - \varepsilon_i} \tag{46}$$

$$= \frac{P_0}{z - \varepsilon_a} + \sum_{i \neq a} \frac{P_i}{\left(\varepsilon_a - \varepsilon_i\right)\left(1 + \frac{z - \varepsilon_a}{\varepsilon_a - \varepsilon_i}\right)} \tag{47}$$

One can then use the Taylor series of $\frac{1}{1+x} = \sum_{n=0}^{\infty} (-1)^n x^n$ for $|x| < 1$.

$$G_0 = \frac{P_0}{z - \varepsilon_a} + \sum_{i \neq a} \frac{1}{\varepsilon_a - \varepsilon_i} \sum_{k=0}^{\infty} (-1)^k \left(\frac{z - \varepsilon_a}{\varepsilon_a - \varepsilon_i}\right)^k P_i \tag{48}$$

A usage of the commutativity and an index shift result in

$$G_0 = \frac{P_0}{z - \varepsilon_a} + \sum_{k=1} (-1)^{k-1} \left(z - \varepsilon_a\right)^{k-1} \sum_{i \neq a} \frac{P_i}{\left(\varepsilon_a - \varepsilon_i\right)^k} \ . \tag{49}$$

One can now define the following operators:

$$S^k = \begin{cases} -P_0 & \text{for } k = 0 \\ \sum_{i \neq a} \frac{P_i}{(\varepsilon_a - \varepsilon_i)^k} & k \geq 1 \end{cases} \tag{50}$$

Because of $(P_i)^k = P_i$, $S^k = (S^1)^k$. With this, equation (49) reads

$$G_0 = \sum_{k=0}^{\infty} (-1)^{k-1} (z - \varepsilon_a)^{k-1} S^k \tag{51}$$

Following equation (44), $C^{(n)}$ is the coefficient of $(z - \varepsilon_a)^{-1}$ in the Laurent series of $G_0 (VG_0)^n$. Thus, in the integrand in equation (44) , there are $n+1$ factors like equation (51):

$$\begin{aligned} G_0 (VG_0)^n = & \left(\sum_{k_1=0}^{\infty} (-1)^{k_1-1} (z - \varepsilon_a)^{k_1-1} S^{k_1} \right) \\ & \cdot V \cdot \left(\sum_{k_2=0}^{\infty} (-1)^{k_2-1} (z - \varepsilon_a)^{k_2-1} S^{k_2} \right) \cdot ... \\ & \cdot V \cdot \left(\sum_{k_{n+1}=0}^{\infty} (-1)^{k_{n+1}-1} (z - \varepsilon_a)^{k_{n+1}-1} S^{k_{n+1}} \right) \end{aligned} \tag{52}$$

For $C^{(n)}$ to be the residue of the pole $(z - \varepsilon_a)$, the exponents $k-1$ of the $n+1$ factors need to add up to -1. Condition $\sum_{i=1}^{n+1}(k_i - 1) = -1$ is equivalent to $\sum_{i=1}^{n+1} k_i = n$. The condition directly yields a resulting factor of -1. $C^{(n)}$ simplifies to:

$$C^{(n)} = - \sum_{\sum_{i=1}^{n+1} k_i = n} S^{k_1} V S^{k_2} V ... V S^{k_n+1} \tag{53}$$

In the first orders, the total projection then reads

$$\begin{aligned} P &= P_0 + \sum_{n=1}^{\infty} \lambda^n C^{(n)} \\ &= P_0 - \lambda \left(S^0 V S^1 + S^1 V S^0 \right) \\ &\quad - \lambda^2 \left(S^0 V S^1 V S^1 + S^1 V S^0 V S^1 + S^1 V S^1 V S^0 \right. \\ &\quad \left. + S^0 V S^0 V S^2 + S^0 V S^2 V S^0 + S^2 V S^0 V S^0 \right) \\ &\quad + \mathcal{O}\left(\lambda^3\right) \end{aligned} \tag{54}$$

With this, the projectors (and eigenvectors) can be calculated. In order to directly calculate the eigenvalues without explicitly determining the eigenvectors, again auxiliary

calculations are needed:

$$(z - H)G(z) = 1 \tag{55}$$

$$HG(z) = zG(z) - 1 \tag{56}$$

$$(H - \varepsilon_a)\, G(z) = (z - \varepsilon_a)\, G(z) - 1 \tag{57}$$

Keeping in mind that $\frac{1}{2\pi i} \oint_{\Gamma_a} G(z)\mathrm{d}z = P$, one obtains:

$$(H - \varepsilon_a)\, P = \frac{1}{2\pi i} \oint_{\Gamma_a} (z - \varepsilon_a)\, G(z)\mathrm{d}z. \tag{58}$$

Next, equation (40) is inserted, and – again assuming sufficient convergence behavior – the order of integration and summation is swapped. ε_a is a first-order pole of $G_0(z)$; $(z - \varepsilon_a)\, G_0(z)$ does not have singularities anymore. The term for $n = 0$ hence yields no contribution to the integral. With the same reasoning as for $A^{(n)}$, an expression for the residue of $(z - \varepsilon_a)$ in the Laurent series of $(z - \varepsilon_a)\, G_0\, (VG_0)^n$ is found:

$$B^{(n)} = \frac{1}{2\pi} \oint_{\Gamma_a} (z - \varepsilon_a)\, G_0\, (VG_0)^n\, \mathrm{d}z. \tag{59}$$

Rewriting equation (58) yields

$$\left(H - E_a^0\right) P = \sum_{n=1}^{\infty} \lambda^n B^{(n)}. \tag{60}$$

Again, the operators S^k are inserted into the residue. Because of the additional factor $(z - \varepsilon_a)$, the exponents k_i now need to add up to -2. The condition $\sum_{i=1}^{n+1}(k_i - 1) = -2$ is equivalent to $\sum_{i=1}^{n+1} k_i = n - 1$.

$$B^{(n)} = \sum_{\sum_{i=1}^{n+1} k_i = n-1} S^{k_1} V S^{k_2} V ... V S^{k_n+1} \tag{61}$$

[4] Now, traces are calculated for both sides of the equation $HP = E_a P$, with E_a being the eigenvalue of the full (perturbed) Hamiltonian H. As an idempotent matrix, the trace of a projection matrix P is equal to its rank [7] A projector belonging to a non-degenerate eigenvalue has rank 1 since its target space is one-dimensional.

$$\mathrm{Tr}(HP) = \mathrm{Tr}(E_a P) \tag{62}$$

$$= E_a \mathrm{Tr} P = E_a \cdot 1 = E_a \tag{63}$$

[4] From this point, all considerations are taken under the assumption that the respective unperturbed state $|a\rangle$ is non-degenerate.

On the other hand, $\mathrm{Tr}(HP)$ can be rewritten:

$$\mathrm{Tr}(HP) = \mathrm{Tr}\left((H-\varepsilon_a)\,P + \varepsilon_a P\right) \tag{64}$$

$$= \mathrm{Tr}\left(\sum_{n=1}^{\infty} \lambda^n B^{(n)}\right) + \varepsilon_a. \tag{65}$$

Therefore, the explicit representation of the perturbational series for E_a in all orders of λ (with $\varepsilon_a = E_a^{(0)}$) reads:

$$E_a = \varepsilon_a + \sum_{n=1}^{\infty} \lambda^n \ \mathrm{Tr} B^{(n)} \tag{66}$$

$$= \sum_{n=0}^{\infty} \lambda^n \ E_a^{(n)} \tag{67}$$

[4, 6]

3.3 Evaluation of the Contributions

The traces in equation (66) have to be evaluated in every order n. For this, it is useful to express each term in $B^{(n)}$ through the sequence of the exponents k_i:

$$S^{k_1} V S^{k_2} V ... V S^{k_{n_n}+1} \widehat{=} (k_1, ... k_{n_{n+1}}). \tag{68}$$

In the following, all general findings for this and instructions to obtain values for the energy correction from equation (67) are taken from [8] and [9].
It can be instructive to work through the examples of $n = 1, 2, 3$ to get a better understanding of the structure of these traces. Later, the general case will be considered.
With the condition $\sum_{i=1}^{n+1} k_i = n - 1$, for $n = 1$, the sum over $n + 1$ terms (exponents) needs to be zero. This only allows one tuple:

$$(0,0) \ \widehat{=}\ \mathrm{Tr}\left(S^0 V S^0\right) = \mathrm{Tr}\left(|a\rangle\langle a|V|a\rangle\langle a|\right) = \langle a|V|a\rangle \ \mathrm{Tr}\left(|a\rangle\langle a|\right) = \langle a|V|a\rangle \tag{69}$$

This matches the first-order energy correction $E_n^{(1)}$ in the Rayleigh-Schrödinger perturbation theory in equation (10). Moreover, a general finding can be taken from this example. The last two steps in equation (69) consisted of considering the "inner matrix element" $\langle a|V|a\rangle$ a number, which then can be taken out of the trace as a constant. The remaining element underneath the trace is a projector, whose trace is, as mentioned above, equal to the dimension of its target space - which here is one. This step will work regardless of the length of the inner matrix element. As long as the outer values of k, in the following called k_L and k_R for the left- and rightmost k, respectively, are both zero, the trace will be equal to this inner matrix element. Due to $k_L = k_R = 0$, it always is a diagonal matrix

element (DME) with respect to the unperturbed state.
For $n = 2$, three terms need to sum up to one, leading to three possible tuples:

1. $(0, 0, 1) \rightarrow \mathrm{Tr}\left(S^0 V S^0 V S^1\right)$
2. $(0, 1, 0) \rightarrow \mathrm{Tr}\left(S^0 V S^1 V S^0\right)$
3. $(0, 0, 1) \rightarrow \mathrm{Tr}\left(S^1 V S^0 V S^0\right)$

As the trace is invariant under cyclic permutations, the first term can be rewritten:

$$\mathrm{Tr}\left(S^0 V S^0 V S^1\right) = \mathrm{Tr}\left(S^1 S^0 V S^0 V\right) \tag{70}$$

However, since $S^1 S^0 = \sum_{i \neq a} \frac{-P_0 P_i}{(\varepsilon_a - \varepsilon_i)} = \sum_{i \neq a} \frac{-\delta_{a,i}}{(\varepsilon_a - \varepsilon_i)} = 0$, the first and – with analogous considerations – the third tuple vanish. As only the first and last elements in a tuple needed to be considered for this finding, this can directly be generalized for all tuples in an arbitrary order n: Whenever exactly one of the first and last numbers k in a tuple is zero, the whole trace vanishes. Here for $n = 2$, therefore, only the tuple $(0, 1, 0)$ remains:

$$\begin{aligned}
(0,1,0) &\widehat{=} \mathrm{Tr}\left(|a\rangle\langle a|VS^1V|a\rangle\langle a|\right) && (71)\\
&= \mathrm{Tr}\left(\langle a|a\rangle\langle a|VS^1V|a\rangle\right) && (72)\\
&= \langle aVS^1V|a\rangle = \sum_{m \neq a} \frac{\langle a|V|m\rangle\langle m|V|a\rangle}{\varepsilon_a - \varepsilon_m} = E_a^{(2)} && (73)
\end{aligned}$$

Again, this coincides with the second-order energy correction from Rayleigh-Schrödinger perturbation theory in equation (12).
For $n = 3$, ten possible tuples can be constructed. Using the finding that $(0, ..., 1) \cup (1, ..., 0) \rightarrow 0$, only four tuples remain nonzero:

1. $(0, 0, 0, 2) \rightarrow 0$
2. $(0, 0, 1, 1) \rightarrow 0$
3. $(0, 0, 2, 0) \rightarrow \mathrm{Tr}\left(S^0 V S^0 V S^2 V S^0\right)$
4. $(0, 1, 0, 1) \rightarrow 0$
5. $(0, 1, 1, 0) \rightarrow \mathrm{Tr}\left(S^0 V S^1 V S^1 V S^0\right)$
6. $(0, 2, 0, 0) \rightarrow \mathrm{Tr}\left(S^0 V S^2 V S^0 V S^0\right)$
7. $(1, 0, 0, 1) \rightarrow \mathrm{Tr}\left(S^1 V S^0 V S^0 V S^1\right)$
8. $(1, 0, 1, 0) \rightarrow 0$
9. $(1, 1, 0, 0) \rightarrow 0$
10. $(2, 0, 0, 0) \rightarrow 0$

Using the invariance of the traces under the construction of the adjoint, i. e. $\mathrm{Tr}\,(A) = \mathrm{Tr}\left(A^\dagger\right)$, some of the remaining terms can be combined. With the hermiticity of both V and the operators S^k, the adjoint of the matrix under the trace is simply expressed by the reversed order of the elements, thus:

$$\mathrm{Tr}\left(S^0VS^0VS^2VS^0\right) = \mathrm{Tr}\left(S^0VS^2VS^0VS^0\right) \tag{74}$$

Moreover, considering the case of $k_L = k_R = 0$, some terms can be directly rewritten as a diagonal matrix element with respect to the unperturbed state. For the remaining term $(1, 0, 0, 1)$, an auxiliary calculation is needed. With $S^0 = -\left(S^0\right)^2$, $S^k + S^k = S^{k_1+k_2}$ and a cyclic permutation, this becomes:

$$\begin{aligned}
\mathrm{Tr}\left(S^1VS^0VS^0VS^1\right) &= \mathrm{Tr}\left(S^0VS^0VS^1S^1V\right) && (75)\\
&= \mathrm{Tr}\left(S^0VS^0VS^2V\right) && (76)\\
&= \mathrm{Tr}\left(-S^0VS^0VS^2VS^0\right) && (77)\\
&= -\langle a|VS^0VS^2V|a\rangle && (78)
\end{aligned}$$

Thus, the contributions for the third-order energy corrections are:

$$\begin{aligned}
E_a^{(3)} &= \langle a|VS^1VS^1V|a\rangle + 2\cdot\langle a|VS^0VS^2V|a\rangle - \langle a|VS^0VS^2V|a\rangle && (79)\\
&= \langle a|VS^1VS^1V|a\rangle + \langle a|VS^0VS^2V|a\rangle && (80)
\end{aligned}$$

Inserting $-|a\rangle\langle a|$ for S^0, this becomes

$$E_a^{(3)} = \langle a|VS^1VS^1V|a\rangle - \langle a|V|a\rangle\langle a|VS^2V|a\rangle \tag{81}$$

And finally with explicit S^1 and S^2:

$$E_a^{(3)} = \sum_{m\neq a}\sum_{l\neq a}\frac{\langle a|V|m\rangle\langle m|V|l\rangle\langle l|V|a\rangle}{(\varepsilon_a-\varepsilon_m)(\varepsilon_a-\varepsilon_l)} - \sum_{m\neq a}\frac{\langle a|V|a\rangle\ |\langle a|V|m\rangle|^2}{(\varepsilon_a-\varepsilon_m)^2} \tag{82}$$

This is the result one would also obtain by inserting $|A^{(2)}\rangle$ (equation (13)) into the formula for $E_a^{(n)}$ (8). Thus, all exemplary evaluations for $E_a^{(n)}$ considered here match the corresponding expressions in Rayleigh-Schrödinger perturbation theory. Moreover, comparing equation (8) to the development of E_a in the Kato formalism, one can reasonably suspect an accordance of both formalisms in all orders n.
In the exemplary evaluation of $\mathrm{Tr}\left(B^{(n)}\right)$, each term could be brought into the form of a diagonal matrix element with respect to the unperturbed state. Here, several rules were used, which can be generalized to the case of an arbitrary n as follows:

1. Outer zeros: In the case of $k_L = k_R = 0$, the respective trace will be equal to the diagonal matrix element $\langle a|VS^{k_L+1}V...VS^{k_R-1}|a\rangle$

2. Single outer zeros: Tuples $(k_L, ..., k_R)$ with either $k_L = 0$ or $k_R = 0$ do not contribute.

3. Outer non-zeros: In case of neither k_L nor k_R being zero, cyclic permutations (leaving the trace unchanged) together with the properties $S^0 = -\left(S^0\right)^2$, $S^{k_1} + S^{k_2} = S^{k_1+k_2}$ help to convert the respective trace into a diagonal matrix element with respect to the unperturbed state. Since with the condition $\sum_{i=1}^{n+1} k_i = n - 1$, any tuple must contain at least two $k_i = 0$, this finding can be applied for any trace in the aforementioned form. With M and N being any $d \times d$ square matrices with $d = \dim H$, this is formally expressed as:

$$\begin{aligned} \mathrm{Tr}(MS^0N) &= \mathrm{Tr}(S^0NM) && (83)\\ &= \mathrm{Tr}(-S^0S^0NM) && (84)\\ &= -\mathrm{Tr}(|a\rangle\langle a|MN|a\rangle\langle a|) && (85)\\ &= -\langle a|MN|a\rangle && (86) \end{aligned}$$

4. Mirrored traces: Due to $\mathrm{Tr}\,(A) = \mathrm{Tr}\left(A^\dagger\right)$ and the hermiticity of all S^k and V, a certain tuple and its mirrored version are associated with the same value.

5. Elementary matrix elements (EME(s)): Explicitly rewriting "inner" [5] operators S^0 into their projector form $-|a\rangle\langle a|$ allows the splitting of diagonal matrix elements into products of shorter ones. If all possible splits are applied, the short diagonal elements are called "elementary matrix elements". Therefore, each diagonal matrix element can be identified by its respective elementary matrix elements.

Using these rules, all contributions for the n-th order energy correction can be written as:

$$E_a^{(n)} = \sum_{\{k_j\}_{n-1}} G_{\{k_j\}_{n-1}} D_{a,\{k_j\}_{n-1}} \tag{87}$$

with the following variables:

- $\{k_j\}_{n-1}$: combinations $(0, k'_1, ..., k'_{n-1}, 0)$ in n-th order that satisfy the condition $\sum_{j=1}^{n-1} k'_j = n - 1$. The $'$ expresses possible changes to the original set $\{k_i\}$ during the conversion of the traces to a diagonal matrix element. These changes, however, do not change the validity of the condition, which holds for both $\{k_i\}$ and $\{k'_i\}$.

- $D_{a,\{k_j\}_{n-1}}$ is the diagonal matrix element with respect to unperturbed state $|a\rangle$.

- $G_{\{k_j\}_{n-1}}$ expresses the respective weight factor for the diagonal matrix element.

[5] i. e. S^0 that do no not come from k_L or k_R.

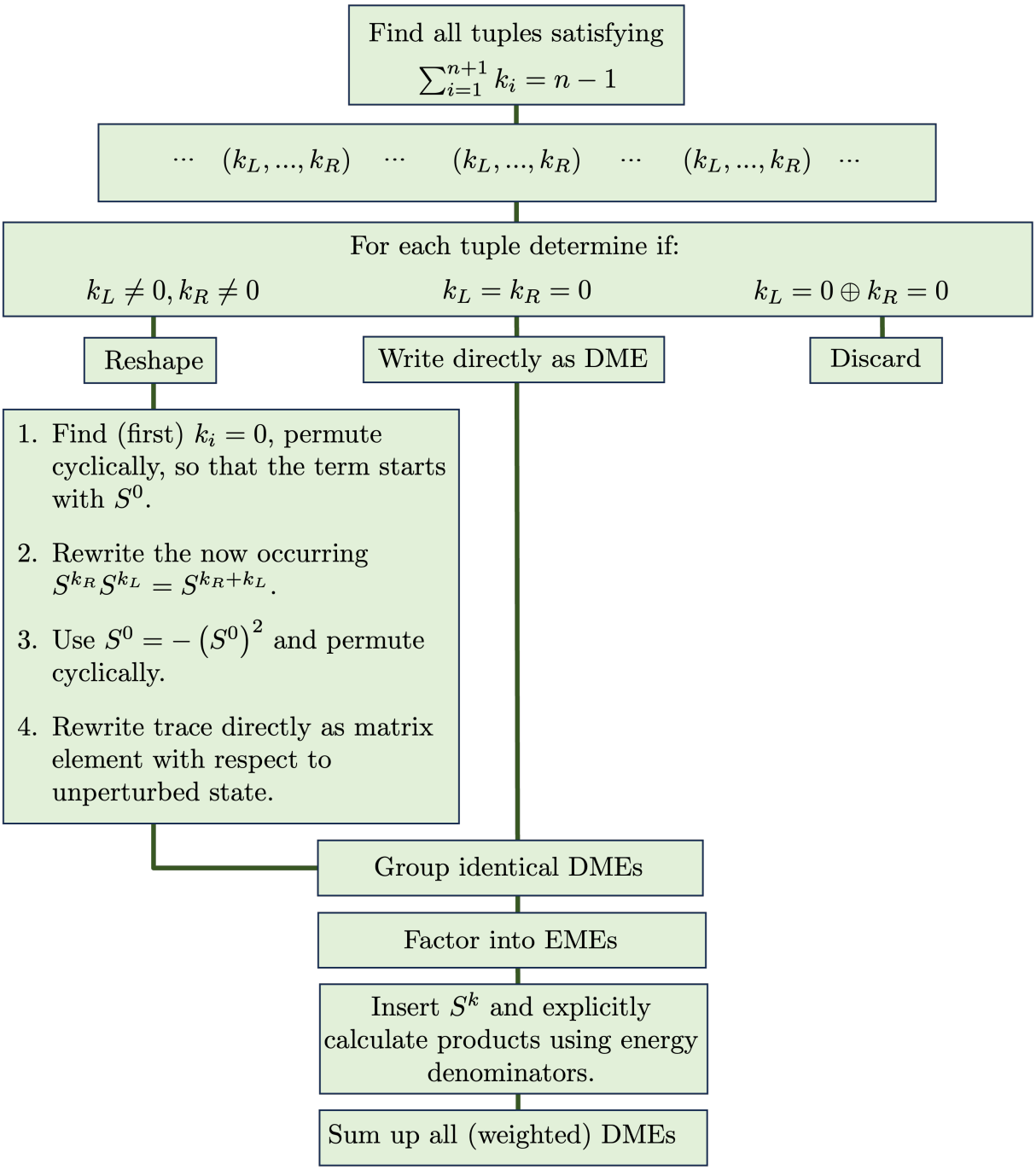

Figure 1: Flowchart for the manual calculation of the n-th order energy correction $E_a^{(n)}$. Relevant tuples only appear for the cases of the outmost k_i being either simultaneously zero or non-zero. Tuples of the former case can be directly written as diagonal matrix elements with respect to the unperturbed state, whereas the latter case needs reshaping. Once all tuples are in the form of DMEs, identical elements are grouped. The resulting terms are then factored into elementary matrix elements that do not contain any S^0 anymore. By inserting the S^k, the energy correction can be calculated for the whole order n.

Teichmann et. al. [9] and Eckardt [8] use a figurative interpretation of the energy corrections: The resulting contributions are sums over *process chains* that each are formed by a sequence of perturbation events that starts and end in the unperturbed state $|a\rangle$. Each of these events within a sequence is then a matrix element of the perturbation operator

$\langle\alpha|V|\beta\rangle$.
Using all of the above, a guide for the evaluation of all contributing terms for a particular n-th order energy correction "by hand" is depicted in Figure 1.

3.4 Algorithmical Computation of the Contributions (Python-Code)

For higher orders, the manual handling of all occurring terms for $\mathrm{Tr}\left(B^{(n)}\right)$ quickly becomes very complicated. Therefore, these terms can better be evaluated using an algorithm executed by a computer. In the following, the adaption of the aforementioned guideline into a computer algorithm will be explained with the help of schematics. A Python-based package was developed that determines all contributing diagonal matrix elements (in the form of products of elementary matrix elements) for any order n. This procedure is abstract and per se not connected to any particular system. However, this package also contains the possibility to evaluate these perturbational contributions for a given Hamiltonian and perturbation. It can therefore be applied to a wide range of systems.
Figure 2 shows the schematics of the algorithmic computation of the general system-independent series, which is implemented in an object-oriented manner. For this, an object of the class `AbstractSeries` is created and its parameters (attributes) are initialized. These parameters and methods are marked by `self.(attributename)`. Here, the initial attributes are caches for the tuples $(k_L, ..., k_R)$ for single order determination, as well as a dictionary[6] for the whole series where all resulting contributions are stored. After the object initialization, for each order n, $\mathrm{Tr}\left(B^{(n)}\right)$ is determined, starting by constructing all permitted tuples. Figure 4 shows the procedure that is conducted for the generation of the whole series, starting by generating the contributions within a single order (see Figure 5). Here, as already discussed, only two types of tuples need to be considered and generated. First, tuples where the outmost k_i are zero, are considered. As these can be directly rewritten into diagonal matrix elements, they will from now on be referred to as *direct DMEs*, whereas tuples with $k_L \neq 0, k_R \neq 0$ will be called *indirect DMEs*.
Both cases for the relevant tuples alter the condition under which the inner k_i need to be generated: For the direct DMEs (see Figure 6 for the code), two k_i are already fixed. Therefore, now, the sum over $n+1-2 = n-1$ objects needs to be equal to $n-1$. This condition leads to the symbolic problem of putting $n-1$ objects onto $n-1$ positions, as calculated by the class method `_configuration()`. This function works through each of the symbolic positions one by one. Starting at the first position, it iteratively fixes each possible number of objects at this position and calls itself again to place the remain-

[6] In Python, a dictionary is a type of data collection where items can be stored in key:value pairs. Aside from a few exceptions, the values can be of an arbitrary datatype. They are indexed by their key, i. e. `mydictionary['key 1']= value1`.

ing objects into the remaining number of places. Each of these function calls contains information on the original problem and the current condition, i. e. how many objects remain to be fixed (`rem_sum`) in how many places (`rem_pos`). Besides, the previously fixed numbers in the hierarchically higher function calls are passed with each function call, as well. In this context, "fixing a number at a position" refers to appending the respective number to the current configuration. Once a configuration is complete, i. e. the numbers of remaining positions and objects are both zero, it is appended to the cache. In this recursive manner, the function `_configuration()` generates a list of all possible configurations satisfying the conditions given by the initial input `init_pos` and `init_sum`.
For the tuples with the two outmost k_i being both nonzero, `_configuration()` is used again – however, with different initial parameters. Figure 7 contains the Python code used for this. During the process of reshaping the tuples into DMEs, the property $S^{k_L} + S^{k_R} = S^{k_L+k_R}$ is used, thus having k_L and k_R appear only within their sum $k_L + k_R := \eta$. As explained by [10], for a fixed value of η, only $n - 1 - \eta$ objects need to be placed onto $n-1$ positions. Therefore, `_configuration()` is called with these parameters to generate all possible inner k_i for this η. These configurations now need to be reshaped according to steps 1 to 4 in Figure 1. For a tuple $(k_{L+1}, ...k_{R-1})$, the following procedure needs to be applied: The rewritten tuple starts with the subset of elements that appear behind the first $k_i = 0$. After that, η is inserted, followed by the rest (beginning) of the original tuple. As there are $\eta - 1$ possibilities to write η as a sum of two summands k_L and k_R and the usage of the property $S^0 = -(S^0)^2$ delivers one minus sign, the weight of the configurations arising from one η is $-(\eta - 1)$. Once all of the indirect DMEs have been generated, they are saved in the respective cache/dictionary `indirect_dmes`.
Following the general scheme in Figure 1, identical DMEs now need to be grouped. The direct DMEs already contain every possible form of the inner part of a DME. Therefore, one can go through them one by one and find possible matching "partners" within the set of indirect DMEs. The weight of the direct DME is then increased by the (η-dependent) weight of the indirect DME. Then, the DME is factored into EMEs using the class method `_factor_into_emes()`. This method iteratively goes through each entry within a DME and cuts the DME into sublists at each occurring zero. This symbolizes the explicit writing of every $S^0 = -P_0$. The respective code is shown in Figure 9. Thus, each split delivers a minus sign for the whole DME. The resulting sign of the DME is then multiplied with the combined weight in order to obtain the final correct prefactor. With the function `_create_eme_occupation_dict()`, the DME is identified with the distribution of its EMEs. The function returns a sorted dictionary, where each EME is assigned its number of occurrences. Here, it will later prove useful to label the different EMEs with a number. For this, the function `_create_eme_occupation_dict()` uses the method `_label_eme()`,

see Figure 11 for the code. For each EME, the following number Z is calculated.

$$Z = \sum_{i=1}^{L} k_i n^{i-1} \tag{88}$$

This labeling procedure recognizes the tuple of an EME as a number representation of an integer Z in base n. L is the length of the EME-tuple. Since reversed tuples will lead to the same value, Z is calculated for both the original and reversed versions of the EME. The smaller value of Z is then kept and stored.

At this point in the process, approximately the first half of the code in Figure 5 has been executed, i. e. the particular DME is assigned a weight and can be represented by its occupation of EMEs. This is all information that is needed in order to later explicitly calculate its value. Therefore, this DME is now represented by a dictionary that stores its weight, its EME occupation, and – for better documentation – its explicit factored form and its original tuple. This dictionary is then appended to the "final" dictionary `dmes_all_orders[n]`. However, this dictionary only needs to be created once for identical DMEs, i. e. EMEs with the same EME occupation. For this, the cache `treated_dmes[n]` is used. Here, already treated DMEs are saved, together with their key within the final dictionary `dmes_all_orders[n]`. Thus, no new entry needs to be created. Instead, only the weight in this already existing entry is increased by `weight`.

After the generation of contributions in every order n, all information is stored in the variable `dmes_all_orders[n]`. At this point, it can be useful to filter this dictionary and only keep all entries for which `'weight'` leads to a nonzero number. This step is not explicitly shown in the shown code, however, it was used. Figure 3 contains an example print of `dmes_all_orders[n]` for `n=5`.

```python
class AbstractSeries:
    def __init__(
        self, max_order: int, return_without_first_order: bool = False
    ) -> None:
        self.max_order = max_order
        # 1. Parameters for the generation of a single order
        self.first_call = True
        self.list_idx = 0
        self.length_cache = [0]
        self.config_cache = []
        self.direct_dmes = []
        self.indirect_dmes = []  # position in list equals 2+eta
        self.weights_indirect_dmes = {}
        self.return_without_first_order = return_without_first_order

        # 2. Parameters for the generation of the full series

        self.dme_all_orders = {}

        for i in range(self.max_order + 1):
            # value should be a dictionary of dictionaries.
            # each dictionary represents a DME.
            # For this: keys: 'weight', 'EMEs', 'factors', 'repres'
            # 'EME' itself is a dict with keys: 'label' and 'occupation'
            self.dme_all_orders[i] = {}

            # list for already treated matrix elements, identification by EME occupation
            # ! list has empty zeroeth entry, so the order indexing is more readable
            self.treated_dmes = [{} for _ in range(self.max_order + 1)]
            # initialize dict for EME labels
            self.dict_eme_labels = {}
```

Figure 2: Constructor of abstract, i.e. system-independent Kato series, implemented in a Python class called `AbstractSeries`. The `AbstractSeries` object receives two input parameters: `max_order`, which is the maximum order up to which the energy corrections $E_a^{(n)}$ are summed up and `return_without_first_order` which flags if the first order correction is already known to vanish. However, when set to `false`, the calculated series will be returned both without and with first order corrections. Using the flag only leads to reduction in memory. Mainly, six initial attributes are mapped to the object upon construction: caches (lists) for the tuples within a single order `direct_dmes` and `indirect_dmes`, a later-needed boolean flag `first_call` and a structured variable `dmes_all_orders` (here: dictionary with entries for every order n) to store the results of each single order. Every item in `dmes_all_orders[n]` contains a dictionary for every unique DME – each with information on its weight, its factorization into EMEs, and the respective occupation representation. `treated_dmes` is a list of dictionaries that is later needed as a cache for the evaluation of the tuples. `dict_eme_labels` is a dictionary that will contain unique labels for each unique elementary matrix element. For each order n, all relevant contributions are calculated. The procedure is shown in Figure 5. After all orders have been generated, `dmes_all_orders` contains the complete information on the Kato series up to order `max_order`.

```
Results:

Order n= 1
{'weight': 1, 'EMEs': {0: 1}, 'factors': [[]], 'repres': [0]}


Order n= 2
{'weight': 1, 'EMEs': {1: 1}, 'factors': [[1]], 'repres': [1]}


Order n= 3
{'weight': -1, 'EMEs': {0: 1, 2: 1}, 'factors': [[], [2]], 'repres': [0, 2]}
{'weight': 1, 'EMEs': {10: 1}, 'factors': [[1, 1]], 'repres': [1, 1]}


Order n= 4
{'weight': 1, 'EMEs': {0: 2, 3: 1}, 'factors': [[], [], [3]], 'repres': [0, 0, 3]}
{'weight': -2, 'EMEs': {0: 1, 11: 1}, 'factors': [[], [1, 2]], 'repres': [0, 1, 2]}
{'weight': -1, 'EMEs': {1: 1, 2: 1}, 'factors': [[1], [2]], 'repres': [1, 0, 2]}
{'weight': 1, 'EMEs': {91: 1}, 'factors': [[1, 1, 1]], 'repres': [1, 1, 1]}


Order n= 5
{'weight': -1, 'EMEs': {0: 3, 4: 1}, 'factors': [[], [], [], [4]], 'repres': [0, 0, 0, 4]}
{'weight': 2, 'EMEs': {0: 2, 12: 1}, 'factors': [[], [], [1, 3]], 'repres': [0, 0, 1, 3]}
{'weight': 2, 'EMEs': {0: 2, 20: 1}, 'factors': [[], [], [2, 2]], 'repres': [0, 0, 2, 2]}
{'weight': 2, 'EMEs': {0: 1, 1: 1, 3: 1}, 'factors': [[], [1], [3]], 'repres': [0, 1, 0, 3]}
{'weight': -2, 'EMEs': {0: 1, 92: 1}, 'factors': [[], [1, 1, 2]], 'repres': [0, 1, 1, 2]}
{'weight': -1, 'EMEs': {0: 1, 100: 1}, 'factors': [[], [1, 2, 1]], 'repres': [0, 1, 2, 1]}
{'weight': 1, 'EMEs': {0: 1, 2: 2}, 'factors': [[], [2], [2]], 'repres': [0, 2, 0, 2]}
{'weight': -2, 'EMEs': {1: 1, 11: 1}, 'factors': [[1], [1, 2]], 'repres': [1, 0, 1, 2]}
{'weight': -1, 'EMEs': {2: 1, 10: 1}, 'factors': [[1, 1], [2]], 'repres': [1, 1, 0, 2]}
{'weight': 1, 'EMEs': {820: 1}, 'factors': [[1, 1, 1, 1]], 'repres': [1, 1, 1, 1]}
```

Figure 3: Results (output from function `print_abstract_result()`) for the first five orders. As visible in the contributions, the first order is not explicitly set to vanish.

```python
def _generate_all_orders(self) -> None:
    "Generate whole (abstract, i.e. symbolic) Kato series up to max_order"
    for n in range(1, self.max_order + 1):
        print(f"Generating order n= {n}...")
        self._generate_single_order_into_cache(n)
        # contributions for order n now lie in self.direct_dmes and self.indirect_dmes
        # Go through each direct DME and add weight from indirect DMEs
        # Then, factor each DME into EMEs and check if a matrix element
        # with the same EMEs has already been treated.
        # In this case: combine the weights. If not create entry in
        # self.treated_matrix_elements
        # and append result to self.dme_all_orders[o]

        for dme_tuple in self.direct_dmes:
            factored = self._factor_into_emes(dme_tuple)
            # each split produces a minus sign
            sign_matrix_element = pow(-1, len(factored) - 1)
            weight_nonzero = self.weights_indirect_dmes.get(tuple(dme_tuple), 0)
            weight = weight_nonzero + 1
            weight = weight * sign_matrix_element
            emes = self._create_eme_occupation_dict(factored)
            if str(emes) in self.treated_dmes[n]:
                key = self.treated_dmes[n][str(emes)]
                self.dme_all_orders[n][key]["weight"] += weight
            else:
                dict_dme = {
                    "weight": weight,
                    "EMEs": emes,
                    "factors": factored,
                    "repres": dme_tuple,
                }
                key_in_dict = (
                    max(self.dme_all_orders[n].keys()) + 1
                    if self.dme_all_orders[n]
                    else 0
                )
                self.dme_all_orders[n][key_in_dict] = dict_dme
                self.treated_dmes[n][str(emes)] = key_in_dict
        print("Generation of abstract Kato series complete.")
```

Figure 4: Python code for the generation of all orders up to `max_order`. The method`self._generate_single_order_into_cache` generates all tuples for the direct and indirect DMEs into the single-order caches `direct_dmes` and `indirect_dmes`. Then, the `for-loop` goes through each direct DME: First, with `factor_into_emes()`, every DME is split into EMEs. The sign, resulting from explicitly writing out S^0 as $-P_0$ is calculated from the number of splits during this. If an indirect DME leads to the same tuple $(0, k'_1, ..., k'_{n-1}, 0)$ as `dme_tuple` the original weight one is increased by the weight of the respective indirect DME before multiplying the resulting combined weight by its sign. Next, a variable `emes` is calculated using `create_occupation_dict()`, which contains the number of occurrences of the EMEs within the DME. As the function sorts the dictionary before returning it, all DMEs with an identical occupation of EMEs will lead to the same variable `emes`. If a DME with the same EME occupation has already been treated within the current order n, it already has an entry in `dmes_all_orders[n]`. The weight of this entry is then increased by `weight`. If no such DME has already been treated, an entry is created in `dmes_all_orders[n]` with the next higher key than the last entry's key. This way, equivalent DMEs are now combined in `dmes_all_orders[n]` under one key.

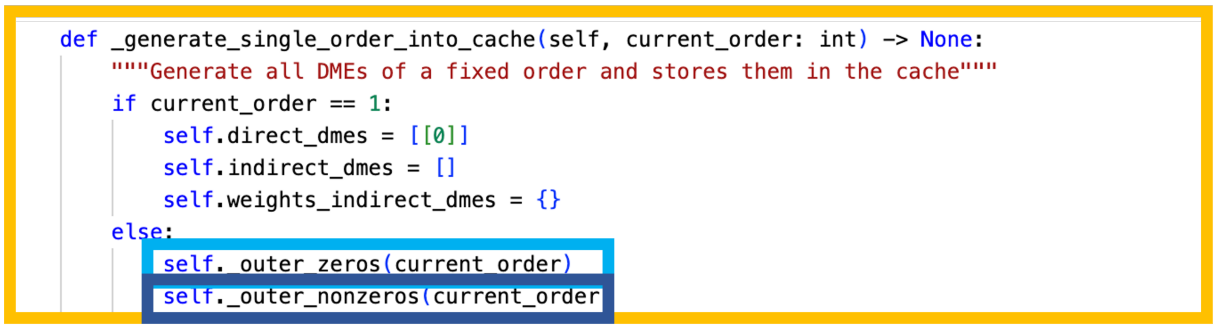

Figure 5: Python code for the generation of a single order n. The methods `self.outer_zeros()` and `self.outer_nonzeros()` generate all tuples for the direct and indirect DMEs into the single-order caches `direct_dmes` and `indirect_dmes`.

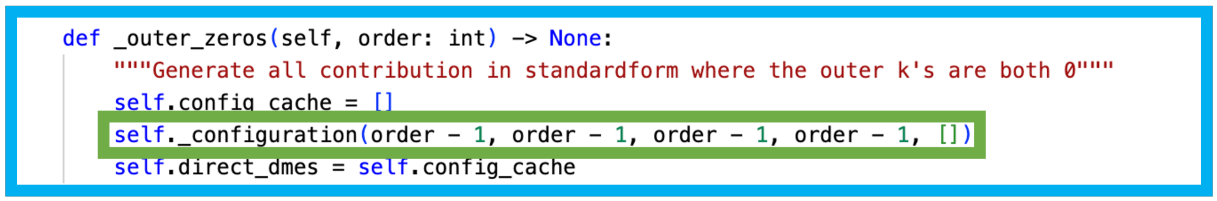

Figure 6: Python-code for the generation of direct DMEs, i. e. DMEs resulting from tuples for which $k_L = k_R = 0$. Therefore, the inner matrix element can be represented as $(k_{L+1}, ...k_{R-1})$ with the condition $\sum_{i=1}^{n-1} k_i = n - 1$ - now that two k_i are already set to be zero. This generation of this inner matrix element is therefore equivalent to finding all possible distributions of $n - 1$ objects that sum up to $n - 1$ onto $n - 1$ places. This is performed by the function `self._configuration()`, which saves all possible inner matrix elements $(k_{L+1}, ...k_{R-1})$ in `self.config_cache`. The content of this cache is then saved in `direct_dmes`.

```python
def _outer_nonzeros(self, order: int) -> None:
    """Generate all contributions where both k_L and k_R are non-zero for
    one fixed order"""
    self.indirect_dmes = (
        []
    )  # becomes a list (eta values) of lists (of fixed-eta configurations)
    self.weights_indirect_dmes = {}
    self.config_cache = []
    eta_range = range(2, order)  # eta can take all values from 2 to order-1
    for eta in eta_range:
        sum_ = order - 1 - eta
        pos = order - 1
        self._configuration(pos, sum_, pos, sum_, [])
        self.indirect_dmes.append(self.config_cache)

    # write contributions in standardform (without explicit outer zeros).
    # For this, look at each contribution
    for eta_idx in range(len(self.indirect_dmes)):
        eta = eta_idx + 2  # eta ranges from 2 to order-1
        for contr_idx in range(len(self.indirect_dmes[eta_idx])):
            k_list = self.indirect_dmes[eta_idx][contr_idx]
            idx = 0
            not_found = True
            while not_found:
                not_found = k_list[idx] != 0
                idx += 1
            # choose first zero, write down part after it,
            # then eta, then rest of the list without
            # (0 is used for DME form)
            idx -= 1  # this way, idx is the index of zero
            new_list = k_list[idx + 1 :]
            new_list.append(eta)
            if idx != 0:
                new_list += k_list[:idx]
            self.indirect_dmes[eta_idx][contr_idx] = new_list
            self.weights_indirect_dmes[tuple(new_list)] = -(eta - 1)
```

Figure 7: Python-code for the generation of indirect DMEs, i. e. DMEs resulting from tuples for which $k_L, k_R \neq 0$. First, the caches `indirect_dmes`, `weights_indirect_dmes`, and `config_cache` are emptied. In the evaluation of these indirect DMEs, k_L and k_R only appear in their sum $k_L + k_R := \eta$. Possible values for η are therefore all whole numbers between 2 and $n-1$. For each value of η, `self._configuration()` is used to find all possibilities of distributing $n-1-\eta$ objects onto $n-1$ positions. These contributions are saved in `indirect_dmes`. In order to write each tuple as a DME, the first $k_i = 0$ is searched for. This then allows the cyclical permutation to have the resulting $k'_L = k'_R = 0$. As for each η, there are $\eta - 1$ possibilities for explicit values of the original k_L and k_R, each of the generated indirect DME carries a weight factor of $-(\eta - 1)$. The minus sign results from the property $S^0 = -\left(S^0\right)^2$.

```python
def _configuration(
    self,
    init_pos: int,
    init_sum: int,
    rem_pos: int,
    rem_sum: int,
    config: list,
) -> None:
    """Generate all possible configurations of init_sum objects onto
    init_pos positions"""
    # if the function is called for the first time, reset the current_list track_keeper
    if self.first_call:
        # create list with enough entries to store all possible configurations
        self.length_cache = int(sp.special.binom(init_sum + init_pos - 1, init_sum))
        self.config_cache = [[] for _ in range(self.length_cache)]
        self.first_call = False
        config = []

    if rem_pos == 1:
        self.config_cache[self.list_idx] = config + [rem_sum]
        self.list_idx += 1
    else:
        for i in range(rem_sum + 1):
            self._configuration(
                init_pos,
                init_sum,
                rem_pos - 1,
                rem_sum - i,
                config + [i],
            )

    if self.list_idx == self.length_cache:
        self.list_idx = 0
        self.first_call = True
```

Figure 8: Python-code for the function `place_objects()`, which generates all possibilities to distribute `rem_sum` objects onto `rem_pos` positions. If the function is called for the first time, the length of the cache for the tuples is set to the number of possible distributions under the aforementioned condition. Besides, the cache is initialized before setting `first_call` to zero. Basically, this function fixes the position of one object and calls itself again to place the remaining objects onto the remaining number of positions. For this, the current configuration, i. e. the already fixed positions, is also passed as an argument. In each recursive iteration, one new possible position is fixed and appended to `config_cache`. The "first" parameters `init_pos` and `init_sum` are passed as well, allowing for a termination condition of the recursive function calling once all objects have been placed in a particular configuration. In this case, the hierarchically higher function call executes the next iteration of the `for`-Loop, until the respective configuration (tuple) is completely fixed for all places. Through the function arguments, each function call "knows" its hierarchical position within all recursive function calls. If all possibilities have been generated for one order n, the list index is reset and the boolean flag `first_call` is set to `True` again.

```python
def _factor_into_emes(self, dme: list) -> list:
    """Split a list of integers at the integers that are 0"""
    if dme == [0]:
        return [[]]
    new_list = []
    last_zero_idx = 0
    for i in range(len(dme)):
        if dme[i] == 0:
            new_list = new_list + [dme[last_zero_idx:i]]
            last_zero_idx = i + 1  # this way, 0 is not included in the next slice
    new_list = new_list + [dme[last_zero_idx:]]
    return new_list
```

Figure 9: Python-code for the function `_factor_into_emes()`, which splits a tuple `dme` at all occuring zeroes. For this, the original DME tuple is passed as a list of integers. Using a `for`-loop, at each $k_i = 0$, a new (sub)list is begun with all following k_i until (possibly) the next zero. The result is returned as a list with sublists for each occurring EME.

```python
def _create_eme_occupation_dict(self, factored_matrixelement: list) -> dict:
    """Create a (sorted) dictionary with the occupation of each EME within a DME"""
    occ_dict = {}
    # itterate through all EMEs within a matrix element
    for i in range(len(factored_matrixelement)):
        label_min = self._label_eme(factored_matrixelement[i])
        occ_dict[label_min] = occ_dict.get(label_min, 0) + 1
        # all dicts should have same order
    return dict(sorted(occ_dict.items()))
```

Figure 10: Python-code for the function `create_occ_dict ()`. The function receives the input `factored_matrixelement` as a list of lists, representing the factorization into elementary matrix elements. For each EME in `factored_matrixelement`, a label according to equation (88) is calculated using the function `_label_eme()`. For each occurrence of the same label, a counter in `occ_dict` is increased by one, thus constructing an occupation representation of the EMEs within the factored DME. To account for the commutativity of EMEs within a factored DME, the resulting dictionary is sorted by the values of the EME labels. Therefore DMEs that have an identical EME occupation are assigned identical occupation dictionaries.

```python
def _label_eme(self, eme: list) -> int:
    """Return the minimal Z and the respective minimal eme,
    adds the calculated values to the global dictionary of eme labels
    """
    Z = 0
    Z_ = 0
    eme_rev = eme[::-1]
    # loop is not executed if len(eme)==0, i.e. if eme = <a|V|a>
    for i in range(len(eme)):
        Z += eme[i] * pow(self.max_order, i)
        Z_ += eme_rev[i] * pow(self.max_order, i)

    if Z > Z_:
        label_min = Z_
        eme_min = eme_rev
    else:
        label_min = Z
        eme_min = eme
    self.dict_eme_labels[label_min] = eme_min
    return label_min
```

Figure 11: Python-code for the function `_label_eme()`, which calculates the the number Z (equation (88)) for each EME. As mirrored EMEs lead to the same values, the minimum of the Z-values for both the original and reversed version of the EME is returned.

References

[1] Tosio Kato. On the Convergence of the Perturbation Method. I. *Prog. Theor. Exp. Phys.*, 4(4):514–523, 1949.

[2] David J. Griffiths. *Introduction to Quantum Mechanics*. Cambridge University Press, Cambridge, third edition. edition, August 2018.

[3] J. J. Sakurai and J. Napolitano. *Modern Quantum Mechanics*. Pearson, second edition, 2013.

[4] Albert Messiah. *Quantenmechanik*, volume 2. de Gruyter, third edition, 1990.

[5] Robert Denk. *Mathematische Grundlagen der Quantenmechanik*, volume 1, chapter 5, pages 117–118. Springer Spektrum, first edition, 2022.

[6] Tosio Kato. *Perturbation Theory for Linear Operators*, volume 132 of *Classics in Mathematics*. Springer Science & Business Media, second edition, 2013.

[7] Oskar Maria Baksalary, Dennis S. Bernstein, and Götz Trenkler. On the equality between rank and trace of an idempotent matrix. *Applied Mathematics and Computation*, 217(8):4076–4080, 2010.

[8] André Eckardt. Process-Chain Approach to High-Order Perturbation Calculus for Quantum Lattice Models. *Phys. Rev. B*, 79(19):195131, May 2009.

[9] Niklas Teichmann, Dennis Hinrichs, Martin Holthaus, and André Eckardt. Process-Chain Approach to the Bose-Hubbard Model: Ground-state Properties and Phase Diagram. *Phys. Rev. B*, 79(22):224515, June 2009.

[10] Martin Holthaus. Higher-Order Perturbation Theory. Unpublished lecture script, Carl von Ossietzky Universität Oldenburg, May 2024.