Electronic Band Structure, Spin Properties, and Magnetism of Monolayer MoS_2 from Quantum ESPRESSO and Wannier90

Contents

Chapter 1

Introduction

Monolayer MoS_2 is a prototypical transition metal dichalcogenide with strong spin orbit coupling, a direct band gap at the K point, and interesting valley physics. It is not intrinsically magnetic in the ground state, but spin orbit coupling induces a sizeable spin splitting in the valence band at K and K′ that is opposite in the two valleys.

The goals of this tutorial are:

- to compute the electronic band structure of monolayer MoS_2 using Quantum ESPRESSO (QE),
- to include spin orbit coupling (SOC) and extract spin splittings and spin expectation values,
- to assess whether MoS_2 is magnetic through spin polarised calculations and spin density analysis,
- to construct a spinor Wannier90 tight binding model that reproduces the SOC band structure and can be used to analyse spin texture.

All workflows are given with explicit input files and Python scripts suitable for direct adaptation.

Chapter 2

Crystal Structure and General DFT Setup

2.1 Monolayer MoS_2 Geometry

Monolayer MoS_2 has a hexagonal lattice with symmetry corresponding to a single S–Mo–S trilayer. A simple model cell uses:

- in plane lattice constant $a \approx 3.18$ Å,
- out of plane lattice constant $c \approx 20.0$ Å to provide vacuum,
- three atoms per cell:
  - one Mo at the center layer,
  - two S atoms above and below the Mo plane.

In many QE inputs, a tetragonal representation is used with `ibrav = 4`, that is simple hexagonal.

2.2 Choice of Pseudopotentials

To capture spin orbit coupling properly, fully relativistic pseudopotentials are required. These must be compatible with noncollinear calculations and have `lspinorb` support.

Examples:

- `Mo.rel-pbe-spn-kjpaw_psl.1.0.0.UPF`
- `S.rel-pbe-n-kjpaw_psl.1.0.0.UPF`

If such pseudopotentials are not available, the SOC effects cannot be computed reliably. You may still perform scalar relativistic (no SOC) calculations to obtain an approximate band structure.

Chapter 3

Electronic Band Structure with Quantum ESPRESSO

This chapter gives a baseline band structure workflow. First we present a scalar relativistic (no SOC) calculation to establish the general band structure, then we move to fully relativistic noncollinear calculations with SOC.

3.1 Scalar Relativistic SCF Calculation (No SOC)

3.1.1 SCF Input: `mos2_scf.in`

```
&control
    calculation = 'scf',
    prefix = 'mos2',
    outdir = './tmp',
    pseudo_dir = './pseudo'
/
&system
    ibrav = 4,
    a = 3.18,
    c = 20.0,
    nat = 3,
    ntyp = 2,
    ecutwfc = 60.0,
    ecutrho = 480.0,
    occupations = 'fixed'
/
&electrons
    conv_thr = 1.0d-8
/
ATOMIC_SPECIES
 Mo 95.94 Mo.pbe-spn-kjpaw_psl.1.0.0.UPF
 S 32.06 S.pbe-n-kjpaw_psl.1.0.0.UPF

ATOMIC_POSITIONS {crystal}
 Mo 0.0000000 0.0000000 0.0000000
 S 0.3333333 0.6666667 0.0790000
 S 0.6666667 0.3333333 -0.0790000
```

```
K_POINTS automatic
 12 12 1 0 0 0
```

Run:

```
pw.x < mos2_scf.in > mos2_scf.out
```

This yields a converged charge density for use in NSCF and band calculations.

3.2 NSCF on a Dense k Grid

3.2.1 NSCF Input: `mos2_nscf.in`

```
&control
    calculation = 'nscf',
    prefix = 'mos2',
    outdir = './tmp'
/
&system
    ibrav = 4,
    a = 3.18,
    c = 20.0,
    nat = 3,
    ntyp = 2,
    ecutwfc = 60.0,
    ecutrho = 480.0,
    occupations = 'fixed',
    nbnd = 60
/
&electrons
    conv_thr = 1.0d-8
/
ATOMIC_SPECIES
 Mo 95.94 Mo.pbe-spn-kjpaw_psl.1.0.0.UPF
 S 32.06 S.pbe-n-kjpaw_psl.1.0.0.UPF

ATOMIC_POSITIONS {crystal}
 Mo 0.0000000 0.0000000 0.0000000
 S 0.3333333 0.6666667 0.0790000
 S 0.6666667 0.3333333 -0.0790000

K_POINTS automatic
 21 21 1 0 0 0
```

Run:

```
pw.x < mos2_nscf.in > mos2_nscf.out
```

This NSCF run is mainly needed if you plan to interface with Wannier90 without SOC. For spin and SOC analysis, we will repeat this workflow with fully relativistic pseudopotentials.

3.3 Bands Along High Symmetry Path (No SOC)

For the hexagonal Brillouin zone, a standard path is $\Gamma \to K \to M \to \Gamma$. In reciprocal crystal coordinates:

$$\Gamma = (0,0,0), \quad K = \left(\tfrac{1}{3}, \tfrac{1}{3}, 0\right), \quad M = \left(\tfrac{1}{2}, 0, 0\right).$$

3.3.1 Bands Input: `mos2_bands.in`

```
&control
    calculation = 'bands',
    prefix = 'mos2',
    outdir = './tmp'
/
&system
    ibrav = 4,
    a = 3.18,
    c = 20.0,
    nat = 3,
    ntyp = 2,
    ecutwfc = 60.0,
    ecutrho = 480.0,
    nbnd = 60
/
&electrons
    conv_thr = 1.0d-8
/
ATOMIC_SPECIES
 Mo 95.94 Mo.pbe-spn-kjpaw_psl.1.0.0.UPF
 S 32.06 S.pbe-n-kjpaw_psl.1.0.0.UPF

ATOMIC_POSITIONS {crystal}
 Mo 0.0000000 0.0000000 0.0000000
 S 0.3333333 0.6666667 0.0790000
 S 0.6666667 0.3333333 -0.0790000

K_POINTS crystal_b
 4
 0.0000000 0.0000000 0.0 40 ! Gamma
 0.3333333 0.3333333 0.0 40 ! K
 0.5000000 0.0000000 0.0 40 ! M
 0.0000000 0.0000000 0.0 40 ! Gamma
```

Run:

```
pw.x    < mos2_bands.in       > mos2_bands.out
bands.x < mos2_bands_post.in  > mos2_bands_post.out
```

The file specified by `filband` in `mos2_bands_post.in` will contain the raw band energies for plotting.

Chapter 4

Spin Orbit and Noncollinear Band Structure

To describe spin properties correctly, we must use fully relativistic pseudopotentials and noncollinear calculations with SOC enabled.

4.1 Noncollinear SCF with SOC

4.1.1 SOC SCF Input: `mos2_soc_scf.in`

```
&control
    calculation = 'scf',
    prefix = 'mos2_soc',
    outdir = './tmp',
    pseudo_dir = './pseudo'
/
&system
    ibrav = 4,
    a = 3.18,
    c = 20.0,
    nat = 3,
    ntyp = 2,
    ecutwfc = 70.0,
    ecutrho = 560.0,
    occupations = 'fixed',
    noncolin = .true.,
    lspinorb = .true.
/
&electrons
    conv_thr = 1.0d-8,
    diagonalization = 'david'
/
ATOMIC_SPECIES
 Mo 95.94 Mo.rel-pbe-spn-kjpaw_psl.1.0.0.UPF
 S 32.06 S.rel-pbe-n-kjpaw_psl.1.0.0.UPF

ATOMIC_POSITIONS {crystal}
 Mo 0.0000000 0.0000000 0.0000000
```

```
 S 0.3333333 0.6666667 0.0790000
 S 0.6666667 0.3333333 -0.0790000

K_POINTS automatic
 12 12 1 0 0 0
```

Run:

```
pw.x < mos2_soc_scf.in > mos2_soc_scf.out
```

4.2 NSCF with SOC on a k Mesh

4.2.1 SOC NSCF Input: `mos2_soc_nscf.in`

```
&control
    calculation = 'nscf',
    prefix = 'mos2_soc',
    outdir = './tmp'
/
&system
    ibrav = 4,
    a = 3.18,
    c = 20.0,
    nat = 3,
    ntyp = 2,
    ecutwfc = 70.0,
    ecutrho = 560.0,
    occupations = 'fixed',
    nbnd = 80,
    noncolin = .true.,
    lspinorb = .true.
/
&electrons
    conv_thr = 1.0d-8
/
ATOMIC_SPECIES
 Mo 95.94 Mo.rel-pbe-spn-kjpaw_psl.1.0.0.UPF
 S 32.06 S.rel-pbe-n-kjpaw_psl.1.0.0.UPF

ATOMIC_POSITIONS {crystal}
 Mo 0.0000000 0.0000000 0.0000000
 S 0.3333333 0.6666667 0.0790000
 S 0.6666667 0.3333333 -0.0790000

K_POINTS automatic
 21 21 1 0 0 0
```

Run:

```
pw.x < mos2_soc_nscf.in > mos2_soc_nscf.out
```

4.3 SOC Band Structure Along High Symmetry Path

4.3.1 SOC Bands Input: `mos2_soc_bands.in`

```
&control
    calculation = 'bands',
    prefix = 'mos2_soc',
    outdir = './tmp'
/
&system
    ibrav = 4,
    a = 3.18,
    c = 20.0,
    nat = 3,
    ntyp = 2,
    ecutwfc = 70.0,
    ecutrho = 560.0,
    nbnd = 80,
    noncolin = .true.,
    lspinorb = .true.
/
&electrons
    conv_thr = 1.0d-8
/
ATOMIC_SPECIES
 Mo 95.94 Mo.rel-pbe-spn-kjpaw_psl.1.0.0.UPF
 S 32.06 S.rel-pbe-n-kjpaw_psl.1.0.0.UPF

ATOMIC_POSITIONS {crystal}
 Mo 0.0000000 0.0000000 0.0000000
 S 0.3333333 0.6666667 0.0790000
 S 0.6666667 0.3333333 -0.0790000

K_POINTS crystal_b
 4
 0.0000000 0.0000000 0.0 40 ! Gamma
 0.3333333 0.3333333 0.0 40 ! K
 0.5000000 0.0000000 0.0 40 ! M
 0.0000000 0.0000000 0.0 40 ! Gamma
```

Then run:

```
pw.x    < mos2_soc_bands.in       > mos2_soc_bands.out
bands.x < mos2_soc_bands_post.in  > mos2_soc_bands_post.out
```

The resulting file (for example `mos2_soc.bands.dat`) contains the SOC band structure for plotting.

Chapter 5

Spin Properties and Magnetism

5.1 Spin Splitting and Spin Expectation Values

Spin orbit coupling lifts Kramers degeneracy in systems without inversion symmetry. Monolayer MoS_2 retains time reversal symmetry but not inversion, so each valley K and K′ has valence band spin splitting with opposite sign.

To access spin expectation values in QE:

- use `projwfc.x` to obtain orbital and spin projected density of states,
- use `pp.x` with proper `plot_num` to obtain spin density in real space.

5.1.1 Spin Projection with `projwfc.x`

Input file `mos2_soc_proj.in`:

```
&projwfc
   prefix = 'mos2_soc',
   outdir = './tmp',
   filproj = 'mos2_soc_proj'
/
```

Run:

```
projwfc.x < mos2_soc_proj.in > mos2_soc_proj.out
```

In noncollinear mode, `projwfc.x` can output spin-resolved quantities. You can examine `mos2_soc_proj.out` and the associated data files (for example `mos2_soc_proj.pdos_tot` and orbital resolved PDOS) to infer spin characters.

5.2 Testing for Magnetism with Spin Polarised Calculations

The ground state of pristine monolayer MoS_2 is nonmagnetic. Nevertheless, it is useful to verify this numerically by starting a spin polarised calculation and checking whether the magnetization relaxes to zero.

5.2.1 Spin Polarised SCF Test: `mos2_mag_scf.in`

```
&control
    calculation = 'scf',
    prefix = 'mos2_mag',
    outdir = './tmp'
/
&system
    ibrav = 4,
    a = 3.18,
    c = 20.0,
    nat = 3,
    ntyp = 2,
    ecutwfc = 70.0,
    ecutrho = 560.0,
    occupations = 'fixed',
    nspin = 2,
    starting_magnetization(1) = 0.3, ! initial spin on Mo
    starting_magnetization(2) = 0.0 ! S
/
&electrons
    conv_thr = 1.0d-8
/
ATOMIC_SPECIES
 Mo 95.94 Mo.pbe-spn-kjpaw_psl.1.0.0.UPF
 S 32.06 S.pbe-n-kjpaw_psl.1.0.0.UPF

ATOMIC_POSITIONS {crystal}
 Mo 0.0000000 0.0000000 0.0000000
 S 0.3333333 0.6666667 0.0790000
 S 0.6666667 0.3333333 -0.0790000

K_POINTS automatic
 12 12 1 0 0 0
```

Run:

```
pw.x < mos2_mag_scf.in > mos2_mag_scf.out
```

Check in the output:

- total magnetization,
- absolute magnetization,
- per atom and per orbital contributions if you enable `lspinorb` and noncollinear mode.

If the ground state is nonmagnetic, the final magnetization should be very close to zero, independent of the starting magnetization.

5.3 Spin Density Visualization with `pp.x`

To obtain spin density components in real space, use `pp.x` with `plot_num = 6` for noncollinear calculations. Example input `mos2_soc_spinpp.in`:

```
&inputpp
    prefix = 'mos2_soc',
    outdir = './tmp',
    plot_num = 6
/
&plot
    iflag = 3,
    output_format = 5,
    fileout = 'mos2_soc_spindensity.dat'
/
```

Run:

```
pp.x < mos2_soc_spinpp.in > mos2_soc_spinpp.out
```

The file `mos2_soc_spindensity.dat` can be converted to formats suitable for visualization in VESTA or similar programs. It contains three components of the spin density at each grid point.

Chapter 6

SOC Wannier90 Workflow and Spinor Tight Binding Model

In this chapter we construct a spinor Wannier90 model that reproduces the SOC band structure and that can be used to compute spin texture.

6.1 Spinor Wannier90 Setup

To generate spinor Wannier functions, the following are required:

- noncollinear SOC calculations in QE with `noncolin = .true.` and `lspinorb = .true.`,
- a seed file `mos2_soc.win` with `spinors = true` defined,
- the QE interface program `pw2wannier90.x`.

6.1.1 Spinor Wannier Seed File: `mos2_soc.win`

Below is an example that uses 22 spinor Wannier functions (11 spatial WFs times 2 spin components) based on Mo d and S p orbitals.

```
num_wann = 22
num_bands = 80

spinors = true

begin unit_cell_cart
ang
  3.18 0.00 0.0
 -1.59 2.754 0.0
  0.00 0.00 20.0
end unit_cell_cart

begin atoms_cart
ang
 Mo 0.0000 0.0000 10.0
 S 1.59 0.9180 11.58
 S 1.59 0.9180 8.42
end atoms_cart
```

```
begin projections
  Mo : d
  S : p
end projections

mp_grid = 21 21 1

dis_win_min = -8.0
dis_win_max = 8.0
dis_froz_min = -3.0
dis_froz_max = 3.0

num_iter = 1000
conv_tol = 1.0d-8

bands_plot = true
begin kpath
  4
  G 0.0000 0.0000 0.0000
  K 0.3333 0.3333 0.0000
  M 0.5000 0.0000 0.0000
  G 0.0000 0.0000 0.0000
end kpath
bands_num_points = 150
bands_plot_format = gnuplot
```

This seed file should be placed in a directory where you have access to `mos2_soc.save` from QE.

6.2 Preprocessing and QE Interface

Run the Wannier90 preprocessing step:

```
wannier90.x -pp mos2_soc
```

This produces `mos2_soc.nnkp` and other auxiliary files.

Next, run the QE interface `pw2wannier90.x` with input `mos2_soc_p2w.in`:

```
&inputpp
    prefix = 'mos2_soc',
    outdir = './tmp',
    seedname = 'mos2_soc',
    write_amn = .true.,
    write_mmn = .true.
/
```

Run:

```
pw2wannier90.x < mos2_soc_p2w.in > mos2_soc_p2w.out
```

6.3 Wannierization and Spinor Real Space Hamiltonian

Finally, run the full Wannierization:

```
wannier90.x mos2_soc
```

On completion, Wannier90 writes:

- `mos2_soc_hr.dat` with the real space Hamiltonian $H_{mn}(\mathbf{R})$,
- `mos2_soc_band.dat` with Wannier interpolated bands along the specified path,
- `mos2_soc_wout` with information on Wannier spreads and centers.

Chapter 7

Python Tools for Band Structure and Spin Analysis

This chapter provides Python scripts to:

- read QE band structure (`mos2_soc.bands.dat`),
- extract spin splitting at K,
- read `mos2_soc_hr.dat` and reconstruct Wannier bands along a path,
- compare QE and Wannier bands,
- compute approximate spin expectation values in the Wannier basis under suitable assumptions.

7.1 Reading QE Bands and Extracting Spin Splitting at K

We assume `bands.x` output `mos2_soc.bands.dat` in a format with blocks:

- header line for each k: `ik kx ky kz`,
- subsequent lines with eigenvalues,
- blank line, then the next k.

7.1.1 Script: `qe_soc_spin_splitting.py`

```
#!/usr/bin/env python3
"""
qe_soc_spin_splitting.py

Reads QE bands.x output (mos2_soc.bands.dat) and computes
the SOC-induced valence band spin splitting at the K point
and the conduction band splitting near K.

Assumes file structure:
  ik kx ky kz
  E1 E2 ... (possibly over several lines)
```

```
  [blank line]
repeated for each k.

This script does not use explicit spin labels since in
noncollinear SOC the spin character is mixed. It simply
looks at the closest k to the K point and extracts the
energy difference between the top two valence bands and
between the lowest two conduction bands.
"""

import numpy as np
from pathlib import Path

BANDS_FILE = "mos2_soc.bands.dat"

# K point in reciprocal crystal coordinates
K_POINT = np.array([1.0/3.0, 1.0/3.0, 0.0])

# Energy window to search around Fermi level for valence / conduction band edges
# These values can be adjusted depending on the band gap.
VALENCE_WINDOW_MAX = 0.5 # eV below EF
VALENCE_WINDOW_MIN = -3.0 # eV below EF
CONDUCTION_WINDOW_MIN = 0.0 # eV above EF
CONDUCTION_WINDOW_MAX = 3.0 # eV above EF

# If you know the Fermi energy from the SCF, set it here.
# Otherwise set to 0 and manually align if needed.
FERMI_ENERGY = 0.0


def read_qe_bands(filename):
    kpts = []
    bands = []

    current_k = None
    current_evals = []

    with open(filename, "r") as f:
        for line in f:
            stripped = line.strip()
            if not stripped:
                if current_k is not None:
                    kpts.append(current_k)
                    bands.append(np.array(current_evals, dtype=float))
                    current_k = None
                    current_evals = []
                continue

            parts = stripped.split()
            if current_k is None:
                if len(parts) < 4:
                    raise RuntimeError("Unexpected k header line: {}".format(line))
                kx = float(parts[1])
                ky = float(parts[2])
```

```
                kz = float(parts[3])
                current_k = np.array([kx, ky, kz])
            else:
                for p in parts:
                    current_evals.append(float(p))

    if current_k is not None:
        kpts.append(current_k)
        bands.append(np.array(current_evals, dtype=float))

    # pad to same number of bands
    nk = len(kpts)
    nb = max(len(b) for b in bands)
    band_arr = np.full((nk, nb), np.nan)
    for i, b in enumerate(bands):
        band_arr[i, :len(b)] = b

    return np.array(kpts), band_arr


def find_k_index_closest(kpts, k_target):
    dk = np.linalg.norm(kpts - k_target[None, :], axis=1)
    return np.argmin(dk)


def main():
    if not Path(BANDS_FILE).is_file():
        raise SystemExit(f"File '{BANDS_FILE}' not found.")

    kpts, evals = read_qe_bands(BANDS_FILE)
    evals_shifted = evals - FERMI_ENERGY

    ik = find_k_index_closest(kpts, K_POINT)
    print("Index of k-point closest to K:", ik)
    print("k(K) =", kpts[ik])

    Ek = evals_shifted[ik, :]
    print("Energies at K (shifted) in eV:")
    for i, e in enumerate(Ek):
        print(f" band {i+1:3d}: {e: .6f}")

    # Select valence-like and conduction-like states by energy window
    val_indices = np.where((Ek <= VALENCE_WINDOW_MAX) & (Ek >= VALENCE_WINDOW_MIN))[0]
    cond_indices = np.where((Ek >= CONDUCTION_WINDOW_MIN) & (Ek <= CONDUCTION_WINDOW_MAX)
        )[0]

    if len(val_indices) < 2:
        print("Not enough valence states in the chosen window for splitting.")
    else:
        # Sort descending for valence
        val_sorted = sorted(val_indices, key=lambda i: Ek[i], reverse=True)
        v1, v2 = val_sorted[0], val_sorted[1]
        dEv = Ek[v1] - Ek[v2]
        print(f"\nValence band splitting at K:")
```

```
        print(f" bands {v1+1} and {v2+1}, energies {Ek[v1]: .6f}, {Ek[v2]: .6f} eV")
        print(f" splitting = {dEv*1000.0:.2f} meV")

    if len(cond_indices) < 2:
        print("Not enough conduction states in the chosen window for splitting.")
    else:
        # Sort ascending for conduction
        cond_sorted = sorted(cond_indices, key=lambda i: Ek[i])
        c1, c2 = cond_sorted[0], cond_sorted[1]
        dEc = Ek[c2] - Ek[c1]
        print(f"\nConduction band splitting at K:")
        print(f" bands {c1+1} and {c2+1}, energies {Ek[c1]: .6f}, {Ek[c2]: .6f} eV")
        print(f" splitting = {dEc*1000.0:.2f} meV")


if __name__ == "__main__":
    main()
```

This script identifies the k point closest to K, prints the band energies, and reports approximate valence and conduction splittings.

7.2 Reading `mos2_soc_hr.dat` and Constructing Wannier Bands

We now read `mos2_soc_hr.dat` and construct the Wannier tight binding Hamiltonian:

$$H_{mn}(\mathbf{k}) = \sum_{\mathbf{R}} H_{mn}(\mathbf{R}) \exp\left(i2\pi\mathbf{k}\cdot\mathbf{R}\right),$$

where $\mathbf{R} = (R_1, R_2, R_3)$ are lattice vectors in units of the primitive vectors, and $\mathbf{k}$ is in reciprocal crystal coordinates.

7.2.1 Script: `wannier_soc_bands_vs_qe.py`

```
#!/usr/bin/env python3
"""
wannier_soc_bands_vs_qe.py

Reads Wannier90 SOC Hamiltonian mos2_soc_hr.dat
and reconstructs the band structure along Gamma - K - M - Gamma.
Optionally reads QE bands in mos2_soc.bands.dat and plots both
for comparison.
"""

import numpy as np
import matplotlib.pyplot as plt
from pathlib import Path

HR_FILE = "mos2_soc_hr.dat"
QE_BANDS_FILE = "mos2_soc.bands.dat" # set to None if not available

# Special k points in reduced coordinates
GAMMA = np.array([0.0, 0.0, 0.0])
KPT = np.array([1.0/3.0, 1.0/3.0, 0.0])
```

```
MPT = np.array([0.5, 0.0, 0.0])

N_GK = 40
N_KM = 40
N_MG = 40

E_REF = 0.0 # shift reference (e.g. Fermi energy) if desired
E_MIN = -4.0
E_MAX = 4.0


def read_wannier_hr(filename):
    with open(filename, "r") as f:
        # First two lines: num_wann, nrpts
        num_wann = int(f.readline().split()[0])
        nrpts = int(f.readline().split()[0])

        # Degeneracies
        deg = []
        while len(deg) < nrpts:
            line = f.readline()
            if not line:
                raise RuntimeError("Unexpected end of file when reading degeneracies")
            parts = line.split()
            for p in parts:
                deg.append(int(p))
        deg = np.array(deg, dtype=int)

        R_list = []
        H_list = []

        for ir in range(nrpts):
            R_ir = None
            H_ir = np.zeros((num_wann, num_wann), dtype=np.complex128)
            for m in range(num_wann):
                for n in range(num_wann):
                    line = f.readline()
                    if not line:
                        raise RuntimeError("Unexpected end of file in Hamiltonian data")
                    parts = line.split()
                    if len(parts) != 7:
                        raise RuntimeError("Expected 7 entries per line, got: {}".format(
                            parts))
                    r1, r2, r3 = map(int, parts[0:3])
                    mm = int(parts[3]) - 1
                    nn = int(parts[4]) - 1
                    re = float(parts[5])
                    im = float(parts[6])
                    if R_ir is None:
                        R_ir = (r1, r2, r3)
                    else:
                        if R_ir != (r1, r2, r3):
                            raise RuntimeError("Inconsistent R for ir = {}".format(ir))
                    H_ir[mm, nn] = re + 1j * im
```

```
            R_list.append(R_ir)
            H_list.append(H_ir)

    return num_wann, np.array(R_list, dtype=int), deg, np.array(H_list, dtype=np.
        complex128)


def interpolate_segment(k_start, k_end, n_points):
    t = np.linspace(0.0, 1.0, n_points, endpoint=False)
    return k_start[None, :] + (k_end - k_start)[None, :] * t[:, None]


def build_k_path():
    k_GK = interpolate_segment(GAMMA, KPT, N_GK)
    k_KM = interpolate_segment(KPT, MPT, N_KM)
    k_MG = interpolate_segment(MPT, GAMMA, N_MG)

    k_path = np.vstack([
        k_GK,
        k_KM[1:, :],
        k_MG[1:, :]
    ])

    dk = np.linalg.norm(np.diff(k_path, axis=0), axis=1)
    x_axis = np.zeros(len(k_path))
    x_axis[1:] = np.cumsum(dk)

    x_G = x_axis[0]
    x_K = x_axis[N_GK - 1]
    x_M = x_axis[N_GK + N_KM - 2]
    x_G2 = x_axis[-1]
    labels = [
        (x_G, r"$\Gamma$"),
        (x_K, r"$K$"),
        (x_M, r"$M$"),
        (x_G2, r"$\Gamma$")
    ]
    return k_path, x_axis, labels


def Hk_from_HR(k_red, R, deg, H_R):
    phase_arg = 2.0 * np.pi * (R @ k_red)
    phase = np.exp(1j * phase_arg) / deg[:, None, None]
    Hk = np.sum(phase * H_R, axis=0)
    return Hk


def compute_wannier_bands(kpts, R, deg, H_R):
    num_k = kpts.shape[0]
    num_wann = H_R.shape[1]
    evals = np.zeros((num_k, num_wann), dtype=float)
    for i, k in enumerate(kpts):
        Hk = Hk_from_HR(k, R, deg, H_R)
        w, _ = np.linalg.eigh(Hk)
```

```
        evals[i, :] = np.real(w)
    return evals


def read_qe_bands(filename):
    kpts = []
    bands = []
    current_k = None
    current_evals = []

    with open(filename, "r") as f:
        for line in f:
            stripped = line.strip()
            if not stripped:
                if current_k is not None:
                    kpts.append(current_k)
                    bands.append(np.array(current_evals, dtype=float))
                    current_k = None
                    current_evals = []
                continue

            parts = stripped.split()
            if current_k is None:
                if len(parts) < 4:
                    raise RuntimeError("Unexpected line in QE bands file: '{}'".format(line
                        ))
                kx = float(parts[1])
                ky = float(parts[2])
                kz = float(parts[3])
                current_k = np.array([kx, ky, kz], dtype=float)
            else:
                for p in parts:
                    current_evals.append(float(p))

    if current_k is not None:
        kpts.append(current_k)
        bands.append(np.array(current_evals, dtype=float))

    kpts = np.array(kpts, dtype=float)
    max_nb = max(len(b) for b in bands)
    bands_array = np.full((len(bands), max_nb), np.nan, dtype=float)
    for i, b in enumerate(bands):
        bands_array[i, :len(b)] = b

    # build x-axis
    dk = np.linalg.norm(np.diff(kpts, axis=0), axis=1)
    x_axis = np.zeros(len(kpts))
    x_axis[1:] = np.cumsum(dk)

    return kpts, x_axis, bands_array


def main():
    if not Path(HR_FILE).is_file():
```

```
        raise SystemExit(f"File '{HR_FILE}' not found.")
    print("Reading Wannier HR file...")
    num_wann, R, deg, H_R = read_wannier_hr(HR_FILE)
    print(" num_wann =", num_wann, "nrpts =", R.shape[0])

    k_path, x_axis, labels = build_k_path()
    print("Computing Wannier bands...")
    wann_evals = compute_wannier_bands(k_path, R, deg, H_R)
    wann_evals -= E_REF

    have_qe = QE_BANDS_FILE is not None and Path(QE_BANDS_FILE).is_file()
    if have_qe:
        print("Reading QE bands...")
        qe_kpts, qe_x_axis, qe_evals = read_qe_bands(QE_BANDS_FILE)
        qe_evals -= E_REF
    else:
        qe_x_axis = None
        qe_evals = None

    plt.figure(figsize=(6,6))

    # plot Wannier bands
    for n in range(num_wann):
        plt.plot(x_axis, wann_evals[:, n], color="black", lw=1.0)

    # plot QE bands if available
    if have_qe:
        nb_qe = qe_evals.shape[1]
        for n in range(nb_qe):
            plt.plot(qe_x_axis, qe_evals[:, n], color="red", lw=0.8, linestyle="--")

    for x, lab in labels:
        plt.axvline(x=x, color="k", linewidth=0.5)
        plt.text(x, E_MIN + 0.05*(E_MAX - E_MIN), lab,
                 ha="center", va="bottom")

    plt.xlim(x_axis[0], x_axis[-1])
    plt.ylim(E_MIN, E_MAX)
    plt.ylabel("Energy (eV)")
    plt.xticks([])

    title = "SOC Wannier bands"
    if have_qe:
        title += " vs QE"
    plt.title(title)
    plt.tight_layout()
    plt.savefig("wannier_soc_vs_qe.png", dpi=300)
    print("Saved plot to wannier_soc_vs_qe.png")
    plt.show()


if __name__ == "__main__":
    main()
```

This script compares the SOC Wannier band structure with the original QE SOC bands.

7.3 Approximate Spin Texture in the Wannier Basis

A rigorous spin texture calculation in a Wannier basis requires knowing how spinor components are organised. Under a simplifying assumption, one can treat the Wannier orbitals as grouped into up and down components:

- indices $0, 1, \ldots, N_{\text{orb}} - 1$ correspond to spin up components,
- indices $N_{\text{orb}}, \ldots, 2N_{\text{orb}} - 1$ correspond to spin down components,

for each spatial Wannier orbital.

Under this assumption, the spin operators in the Wannier basis can be written in block form using Pauli matrices. For instance, for the z component:

$$\hat{S}_z = \frac{\hbar}{2} \begin{pmatrix} I_{N_{\text{orb}}} & 0 \\ 0 & -I_{N_{\text{orb}}} \end{pmatrix}.$$

Then, the spin expectation value in band n at k is:

$$\langle S_z \rangle_{n\mathbf{k}} = \langle u_{n\mathbf{k}} | \hat{S}_z | u_{n\mathbf{k}} \rangle,$$

where $u_{n\mathbf{k}}$ is the eigenvector of $H(\mathbf{k})$.

Below is a simple script that illustrates this idea. It must be adapted if the spinor ordering differs.

7.3.1 Script: `wannier_soc_spin_texture.py`

```
#!/usr/bin/env python3
"""
wannier_soc_spin_texture.py

Illustrative script to compute approximate spin expectation values
from a spinor Wannier Hamiltonian, assuming a simple up/down block
structure in the Wannier basis.

For a given k-grid in the Brillouin zone, this script computes
Sx, Sy, Sz expectation values for a selected band near the gap
and produces a scatter plot of spin texture around K.

This is an illustrative example and may require adaptation
to the actual ordering of spinor Wannier functions.
"""

import numpy as np
import matplotlib.pyplot as plt
from pathlib import Path

HR_FILE = "mos2_soc_hr.dat"

# define a small k-mesh around K in reduced coordinates
```

```
KPT = np.array([1.0/3.0, 1.0/3.0, 0.0])
DELTA = 0.1
NKX = 21
NKY = 21

# select which band index to analyse (0-based)
BAND_INDEX = 10 # example, adjust after inspecting band structure

HBAR = 1.0 # for convenience; overall scale is arbitrary for normalized spins


def read_wannier_hr(filename):
    with open(filename, "r") as f:
        num_wann = int(f.readline().split()[0])
        nrpts = int(f.readline().split()[0])

        deg = []
        while len(deg) < nrpts:
            line = f.readline()
            if not line:
                raise RuntimeError("Unexpected end of file reading degeneracies")
            parts = line.split()
            for p in parts:
                deg.append(int(p))
        deg = np.array(deg, dtype=int)

        R_list = []
        H_list = []
        for ir in range(nrpts):
            R_ir = None
            H_ir = np.zeros((num_wann, num_wann), dtype=np.complex128)
            for m in range(num_wann):
                for n in range(num_wann):
                    line = f.readline()
                    if not line:
                        raise RuntimeError("Unexpected end of file in H(R)")
                    parts = line.split()
                    r1, r2, r3 = map(int, parts[0:3])
                    mm = int(parts[3]) - 1
                    nn = int(parts[4]) - 1
                    re = float(parts[5])
                    im = float(parts[6])
                    if R_ir is None:
                        R_ir = (r1, r2, r3)
                    else:
                        if R_ir != (r1, r2, r3):
                            raise RuntimeError("Inconsistent R at ir = {}".format(ir))
                    H_ir[mm, nn] = re + 1j * im
            R_list.append(R_ir)
            H_list.append(H_ir)

    return num_wann, np.array(R_list, dtype=int), deg, np.array(H_list, dtype=np.
        complex128)
```

```
def Hk_from_HR(k_red, R, deg, H_R):
    phase_arg = 2.0 * np.pi * (R @ k_red)
    phase = np.exp(1j * phase_arg) / deg[:, None, None]
    Hk = np.sum(phase * H_R, axis=0)
    return Hk


def build_spin_operators(num_wann):
    if num_wann % 2 != 0:
        raise ValueError("num_wann must be even for simple up/down block structure.")

    Norb = num_wann // 2
    I = np.eye(Norb)
    zero = np.zeros((Norb, Norb))

    # Pauli matrices in spin space
    sx = np.array([[0, 1], [1, 0]], dtype=complex)
    sy = np.array([[0, -1j], [1j, 0]], dtype=complex)
    sz = np.array([[1, 0], [0, -1]], dtype=complex)

    # Lift to Wannier basis: block Kronecker product
    # basis ordering [orb1_up, orb2_up, ..., orbN_up, orb1_dn, ..., orbN_dn]
    Sx = np.block([[zero, I],
                   [I, zero]]) * (HBAR / 2.0)
    Sy = np.block([[zero, -1j*I],
                   [1j*I, zero]]) * (HBAR / 2.0)
    Sz = np.block([[I, zero],
                   [zero, -I]]) * (HBAR / 2.0)
    return Sx, Sy, Sz


def main():
    if not Path(HR_FILE).is_file():
        raise SystemExit(f"File '{HR_FILE}' not found.")

    num_wann, R, deg, H_R = read_wannier_hr(HR_FILE)
    print("num_wann =", num_wann)
    Sx, Sy, Sz = build_spin_operators(num_wann)

    kxs = np.linspace(KPT[0] - DELTA, KPT[0] + DELTA, NKX)
    kys = np.linspace(KPT[1] - DELTA, KPT[1] + DELTA, NKY)

    KX, KY = np.meshgrid(kxs, kys, indexing="ij")

    Sx_map = np.zeros_like(KX, dtype=float)
    Sy_map = np.zeros_like(KY, dtype=float)
    Sz_map = np.zeros_like(KY, dtype=float)

    for ix in range(NKX):
        for iy in range(NKY):
            k = np.array([KX[ix, iy], KY[ix, iy], KPT[2]])
            Hk = Hk_from_HR(k, R, deg, H_R)
            w, v = np.linalg.eigh(Hk)
```

```
            u = v[:, BAND_INDEX]

            Sx_map[ix, iy] = np.real(np.vdot(u, Sx @ u))
            Sy_map[ix, iy] = np.real(np.vdot(u, Sy @ u))
            Sz_map[ix, iy] = np.real(np.vdot(u, Sz @ u))

    # normalise spin vectors for plotting arrows
    norm = np.sqrt(Sx_map**2 + Sy_map**2 + Sz_map**2)
    norm[norm == 0] = 1.0
    Sx_n = Sx_map / norm
    Sy_n = Sy_map / norm
    Sz_n = Sz_map / norm

    plt.figure(figsize=(6,5))
    plt.quiver(KX, KY, Sx_n, Sy_n, Sz_n,
               cmap="coolwarm", scale=30, pivot="mid")
    plt.colorbar(label="Sz (normalised)")
    plt.xlabel("kx (reduced)")
    plt.ylabel("ky (reduced)")
    plt.title(f"Approximate spin texture near K, band index {BAND_INDEX}")
    plt.tight_layout()
    plt.savefig("spin_texture_near_K.png", dpi=300)
    print("Saved spin texture plot to spin_texture_near_K.png")
    plt.show()


if __name__ == "__main__":
    main()
```

This script is illustrative. The correctness of the spin texture depends critically on the actual ordering and nature of the spinor Wannier functions, which should be checked by inspecting Wannier outputs and possibly by comparing to `projwfc.x` projections.

Chapter 8

Summary

This tutorial provided a detailed workflow for:

- computing the band structure of monolayer MoS_2 with Quantum ESPRESSO,
- including spin orbit coupling in a noncollinear framework, and extracting spin splittings and spin expectation values,
- verifying the nonmagnetic character of MoS_2 by spin polarised calculations and spin density analysis,
- constructing a spinor Wannier90 model that reproduces the SOC band structure,
- and using Python scripts to reconstruct Wannier bands, compare with QE results, and explore spin texture.

The same methodology can be adapted to other two dimensional materials with strong spin orbit coupling, or extended to include dopants and external fields to study induced magnetism or modified spin textures.