Photoemission Orbital Tomography for an LCAO–GTO Basis Set

PhysikMDB — theory notes

Abstract

These notes collect the theory behind PhysikMDB: what a photoemission momentum map is, how it follows analytically from a molecular orbital expanded in contracted Gaussians, what the measurement adds on top of it, and how the whole picture extends from ground-state orbitals to excitons. The first seven sections are physics and can be read without ever opening the code; Appendix A maps them onto the C kernels the website runs, for a reader who wants to check one against the other. Everything is in Hartree atomic units unless a unit is named.

Contents

1 Introduction

Interfaces between organic molecules and metals are central to organic electronics, and their electronic structure is set by several processes at once: van der Waals attraction, renormalisation of the ionisation potential and electron affinity by the polarisable substrate,

hybridisation of molecular orbitals with metallic states, and charge transfer across the interface. Density functional theory (DFT) is the standard tool for describing them, but relating Kohn–Sham orbital energies to measured single-electron excitation energies has no rigorous justification except for the highest occupied orbital.

Photoemission turns that difficulty into an opportunity. By measuring not only the energy of the photoelectron but also the *angular* distribution of the photocurrent — angle-resolved photoemission spectroscopy (ARPES) — one obtains a pattern that carries an orbital fingerprint, not just an energy. This is the basis of photoemission orbital tomography (POT) [1, 2, 3]: for oriented films of π-conjugated molecules the measured angular distribution is, to a very good approximation, the modulus squared of the initial orbital in *momentum* space. Orbitals overlapping in binding energy can then still be told apart by their patterns [4], and the same machinery run in reverse reconstructs real-space orbital densities from data [1].

This database stores the ingredients POT needs — geometries, basis sets and molecular orbital coefficients from DFT — and evaluates the maps in the browser rather than shipping pictures. Sections 2–6 derive what is evaluated. Section 7 extends it to excited states following Kern, Windischbacher and Puschnig [5]. The same theory underlies the standalone simulation program kMap.py [6], which solves the complementary problem of fitting simulated maps to measured ones.

2 Photoemission Orbital Tomography

2.1 The photocurrent

Within the one-step model, Fermi's golden rule gives the photocurrent as a sum over initial states Ψ_i reaching a final state Ψ_f at kinetic energy E_{kin} and parallel momentum (k_x, k_y),

$$I(k_x, k_y; E_{kin}) \;\propto\; \sum_i \left| \langle \Psi_f(k_x, k_y; E_{kin}) |\, \vec{A} \cdot \vec{p} \,| \Psi_i \rangle \right|^2 \delta(E_i + \Phi + E_{kin} - h\nu)\,, \qquad (1)$$

with $\vec{A}$ the vector potential of the exciting light, $\vec{p}$ the momentum operator, Φ the sample work function, E_i the binding energy of the initial state and $h\nu$ the photon energy. The δ-function is energy conservation and is what ties a pattern to a binding energy.

The emission direction fixes the momentum. With polar and azimuthal emission angles ϑ, φ,

$$k_x = k \sin\vartheta \cos\varphi, \qquad k_y = k \sin\vartheta \sin\varphi, \qquad E_{kin} = \frac{\hbar^2 k^2}{2m_e}, \qquad (2)$$

so scanning the analyser over emission angles at fixed E_{kin} scans a *hemisphere* of radius $k(E_{kin})$ in momentum space, and the map the detector records is that hemisphere projected onto (k_x, k_y).

2.2 The plane-wave final state

POT rests on one approximation: the final state of the photoelectron is taken to be a plane wave, $\Psi_f \to e^{i\vec{k}\cdot\vec{r}}$. The matrix element in Eq. (1) then factorises into a polarisation prefactor and the Fourier transform of the initial orbital,

$$I_i(k_x, k_y; E_{kin}) \;\propto\; \left|\vec{A}\cdot\vec{k}\right|^2 \left|\tilde{\psi}_i(\vec{k})\right|^2, \qquad \tilde{\psi}_i(\vec{k}) = \int \psi_i(\vec{r})\, e^{-i\vec{k}\cdot\vec{r}}\, d\vec{r}. \tag{3}$$

Equation (3) is the central statement of POT: up to the smooth envelope $|\vec{A}\cdot\vec{k}|^2$, the ARPES pattern of a single orbital *is* that orbital in momentum space. It is why a measured momentum map can be read as an orbital at all.

Two caveats belong with it. First, the plane wave ignores scattering of the outgoing electron by the ion cores; it works well for the delocalised π states of planar aromatic molecules at ultraviolet photon energies, and less well for localised σ states and for atoms with large scattering cross sections [7, 4]. Second, ψ_i should strictly be the Dyson orbital of the ionisation process rather than a Kohn–Sham orbital [7]; identifying the two is the usual working assumption here, and Section 7 shows what happens when it is dropped.

2.3 The photoemission hemisphere

Setting $\hbar = m_e = 1$, Eq. (2) collapses to a parameter-free relation between the kinetic energy and the radius of the sampled sphere,

$$k(E_{kin}) = \sqrt{2\,E_{kin}}, \qquad k_z = \sqrt{k^2 - k_x^2 - k_y^2}, \tag{4}$$

with no empirical eV-to-Å^{-1} prefactor anywhere. Pixels with $k_x^2 + k_y^2 > k^2$ lie outside the emission horizon and carry no intensity: a map computed over a wider range than $k(E_{kin})$ simply gains an empty margin.

The task of the code is therefore to evaluate $\tilde{\psi}_i(\vec{k})$ for an LCAO–GTO orbital: on the hemisphere (4) for a momentum map, on a full Cartesian k-grid for the three-dimensional density, or — trivially — in real space.

3 The LCAO–GTO Wavefunction

Each molecular orbital is a linear combination of atom-centred basis functions,

$$\psi_i(\vec{r}) = \sum_{\mu=1}^{N_{basis}} c_{i\mu}\, \phi_\mu(\vec{r} - \vec{R}_{A(\mu)}), \tag{5}$$

where i indexes the molecular orbital, μ the basis function and $A(\mu)$ the atom it is centred on. A basis function is a real solid harmonic times a contracted radial Gaussian,

$$\phi_\mu(\vec{r}) = R_{l_\mu}(r)\, X_{l_\mu m_\mu}(\hat{r}), \qquad R_l(r) = r^l \sum_{p=1}^{M_\mu} c_p\, N_p\, e^{-\alpha_p r^2}, \tag{6}$$

with contraction coefficients c_p, Gaussian exponents α_p (in Bohr^{-2}) and M_μ primitives. Basis functions sharing the same $(A, l, \{\alpha_p\})$ and differing only in m form one *shell* of $2l+1$ members.

3.1 The primitive normalisation

Quantum-chemistry codes report contraction coefficients for *unnormalised* primitives, so the primitive normalisation

$$N_p = \sqrt{\frac{2\,(2\alpha_p)^{\,l+3/2}}{\Gamma(l+3/2)}} \tag{7}$$

has to be folded into c_p before the basis is used. This is not an overall scale factor and cannot be absorbed into a later normalisation: N_p grows as $\alpha_p^{\,l/2+3/4}$, so within one contracted shell it reweights the tight primitives against the diffuse ones and therefore changes the *shape* of the basis function, not only its size. Getting it wrong shifts peak-normalised momentum maps by a few per cent and real-space amplitudes by factors of order unity.

4 Fourier Transform of a Contracted Gaussian

The transform of Eq. (5) is done once, analytically, and then reused at every sampling point.

4.1 Separating centre, angle and radius

The shift theorem moves each atomic centre into a phase,

$$\int \phi_\mu(\vec{r} - \vec{R}_A)\, e^{-i\vec{k}\cdot\vec{r}}\, d\vec{r} = e^{-i\vec{k}\cdot\vec{R}_A}\, \tilde{\phi}_\mu(\vec{k}), \tag{8}$$

and the Rayleigh expansion of the plane wave,

$$e^{-i\vec{k}\cdot\vec{r}} = 4\pi \sum_{l'=0}^{\infty} \sum_{m'=-l'}^{l'} (-i)^{l'}\, j_{l'}(kr)\, Y_{l'm'}(\hat{k})\, Y^*_{l'm'}(\hat{r}), \tag{9}$$

together with the orthonormality of the spherical harmonics collapses the angular integral onto the single term $(l', m') = (l_\mu, m_\mu)$. A Gaussian basis function therefore keeps its

angular character exactly: an l function transforms into an l function, with a radial part left to do,

$$\tilde{\phi}_\mu(\vec{k}) = 4\pi\,(-i)^{l_\mu}\,X_{l_\mu m_\mu}(\hat{k}) \int_0^\infty r^2\, j_{l_\mu}(kr)\, R_{l_\mu}(r)\, dr. \tag{10}$$

The real solid harmonics X_{lm} replace the complex Y_{lm} here; they diagonalise the angular integral the same way and are what a real-valued implementation wants (Section 5.2).

4.2 The radial integral

Writing $j_l(x) = \sqrt{\pi/2x}\, J_{l+1/2}(x)$ and inserting Eq. (6), each primitive contributes

$$\int_0^\infty r^{l+2}\, j_l(kr)\, e^{-\alpha_p r^2}\, dr = \sqrt{\frac{\pi}{2k}} \int_0^\infty r^{\,l+3/2}\, J_{l+1/2}(kr)\, e^{-\alpha_p r^2}\, dr. \tag{11}$$

The standard integral [8, Eq. 6.631.4]

$$\int_0^\infty r^{\nu+1}\, J_\nu(\beta r)\, e^{-\alpha r^2}\, dr = \frac{\beta^\nu}{(2\alpha)^{\nu+1}}\, e^{-\beta^2/4\alpha}, \tag{12}$$

with $\nu = l + \frac{1}{2}$, $\beta = k$ and $\alpha = \alpha_p$, gives the closed form

$$\int_0^\infty r^2\, j_l(kr)\, e^{-\alpha_p r^2}\, dr = \sqrt{\frac{\pi}{2}}\, \frac{k^l}{(2\alpha_p)^{\,l+3/2}}\, e^{-k^2/4\alpha_p}. \tag{13}$$

A Gaussian is its own Fourier transform up to a width inversion, and Eq. (13) is that statement for an l-th partial wave: a tight primitive (large α_p) becomes a broad feature in k, and a diffuse one becomes a sharp one.

4.3 The result

Collecting constants, the transform of one contracted basis function is

$$\tilde{\phi}_\mu(\vec{k}) \;\propto\; (-i)^{l_\mu}\, X_{l_\mu m_\mu}(\hat{k})\, \underbrace{\left(\frac{k}{2}\right)^{l_\mu} \sum_{p=1}^{M_\mu} c_p\, \alpha_p^{-(l_\mu+3/2)}\, e^{-k^2/4\alpha_p}}_{S_\mu(k)}, \tag{14}$$

where the shell-wide, direction-independent scalar $S_\mu(k)$ absorbs the whole primitive sum. It depends on $|\vec{k}|$ alone, which is what makes the momentum map cheap: on the hemisphere $|\vec{k}|$ is constant, so S_μ is evaluated once per shell rather than once per pixel.

The constant $\pi^{3/2}$ and the harmonic normalisations common to a shell are dropped throughout: maps are normalised to their own maximum and isosurfaces are drawn relative to the field maximum, so only relative amplitudes survive to the screen.

5 The Photoemission Matrix Element

5.1 Atom-major form

Summing Eq. (14) over the basis and grouping by atom,

$$\tilde{\psi}_i(\vec{k}) = \sum_{A=1}^{N_{atom}} e^{-i\vec{k}\cdot\vec{R}_A}\, F_A(\vec{k}), \qquad F_A(\vec{k}) = \sum_{\mu\in A} c_{i\mu}\,(-i)^{l_\mu}\, X_{l_\mu m_\mu}(\hat{k})\, S_\mu(k), \tag{15}$$

with F_A the atomic form factor. The structure factor $e^{-i\vec{k}\cdot\vec{R}_A}$ depends only on where the atom is, so the interference between atoms — the part of the pattern that reports the molecular geometry — is separated cleanly from the atomic form factors, which report the basis. The measured momentum map is $|\tilde{\psi}_i|^2$ restricted to the hemisphere (4) and dressed with the measurement factors of Section 6.

5.2 Real cubic harmonics

Because the map is evaluated on a regular (k_x, k_y) grid and every quantity is real, the real (cubic) solid harmonics $X_{lm}(\hat{k})$ are used, in the ordering the quantum-chemistry code writes its coefficients.[1] With the normalised direction $\hat{k} = (\hat{k}_x, \hat{k}_y, \hat{k}_z)$:

$$l = 0: \quad X_s = \tfrac{1}{2}\sqrt{\tfrac{1}{\pi}}, \tag{16}$$

$$l = 1: \quad X_{p_z} = \tfrac{1}{2}\sqrt{\tfrac{3}{\pi}}\,\hat{k}_z, \quad X_{p_x} = \tfrac{1}{2}\sqrt{\tfrac{3}{\pi}}\,\hat{k}_x, \quad X_{p_y} = \tfrac{1}{2}\sqrt{\tfrac{3}{\pi}}\,\hat{k}_y, \tag{17}$$

$$l = 2: \quad X_{d_{z^2}} = \tfrac{1}{4}\sqrt{\tfrac{5}{\pi}}\,(3\hat{k}_z^2 - 1), \quad X_{d_{xz}}, X_{d_{yz}}, X_{d_{x^2-y^2}}, X_{d_{xy}} \;\propto\; \hat{k}_a\hat{k}_b. \tag{18}$$

They are homogeneous polynomials of degree l in $\hat{k}$; the f and g harmonics follow the same convention and are listed in the kernel source. The ordering within a shell is not a free choice: it must match the order the coefficients were written in. Getting it wrong permutes lobes and produces a plausible, wrong map.

5.3 The $(-i)^l$ phase

The prefactor $(-i)^{l_\mu}$ is purely real or purely imaginary depending on $l \bmod 4$ ($l = 0 \to 1$, $1 \to -i$, $2 \to -1$, $3 \to i$, $4 \to 1$). Contributions of even and odd l therefore never mix in a single accumulator, and no complex arithmetic is needed anywhere: each shell is routed into a real or an imaginary accumulator, and the per-atom phase recombines them as

$$\mathrm{Re}\Big[e^{-i\vec{k}\cdot\vec{R}_A} F_A\Big] = \cos(\vec{k}\cdot\vec{R}_A)\,\mathrm{Re}\,F_A + \sin(\vec{k}\cdot\vec{R}_A)\,\mathrm{Im}\,F_A, \tag{19}$$

and analogously for the imaginary part, before $|\tilde{\psi}_i|^2$ is formed.

[1] Real-solid-harmonic ordering and normalisation follow the ORCA `orca_2json` definition [9], `https://www.faccts.de/docs/orca/6.1/manual/contents/utilitiesvisualization/orca_2json.html`.

6 The Measurement Model

Everything so far is a property of the orbital. What follows is a property of the experiment, and is applied to the finished map.

6.1 Molecular orientation

The molecule is rotated relative to the analyser frame by three Euler angles (ϕ, θ, ψ) in the ZXZ convention. Rather than rotating the molecule, the implementation rotates each sampling vector, $\vec{k} \mapsto \mathsf{R}\vec{k}$, which is the same thing and costs one matrix per map instead of one per atom. For $\psi = 0$,

$$\mathsf{R}(\phi,\theta) = \begin{pmatrix} \cos\phi & \sin\phi & 0 \\ -\sin\phi\cos\theta & \cos\phi\cos\theta & \sin\theta \\ \sin\phi\sin\theta & -\cos\phi\sin\theta & \cos\theta \end{pmatrix}. \tag{20}$$

A momentum map is meaningless without a stated orientation: θ decides whether a planar molecule lies flat, with its π system pointing along the surface normal, or stands upright.

6.2 Light polarisation

The prefactor $|\vec{A}\cdot\vec{k}|^2$ of Eq. (3) modulates the map with a smooth envelope set by the geometry of the light. With the vector potential at polar angle ϑ_A from the surface plane and azimuth φ_A, the two linear components are

$$\vec{A}_p \cdot \vec{k} = k_x \cos\vartheta_A \cos\varphi_A + k_y \cos\vartheta_A \sin\varphi_A + k_z \sin\vartheta_A, \tag{21}$$

$$\vec{A}_s \cdot \vec{k} = -k_x \sin\varphi_A + k_y \cos\varphi_A. \tag{22}$$

The finite escape depth of the photoelectron enters as a damping γ, the reciprocal of the inelastic mean free path, which gives the p-component an imaginary part and so contributes an extra $\gamma^2 \sin^2\vartheta_A$. Writing $g \equiv \gamma \sin\vartheta_A$, the implemented cases are

$$\text{linear } (s\text{-share } s): \quad |\vec{A}\cdot\vec{k}|^2 = s\,(\vec{A}_s\cdot\vec{k})^2 + (1-s)\left[(\vec{A}_p\cdot\vec{k})^2 + g^2\right], \tag{23}$$

$$\text{circular, left/right}: \quad |\vec{A}\cdot\vec{k}|^2 = \tfrac{1}{2}\left[(\vec{A}_p\cdot\vec{k})^2 + g^2\right] + \tfrac{1}{2}(\vec{A}_s\cdot\vec{k})^2 \mp (k_x\sin\varphi_A - k_y\cos\varphi_A)\,g, \tag{24}$$

$$\text{circular dichroism}: \quad |\vec{A}\cdot\vec{k}|^2_R - |\vec{A}\cdot\vec{k}|^2_L = 2\,(k_x\sin\varphi_A - k_y\cos\varphi_A)\,g, \tag{25}$$

$$\text{toroid } (p\text{-polarised}): \quad \vec{A}\cdot\vec{k} = k_\parallel \cos\vartheta_A + k_z \sin\vartheta_A. \tag{26}$$

Three consequences are worth stating. Equation (25) is *signed*, so a dichroism map runs through zero and needs a colour range symmetric about it. It is also proportional to $g = \gamma\sin\vartheta_A$ alone: in this model the circular dichroism comes entirely from the finite

escape depth, and vanishes for an infinite mean free path or for light along the surface plane. The toroid case depends on $|\vec{k}_{\parallel}|$ rather than on its direction, so its envelope is azimuthally symmetric.

The damping itself is taken from the empirical universal curve [10],

$$\lambda\,[\text{Å}] = 10\left(\frac{143}{E^2} + 0.054\sqrt{E}\right), \qquad E = E_{kin}\,[\text{eV}], \qquad \gamma = \frac{1}{\lambda}. \tag{27}$$

This is the one place where an empirical, unit-bearing formula enters, and it is evaluated in eV and Å before being converted.

6.3 Substrate domains

A molecule adsorbed on a crystalline surface occupies several symmetry-equivalent orientations at once, and an experiment on a real crystal measures all of them together. The simulated map is therefore the *incoherent* sum over the domain operations of the surface: rotations about the surface normal, together with the mirror that relates the two enantiomorphic domains. For the three implemented surfaces,

Surface	rotations	domains	
fcc(100)	$0°, 90°, 180°, 270°$	8	(C_{4v})
fcc(110)	$0°, 180°$	4	(C_{2v})
fcc(111)	$0°, 120°, 240°$	6	(C_{3v})

The domain count is the order of the point group generated by those rotations together with the mirror, which is twice the number of rotations in each case. Quoting rotations and domains interchangeably is a common source of confusion.

6.4 Detector resolution

A real analyser integrates over a finite acceptance in momentum. The ideal map is therefore convolved with a Gaussian of full width at half maximum Δk,

$$I_{\text{meas}}(\vec{k}_{\parallel}) = \int I(\vec{k}'_{\parallel})\, G(\vec{k}_{\parallel} - \vec{k}'_{\parallel})\, d^2k'_{\parallel}, \qquad \sigma = \frac{\Delta k}{2\sqrt{2\ln 2}}, \tag{28}$$

truncated at 3σ, where a normalised Gaussian has 0.3 % of its weight left outside. This is a property of the instrument, not of the orbital, and it is what makes a calculated map directly comparable with a measured one instead of obviously sharper.

6.5 What reaches the eye

Photoemission intensity in Eq. (3) is defined only up to a constant, so every map is displayed on a range chosen for it. Scaling each map to its own maximum makes each one legible

and makes none of them comparable with another; a range shared across maps preserves relative intensities at the cost of leaving weak orbitals dark. Intensities are only strictly comparable between orbitals sampled on the *same* hemisphere, that is at constant E_{kin} with the photon energy retuned per orbital, since the transform is evaluated at a different $|\vec{k}|$ otherwise.

7 Excited States: POT for Excitons

Time-resolved photoemission can probe a molecule after optical excitation, and POT extends to that case. This section follows Kern, Windischbacher and Puschnig [5]. The essential new feature is that an exciton is a *correlated* electron–hole pair, so photoemission from it cannot be described by a single orbital.

7.1 The exciton wave function

Linear-response TDDFT [11] returns, for the m-th excited state of excitation energy Ω_m, a set of amplitudes $X_{vc}^{(m)}$ over occupied (valence) orbitals φ_v and unoccupied (conduction) orbitals χ_c. The two-particle wave function is

$$\psi_m(\vec{r}_h, \vec{r}_e) = \sum_{v,c} X_{vc}^{(m)}\, \varphi_v^*(\vec{r}_h)\, \chi_c(\vec{r}_e), \qquad \sum_{v,c} \left|X_{vc}^{(m)}\right|^2 = 1. \tag{29}$$

The normalisation is exact and is what lets a truncated sum report honestly how much of the state it still covers.

7.2 Natural transition orbitals

A singular value decomposition $X = V \Lambda C^{\mathsf{T}}$ turns Eq. (29) into a sum over pairs [12],

$$\psi_m(\vec{r}_h, \vec{r}_e) = \sum_{\lambda} \Lambda_\lambda\, \tilde{\varphi}_\lambda^*(\vec{r}_h)\, \tilde{\chi}_\lambda(\vec{r}_e), \qquad \sum_{\lambda} \Lambda_\lambda^2 = 1. \tag{30}$$

The natural transition orbitals $\tilde{\varphi}_\lambda, \tilde{\chi}_\lambda$ usually compress a state into one or two dominant pairs, and the number of pairs that matter is what distinguishes the cases below.

7.3 Photoemission from an exciton

Removing the electron of the pair leaves an $(N-1)$-electron ion which can be in any of several hole configurations j. Energy conservation for that channel reads

$$E_{kin} = \omega - \left(E_j^{f,N-1} - E_{i,0}^{N}\right) + \Omega_m = \omega - \varepsilon_j + \Omega_m, \tag{31}$$

where $\varepsilon_j > 0$ is the ionisation potential of hole j and ω the probe photon energy. The excitation energy Ω_m enters with a plus sign: the pump has already put that energy into the system, so a photoelectron from an excited molecule comes out correspondingly faster.

Carrying the plane-wave final state through gives the angular distribution of the m-th exciton,

$$I_m(\vec{k}) \;\propto\; \left|\vec{A}\cdot\vec{k}\right|^2 \sum_j \left| \sum_c X_{jc}^{(m)}\, \mathcal{F}[\chi_c](\vec{k}) \right|^2 \delta(\omega - E_{kin} - \varepsilon_j + \Omega_m)\,. \tag{32}$$

7.4 Dyson orbitals, and why one sum is coherent

The inner sum in Eq. (32) defines the j-th Dyson orbital of state m,

$$D_{j,m}(\vec{r}) = \sum_c X_{jc}^{(m)}\, \chi_c(\vec{r}), \tag{33}$$

so that $I_m(\vec{k}) \propto |\vec{A}\cdot\vec{k}|^2 \sum_j |\tilde{D}_{j,m}(\vec{k})|^2$ with each term at its own kinetic energy. The structure of Eq. (32) is the whole physics of the section:

- The sum over conduction orbitals c is over *amplitudes*. Those channels are indistinguishable — they leave the ion in the same hole state — so they interfere, and the pattern is the transform of one coherent superposition rather than a sum of orbital patterns.
- The sum over holes j is over *intensities*. Those channels leave the ion in distinguishable states and, by Eq. (31), at different kinetic energies, so they do not interfere.

Because a Dyson orbital is a linear combination of orbitals in the same basis, it is itself an LCAO expansion,

$$d_\mu^{(j,m)} = \sum_c X_{jc}^{(m)}\, c_{c\mu}, \tag{34}$$

and everything derived in Sections 2–5 applies to it unchanged. An exciton with hundreds of significant (v, c) pairs still costs one transform per photohole, not one per pair.

7.5 Four generic cases

Counting how many holes and how many conduction orbitals carry appreciable weight sorts excitons into four types [5]:

(i) one hole, one electron orbital. A single momentum map at a single kinetic energy, and it is the Fourier transform of that one orbital. The intuitive orbital picture of ground-state POT survives intact.

(ii) several holes, different electron orbitals. The entangled case: a *different* map appears at each kinetic energy. This is the signature that the electron and hole are genuinely correlated.

(iii) several holes, one shared electron orbital. The *same* map appears at two or more kinetic energies.

(iv) one hole, several electron orbitals. One map at one kinetic energy, but of a coherent superposition — so it need not resemble any single orbital.

7.6 The conditional wave function

Equation (29) is a function of two positions and cannot be drawn directly. Fixing the hole at a point $\vec{R}_h$ gives a function of the electron coordinate alone,

$$\Psi(\vec{r}_e;\vec{R}_h) = \sum_c \Big[\sum_v X_{vc}\,\varphi_v(\vec{R}_h)\Big]\chi_c(\vec{r}_e), \tag{35}$$

which is the most literal picture of where the electron sits given where the hole is. It has one trap: a planar molecule lies in the nodal plane of its own π system, so every π hole orbital vanishes at $z = 0$ and the conditional wave function is identically zero there. The hole has to be placed about an Ångström above the plane for the picture to exist at all.

7.7 Amplitudes, weights and truncation

Two numbers are easy to confuse. The amplitude X_{vc} is signed and is what Eq. (33) is built from; the weight X_{vc}^2 is positive and is the only one that sums — to one, over the whole state. A truncation applied to $|X|$ therefore does not correspond simply to one applied to $|X|^2$: a cutoff of 0.05 on the amplitude discards configurations worth less than 0.25 % of the state each, and a photohole can carry a small share of a state and still be only partly resolved within it.

7.8 Beyond the Tamm–Dancoff approximation

Equation (29) assumes the excited state is built from excitations only. A full linear-response solution also carries de-excitation amplitudes Y_{vc}, and Eqs. (32)–(33) do not apply unchanged to such a state. States outside the Tamm–Dancoff approximation are therefore marked and left unmapped rather than mapped approximately.

7.9 Optical absorption

For completeness, the same roots give the optical absorption cross-section,

$$\sigma(E) = C\sum_m f_m\, g(E-\Omega_m), \qquad C = 1.0976\ \text{Å}^2\,\text{eV}, \tag{36}$$

with f_m the oscillator strength in the length representation and g a normalised line shape, so that $\int \sigma \, dE = C \sum_m f_m$. A dark state has $f_m = 0$ and contributes nothing to the curve, but it is still a state, and it can still be reached and mapped by photoemission.

8 Units

All of the above, and the stored data, use Hartree atomic units: positions in Bohr, momenta in Bohr^{-1}, energies in Hartree. This is the native system of a Gaussian basis, since quantum-chemistry codes report exponents α_p in Bohr^{-2}; no conversion factor appears in Eq. (14) or in the phase $\vec{k} \cdot \vec{R}_A$. With $\hbar = m_e = 1$ the free-electron dispersion collapses to the exact, parameter-free $k(E_{kin}) = \sqrt{2E_{kin}}$ of Eq. (4).

The one exception is the universal curve, Eq. (27), which is an empirical fit calibrated in eV and Å and has to be evaluated there.

The Å and eV numbers shown in the web interface are a display-only conversion ($1\,\text{Bohr} = 0.529177\,\text{Å}$, $1\,\text{Hartree} = 27.2114\,\text{eV}$) applied at the boundaries where a human reads a value: the plot axes and the energy listings. Note also the sign convention there: orbital energies are quoted relative to the vacuum level and are therefore *negative* for a bound orbital, so Eq. (31) appears in the interface as $E_{kin} = h\nu + E_{orbital} + \Omega$ with $E_{orbital} = -\varepsilon_j$.

A Implementation

This appendix maps the equations above onto the C kernels in `src/kernels/`, which are compiled to WebAssembly for the browser and to a native library for the Python package. It can be skipped entirely by a reader interested only in the physics.

A.1 Data layout

All molecular data are flat, contiguous, shell-major arrays — atom positions, shells per atom, angular momentum and primitive count per shell, exponents, contraction coefficients and MO coefficients — read once per page from `basis.bin` and `coefficients.bin`. The three kernels share this layout and the harmonic machinery, and differ only in where the transform is sampled and in what can be precomputed. The primitive normalisation N_p of Eq. (7) is folded into the stored contraction coefficients at ingest, so the kernels never apply it.

The shell-offset table `SHELL_START` $= \{0, 1, 4, 9, 16\}$ places member m of a shell of angular momentum l at index `SHELL_START`$[l] + m$ in the harmonics array.

A.2 Momentum map — `momentum_map()`

On the hemisphere $|\vec{k}| = k(E_{kin})$ is constant, so $S_\mu(k)$ of Eq. (14) is the same at every pixel. It is precomputed once per shell, folded together with $(-i)^l$ and the MO coefficient, which reduces the per-pixel cost from $O(N_{pixels}N_{prim})$ to $O(N_{pixels}N_{basis})$. Per pixel the kernel skips points outside the emission horizon, forms k_z, rotates $\vec{k}$, evaluates the harmonics at $\hat{k}$, accumulates F_A per atom, applies the per-atom phase and stores $|\tilde{\psi}_i|^2$. The measurement factors of Section 6 are applied afterwards, to the finished image.

Several orbitals are summed by `momentum_map_sum()`, which sums *intensities* with a weight each: photoelectrons from different initial states do not interfere, and $|\tilde{\psi}|^2$ is not linear in the coefficients, so pre-summing coefficient rows would give a different — and wrong — answer. The one case where amplitudes may be summed first is the Dyson orbital of Eq. (33), which is a genuine coherent superposition.

A.3 Real-space orbital — `scalar_field()`

$\psi_i(\vec{r})$ is evaluated directly from Eq. (6) on a Cartesian grid. Because GTOs decay as $e^{-\alpha r^2}$ the traversal is pruned at three levels: a per-shell cutoff radius from the shell's most diffuse primitive bounds each atom's influence to a box of voxels, shells with negligible coefficients are skipped, and the $2l + 1$ members of a shell share one radial evaluation.

A.4 Momentum density — `momentum_density()`

The same transform, Eq. (15), on a Cartesian $[-k_{max}, k_{max}]^3$ grid, storing $\rho_i(\vec{k}) = |\tilde{\psi}_i(\vec{k})|^2$. Two hemisphere simplifications are lost. First, $|\vec{k}|$ varies per voxel, so S_μ cannot be hoisted; it is tabulated in k per shell and interpolated, which is cheap because it is a smooth sum of Gaussians. Second, there is no emission horizon: the density exists everywhere. At the origin $\hat{k}$ is undefined, but every $l > 0$ term carries k^l and vanishes there, so only s-type functions survive the limit.

Intersecting ρ_i with the hemisphere $|\vec{k}| = k(E_{kin})$ reproduces the momentum map of the same orbital, up to the measurement factors — which is the relation an energy-dependent measurement exploits to reconstruct the third momentum component [13].

A.5 Post-processing — `map_image.c`

Substrate symmetrisation, rotation of the finished image and rescaling to a display range work on a plain $N_k \times N_k$ float image and know nothing about orbitals or basis sets. Right-angle rotations are performed as index permutations rather than by interpolation, so that a symmetrised map does not depend on floating-point rounding at its outermost ring.

B The kernels in full

The four kernels and their three headers are reproduced here in full, so that this document is a complete and self-contained description of what the numbers on the website are: Sections 2–7 say what is computed, Appendix A says how, and this appendix is the thing itself. The listings are included from `src/kernels/` at build time rather than pasted, so they cannot drift away from the code that actually runs.

The kernels are released under the EUPL-1.2. The calculation data served by the website is a separate matter and is not covered by that licence.

Read the headers first: `kernels.h` is the whole public surface, and every entry point is documented there with its units.

B.1 `kernels.h` — the public API

```c
/* PhysikMDB physics kernels - the public API.
 *
 * These turn one LCAO-GTO molecular orbital into the three things the site
 * shows: the 2D photoemission momentum map on the constant-|k| hemisphere,
 * the real-space orbital psi(r), and the momentum-space density |psi~(k)|^2.
 * The Fourier transform is analytic (closed-form radial integral, Rayleigh
 * expansion, real cubic harmonics); see docs/physics-compute.md for what is
 * computed and theory/main.tex for why it is right.
 *
 * Units are Hartree atomic units throughout - Bohr, Bohr^-1, Bohr^-2, Hartree.
 * There is no conversion factor anywhere inside these kernels; a caller
 * working in Angstrom or eV converts at its own boundary. A number arriving
 * here in eV is a bug, not a scale factor to compensate for.
 *
 * The basis arrives as nine flat arrays in the shell-major layout of
 * basis.bin (docs/binary-format.md): atoms in order, each atom's shells in
 * order, and per shell one angular momentum l, one list of shell_Nprim
 * primitives, and 2l+1 consecutive MO coefficients with m in ORCA's order.
 * Every entry point takes them the same way, so one parsed basis serves all
 * three.
 *
 * Standalone use: this is plain C99 with no dependency on emscripten or on
 * the web app. Compile the four .c files in this directory and link with -lm.
 * `make kernels-native` does that.
 */
#ifndef PHYSIKMDB_KERNELS_H
#define PHYSIKMDB_KERNELS_H

/* EMSCRIPTEN_KEEPALIVE marks an export for the emscripten linker and means
   nothing to a native compiler. Defining it away here is why a standalone
   build needs no stub header and no extra include path. */
#ifdef __EMSCRIPTEN__
#include <emscripten.h>
#else
#define EMSCRIPTEN_KEEPALIVE
#endif

/* OpenMP is optional, and native only.
 *
```

```
 * The three grid loops each write disjoint output elements, so
 * `#pragma omp parallel for` over the outer index needs no reduction clause
 * and cannot move a single bit: every element is still the sum of the same
 * terms in the same order. Without -fopenmp the pragmas are ignored and these
 * two macros collapse to one thread, which is the only thing that changes.
 *
 * The browser build never gets it. OpenMP under emscripten needs pthreads,
 * which needs SharedArrayBuffer, which needs the COOP/COEP headers that
 * deploy/security-headers.conf does not set - and the page already uses two
 * cores by running the map and field workers side by side. */
#ifdef _OPENMP
#include <omp.h>
#define KERNEL_MAX_THREADS omp_get_max_threads()
#define KERNEL_THREAD_ID omp_get_thread_num()
#else
#define KERNEL_MAX_THREADS 1
#define KERNEL_THREAD_ID 0
#endif

/* --- momentum_map.c: the photoemission hemisphere --- */

/**
 * The 2D photoemission momentum map I(kx, ky) on the constant-|k| hemisphere
 * k(E_kin) = sqrt(2*E_kin), for one orbital.
 *
 * Output index is `ix * Nk + iy` - kx-major, unlike the two field kernels.
 *
 * The grid is given as origin + spacing, like scalar_field(), rather than a
 * half-extent: a caller who wants to fix the resolution rather than the range
 * then needs no kernel change. For the symmetric grid the page uses today,
 * k0 = -k_max and dk = 2*k_max/(Nk-1).
 *
 * @param E_kin kinetic energy [Hartree]
 * @param k0 momentum of the first grid point [Bohr^-1]
 * @param dk grid spacing [Bohr^-1]
 * @param Nk grid points per axis
 * @param phi Euler angle, ZXZ convention, gamma [deg]
 * @param theta Euler angle, ZXZ convention, beta [deg]
 * @param psi Euler angle, ZXZ convention, alpha [deg]
 * @param Natoms number of atoms
 * @param Nshells shells per atom; length Natoms
 * @param Rx atom x-positions [Bohr]; length Natoms
 * @param Ry atom y-positions [Bohr]; length Natoms
 * @param Rz atom z-positions [Bohr]; length Natoms
 * @param shell_l angular momentum per shell; length sum(Nshells)
 * @param shell_Nprim primitives per shell; length sum(Nshells)
 * @param exponents alpha_p [Bohr^-2]; length sum(shell_Nprim)
 * @param contractions c_p per primitive; length sum(shell_Nprim)
 * @param coefficients MO coefficients c_mu; length sum over shells of (2l+1)
 * @param momentum_map out; length Nk*Nk
 */
EMSCRIPTEN_KEEPALIVE
void momentum_map(
    float E_kin, float k0, float dk, int Nk,
    float phi, float theta, float psi,
    int Natoms, int *restrict Nshells,
    float *restrict Rx, float *restrict Ry, float *restrict Rz,
    int *restrict shell_l, int *restrict shell_Nprim,
```

```
    float *restrict exponents, float *restrict contractions,
    float *restrict coefficients, float *restrict momentum_map);

/**
 * Multiply a momentum map in place by the |A.k|^2 polarisation factor:
 * linear (p/s blend), circular (left/right/CD) or toroid. Zeroes pixels off
 * the hemisphere.
 *
 * @param E_kin kinetic energy [Hartree]
 * @param k0 momentum of the first grid point [Bohr^-1]
 * @param dk grid spacing [Bohr^-1]
 * @param Nk grid points per axis
 * @param polar polar angle of the vector potential A [deg]
 * @param azimuth azimuth angle of the vector potential A [deg]
 * @param polarisation type of polarisation [0 == linear, 1 == circular, 2 == toroid]
 * @param s_share share of s-polarisation in linear polarised light [%]
 * @param handedness handedness of circular polarised light [0 == left, 1 == right, 2 == cd]
 * @param gamma inelastic-mean-free-path damping [Bohr^-1] - the caller's job,
 * not this kernel's: the empirical formula it comes from is calibrated in
 * eV/Angstrom, and there is no conversion factor in here to evaluate it with
 * @param momentum_map in/out; length Nk*Nk
 */
EMSCRIPTEN_KEEPALIVE
void polarisation_factor(float E_kin, float k0, float dk, int Nk,
                         float polar, float azimuth,
                         int polarisation, float s_share, int handedness,
                         float gamma,
                         float *restrict momentum_map);

/**
 * Incoherent sum of several orbitals' momentum maps, each at its own kinetic
 * energy and each post-processed on its own (map, then symmetrisation, then
 * optionally polarisation). Photoelectrons from different initial states do
 * not interfere, so this sums intensities, not amplitudes - it cannot be done
 * by pre-summing coefficient rows.
 *
 * Does NOT normalise; call normalisation() afterwards if wanted.
 *
 * @param Norbitals orbitals to sum; 0 leaves a zeroed map
 * @param E_kin kinetic energy per orbital [Hartree]; <= 0 contributes nothing
 * @param weights weight per orbital, or NULL for all ones
 * @param coefficients Norbitals rows of Nbasis coefficients, row-major
 * @param k0 momentum of the first grid point [Bohr^-1]
 * @param dk grid spacing [Bohr^-1]
 * @param Nk grid points per axis
 * @param phi Euler angle, ZXZ, gamma [deg]
 * @param theta Euler angle, ZXZ, beta [deg]
 * @param psi Euler angle, ZXZ, alpha [deg]
 * @param substrate -1 = none, 0 = fcc100, 1 = fcc110, 2 = fcc111
 * @param apply_polarisation non-zero to apply the |A.k|^2 factor
 * @param polar polar angle of A [deg]
 * @param azimuth azimuth angle of A [deg]
 * @param polarisation type of polarisation [0 == linear, 1 == circular, 2 == toroid]
 * @param s_share share of s-polarisation in linear polarised light [%]
 * @param handedness handedness of circular polarised light [0 == left, 1 == right, 2 == cd]
 * @param gamma inelastic-mean-free-path damping per orbital [Bohr^-1]; length
 * Norbitals - see polarisation_factor() for why the caller computes this
 * @param Natoms number of atoms
```

```
 * @param Nshells shells per atom; length Natoms
 * @param Rx atom x-positions [Bohr]; length Natoms
 * @param Ry atom y-positions [Bohr]; length Natoms
 * @param Rz atom z-positions [Bohr]; length Natoms
 * @param shell_l angular momentum per shell
 * @param shell_Nprim primitives per shell
 * @param exponents alpha_p [Bohr^-2]
 * @param contractions c_p per primitive
 * @param out summed map; length Nk*Nk
 */
EMSCRIPTEN_KEEPALIVE
void momentum_map_sum(
    int Norbitals, float *restrict E_kin, float *restrict weights,
    float *restrict coefficients,
    float k0, float dk, int Nk, float phi, float theta, float psi,
    int substrate, int apply_polarisation, float polar, float azimuth,
    int polarisation, float s_share, int handedness, float *restrict gamma,
    int Natoms, int *restrict Nshells,
    float *restrict Rx, float *restrict Ry, float *restrict Rz,
    int *restrict shell_l, int *restrict shell_Nprim,
    float *restrict exponents, float *restrict contractions,
    float *restrict out);

/* --- map_image.c: post-processing on the finished map --- */

/**
 * Sum rotated and mirrored copies of a momentum map in place, approximating
 * the average over a substrate's rotational and mirror domains.
 *
 * @param Nk grid points per axis
 * @param substrate -1 = none (no-op), 0 = fcc100, 1 = fcc110, 2 = fcc111
 * @param momentum_map in/out; length Nk*Nk
 */
EMSCRIPTEN_KEEPALIVE
void symmetrisation(int Nk, int substrate, float *restrict momentum_map);

/**
 * Scale a momentum map in place so its maximum becomes 255 (image data).
 * An all-zero map is left alone rather than turned into NaN.
 *
 * Deliberately a separate call, not folded into momentum_map(): showing
 * un-normalised intensities is a wanted option (issue #62).
 *
 * @param Nk grid points per axis
 * @param momentum_map in/out; length Nk*Nk
 */
EMSCRIPTEN_KEEPALIVE
void normalisation(int Nk, float *restrict momentum_map);

/* --- realspace.c --- */

/**
 * The real-space orbital psi(r) on a Cartesian grid, for one orbital.
 *
 * Output index is `ix + iy*N + iz*N*N` - x fastest, matching three.js's
 * MarchingCubes layout.
 *
 * @param N grid points per axis (cubic grid)
```

```
 * @param x0 grid origin x [Bohr]
 * @param y0 grid origin y [Bohr]
 * @param z0 grid origin z [Bohr]
 * @param spacing grid spacing [Bohr]
 * @param Natoms number of atoms
 * @param Nshells shells per atom; length Natoms
 * @param Rx atom x-positions [Bohr]; length Natoms
 * @param Ry atom y-positions [Bohr]; length Natoms
 * @param Rz atom z-positions [Bohr]; length Natoms
 * @param shell_l angular momentum per shell; length sum(Nshells)
 * @param shell_Nprim primitives per shell; length sum(Nshells)
 * @param exponents alpha_p [Bohr^-2]; length sum(shell_Nprim)
 * @param contractions c_p per primitive; length sum(shell_Nprim)
 * @param coefficients MO coefficients c_mu; length sum over shells of (2l+1)
 * @param field out; length N^3
 */
EMSCRIPTEN_KEEPALIVE
void scalar_field(
    int N, float x0, float y0, float z0, float spacing,
    int Natoms, int *restrict Nshells,
    float *restrict Rx, float *restrict Ry, float *restrict Rz,
    int *restrict shell_l, int *restrict shell_Nprim,
    float *restrict exponents, float *restrict contractions,
    float *restrict coefficients, float *restrict field);

/**
 * sum_i w_i |psi_i(r)|^2 over several orbitals. Squares each orbital's own
 * field, since |psi|^2 is not linear in the coefficients. For one orbital's
 * signed psi(r) call scalar_field() directly.
 *
 * @param N grid points per axis (cubic grid)
 * @param x0 grid origin x [Bohr]
 * @param y0 grid origin y [Bohr]
 * @param z0 grid origin z [Bohr]
 * @param spacing grid spacing [Bohr]
 * @param Norbitals orbitals to sum; 0 leaves a zeroed field
 * @param weights weight per orbital, or NULL for all ones
 * @param coefficients Norbitals rows of Nbasis coefficients, row-major
 * @param Natoms number of atoms
 * @param Nshells shells per atom; length Natoms
 * @param Rx atom x-positions [Bohr]; length Natoms
 * @param Ry atom y-positions [Bohr]; length Natoms
 * @param Rz atom z-positions [Bohr]; length Natoms
 * @param shell_l angular momentum per shell
 * @param shell_Nprim primitives per shell
 * @param exponents alpha_p [Bohr^-2]
 * @param contractions c_p per primitive
 * @param out summed density; length N^3
 */
EMSCRIPTEN_KEEPALIVE
void charge_density_sum(
    int N, float x0, float y0, float z0, float spacing,
    int Norbitals, float *restrict weights, float *restrict coefficients,
    int Natoms, int *restrict Nshells,
    float *restrict Rx, float *restrict Ry, float *restrict Rz,
    int *restrict shell_l, int *restrict shell_Nprim,
    float *restrict exponents, float *restrict contractions,
    float *restrict out);
```

```c

/* --- momentumspace.c --- */

/**
 * The momentum-space density |psi~(k)|^2 on a Cartesian k-grid, for one
 * orbital. Same index convention as scalar_field().
 *
 * @param N grid points per axis (cubic grid)
 * @param k0 momentum of the first grid point, each axis [Bohr^-1]
 * @param dk grid spacing [Bohr^-1]
 * @param Natoms number of atoms
 * @param Nshells shells per atom; length Natoms
 * @param Rx atom x-positions [Bohr]; length Natoms
 * @param Ry atom y-positions [Bohr]; length Natoms
 * @param Rz atom z-positions [Bohr]; length Natoms
 * @param shell_l angular momentum per shell; length sum(Nshells)
 * @param shell_Nprim primitives per shell; length sum(Nshells)
 * @param exponents alpha_p [Bohr^-2]; length sum(shell_Nprim)
 * @param contractions c_p per primitive; length sum(shell_Nprim)
 * @param coefficients MO coefficients c_mu; length sum over shells of (2l+1)
 * @param field out; length N^3
 */
EMSCRIPTEN_KEEPALIVE
void momentum_density(
    int N, float k0, float dk,
    int Natoms, int *restrict Nshells,
    float *restrict Rx, float *restrict Ry, float *restrict Rz,
    int *restrict shell_l, int *restrict shell_Nprim,
    float *restrict exponents, float *restrict contractions,
    float *restrict coefficients, float *restrict field);

/**
 * sum_i w_i |psi~_i(k)|^2 over several orbitals, for the same reason
 * charge_density_sum() squares per orbital.
 *
 * @param N grid points per axis (cubic grid)
 * @param k0 momentum of the first grid point, each axis [Bohr^-1]
 * @param dk grid spacing [Bohr^-1]
 * @param Norbitals orbitals to sum; 0 leaves a zeroed field
 * @param weights weight per orbital, or NULL for all ones
 * @param coefficients Norbitals rows of Nbasis coefficients, row-major
 * @param Natoms number of atoms
 * @param Nshells shells per atom; length Natoms
 * @param Rx atom x-positions [Bohr]; length Natoms
 * @param Ry atom y-positions [Bohr]; length Natoms
 * @param Rz atom z-positions [Bohr]; length Natoms
 * @param shell_l angular momentum per shell
 * @param shell_Nprim primitives per shell
 * @param exponents alpha_p [Bohr^-2]
 * @param contractions c_p per primitive
 * @param out summed density; length N^3
 */
EMSCRIPTEN_KEEPALIVE
void momentum_density_sum(
    int N, float k0, float dk,
    int Norbitals, float *restrict weights, float *restrict coefficients,
    int Natoms, int *restrict Nshells,
    float *restrict Rx, float *restrict Ry, float *restrict Rz,
```

```
    int *restrict shell_l, int *restrict shell_Nprim,
    float *restrict exponents, float *restrict contractions,
    float *restrict out);

#endif
```

B.2 `harmonics.h` — the real cubic harmonics

Section 5.2. The 25 harmonics for $l = 0 \dots 4$, with the shell-offset table that indexes them.

```
/* Real cubic harmonics, shared by all three kernels.
 *
 * The table was copy-pasted three times before this header. Two copies (the
 * momentum map and the momentum density) wrote the unit-vector form, valid
 * only on |k| = 1; the real-space copy wrote the general solid-harmonic form
 * with an explicit r^2. They are the same polynomials - the general form
 * evaluated at r^2 = 1 is the unit-vector form - so one helper covers all
 * three, with the k-space callers passing the literal 1.0f.
 *
 * That literal matters. Passing kx^2 + ky^2 + kz^2 instead would be "the same"
 * only in exact arithmetic; in float it differs from 1 in the last bits and
 * shifts every d, f and g harmonic. tests/test_wasm_physics.py has a
 * regression test for exactly that mistake.
 *
 * Normalisations follow ORCA's real solid harmonic convention:
 *
     https://www.faccts.de/docs/orca/6.1/manual/contents/utilitiesvisualization/orca_2json.html#definition-of-the-real-
 */
#ifndef PHYSIKMDB_HARMONICS_H
#define PHYSIKMDB_HARMONICS_H

#define C_S 0.28209479177387814 // 1/2 * sqrt(1 / pi)
#define C_P 0.4886025119029199 // 1/2 * sqrt(3 / pi)
#define C_D 1.0925484305920792 // 1/2 * sqrt(15 / pi)
#define C_F 2.890611442640554 // 1/2 * sqrt(105 / pi)
#define C_G 5.006685883593409 // 3/2 * sqrt(35 / pi)

// First cubic-harmonic index of each shell (s, p, d, f, g); a shell of angular
// momentum l occupies SHELL_START[l] .. SHELL_START[l] + 2l
static const int SHELL_START[5] = {0, 1, 4, 9, 16};

// The highest angular momentum the table covers (g)
#define MAX_ANGULAR_MOMENTUM 4

/**
 * Evaluate the real cubic harmonics at one point, up to a given shell.
 *
 * Filled cumulatively: only entries belonging to shells the caller actually
 * needs are written, since for one orbital most of a basis set contributes
 * nothing. `max_l` is a parameter rather than something derived here because
 * the three kernels each decide it differently - over every shell, over the
 * significant shells only, or per atom.
 *
 * @param x [Bohr or Bohr^-1] first Cartesian component
 * @param y [Bohr or Bohr^-1] second Cartesian component
 * @param z [Bohr or Bohr^-1] third Cartesian component
 * @param r2 [x^2 + y^2 + z^2] in the same units squared; pass the literal
 * 1.0f when x/y/z are already a unit vector
```

```
 * @param max_l highest angular momentum to fill (0 = s only, 4 = through g)
 * @param H out; entries SHELL_START[0] .. SHELL_START[max_l] + 2*max_l written
 */
static inline void cubic_harmonics(float x, float y, float z, float r2,
                                   int max_l, float H[25])
{
    H[0] = C_S; // s

    if (max_l >= 1)
    {
        H[1] = C_P * z; // pz
        H[2] = C_P * x; // px
        H[3] = C_P * y; // py
    }

    if (max_l >= 2)
    {
        float x2 = x * x;
        float y2 = y * y;
        float z2 = z * z;

        H[4] = C_D * 0.28867513459481287f * (3.0f * z2 - r2); // dz2
        H[5] = C_D * x * z; // dxz
        H[6] = C_D * y * z; // dyz
        H[7] = C_D * 0.5f * (x2 - y2); // dx2-y2
        H[8] = C_D * x * y; // dxy

        if (max_l >= 3)
        {
            H[9] = C_F * 0.12909944487358058f * z * (5.0f * z2 - 3.0f * r2); // fz3
            H[10] = C_F * 0.15811388300841897f * x * (5.0f * z2 - r2); // fxz2
            H[11] = C_F * 0.15811388300841897f * y * (5.0f * z2 - r2); // fyz2
            H[12] = C_F * 0.5f * z * (x2 - y2); // fz(x2-y2)
            H[13] = C_F * x * y * z; // fxyz
            H[14] = -C_F * 0.20412414523193148f * x * (x2 - 3.0f * y2); // fx(x2-3y2)
            H[15] = -C_F * 0.20412414523193148f * y * (3.0f * x2 - y2); // fy(3x2-y2)
        }

        if (max_l >= 4)
        {
            float r4 = r2 * r2;

            H[16] = C_G * 0.021128856368212916f * (35.0f * z2 * z2 - 30.0f * z2 * r2 + 3.0f *
                r4); // gz4
            H[17] = C_G * 0.1336306209562122f * x * z * (7.0f * z2 - 3.0f * r2); // gxz3
            H[18] = C_G * 0.1336306209562122f * y * z * (7.0f * z2 - 3.0f * r2); // gyz3
            H[19] = C_G * 0.09449111825230681f * (x2 - y2) * (7.0f * z2 - r2); // gx2-y2
            H[20] = C_G * 0.18898223650461363f * x * y * (7.0f * z2 - r2); // gxy
            H[21] = -C_G * 0.3535533905932738f * x * z * (x2 - 3.0f * y2); // gx(x2-3y2)
            H[22] = -C_G * 0.3535533905932738f * y * z * (3.0f * x2 - y2); // gy(3x2-y2)
            H[23] = -C_G * 0.125f * (x2 * x2 - 6.0f * x2 * y2 + y2 * y2); // gx4-6x2y2+y4
            H[24] = -C_G * 0.5f * x * y * (x2 - y2); // gxy(x2-y2)
        }
    }
}

#endif
```

B.3 `basis.h` — the basis walk and the radial evaluators

The two radial amplitudes: the real-space one, and $S_\mu(k)$ of Eq. (14), which is the closed form derived in Section 4.

```c
/* The LCAO-GTO basis set, and how one shell is evaluated.
 *
 * All three kernels take the same nine flat arrays straight out of basis.bin
 * (see docs/binary-format.md) and walked them with their own ishell/iprim/
 * icoeff running counters - three copies of the same bookkeeping, and the
 * largest index-bug surface in the kernels. `Basis` bundles the arrays;
 * `radial_*` are the two per-shell radial amplitudes, which were written out
 * twice each.
 *
 * The layout is shell-major: atoms in order, each atom's shells in order, and
 * per shell one angular momentum l, one list of shell_Nprim primitives, and
 * 2l+1 consecutive MO coefficients (m in ORCA's order). So walking it means
 * carrying three offsets at once - the shell, its first primitive, and its
 * first coefficient.
 *
 * Everything here is Hartree atomic units: Bohr, Bohr^-1, Bohr^-2.
 */
#ifndef PHYSIKMDB_BASIS_H
#define PHYSIKMDB_BASIS_H

#include <math.h>

/**
 * One orbital's basis set. Borrows the caller's arrays - it copies nothing and
 * owns nothing, so they must outlive it.
 */
typedef struct
{
    int Natoms;
    int Nshells; // total over all atoms, i.e. sum(Nshells_per_atom)
    int Nbasis; // sum of 2l+1 over all shells; the length of one coefficient row

    const int *Nshells_per_atom; // length Natoms
    const float *Rx; // atom x-positions [Bohr]; length Natoms
    const float *Ry; // atom y-positions [Bohr]; length Natoms
    const float *Rz; // atom z-positions [Bohr]; length Natoms
    const int *shell_l; // angular momentum per shell; length Nshells
    const int *shell_Nprim; // primitives per shell; length Nshells
    const float *exponents; // alpha_p [Bohr^-2]; length sum(shell_Nprim)
    const float *contractions; // c_p per primitive; length sum(shell_Nprim)
} Basis;

/**
 * Bundle the flat arrays, counting the shell and basis-function totals the
 * kernels each used to count for themselves.
 *
 * @param Natoms number of atoms
 * @param Nshells_per_atom shells per atom; length Natoms
 * @param Rx x-positions [Bohr]; length Natoms
 * @param Ry y-positions [Bohr]; length Natoms
 * @param Rz z-positions [Bohr]; length Natoms
 * @param shell_l angular momentum per shell
 * @param shell_Nprim primitives per shell
```

```
 * @param exponents alpha_p [Bohr^-2]
 * @param contractions c_p per primitive
 * @return the bundle, with Nshells and Nbasis filled in
 */
static inline Basis basis_from_arrays(
    int Natoms,
    const int *Nshells_per_atom,
    const float *Rx, const float *Ry, const float *Rz,
    const int *shell_l, const int *shell_Nprim,
    const float *exponents, const float *contractions)
{
    Basis basis;
    basis.Natoms = Natoms;
    basis.Nshells_per_atom = Nshells_per_atom;
    basis.Rx = Rx;
    basis.Ry = Ry;
    basis.Rz = Rz;
    basis.shell_l = shell_l;
    basis.shell_Nprim = shell_Nprim;
    basis.exponents = exponents;
    basis.contractions = contractions;

    basis.Nshells = 0;
    for (int a = 0; a < Natoms; a++)
    {
        basis.Nshells += Nshells_per_atom[a];
    }

    basis.Nbasis = 0;
    for (int s = 0; s < basis.Nshells; s++)
    {
        basis.Nbasis += 2 * shell_l[s] + 1;
    }

    return basis;
}

/* The two approximations, shared by all three kernels.
 *
 * Both are RELATIVE AMPLITUDES: a contribution below this fraction of what it
 * would otherwise be is dropped. So smaller is always closer to exact, and
 * always slower - which is the only reason they exist, since most of a large
 * basis set contributes nothing to any single orbital.
 *
 * They used to differ per kernel, and the difference was not a considered
 * accuracy budget so much as drift: the momentum map applied no cutoff at all,
 * momentumspace.c cut primitives at exp(-30) = 9e-14, and realspace.c at
 * exp(-7) = 9e-4 - which, because it turns that into a cutoff radius
 * sqrt(eps/alpha), discarded up to 5% of a g shell's peak amplitude. Being
 * consistent and being right both point the same way here.
 *
 * The primitive cutoff was written as the exponent argument rather than the
 * amplitude, which made it read backwards: larger meant more exact. Stating
 * the amplitude and deriving the exponent is the same arithmetic under a name
 * that means what it says. */
#define PRIMITIVE_EPSILON 1e-13f // exp(-x) below this: primitive dropped
#define COEFFICIENT_EPSILON 1e-6f // |c_mu| below this: whole shell dropped

```

```c
/**
 * The exponent argument at which a primitive reaches PRIMITIVE_EPSILON.
 *
 * `alpha_p * r^2` in real space, `k^2 / (4 alpha_p)` in momentum space; both
 * enter as exp(-x). Derived rather than written down twice, so the two cannot
 * drift apart.
 *
 * @return the cutoff in x, i.e. -log(PRIMITIVE_EPSILON)
 */
static inline float primitive_exponent_cutoff(void)
{
    return -logf(PRIMITIVE_EPSILON);
}

/**
 * The radial amplitude of one shell in real space: sum_p c_p exp(-alpha_p r^2).
 *
 * The 2l+1 members of a shell share it, so it is evaluated once per shell and
 * per point, not once per basis function.
 *
 * @param basis the basis set
 * @param shell shell index, 0 .. Nshells-1
 * @param first_primitive index of the shell's first primitive in `exponents`
 * @param r2 squared distance from the shell's atom [Bohr^2]
 * @param cutoff skip primitives whose alpha_p * r^2 reaches this; normally
 * primitive_exponent_cutoff()
 * @return the radial amplitude
 */
static inline float radial_realspace(const Basis *basis, int shell,
                                     int first_primitive, float r2, float cutoff)
{
    float radial = 0.0f;

    for (int p = 0; p < basis->shell_Nprim[shell]; p++)
    {
        float exponent = basis->exponents[first_primitive + p] * r2;

        // Skip primitives that have underflowed to zero anyway
        if (exponent < cutoff)
        {
            radial += basis->contractions[first_primitive + p] * expf(-exponent);
        }
    }

    return radial;
}

/**
 * The radial amplitude of one shell in momentum space:
 * (k/2)^l * sum_p c_p alpha_p^(-l-3/2) exp(-k^2 / 4 alpha_p).
 *
 * This is the closed-form radial Fourier integral of a contracted Gaussian
 * (see theory/main.tex); the momentum map evaluates it once per shell at the
 * hemisphere's fixed |k|, the momentum density tabulates it over k.
 *
 * @param basis the basis set
 * @param shell shell index, 0 .. Nshells-1
 * @param first_primitive index of the shell's first primitive in `exponents`
```

```
 * @param k momentum magnitude [Bohr^-1]
 * @param cutoff skip primitives whose k^2 / (4 alpha_p) reaches this;
 * normally primitive_exponent_cutoff()
 * @return the radial amplitude
 */
static inline float radial_momentum(const Basis *basis, int shell,
                                    int first_primitive, float k, float cutoff)
{
    int l = basis->shell_l[shell];
    float k2 = k * k;
    float radial = 0.0f;

    for (int p = 0; p < basis->shell_Nprim[shell]; p++)
    {
        float exponent = k2 / (4.0f * basis->exponents[first_primitive + p]);

        if (exponent < cutoff)
        {
            radial += basis->contractions[first_primitive + p] *
                      powf(basis->exponents[first_primitive + p], -l - 1.5f) *
                      expf(-exponent);
        }
    }

    // Factor independent of alpha_p and c_p. Dividing by 2^l is exact, so it
    // does not matter that the two callers used to bracket this differently.
    return radial * (powf(k, l) / powf(2.0f, l));
}

#endif
```

B.4 momentum_map.c — the momentum map and the polarisation factor

Equations (15) and (4), plus the polarisation factor of Section 6.2.

```
/* The 2D photoemission momentum map, on the constant-|k| hemisphere.
 *
 * This is the ARPES simulation proper: the analytic Fourier transform of the
 * rotated orbital evaluated where photoemission puts it, plus the detector's
 * polarisation factor. Post-processing of the finished image - substrate
 * domains, scaling to image range - is map_image.c.
 */
#include <math.h>
#include <stdlib.h>

#include "basis.h"
#include "harmonics.h"
#include "kernels.h"

// Degrees to radians
#define DEG_TO_RAD 0.01745329252

/**
 * Multiply a momentum map by the |A.k|^2 polarisation factor: linear (p/s
 * blend), circular (left/right/CD) or toroid.
 * @param E_kin kinetic energy [Hartree]
 * @param k0 momentum of the first grid point [Bohr^-1]
```

```
 * @param dk grid spacing [Bohr^-1]
 * @param Nk momentum map grid size (per axis)
 * @param polar polar angle of the vector potential A [deg]
 * @param azimuth azimuth angle of the vector potential A [deg]
 * @param polarisation type of polarisation [0 == linear, 1 == circular, 2 == toroid]
 * @param s_share share of s-polarisation in linear polarised light [%]
 * @param handedness handedness of circular polarised light [0 == left, 1 == right, 2 == cd]
 * @param gamma inelastic-mean-free-path damping [Bohr^-1], the reciprocal of
 * the mean free path. The empirical "universal curve" it comes from
 * (Seah and Dench: lambda[Angstrom] = 10*(143/E^2 + 0.054*sqrt(E)), E in eV)
 * is calibrated in eV/Angstrom, so - unlike every other parameter here - it
 * cannot be evaluated in this kernel's Hartree/Bohr units without a
 * conversion factor; the caller (map-worker.js) computes it once per
 * orbital, next to the other eV<->Hartree conversions at the JS/WASM
 * boundary, and passes the Bohr^-1 result straight in
 * @param momentum_map in/out; length Nk*Nk
 */
EMSCRIPTEN_KEEPALIVE
void polarisation_factor(
    float E_kin,
    float k0,
    float dk,
    int Nk,
    float polar,
    float azimuth,
    int polarisation,
    float s_share,
    int handedness,
    float gamma,
    float *restrict momentum_map)
{
    s_share /= 100;
    float k = sqrtf(2.0f * E_kin); // Radius of photoemission sphere [Bohr^-1]; E = k^2/2 in
        atomic units
    float k2 = k * k;

    // For later
    float cos_polar = cos(polar * DEG_TO_RAD);
    float cos_azimuth = cos(azimuth * DEG_TO_RAD);
    float sin_polar = sin(polar * DEG_TO_RAD);
    float sin_azimuth = sin(azimuth * DEG_TO_RAD);

    float gs = gamma * sin_polar;
    float gs2 = gs * gs;

    // Loop over kx (== rows)
    for (int ix = 0; ix < Nk; ix++)
    {
        float kx = k0 + ix * dk; // x-Position in reciprocal space [Bohr^-1]

        // Loop over ky (== columns)
        for (int iy = 0; iy < Nk; iy++)
        {
            int i = ix * Nk + iy; // Index in the grid
            float ky = k0 + iy * dk; // y-Position in reciprocal space [Bohr^-1]

            // Outside photoemission sphere set map to 0 and skip pixel calculation
            float kz2 = k2 - kx * kx - ky * ky;
```

```
80
81          if (kz2 < 0)
82          {
83              momentum_map[i] = 0.0f;
84              continue;
85          }
86          float kz = sqrtf(kz2); // z- Position in reciprocal space [Bohr^-1]
87
88          // Only NanoESCA p-polarisation for now
89          float Ak2;
90          switch (polarisation)
91          {
92          case 0: // linear
93          {
94              float Ak_p = kx * cos_polar * cos_azimuth + ky * cos_polar * sin_azimuth + kz *
                    sin_polar;
95              float Ak_s = -kx * sin_azimuth + ky * cos_azimuth;
96              Ak2 = s_share * Ak_s * Ak_s + (1 - s_share) * (Ak_p * Ak_p + gs2);
97              break;
98          }
99          case 1: // circular
100             switch (handedness)
101             {
102             case 0: // left-handed
103             {
104                 float Ak_p = kx * cos_polar * cos_azimuth + ky * cos_polar * sin_azimuth + kz
                        * sin_polar;
105                 float Ak_s = -kx * sin_azimuth + ky * cos_azimuth;
106                 Ak2 = 0.5 * (Ak_p * Ak_p + gs2) + 0.5 * Ak_s * Ak_s - (sin_azimuth * kx -
                        cos_azimuth * ky) * gs;
107                 break;
108             }
109             case 1: // right-handed
110             {
111                 float Ak_p = kx * cos_polar * cos_azimuth + ky * cos_polar * sin_azimuth + kz
                        * sin_polar;
112                 float Ak_s = -kx * sin_azimuth + ky * cos_azimuth;
113                 Ak2 = 0.5 * (Ak_p * Ak_p + gs2) + 0.5 * Ak_s * Ak_s + (sin_azimuth * kx -
                        cos_azimuth * ky) * gs;
114                 break;
115             }
116             default: // cd
117             {
118                 Ak2 = 2 * (sin_azimuth * kx - cos_azimuth * ky) * gs;
119                 break;
120             }
121             }
122             break;
123         default: // toroid
124         {
125             float k_par = sqrtf(k2 - kz2);
126             float Ak = k_par * cos_polar + kz * sin_polar;
127             Ak2 = Ak * Ak;
128             break;
129         }
130         }
131
132         momentum_map[i] *= Ak2;
```

```
        }
    }
}

/**
 * Compute the 2D photoemission momentum map I(kx, ky) on the constant-|k|
 * hemisphere k(E_kin) = sqrt(2*E_kin), via the analytic Fourier transform of
 * the rotated LCAO-GTO orbital (see docs/physics-compute.md). All lengths/
 * energies are Hartree atomic units. Parameters are documented inline below;
 * the basis arrives shell-major (see docs/binary-format.md): per shell one
 * angular momentum, one primitive list, and 2l+1 consecutive MO coefficients
 * (atom-major -> shell -> m in ORCA order).
 * @return via the `momentum_map` out-parameter; no return value
 */
EMSCRIPTEN_KEEPALIVE
void momentum_map(
    float E_kin, // Kinetic energy [Hartree]
    float k0, // Momentum of the first grid point [Bohr^-1]
    float dk, // Grid spacing [Bohr^-1]
    int Nk, // Momentum map grid size
    float phi, // Euler angle for molecular rotation, ZXZ convention, gamma [deg]
    float theta, // Euler angle for molecular rotation, ZXZ convention, beta [deg]
    float psi, // Euler angle for molecular rotation, ZXZ convention, alpha [deg]
    int Natoms, // Number of atoms
    int *restrict Nshells, // Shells per atom; length Natoms
    float *restrict Rx, // x-Positions of the atoms [Bohr]; length Natoms
    float *restrict Ry, // y-Positions of the atoms [Bohr]; length Natoms
    float *restrict Rz, // z-Positions of the atoms [Bohr]; length Natoms
    int *restrict shell_l, // Angular momentum per shell; length sum(Nshells)
    int *restrict shell_Nprim, // Primitives per shell; length sum(Nshells)
    float *restrict exponents, // Exponent alpha_p [Bohr^-2]; length sum(shell_Nprim)
    float *restrict contractions, // Contraction c_p per primitive; length sum(shell_Nprim)
    float *restrict coefficients, // MO coefficients c_mu; length sum over shells of (2l+1)
    float *restrict momentum_map) // Final momentum map; length Nk*Nk)
{
    Basis basis = basis_from_arrays(Natoms, Nshells, Rx, Ry, Rz,
                                    shell_l, shell_Nprim, exponents, contractions);

    float k = sqrtf(2.0f * E_kin); // Radius of photoemission sphere [Bohr^-1]; E = k^2/2 in
        atomic units
    float k2 = k * k;
    float k_inv = 1.0f / k; // Inverse of k for later

    float cos_phi = cos(-phi * DEG_TO_RAD);
    float cos_theta = cos(-theta * DEG_TO_RAD);
    float cos_psi = cos(-psi * DEG_TO_RAD);
    float sin_phi = sin(-phi * DEG_TO_RAD);
    float sin_theta = sin(-theta * DEG_TO_RAD);
    float sin_psi = sin(-psi * DEG_TO_RAD);

    // ZXZ convention, alpha = psi, beta = theta, gamma = phi
    float R_00 = cos_psi * cos_phi - cos_theta * sin_psi * sin_phi;
    float R_01 = -cos_psi * sin_phi - cos_theta * cos_phi * sin_psi;
    float R_02 = sin_psi * sin_theta;
    float R_10 = cos_phi * sin_psi + cos_psi * cos_theta * sin_phi;
    float R_11 = cos_psi * cos_theta * cos_phi - sin_psi * sin_phi;
    float R_12 = -cos_psi * sin_theta;
    float R_20 = sin_theta * sin_phi;
```

```
    float R_21 = cos_phi * sin_theta;
    float R_22 = cos_theta;

    // Pre-compute the k-radius radial weight per basis function (constant on
    // the photoemission sphere) and its cubic-harmonic index; the (-i)^l
    // factor sorts each shell into the real or imaginary part
    // On the heap, not the stack: these are sized by the basis set, and a
    // large calculation (FePc has 1342 basis functions) would otherwise
    // silently overrun the small fixed WASM stack. One allocation carved into
    // four arrays - int and float are both 4 bytes here, so no padding.
    size_t scratch_floats = (size_t)Natoms + 3 * (size_t)basis.Nbasis;
    int *scratch = (int *)malloc(scratch_floats * sizeof(int));
    if (scratch == NULL)
    {
        for (int j = 0; j < Nk * Nk; j++)
        {
            momentum_map[j] = 0.0f;
        }
        return;
    }
    int *atom_Nbasis = scratch; // Basis functions per atom (for the pixel loop)
    float *basis_weights_real = (float *)(atom_Nbasis + Natoms);
    float *basis_weights_imag = basis_weights_real + basis.Nbasis;
    int *H_index = (int *)(basis_weights_imag + basis.Nbasis);

    // Only shells that contribute are written, so the pixel loop walks a
    // compacted list: for one orbital most of a large basis set is below
    // COEFFICIENT_EPSILON, and that work was previously done 62 500 times over.
    float cutoff_exponent = primitive_exponent_cutoff();
    int max_l = 0;

    int ishell = 0;
    int iprim = 0;
    int icoeff = 0;
    int ikept = 0;
    for (int a = 0; a < Natoms; a++)
    {
        int nkept = 0;
        for (int s = 0; s < Nshells[a]; s++)
        {
            int l = shell_l[ishell];
            int members = 2 * l + 1;

            int shell_significant = 0;
            for (int m = 0; m < members; m++)
            {
                if (fabsf(coefficients[icoeff + m]) > COEFFICIENT_EPSILON)
                {
                    shell_significant = 1;
                }
            }

            if (shell_significant)
            {
                if (l > max_l)
                {
                    max_l = l;
                }
```

```
248
249                 // The 2l+1 members of a shell share one radial sum
250                 float radial = radial_momentum(&basis, ishell, iprim, k, cutoff_exponent);
251
252                 for (int m = 0; m < members; m++)
253                 {
254                     float sum = coefficients[icoeff + m] * radial;
255
256                     // (-i)^l sorts the contribution into the real or imaginary
257                     // part. Modulo 4, as in momentumspace.c: an l the harmonics
258                     // table does not cover would otherwise fall through and
259                     // leave these two uninitialised.
260                     switch (l % 4)
261                     {
262                     case 0: //(-i)^0 = 1
263                         basis_weights_real[ikept] = sum;
264                         basis_weights_imag[ikept] = 0.0f;
265                         break;
266                     case 1: //(-i)^1 = -i
267                         basis_weights_real[ikept] = 0.0f;
268                         basis_weights_imag[ikept] = -sum;
269                         break;
270                     case 2: //(-i)^2 = -1
271                         basis_weights_real[ikept] = -sum;
272                         basis_weights_imag[ikept] = 0.0f;
273                         break;
274                     default: //(-i)^3 = i
275                         basis_weights_real[ikept] = 0.0f;
276                         basis_weights_imag[ikept] = sum;
277                         break;
278                     }
279
280                     H_index[ikept] = SHELL_START[l] + m;
281                     ikept++;
282                     nkept++;
283                 }
284             }
285
286             iprim += shell_Nprim[ishell];
287             icoeff += members;
288             ishell++;
289         }
290         atom_Nbasis[a] = nkept;
291     }
292
293     // Loop over kx (== rows). Each ix owns its own row of the map and reads
294     // nothing the others write, so splitting it across threads leaves every
295     // pixel the sum of the same terms in the same order (see kernels.h).
296 #pragma omp parallel for schedule(static)
297     for (int ix = 0; ix < Nk; ix++)
298     {
299         float kx = k0 + ix * dk; // x-Position in reciprocal space [Bohr^-1]
300
301         // Loop over ky (== columns)
302         for (int iy = 0; iy < Nk; iy++)
303         {
304             int i = ix * Nk + iy; // Index in the grid
305             float ky = k0 + iy * dk; // y-Position in reciprocal space [Bohr^-1]
```

```
306
307             // Outside photoemission sphere set map to 0 and skip pixel calculation
308             float kz2 = k2 - kx * kx - ky * ky;
309             if (kz2 < 0)
310             {
311                 momentum_map[i] = 0.0f;
312                 continue;
313             }
314             float kz = sqrtf(kz2); // z- Position in reciprocal space [Bohr^-1]
315
316             // Rotate the k point
317             float kx_rot = R_00 * kx + R_10 * ky + R_20 * kz;
318             float ky_rot = R_01 * kx + R_11 * ky + R_21 * kz;
319             float kz_rot = R_02 * kx + R_12 * ky + R_22 * kz;
320
321             // Calculate k_hat
322             float kx_hat = kx_rot * k_inv; // Normalised rotated x-Position in reciprocal space
                      [Bohr^-1]
323             float ky_hat = ky_rot * k_inv; // Normalised rotated y-Position in reciprocal space
                      [Bohr^-1]
324             float kz_hat = kz_rot * k_inv; // Normalised rotated z-Position in reciprocal space
                      [Bohr^-1]
325
326             // k_hat is a unit vector, so the harmonics take r^2 = 1 exactly -
327             // never kx_hat^2 + ky_hat^2 + kz_hat^2, which differs from 1 in
328             // the last bits (see harmonics.h)
329             float H[25];
330             cubic_harmonics(kx_hat, ky_hat, kz_hat, 1.0f, max_l, H);
331
332             float pixel_real = 0.0f;
333             float pixel_imag = 0.0f;
334             int ibasis = 0; // Index in the basis function arrays
335             for (int a = 0; a < Natoms; a++)
336             {
337                 float atom_contribution_real = 0; // Total contribution to momentum map from
                          single atom
338                 float atom_contribution_imag = 0; // Total contribution to momentum map from
                          single atom
339
340                 // Calculate phase factor
341                 float R_dot_k = Rx[a] * kx_rot + Ry[a] * ky_rot + Rz[a] * kz_rot;
342                 float phase_cos = cosf(R_dot_k);
343                 float phase_sin = sinf(R_dot_k);
344
345                 for (int b = 0; b < atom_Nbasis[a]; b++)
346                 {
347                     float basis_weight_real = basis_weights_real[ibasis] * H[H_index[ibasis]];
348                     float basis_weight_imag = basis_weights_imag[ibasis] * H[H_index[ibasis]];
349
350                     atom_contribution_real += basis_weight_real;
351                     atom_contribution_imag += basis_weight_imag;
352                     ibasis++;
353                 }
354
355                 // Add atomic contribution to map and phase factor
356                 pixel_real += phase_cos * atom_contribution_real + phase_sin *
                          atom_contribution_imag;
```

```
357                 pixel_imag += phase_cos * atom_contribution_imag - phase_sin *
                        atom_contribution_real;
358             }
359             momentum_map[i] = pixel_real * pixel_real + pixel_imag * pixel_imag;
360         }
361     }
362
363     free(scratch);
364 }
365
366 /**
367  * Incoherent sum of several orbitals' momentum maps, each computed at its own
368  * kinetic energy and post-processed on its own.
369  *
370  * Photoelectrons from different initial states do not interfere, so this is a
371  * sum of intensities, not of amplitudes - which is also why it cannot be done
372  * by pre-summing the coefficient rows. It used to be done in JavaScript, which
373  * meant copying every orbital's map out of the WASM heap, adding it there, and
374  * copying the total back in for normalisation(); the TODO asking for this to
375  * move into C is the reason the step exists.
376  *
377  * Each orbital runs the same chain the page always ran, in the same order:
378  * momentum_map, then symmetrisation, then optionally polarisation, then
379  * accumulate. Symmetrisation is linear and could be hoisted out of the loop,
380  * but that would change the arithmetic, so it is not.
381  *
382  * Does NOT normalise: showing un-normalised intensities is a wanted option
383  * (issue #62), so normalisation() stays a separate call.
384  *
385  * @param Norbitals number of orbitals to sum; 0 leaves a zeroed map
386  * @param E_kin kinetic energy per orbital [Hartree]; length Norbitals. An
387  * orbital with E_kin <= 0 lies below the photon energy and contributes
388  * nothing
389  * @param weights weight per orbital, or NULL for all ones. An exciton map is a
390  * weighted sum over electron-hole pairs; nothing passes weights yet
391  * @param coefficients Norbitals rows of Nbasis MO coefficients, row-major
392  * @param k0 momentum of the first grid point [Bohr^-1]
393  * @param dk grid spacing [Bohr^-1]
394  * @param Nk grid points per axis
395  * @param phi Euler angle, ZXZ convention, gamma [deg]
396  * @param theta Euler angle, ZXZ convention, beta [deg]
397  * @param psi Euler angle, ZXZ convention, alpha [deg]
398  * @param substrate -1 = none, 0 = fcc100, 1 = fcc110, 2 = fcc111
399  * @param apply_polarisation non-zero to apply the |A.k|^2 factor
400  * @param polar polar angle of the vector potential A [deg]
401  * @param azimuth azimuth angle of the vector potential A [deg]
402  * @param polarisation type of polarisation [0 == linear, 1 == circular, 2 == toroid]
403  * @param s_share share of s-polarisation in linear polarised light [%]
404  * @param handedness handedness of circular polarised light [0 == left, 1 == right, 2 == cd]
405  * @param gamma inelastic-mean-free-path damping per orbital [Bohr^-1]; length
406  * Norbitals - see polarisation_factor() for why this is computed by the
407  * caller rather than from E_kin here
408  * @param Natoms number of atoms
409  * @param Nshells shells per atom; length Natoms
410  * @param Rx atom x-positions [Bohr]; length Natoms
411  * @param Ry atom y-positions [Bohr]; length Natoms
412  * @param Rz atom z-positions [Bohr]; length Natoms
413  * @param shell_l angular momentum per shell; length sum(Nshells)
```

```
 * @param shell_Nprim primitives per shell; length sum(Nshells)
 * @param exponents alpha_p [Bohr^-2]; length sum(shell_Nprim)
 * @param contractions c_p per primitive; length sum(shell_Nprim)
 * @param out summed map; length Nk*Nk
 */
EMSCRIPTEN_KEEPALIVE
void momentum_map_sum(
    int Norbitals,
    float *restrict E_kin,
    float *restrict weights,
    float *restrict coefficients,
    float k0, float dk, int Nk,
    float phi, float theta, float psi,
    int substrate, int apply_polarisation, float polar, float azimuth,
    int polarisation, float s_share, int handedness, float *restrict gamma,
    int Natoms, int *restrict Nshells,
    float *restrict Rx, float *restrict Ry, float *restrict Rz,
    int *restrict shell_l, int *restrict shell_Nprim,
    float *restrict exponents, float *restrict contractions,
    float *restrict out)
{
    int total_pixels = Nk * Nk;
    for (int i = 0; i < total_pixels; i++)
    {
        out[i] = 0.0f;
    }

    Basis basis = basis_from_arrays(Natoms, Nshells, Rx, Ry, Rz,
                                    shell_l, shell_Nprim, exponents, contractions);

    float *one = (float *)malloc((size_t)total_pixels * sizeof(float));
    if (one == NULL)
    {
        return; // the zeroed map above is the honest answer
    }

    for (int o = 0; o < Norbitals; o++)
    {
        // E_kin <= 0 means the photon energy is below this orbital's binding
        // energy: no photoemission, so it contributes nothing
        if (E_kin[o] <= 0.0f)
        {
            continue;
        }

        momentum_map(E_kin[o], k0, dk, Nk, phi, theta, psi,
                     Natoms, Nshells, Rx, Ry, Rz, shell_l, shell_Nprim,
                     exponents, contractions,
                     coefficients + (size_t)o * basis.Nbasis, one);

        symmetrisation(Nk, substrate, one);
        if (apply_polarisation)
        {
            polarisation_factor(E_kin[o], k0, dk, Nk, polar, azimuth, polarisation, s_share,
                 handedness, gamma[o], one);
        }

        float weight = weights ? weights[o] : 1.0f;
```

```
        for (int i = 0; i < total_pixels; i++)
        {
            out[i] += weight * one[i];
        }
    }

    free(one);
}
```

B.5 `realspace.c` — the orbital in real space

Equation (6) evaluated on a Cartesian grid.

```
#include <math.h>
#include <stdlib.h>

#include "basis.h"
#include "harmonics.h"
#include "kernels.h"

/**
 * Compute the real-space orbital psi(r) on a Cartesian r-grid (see
 * docs/physics-compute.md). All lengths/energies are Hartree atomic units.
 * Parameters are documented inline below; the basis arrives shell-major (see
 * docs/binary-format.md): per shell one angular momentum, one primitive
 * list, and 2l+1 consecutive MO coefficients (atom-major -> shell -> m in
 * ORCA order).
 * @return via the `field` out-parameter; no return value
 */
EMSCRIPTEN_KEEPALIVE
void scalar_field(
    int N, // Grid points per axis (cubic grid)
    float x0, // Grid origin x [Bohr]
    float y0, // Grid origin y [Bohr]
    float z0, // Grid origin z [Bohr]
    float spacing, // Grid spacing [Bohr]
    int Natoms, // Number of atoms
    int *restrict Nshells, // Shells per atom; length Natoms
    float *restrict Rx, // x-Positions of the atoms [Bohr]; length Natoms
    float *restrict Ry, // y-Positions of the atoms [Bohr]; length Natoms
    float *restrict Rz, // z-Positions of the atoms [Bohr]; length Natoms
    int *restrict shell_l, // Angular momentum per shell; length sum(Nshells)
    int *restrict shell_Nprim, // Primitives per shell; length sum(Nshells)
    float *restrict exponents, // Exponent alpha_p [Bohr^-2]; length sum(shell_Nprim)
    float *restrict contractions, // Contraction c_p per primitive; length sum(shell_Nprim)
    float *restrict coefficients, // MO coefficients c_mu; length sum over shells of (2l+1)
    float *restrict field) // Output psi; length N^3, x fastest (i = ix + iy*N + iz*N*N)
{
    Basis basis = basis_from_arrays(Natoms, Nshells, Rx, Ry, Rz,
                                    shell_l, shell_Nprim, exponents, contractions);

    // The two cutoffs are shared with the other kernels; see basis.h
    float cutoff_exponent = primitive_exponent_cutoff();

    // We skip work at three levels:
    // 1. Primitives: inside the hot loop, primitives with alpha * r^2 past the cutoff
    // 2. Shells: shells whose 2l+1 coefficients are all ~0, and voxels beyond the
```

```
    // shell's cutoff radius (from its most diffuse primitive)
    // 3. Atoms: atoms with no significant shell, and voxels beyond the atom's reach

    // On the heap, not the stack: these are sized by the basis set, and a large
    // calculation would otherwise silently overrun the small fixed WASM stack.
    // One allocation carved into eight arrays - int and float are both 4 bytes
    // here, so no padding is needed between them.
    size_t scratch_words = 6 * (size_t)Natoms + 2 * (size_t)basis.Nshells;
    int *scratch = (int *)malloc(scratch_words * sizeof(int));
    if (scratch == NULL)
    {
        for (int j = 0; j < N * N * N; j++)
        {
            field[j] = 0.0f;
        }
        return;
    }
    int *shell_offset = scratch; // First shell index of each atom
    int *prim_offset = shell_offset + Natoms; // First primitive index of each atom
    int *coeff_offset = prim_offset + Natoms; // First coefficient index of each atom
    int *atom_max_l = coeff_offset + Natoms; // Highest significant angular momentum per atom
    int *significant_atoms = atom_max_l + Natoms; // Atom has any significant shell
    int *significant_shells = significant_atoms + Natoms; // Shell has any significant
         coefficient
    float *r2_cutoff_atoms = (float *)(significant_shells + basis.Nshells); // Squared reach of
         the atom [Bohr^2]
    float *r2_cutoff_shells = r2_cutoff_atoms + Natoms; // Squared reach of the shell [Bohr^2]

    int ishell = 0;
    int iprim = 0;
    int icoeff = 0;
    for (int a = 0; a < Natoms; a++)
    {
        shell_offset[a] = ishell;
        prim_offset[a] = iprim;
        coeff_offset[a] = icoeff;
        atom_max_l[a] = 0;
        significant_atoms[a] = 0;

        // The most diffuse (smallest) exponent of the atom's significant
        // shells dictates how far the atom reaches
        float alpha_min_atom = 1e30f;

        for (int s = 0; s < Nshells[a]; s++)
        {
            int l = shell_l[ishell];
            int members = 2 * l + 1;

            int shell_significant = 0;
            for (int m = 0; m < members; m++)
            {
                if (fabsf(coefficients[icoeff + m]) > COEFFICIENT_EPSILON)
                {
                    shell_significant = 1;
                }
            }
            significant_shells[ishell] = shell_significant;

```

```
            // The shell's most diffuse (smallest) exponent -> cutoff radius
            float alpha_min_shell = 1e30f;
            for (int p = 0; p < shell_Nprim[ishell]; p++)
            {
                if (exponents[iprim + p] < alpha_min_shell)
                {
                    alpha_min_shell = exponents[iprim + p];
                }
            }
            r2_cutoff_shells[ishell] = cutoff_exponent / alpha_min_shell;

            if (shell_significant)
            {
                significant_atoms[a] = 1;
                if (l > atom_max_l[a])
                {
                    atom_max_l[a] = l;
                }
                if (alpha_min_shell < alpha_min_atom)
                {
                    alpha_min_atom = alpha_min_shell;
                }
            }

            iprim += shell_Nprim[ishell];
            icoeff += members;
            ishell++;
        }

        r2_cutoff_atoms[a] = cutoff_exponent / alpha_min_atom;
    }

    // Zero the field, then accumulate atom by atom
    for (int j = 0; j < N * N * N; j++)
    {
        field[j] = 0.0f;
    }

    for (int a = 0; a < Natoms; a++)
    {
        if (!significant_atoms[a])
        {
            continue; // No shell of this atom contributes to the orbital
        }

        float ax = Rx[a];
        float ay = Ry[a];
        float az = Rz[a];
        float r_cutoff_atom = sqrtf(r2_cutoff_atoms[a]);
        int max_l = atom_max_l[a];

        // Voxel bounds overlapping the cutoff sphere, clamped to the grid
        int ix_min = (int)((ax - r_cutoff_atom - x0) / spacing); // i.e. floor
        int iy_min = (int)((ay - r_cutoff_atom - y0) / spacing);
        int iz_min = (int)((az - r_cutoff_atom - z0) / spacing);
        int ix_max = (int)((ax + r_cutoff_atom - x0) / spacing) + 1; // i.e. ceil
        int iy_max = (int)((ay + r_cutoff_atom - y0) / spacing) + 1;
        int iz_max = (int)((az + r_cutoff_atom - z0) / spacing) + 1;
```

```
        if (ix_min < 0)
            ix_min = 0;
        if (iy_min < 0)
            iy_min = 0;
        if (iz_min < 0)
            iz_min = 0;
        if (ix_max > N - 1)
            ix_max = N - 1;
        if (iy_max > N - 1)
            iy_max = N - 1;
        if (iz_max > N - 1)
            iz_max = N - 1;

        // Split the atom's slab of voxels across threads. The atom loop above
        // stays serial, so each voxel still accumulates its atoms in the same
        // order, and within a slab no two iz touch the same element.
#pragma omp parallel for schedule(static)
        for (int iz = iz_min; iz <= iz_max; iz++)
        {
            // dx/dy/dz are the position in space relative to the atom
            float dz = z0 + iz * spacing - az;
            float dz2 = dz * dz;

            for (int iy = iy_min; iy <= iy_max; iy++)
            {
                float dy = y0 + iy * spacing - ay;
                float dy2 = dy * dy;
                float dyz2 = dy2 + dz2;

                // Row is entirely outside the sphere
                if (dyz2 > r2_cutoff_atoms[a])
                {
                    continue;
                }

                for (int ix = ix_min; ix <= ix_max; ix++)
                {
                    float dx = x0 + ix * spacing - ax;
                    float dx2 = dx * dx;
                    float r2 = dx2 + dyz2;

                    if (r2 > r2_cutoff_atoms[a])
                    {
                        continue;
                    }

                    // The general solid-harmonic form: unlike the two k-space
                    // kernels, r^2 here is a real distance, not 1
                    float H[25];
                    cubic_harmonics(dx, dy, dz, r2, max_l, H);

                    float psi = 0.0f;
                    int is = shell_offset[a];
                    int ip = prim_offset[a];
                    int ic = coeff_offset[a];
                    for (int s = 0; s < Nshells[a]; s++)
                    {
                        int l = shell_l[is];
```

```
                    int members = 2 * l + 1;

                    if (significant_shells[is] && r2 <= r2_cutoff_shells[is])
                    {
                        // The 2l+1 members of a shell share one radial part
                        float radial = radial_realspace(&basis, is, ip, r2, cutoff_exponent);

                        for (int m = 0; m < members; m++)
                        {
                            psi += coefficients[ic + m] * radial * H[SHELL_START[l] + m];
                        }
                    }

                    ip += shell_Nprim[is];
                    ic += members;
                    is++;
                }

                field[ix + iy * N + iz * N * N] += psi;
            }
        }
    }
}

    free(scratch);
}

/**
 * The charge density of several orbitals summed: sum_i w_i |psi_i(r)|^2.
 *
 * |psi|^2 is not linear in the coefficients, so this squares each orbital's
 * own field rather than squaring a pre-summed row - a coherent sum of
 * amplitudes is not physically meaningful for non-degenerate orbitals. The
 * summation used to happen in JavaScript, one heap round-trip per orbital.
 *
 * For a single orbital's signed psi(r), call scalar_field() directly; that is
 * the wavefunction view, and it is only offered when one orbital is selected.
 *
 * Each orbital still gets its own scalar_field() call rather than sharing one
 * voxel loop: the cutoff bounds are per-orbital, and a shared loop would have
 * to visit the union of them.
 *
 * @param N grid points per axis (cubic grid)
 * @param x0 grid origin x [Bohr]
 * @param y0 grid origin y [Bohr]
 * @param z0 grid origin z [Bohr]
 * @param spacing grid spacing [Bohr]
 * @param Norbitals number of orbitals to sum; 0 leaves a zeroed field
 * @param weights weight per orbital, or NULL for all ones
 * @param coefficients Norbitals rows of Nbasis MO coefficients, row-major
 * @param Natoms number of atoms
 * @param Nshells shells per atom; length Natoms
 * @param Rx atom x-positions [Bohr]; length Natoms
 * @param Ry atom y-positions [Bohr]; length Natoms
 * @param Rz atom z-positions [Bohr]; length Natoms
 * @param shell_l angular momentum per shell; length sum(Nshells)
 * @param shell_Nprim primitives per shell; length sum(Nshells)
 * @param exponents alpha_p [Bohr^-2]; length sum(shell_Nprim)
```

```
 * @param contractions c_p per primitive; length sum(shell_Nprim)
 * @param out summed density; length N^3
 */
EMSCRIPTEN_KEEPALIVE
void charge_density_sum(
    int N, float x0, float y0, float z0, float spacing,
    int Norbitals, float *restrict weights, float *restrict coefficients,
    int Natoms, int *restrict Nshells,
    float *restrict Rx, float *restrict Ry, float *restrict Rz,
    int *restrict shell_l, int *restrict shell_Nprim,
    float *restrict exponents, float *restrict contractions,
    float *restrict out)
{
    int voxels = N * N * N;
    for (int i = 0; i < voxels; i++)
    {
        out[i] = 0.0f;
    }

    Basis basis = basis_from_arrays(Natoms, Nshells, Rx, Ry, Rz,
                                shell_l, shell_Nprim, exponents, contractions);

    float *one = (float *)malloc((size_t)voxels * sizeof(float));
    if (one == NULL)
    {
        return;
    }

    for (int o = 0; o < Norbitals; o++)
    {
        scalar_field(N, x0, y0, z0, spacing,
                    Natoms, Nshells, Rx, Ry, Rz, shell_l, shell_Nprim,
                    exponents, contractions,
                    coefficients + (size_t)o * basis.Nbasis, one);

        float weight = weights ? weights[o] : 1.0f;
        for (int i = 0; i < voxels; i++)
        {
            out[i] += weight * one[i] * one[i];
        }
    }

    free(one);
}
```

B.6 `momentumspace.c` — the 3D momentum density

The same transform as the momentum map, on a Cartesian k-grid.

```
#include <math.h>
#include <stdlib.h>

#include "basis.h"
#include "harmonics.h"
#include "kernels.h"

// Samples of the per-shell radial amplitude over k in [0, the box's far corner].
```

```c
// The radial part is a smooth sum of Gaussians in k, so linear interpolation
// on this table is far cheaper than evaluating the primitives per voxel.
#define RADIAL_TABLE_SIZE 1024

/**
 * The voxel loop behind both entry points: sum_o w_o |psi~_o(k)|^2 on a
 * Cartesian k-grid, via the analytic Fourier transform of each LCAO-GTO
 * orbital (see docs/physics-compute.md). Hartree atomic units throughout.
 *
 * One voxel loop for every orbital, not one per orbital. What a voxel costs
 * that does not depend on which orbital is asking - the direction k_hat, the
 * cubic harmonics, the radial tables, and each atom's cos/sin phase factor -
 * is computed once and then reused, where summing S orbitals used to evaluate
 * all of it S times over. On a 64^3 grid over a 77-atom molecule the two
 * transcendentals per atom per voxel are the single largest term.
 *
 * The per-orbital part is deliberately left as it was, accumulating into plain
 * locals over the atoms in the same order and behind the same significance
 * test. Per-orbital accumulator *arrays* were tried instead and cost 20% on the
 * one-orbital case, which is the common one, for no gain beyond this.
 *
 * @param N grid points per axis (cubic grid)
 * @param k0 momentum of the first grid point, each axis [Bohr^-1]
 * @param dk grid spacing [Bohr^-1]
 * @param Norbitals number of orbitals to sum; 0 leaves a zeroed field
 * @param weights weight per orbital, or NULL for all ones
 * @param coefficients Norbitals rows of Nbasis MO coefficients, row-major
 * @param basis the basis set
 * @param out summed density; length N^3, x fastest (i = ix + iy*N + iz*N*N)
 * @return via `out`; no return value
 */
static void momentum_density_voxels(
    int N, float k0, float dk,
    int Norbitals, const float *weights, const float *coefficients,
    const Basis *basis, float *restrict out)
{
    int voxels = N * N * N;
    for (int i = 0; i < voxels; i++)
    {
        out[i] = 0.0f;
    }
    if (Norbitals <= 0)
    {
        return;
    }

    int Natoms = basis->Natoms;
    int Nshells = basis->Nshells;
    int Nbasis = basis->Nbasis;

    // Local restrict-qualified copies: the compiler needs to know these do not
    // alias `out` to keep the inner loops vectorised
    const int *restrict Nshells_atom = basis->Nshells_per_atom;
    const int *restrict shell_l = basis->shell_l;
    const float *restrict Rx = basis->Rx;
    const float *restrict Ry = basis->Ry;
    const float *restrict Rz = basis->Rz;

```

```
    // Everything sized by the basis or the selection goes on the heap: a large
    // calculation would silently overrun arrays sized this way on the stack.
    // `significant` carries one row per orbital plus a final row flagging the
    // shells some orbital needs, which is what decides the radial tables.
    // The phase scratch is per voxel, so under OpenMP each thread needs its
    // own slice; without it KERNEL_MAX_THREADS is 1 and this is one pair.
    size_t phase_stride = 2 * (size_t)Natoms;
    char *significant = (char *)malloc((size_t)(Norbitals + 1) * (size_t)Nshells);
    float *phases = (float *)malloc((size_t)KERNEL_MAX_THREADS * phase_stride * sizeof(float));
    float *radial_tables = (float *)calloc((size_t)Nshells * RADIAL_TABLE_SIZE, sizeof(float));
    if (significant == NULL || phases == NULL || radial_tables == NULL)
    {
        free(significant);
        free(phases);
        free(radial_tables);
        return;
    }
    // Plain pointer, not restrict: it is a slice of the same allocation, so
    // promising the compiler it cannot alias `significant` would be a lie
    char *used = significant + (size_t)Norbitals * Nshells;

    // Setup pass: which shells each orbital needs, and the highest angular
    // momentum any of them reaches. Filling the harmonics up to the highest l
    // over all orbitals gives every orbital the same values it would get on its
    // own - cubic_harmonics() writes one independent block per l.
    int max_l = 0;
    for (int s = 0; s < Nshells; s++)
    {
        used[s] = 0;
    }
    for (int o = 0; o < Norbitals; o++)
    {
        const float *row = coefficients + (size_t)o * Nbasis;
        char *flags = significant + (size_t)o * Nshells;
        int icoeff = 0;

        for (int s = 0; s < Nshells; s++)
        {
            int members = 2 * shell_l[s] + 1;

            char shell_significant = 0;
            for (int m = 0; m < members; m++)
            {
                if (fabsf(row[icoeff + m]) > COEFFICIENT_EPSILON)
                {
                    shell_significant = 1;
                }
            }
            flags[s] = shell_significant;
            if (shell_significant)
            {
                used[s] = 1;
                if (shell_l[s] > max_l)
                {
                    max_l = shell_l[s];
                }
            }

```

```
            icoeff += members;
        }
    }

    // Precompute the radial amplitude (k/2)^l * sum_p c_p alpha_p^(-l-3/2)
    // exp(-k^2 / 4 alpha_p) of every needed shell on a 1D k-table; the voxel
    // loop then only interpolates. A shell's table does not depend on which
    // orbital asked for it, so it is built once for the whole selection.
    // Largest |k| the voxel loop can ask for: a corner of the box. k_last is
    // written exactly as the loop writes it, so the table always covers the
    // range actually sampled - which is what lets the grid be asymmetric or
    // arbitrarily wide rather than a fixed [-k_max, k_max]^3.
    float k_last = k0 + (float)(N - 1) * dk;
    float k_extreme = fmaxf(fabsf(k0), fabsf(k_last));
    float k_table_max = sqrtf(3.0f) * k_extreme;
    float dk_table = k_table_max / (float)(RADIAL_TABLE_SIZE - 1);

    // The two cutoffs are shared with the other kernels; see basis.h
    float cutoff_exponent = primitive_exponent_cutoff();

    int iprim = 0;
    for (int s = 0; s < Nshells; s++)
    {
        if (used[s])
        {
            float *table = radial_tables + (size_t)s * RADIAL_TABLE_SIZE;

            for (int it = 0; it < RADIAL_TABLE_SIZE; it++)
            {
                table[it] = radial_momentum(basis, s, iprim, it * dk_table, cutoff_exponent);
            }
        }

        iprim += basis->shell_Nprim[s];
    }

    // Loop over the k-grid; x fastest to match the three.js MarchingCubes
    // layout. Each iz owns its own N*N plane of the output and only reads
    // shared setup, so the threads leave every voxel's sum untouched.
#pragma omp parallel for schedule(static)
    for (int iz = 0; iz < N; iz++)
    {
        float kz = k0 + iz * dk;
        float *phase_cos = phases + (size_t)KERNEL_THREAD_ID * phase_stride;
        float *phase_sin = phase_cos + Natoms;

        for (int iy = 0; iy < N; iy++)
        {
            float ky = k0 + iy * dk;

            for (int ix = 0; ix < N; ix++)
            {
                float kx = k0 + ix * dk;
                float k2 = kx * kx + ky * ky + kz * kz;
                float k = sqrtf(k2);

                // At the origin the direction k_hat is undefined; only s-type
                // functions contribute there (all l > 0 carry a factor k^l)
```

```
            float k_inv = k2 > 1e-12f ? 1.0f / k : 0.0f;
            float kx_hat = kx * k_inv;
            float ky_hat = ky * k_inv;
            float kz_hat = kz * k_inv;

            // Table lookup position for the radial interpolation
            float t = k / dk_table;
            int it = (int)t;
            if (it > RADIAL_TABLE_SIZE - 2)
            {
                it = RADIAL_TABLE_SIZE - 2;
            }
            float fraction = t - (float)it;

            // k_hat is a unit vector (or zero at the origin), so r^2 = 1
            // exactly. Same call as momentum_map.c, which is what makes the
            // hemisphere cut of this field match the momentum maps.
            float H[25];
            cubic_harmonics(kx_hat, ky_hat, kz_hat, 1.0f, max_l, H);

            // Phase factor e^(-i k.R_A), the same for every orbital
            for (int a = 0; a < Natoms; a++)
            {
                float R_dot_k = Rx[a] * kx + Ry[a] * ky + Rz[a] * kz;
                phase_cos[a] = cosf(R_dot_k);
                phase_sin[a] = sinf(R_dot_k);
            }

            // |psi~|^2 is not linear in the coefficients, so each orbital's
            // own density is summed rather than a pre-summed row transformed
            float total = 0.0f;
            for (int o = 0; o < Norbitals; o++)
            {
                const float *row = coefficients + (size_t)o * Nbasis;
                const char *flags = significant + (size_t)o * Nshells;

                float pixel_real = 0.0f;
                float pixel_imag = 0.0f;
                int is = 0;
                int ic = 0;
                for (int a = 0; a < Natoms; a++)
                {
                    float atom_real = 0.0f; // Atomic form factor F_A
                    float atom_imag = 0.0f;

                    for (int s = 0; s < Nshells_atom[a]; s++)
                    {
                        int l = shell_l[is];
                        int members = 2 * l + 1;

                        if (flags[is])
                        {
                            // The 2l+1 members of a shell share one radial part
                            const float *table = radial_tables + (size_t)is * RADIAL_TABLE_SIZE;
                            float radial = table[it] + fraction * (table[it + 1] - table[it]);

                            float shell_real = 0.0f;
                            for (int m = 0; m < members; m++)
```

```
                            {
                                shell_real += row[ic + m] * H[SHELL_START[l] + m];
                            }
                            shell_real *= radial;

                            // (-i)^l sorts the contribution into real/imaginary part
                            switch (l % 4)
                            {
                            case 0: //(-i)^0 = 1
                                atom_real += shell_real;
                                break;
                            case 1: //(-i)^1 = -i
                                atom_imag -= shell_real;
                                break;
                            case 2: //(-i)^2 = -1
                                atom_real -= shell_real;
                                break;
                            case 3: //(-i)^3 = i
                                atom_imag += shell_real;
                                break;
                            }
                        }

                        ic += members;
                        is++;
                    }

                    pixel_real += phase_cos[a] * atom_real + phase_sin[a] * atom_imag;
                    pixel_imag += phase_cos[a] * atom_imag - phase_sin[a] * atom_real;
                }

                float weight = weights ? weights[o] : 1.0f;
                total += weight * (pixel_real * pixel_real + pixel_imag * pixel_imag);
            }
            out[ix + iy * N + iz * N * N] = total;
        }
    }
}

    free(radial_tables);
    free(phases);
    free(significant);
}

/**
 * Compute the 3D momentum-space density |psi~(k)|^2 of one orbital on a
 * Cartesian k-grid. All lengths/energies are Hartree atomic units. Parameters
 * are documented inline below; the basis arrives shell-major (see
 * docs/binary-format.md): per shell one angular momentum, one primitive list,
 * and 2l+1 consecutive MO coefficients (atom-major -> shell -> m in ORCA
 * order).
 * @return via the `field` out-parameter; no return value
 */
EMSCRIPTEN_KEEPALIVE
void momentum_density(
    int N, // Grid points per axis (cubic grid)
    float k0, // Momentum of the first grid point, each axis [Bohr^-1]
    float dk, // Grid spacing [Bohr^-1]
```

```
    int Natoms, // Number of atoms
    int *restrict Nshells_atom, // Shells per atom; length Natoms
    float *restrict Rx, // x-Positions of the atoms [Bohr]; length Natoms
    float *restrict Ry, // y-Positions of the atoms [Bohr]; length Natoms
    float *restrict Rz, // z-Positions of the atoms [Bohr]; length Natoms
    int *restrict shell_l, // Angular momentum per shell; length sum(Nshells_atom)
    int *restrict shell_Nprim, // Primitives per shell; length sum(Nshells_atom)
    float *restrict exponents, // Exponent alpha_p [Bohr^-2]; length sum(shell_Nprim)
    float *restrict contractions, // Contraction c_p per primitive; length sum(shell_Nprim)
    float *restrict coefficients, // MO coefficients c_mu; length sum over shells of (2l+1)
    float *restrict field) // Output |psi~(k)|^2; length N^3, x fastest (i = ix + iy*N + iz*N*N)
{
    Basis basis = basis_from_arrays(Natoms, Nshells_atom, Rx, Ry, Rz,
                                    shell_l, shell_Nprim, exponents, contractions);

    momentum_density_voxels(N, k0, dk, 1, NULL, coefficients, &basis, field);
}

/**
 * The momentum-space density of several orbitals summed:
 * sum_i w_i |psi~_i(k)|^2.
 *
 * The summation used to happen in JavaScript, one heap round-trip per orbital.
 *
 * @param N grid points per axis (cubic grid)
 * @param k0 momentum of the first grid point, each axis [Bohr^-1]
 * @param dk grid spacing [Bohr^-1]
 * @param Norbitals number of orbitals to sum; 0 leaves a zeroed field
 * @param weights weight per orbital, or NULL for all ones
 * @param coefficients Norbitals rows of Nbasis MO coefficients, row-major
 * @param Natoms number of atoms
 * @param Nshells shells per atom; length Natoms
 * @param Rx atom x-positions [Bohr]; length Natoms
 * @param Ry atom y-positions [Bohr]; length Natoms
 * @param Rz atom z-positions [Bohr]; length Natoms
 * @param shell_l angular momentum per shell; length sum(Nshells)
 * @param shell_Nprim primitives per shell; length sum(Nshells)
 * @param exponents alpha_p [Bohr^-2]; length sum(shell_Nprim)
 * @param contractions c_p per primitive; length sum(shell_Nprim)
 * @param out summed density; length N^3
 */
EMSCRIPTEN_KEEPALIVE
void momentum_density_sum(
    int N, float k0, float dk,
    int Norbitals, float *restrict weights, float *restrict coefficients,
    int Natoms, int *restrict Nshells,
    float *restrict Rx, float *restrict Ry, float *restrict Rz,
    int *restrict shell_l, int *restrict shell_Nprim,
    float *restrict exponents, float *restrict contractions,
    float *restrict out)
{
    Basis basis = basis_from_arrays(Natoms, Nshells, Rx, Ry, Rz,
                                    shell_l, shell_Nprim, exponents, contractions);

    momentum_density_voxels(N, k0, dk, Norbitals, weights, coefficients, &basis, out);
}
```

B.7 `map_image.c` — substrate domains and image post-processing

Section 6: the domain sum, the right-angle rotations done as index permutations, and the rescaling to a display range.

```c
/* Post-processing on a finished momentum map.
 *
 * Everything here works on a plain Nk*Nk float image and knows nothing about
 * orbitals, basis sets or physics - which is why it lives apart from
 * momentum_map.c. The rotation and the two mirrors are internal; the two
 * exported entry points are the substrate-domain average and the scaling to
 * image range.
 */
#include <math.h>
#include <stdlib.h>

#include "kernels.h"

// Degrees to radians
#define DEG_TO_RAD 0.01745329252

/**
 * Scale a momentum map in place so its maximum value becomes 255 (image data).
 * @param Nk momentum map grid size (per axis)
 * @param momentum_map in/out; length Nk*Nk
 */
EMSCRIPTEN_KEEPALIVE
void normalisation(
    int Nk,
    float *restrict momentum_map)
{
    int total_pixels = Nk * Nk;
    float maximum = 0.0f;

    // Find maximum
    for (int i = 0; i < total_pixels; i++)
    {
        if (momentum_map[i] > maximum)
        {
            maximum = momentum_map[i];
        }
    }

    if (maximum == 0.0f) // Nothing to normalise; avoids 255/0 -> inf -> NaN map
    {
        return;
    }

    float norm = 255.0f / maximum; // Normalise to 255 for image data

    // Normalise
    for (int i = 0; i < total_pixels; i++)
    {
        momentum_map[i] *= norm;
    }
}

/**
```

```
 * Rotate a momentum map by exactly 90, 180 or 270 degrees about its center.
 *
 * A rotation by a right angle maps every grid point onto another grid point,
 * so this is an exact index permutation - no trig, no interpolation, and (the
 * reason it exists) no floating-point rounding to go wrong. The general path
 * below computes the same thing via cosf/sinf, but cosf(90 deg) is not
 * exactly 0.0f in float; multiplied by the largest dx/dy in the grid (near
 * the edges) that residual is just large enough to land the interpolation
 * source exactly on the strict bounds test a few lines down, so a sub-ULP
 * difference - between compilers, between -ffast-math on and off, between
 * anything - flipped whole edge pixels between "interpolated" and "zeroed"
 * (measured up to 14% of peak on fcc100, the only substrate using 90/270).
 * Bypassing the trig removes that fragility rather than papering over one
 * measurement of it.
 *
 * Preserves the general path's own boundary convention exactly: a pixel whose
 * rotated source would be the far edge row/column (`Nk-1`) is zeroed, just as
 * the strict `< Nk-1` check below already zeroes it for every other angle.
 *
 * @param quarter_turns 1, 2 or 3 - the rotation as a multiple of 90 degrees
 * @param Nk momentum map grid size (per axis)
 * @param momentum_map source map; length Nk*Nk
 * @return newly malloc'd rotated map, length Nk*Nk (caller frees)
 */
static float *rotate_right_angle(
    int quarter_turns,
    int Nk,
    float *restrict momentum_map)
{
    size_t map_size = (size_t)Nk * Nk * sizeof(float);
    float *rotated_map = (float *)malloc(map_size);

    for (int x = 0; x < Nk; x++)
    {
        for (int y = 0; y < Nk; y++)
        {
            int i = x * Nk + y;
            int x_src, y_src;

            switch (quarter_turns)
            {
            case 1: // 90 deg: source (y, Nk-1-x)
                x_src = y;
                y_src = Nk - 1 - x;
                break;
            case 2: // 180 deg: source (Nk-1-x, Nk-1-y)
                x_src = Nk - 1 - x;
                y_src = Nk - 1 - y;
                break;
            default: // 270 deg: source (Nk-1-y, x)
                x_src = Nk - 1 - y;
                y_src = x;
                break;
            }

            // Same exclusion as the general path's `x_src < Nk - 1` check:
            // a source index of exactly Nk-1 is never read
            rotated_map[i] = (x_src == Nk - 1 || y_src == Nk - 1)
```

```
                                ? 0.0f
                                : momentum_map[x_src * Nk + y_src];
        }
    }
    return rotated_map;
}

/**
 * Rotate a momentum map by `psi` about its center (bilinear interpolation),
 * used by symmetrisation() to build substrate-domain copies.
 *
 * Exact right angles (90/180/270, the only rotations symmetrisation() ever
 * asks for) go through rotate_right_angle() instead - see its comment.
 * @param psi rotation angle [deg]
 * @param Nk momentum map grid size (per axis)
 * @param momentum_map source map; length Nk*Nk
 * @return newly malloc'd rotated map, length Nk*Nk (caller frees)
 */
float *rotation(
    float psi,
    int Nk,
    float *restrict momentum_map)
{
    // Exact float literal comparisons are safe here: symmetrisation() always
    // passes these as the literal constants below, never a computed angle
    if (psi == 90.0f)
    {
        return rotate_right_angle(1, Nk, momentum_map);
    }
    if (psi == 180.0f)
    {
        return rotate_right_angle(2, Nk, momentum_map);
    }
    if (psi == 270.0f)
    {
        return rotate_right_angle(3, Nk, momentum_map);
    }

    size_t map_size = (size_t)Nk * Nk * sizeof(float);
    float *rotated_map = (float *)malloc(map_size);
    float cos_psi = cosf(psi * DEG_TO_RAD);
    float sin_psi = sinf(psi * DEG_TO_RAD);

    // Pivot point of the rotation
    float center = (Nk - 1) / 2.0f;

    // Loop over x (== rows) of new image
    for (int x = 0; x < Nk; x++)
    {
        // Find distance from center
        float dx = (float)x - center;

        // Loop over y (== columns) of new image
        for (int y = 0; y < Nk; y++)
        {
            int i = x * Nk + y;
            float dy = (float)y - center;

```

```
            // Apply inverse rotation to find source of the new pixel
            float x_src = dx * cos_psi + dy * sin_psi + center;
            float y_src = -dx * sin_psi + dy * cos_psi + center;

            // Check if source pixel is inside the image
            if (x_src >= 0.0f && x_src < (float)(Nk - 1) && y_src >= 0.0f && y_src < (float)(Nk -
                1))
            {
                // Bilinear interpolation
                // Find the "previous" neighbour to the left/bottom
                int x_prev = (int)x_src;
                int y_prev = (int)y_src;

                // Get the fractional distance from "previous" neighbour
                float fx = x_src - (float)x_prev;
                float fy = y_src - (float)y_prev;

                // Calculate bilinear coefficients
                float p00 = momentum_map[x_prev * Nk + y_prev]; // (x, y)
                float p10 = momentum_map[(x_prev + 1) * Nk + y_prev]; // (x+1, y)
                float p01 = momentum_map[x_prev * Nk + (y_prev + 1)]; // (x, y+1)
                float p11 = momentum_map[(x_prev + 1) * Nk + (y_prev + 1)]; // (x+1, y+1)

                rotated_map[i] = p00 * (1.0f - fx) * (1.0f - fy) +
                                 p10 * fx * (1.0f - fy) +
                                 p01 * (1.0f - fx) * fy +
                                 p11 * fx * fy;
            }
            else
            {
                rotated_map[i] = 0.0f;
            }
        }
    }
    return rotated_map;
}

/**
 * Mirror a momentum map across its x-axis.
 * @param Nk momentum map grid size (per axis)
 * @param momentum_map source map; length Nk*Nk
 * @return newly malloc'd flipped map, length Nk*Nk (caller frees)
 */
float *flip_over_x(
    int Nk,
    float *restrict momentum_map)
{
    size_t map_size = (size_t)Nk * Nk * sizeof(float);
    float *flipped_map = (float *)malloc(map_size);

    for (int x = 0; x < Nk; x++)
    {
        for (int y = 0; y < Nk; y++)
        {
            flipped_map[Nk * x + y] = momentum_map[Nk * ((Nk - x - 1) % Nk) + y];
        }
    }
    return flipped_map;
```

```
}

/**
 * Mirror a momentum map across its y-axis.
 * @param Nk momentum map grid size (per axis)
 * @param momentum_map source map; length Nk*Nk
 * @return newly malloc'd flipped map, length Nk*Nk (caller frees)
 */
float *flip_over_y(
    int Nk,
    float *restrict momentum_map)
{
    size_t map_size = (size_t)Nk * Nk * sizeof(float);
    float *flipped_map = (float *)malloc(map_size);

    for (int x = 0; x < Nk; x++)
    {
        for (int y = 0; y < Nk; y++)
        {
            flipped_map[Nk * x + y] = momentum_map[Nk * x + ((Nk - y - 1) % Nk)];
        }
    }
    return flipped_map;
}

/**
 * Sum rotated + mirrored copies of a momentum map to approximate the
 * averaging over a substrate's rotational/mirror domains. Using mirror
 * symmetry could be improved but it is cheap to begin with.
 * @param Nk momentum map grid size (per axis)
 * @param substrate -1 = no substrate, 0 = fcc100, 1 = fcc110, 2 = fcc111
 * @param momentum_map in/out; length Nk*Nk
 */
EMSCRIPTEN_KEEPALIVE
void symmetrisation(
    int Nk,
    int substrate,
    float *restrict momentum_map)
{
    if (substrate == -1) // do nothing
    {
        return;
    }

    int total_pixels = Nk * Nk;
    int Nangles = 0;
    float angles[3]; // 3 is the maximum number of additional domains (fcc100)
    size_t map_size = (size_t)total_pixels * sizeof(float);
    float *symmetrised_map = (float *)malloc(map_size);
    if (symmetrised_map == NULL)
    {
        return; // leave the caller's map alone rather than write a partial one
    }

    // Base rotation of 0°
    for (int j = 0; j < total_pixels; j++)
    {
        symmetrised_map[j] = momentum_map[j];
```

```
    }

    if (substrate == 0) // fcc100
    {
        angles[0] = 90.0f;
        angles[1] = 180.0f;
        angles[2] = 270.0f;
        Nangles = 3;
    }
    else if (substrate == 1) // fcc110
    {
        angles[0] = 180.0f;
        Nangles = 1;
    }
    else // fcc111
    {
        angles[0] = 120.0f;
        angles[1] = 240.0f;
        Nangles = 2;
    }

    for (int a = 0; a < Nangles; a++)
    {
        // Rotate the original momentum map
        float *rotated_map = rotation(angles[a], Nk, momentum_map);

        // Add to symmetrised_map
        for (int j = 0; j < total_pixels; j++)
        {
            symmetrised_map[j] += rotated_map[j];
        }

        free(rotated_map);
    }

    // Mirror. Each flip returns a fresh buffer, so each one is freed straight
    // after it has been added in - this used to overwrite a single pointer and
    // leak one or two Nk*Nk buffers on every call.
    float *flipped_map = flip_over_y(Nk, symmetrised_map);
    for (int j = 0; j < total_pixels; j++)
    {
        symmetrised_map[j] += flipped_map[j];
    }
    free(flipped_map);

    if (substrate == 0) // fcc100
    {
        flipped_map = flip_over_x(Nk, symmetrised_map);
        for (int j = 0; j < total_pixels; j++)
        {
            symmetrised_map[j] += flipped_map[j];
        }
        free(flipped_map);
    }

    // Write out the result to the input pointer location
    for (int j = 0; j < total_pixels; j++)
    {
```

```
343         momentum_map[j] = symmetrised_map[j];
344     }
345     free(symmetrised_map);
346 }
```

References

[1] P. Puschnig, S. Berkebile, A. J. Fleming, G. Koller, K. Emtsev, T. Seyller, J. D. Riley, C. Ambrosch-Draxl, F. P. Netzer, and M. G. Ramsey. Reconstruction of molecular orbital densities from photoemission data. *Science*, 326(5953):702–706, 2009.

[2] D. Lüftner, T. Ules, E. M. Reinisch, G. Koller, S. Soubatch, F. S. Tautz, M. G. Ramsey, and P. Puschnig. Imaging the wave functions of adsorbed molecules. *Proceedings of the National Academy of Sciences*, 111(2):605–610, 2014.

[3] P. Puschnig and M. G. Ramsey. Photoemission tomography: Valence band photoemission as a quantitative method for investigating molecular films. In *Encyclopedia of Interfacial Chemistry*, pages 380–391. Elsevier, 2018.

[4] A. Haags, D. Brandstetter, X. Yang, et al. Tomographic identification of all molecular orbitals in a wide binding-energy range. *Physical Review B*, 111:165402, 2025.

[5] C. S. Kern, A. Windischbacher, and P. Puschnig. Photoemission orbital tomography for excitons in organic molecules. *Physical Review B*, 108:085132, 2023.

[6] D. Brandstetter, X. Yang, D. Lüftner, F. S. Tautz, and P. Puschnig. kmap.py: A python program for simulation and data analysis in photoemission tomography. *Computer Physics Communications*, 263:107905, 2021.

[7] M. Dauth, M. Wiessner, V. Feyer, A. Schöll, P. Puschnig, F. Reinert, and S. Kümmel. Angle resolved photoemission from organic semiconductors: orbital imaging beyond the molecular orbital interpretation. *New Journal of Physics*, 16:103005, 2014.

[8] Izrail Solomonovich Gradshtein, Iosif Moiseevich Ryzhik, Alan Jeffrey, and Daniel Zwillinger. *Table of integrals, series, and products*. Elsevier : Academic press, 2007.

[9] F. Neese. Software update: The orca program system, version 5.0. *WIREs Computational Molecular Science*, 12(5):e1606, 2022.

[10] M. P. Seah and W. A. Dench. Quantitative electron spectroscopy of surfaces: A standard data base for electron inelastic mean free paths in solids. *Surface and Interface Analysis*, 1(1):2–11, 1979.

[11] M. E. Casida. Time-dependent density functional response theory for molecules. In D. P. Chong, editor, *Recent Advances in Density Functional Methods, Part I*, pages 155–192. World Scientific, 1995.

[12] R. L. Martin. Natural transition orbitals. *The Journal of Chemical Physics*, 118(11):4775–4777, 2003.

[13] S. Weiß, D. Lüftner, T. Ules, E. M. Reinisch, H. Kaser, A. Gottwald, M. Richter, S. Soubatch, G. Koller, M. G. Ramsey, F. S. Tautz, and P. Puschnig. Exploring three-dimensional orbital imaging with energy-dependent photoemission tomography. *Nature Communications*, 6:8287, 2015.