Supplementary Note 1

Conversion Record: the `KernSmooth` R Package to Python

Yufei Cai Jun Li

Department of Applied and Computational Mathematics and Statistics,
University of Notre Dame

About this document. This is a contemporaneous engineering record of the `KernSmooth` conversion reported in *r2py: AI-Assisted Conversion of R Statistical Packages to Python*. It documents each phase as it was carried out, including diagnostic paths through defects that were subsequently fixed, and is supplied so that the process described in the main text can be audited. It is not intended to be read end to end; the Supplementary Information gives an index from claims in the main text to the evidence for them.

Phase numbering. `KernSmooth` is reached through R's array-based `.Fortran()` interface and so required no prologue. Phases 1–7 in this record are the seven core phases of the main text, in the same order.

Contents

Abstract

This record documents the conversion of the R package `KernSmooth` (version 2.23-26, released 2024-12-10) into an installable Python package named `r2py_kernsmooth`. `KernSmooth` is a *recommended* package shipped with every standard R installation; it implements kernel smoothing methods from Wand & Jones [1] and has no runtime dependencies on any external CRAN package.

The conversion strategy retains the original Fortran 77 computational core verbatim, compiles it into a Python C-extension via `numpy.f2py`, and replaces the R wrapper layer with semantically equivalent Python functions. The project proceeds in five phases: (1) static architectural analysis of the R package, (2) Python build-system establishment, (3) language-dependency cataloging and per-dependency conversion guide generation, (4) automated R-to-Python function conversion and package assembly, and (5) code-quality audit and namespace refinement. All five phases have been executed. The resulting package installs under Python 3.14 (CPython) in a `conda` environment on a Red Hat Enterprise Linux 9 HPC node and produces numerically correct output.

1 Introduction

1.1 Background and Motivation

The R package `KernSmooth` implements the statistical methods described in

> M. P. Wand and M. C. Jones (1995). *Kernel Smoothing*. Chapman and Hall, London.

It is a *recommended* package — one of the roughly fifteen packages that ship as part of the base R distribution and are maintained by the R Core Team. The package was authored by M. P. Wand, contributed LINPACK Fortran routines are attributed to Cleve Moler, and the current R-language port and maintenance is by Brian Ripley. It is released under an "Unlimited" license.

The package provides seven public functions covering kernel density estimation, plug-in bandwidth and bin-width selection, and local polynomial regression. Because it delegates all numerically intensive loops to compiled Fortran 77 code, it is both correct and performant.

The motivation for a Python port is to make these well-tested implementations available to Python-based scientific workflows without requiring an R runtime. A port that re-uses the *original Fortran routines* offers a stronger correctness guarantee than a ground-up Python reimplementation: numerical differences between the Python and R packages can then originate only in the R-to-Python wrapper translation, which is far easier to audit systematically.

The broader scientific motivation, experimental design, and validation plan for this project

are described in two planning documents that are not reproduced in this report:

- `docs/methodology_and_plan_Jun.pdf` — the primary methodology and project plan.
- `docs/methodology_addendum_20260511.pdf` — an addendum dated 2026-05-11.

The original R package reference manual is archived at `docs/KernSmooth.pdf`. The contents of these PDF files were not available during the writing of this report and are therefore not summarized here.

1.2 Repository Structure

The working directory `python-KernSmooth/` is the working tree of the `python-KernSmooth` repository hosted at `github.com/r2py-project/python-KernSmooth`. The `r2py_kernsmooth/` subdirectory is additionally managed as a git subtree linked to a separate repository at `github.com/r2py-project/r2py_kernsmooth`, allowing the Python package to be versioned and published to PyPI independently of the broader project. The cluster scripts `git_pull.sh` and `git_push.sh` at the project root encapsulate the subtree synchronisation commands required to keep both remotes up to date from the HPC node.

Path	Contents
`KernSmooth/`	Unmodified R package source (v2.23-26).
`r2py_kernsmooth/`	Git subtree linked to `github.com/r2py-project/r2py_kernsmooth`; the installable Python package. Internal structure detailed in the table below.
`structural_analysis/`	Dependency-graph visualisation artefacts (`dependency_graph.{py,html,pdf,png}`), the function-level CSV (`dependency_levels.csv`), and the JSON analysis outputs per R file.
`language_dependency_analysis/`	Per-file CSV dependency tables, the CSV merge script, the combined table, and the 46 per-dependency conversion guides in `conversion_guides/`.
`conversion_results/`	Per-function JSON translation artefacts produced by the conversion pipeline.
`docs/`	Architecture document, planning PDFs, per-phase summaries in `daily_summaries/`, and this report in `report/`.
`.claude/agents/`	Sub-agent specification Markdown files.
`.claude/commands/`	Skill (slash command) specification Markdown files.
`git_pull.sh`	Cluster batch script; pulls from the `python-KernSmooth` main remote and merges changes from the `r2py_kernsmooth` subtree remote into `r2py_kernsmooth/`.
`git_push.sh`	Cluster batch script; pushes to the `python-KernSmooth` main remote and propagates `r2py_kernsmooth/` changes to the `r2py_kernsmooth` subtree remote via `git subtree push`.

The internal structure of `r2py_kernsmooth/` is as follows.

Path (relative to `r2py_kernsmooth/`)	Contents
`r2py_kernsmooth/__init__.py`	The main Python module; defines the public API (`__all__`) and all wrapper functions.
`meson.build`	Meson build definition: f2py wrapper generation, NumPy include discovery, BLAS detection, and extension module compilation.
`pyproject.toml`	Package metadata, build-system declaration (`meson-python`), runtime dependencies, and cibuildwheel platform configuration.
`src/`	Thirteen Fortran 77 source files: eleven from the original `KernSmooth` package and two LINPACK routines sourced from Netlib.
`tests/`	Test scripts for validating the package against the R `KernSmooth` package.
`pip_install.sh`	Cluster batch script for local development installation via `pip install -no-build-isolation`.
`.github/workflows/ python-publish.yml`	GitHub Actions workflow; builds binary wheels for five platforms and publishes them to PyPI on every release via OIDC trusted publishing.

1.3 Automated Workflow Infrastructure

This project employs **Claude Code** (Anthropic) as an AI-assisted development environment. Recurring multi-step tasks are encoded as *skills* — top-level slash commands whose specifications are stored as Markdown files under `.claude/commands/`. Skills orchestrate one or more *sub-agents* whose specifications are stored under `.claude/agents/`. Both types of specification files use YAML front matter to declare a name and description, followed by a structured natural-language body describing execution steps and output schemas. Sub-agents are dispatched via the `@agent-name` notation from within a skill.

The full specifications for all agents and commands referenced in this report appear in Appendices A and B.

At the end of each phase, the `/summarize-research-report` skill was invoked:

```
/summarize-research-report
```

This produced a structured Markdown summary of the phase's activities, which was saved

to `docs/daily_summaries/`. Those summaries are the primary source material for this report.

2 Phase 1: Static Analysis of the KernSmooth R Package

2.1 Motivation

Before any translation work could begin, a precise, machine-readable map of the R package's internal structure was required. The goals of this phase were:

1. Classify every R function by dependency type: calls to R standard-library functions, calls to other functions defined within the same (or other self-written related) R source file, and calls through the Fortran foreign-function interface.
2. Establish the public API surface and the complete internal call graph.
3. Understand the Fortran integration layer and the symbol-registration convention it uses.
4. Produce a comprehensive architecture document to serve as a reference throughout all subsequent phases.

Without this foundation, later phases would need to infer architectural facts on a per-function basis, risking inconsistency and missed dependencies. The structural analysis JSON produced here also serves as a direct input to the language-dependency extraction skill in Phase 3 and to the conversion ordering logic in Phase 4.

2.2 R Dependency Analysis

The `/analyze-r-folder-dependencies` skill (Appendix B, §B.1) was invoked against the R source directory:

```
/analyze-r-folder-dependencies KernSmooth/R/
```

The skill performs a recursive scan of the target folder for `.R` and `.r` files, then dispatches the `@analyze-r-file-dependencies` sub-agent (Appendix A, §A.1) on each file. Because `KernSmooth` packages all R code into a single source file (`KernSmooth/R/all.R`), exactly one agent invocation was required.

The sub-agent parses the R file, identifies every user-defined function, and classifies each function call in its body into one of three mutually exclusive categories:

`language_dependencies` Calls to R built-in functions or functions from installed packages, identified by namespace prefix or by not being locally defined (e.g. `fft`, `var`, `dnorm`, `stats::quantile`).

`internal_dependencies` Calls to other functions defined within the same R file.

external_dependencies Calls via `.Fortran()`, `.C()`, `.Call()`, or `.External()` to compiled native-code entry points.

The result is a JSON object keyed by function name, with each value containing the three dependency arrays, each listing unique function names for the dependencies. The output was written to `structural_analysis/R/all.json`.

2.3 Architecture Synthesis

To produce a comprehensive architecture document, the following files were read in parallel and synthesised into `docs/architecture-analysis.md`:

File	Purpose
`KernSmooth/DESCRIPTION`	Package metadata, authorship, license, declared dependencies.
`KernSmooth/NAMESPACE`	Exported symbols; `useDynLib` registration directive.
`structural_analysis/R/all.json`	Per-function dependency classification (produced above).
`structural_analysis/dependency_levels.csv`	Function call-depth hierarchy (pre-existing artefact).
`KernSmooth/man/bkde.Rd`	Density estimation API and algorithm description.
`KernSmooth/man/bkde2D.Rd`	Two-dimensional density estimation API.
`KernSmooth/man/bkfe.Rd`	Kernel functional estimator API.
`KernSmooth/man/dpih.Rd`	Histogram bin-width selector API.
`KernSmooth/man/dpik.Rd`	Kernel density bandwidth selector API.
`KernSmooth/man/dpill.Rd`	Local linear regression bandwidth selector API.
`KernSmooth/man/locpoly.Rd`	Local polynomial estimator API.

A check for `KernSmooth/vignettes/` confirmed that the directory does not exist; the package has no vignette sources.

2.4 Key Findings

2.4.1 Package Scope and External Dependencies

`KernSmooth` has *zero* runtime dependencies on contributed CRAN packages. It imports exactly five functions from the base `stats` package: `dbeta`, `dnorm`, `fft`, `quantile`, and `var`. `MASS` and `carData` appear in the `Suggests` field of `DESCRIPTION` but are referenced only in `man/` examples and play no role in the package's computational logic. This minimal dependency footprint simplifies the Python port considerably.

2.4.2 Function Inventory

`R/all.R` defines exactly 16 functions in a single file. Seven are exported via `NAMESPACE`; nine are internal (unexported).

Function	Exported	Level	Fortran entry point
`bkde`	Yes	0	`F_linbin` (via `linbin`)
`bkde2D`	Yes	0	`F_lbtwod` (via `linbin2D`)
`dpih`	Yes	0	`F_linbin` (via `linbin`)
`dpik`	Yes	0	`F_linbin` (via `linbin`)
`dpill`	Yes	0	multiple (via six helpers)
`locpoly`	Yes	0	`F_locpol` and `F_rlbin`
`bkfe`	Yes	1	`F_linbin` (via `linbin`)
`linbin`	No	2	`F_linbin`
`rlbin`	No	2	`F_rlbin`
`blkest`	No	1	`F_blkest`
`cpblock`	No	1	`F_cp`
`linbin2D`	No	1	`F_lbtwod`
`sdiag`	No	1	`F_sdiag`
`sstdiag`	No	1	`F_sstdg`
`.onAttach`	No	0	—
`.onUnload`	No	0	—

`bkfe` is notable for being both an *exported* function and an *internal dependency*: `dpih` and `dpik` both call it to evaluate the density functionals $\int f(x)\, f^{(r)}(x)\, \mathrm{d}x$ required for plug-in bandwidth selection.

2.4.3 Three-Level Dependency Hierarchy

The internal call graph has three levels, stored in `structural_analysis/dependency_levels.csv`:

Level 0 (entry points): Functions callable directly by the user: `bkde`, `bkde2D`, `dpih`, `dpik`, `dpill`, `.onAttach`, `.onUnload`.

Level 1 (mid-tier helpers): Functions called by one or more level-0 functions: `bkfe`, `blkest`, `cpblock`, `linbin2D`, `locpoly`, `sdiag`, `sstdiag`.

Level 2 (binning primitives): `linbin` and `rlbin`.

`linbin` (one-dimensional linear binning, backed by Fortran `F_linbin`) is the single most central function: it is called by seven distinct callers (`bkde`, `bkfe`, `dpih`, `dpik`, `locpoly`, `sdiag`, `sstdiag`), meaning that every density estimation and functional estimation path passes through it.

`dpill` is the most internally complex exported function, orchestrating six internal helpers

(`rlbin`, `cpblock`, `blkest`, `locpoly`, `sdiag`, `sstdiag`) to produce a single bandwidth estimate.

2.4.4 Fortran Integration Convention

The `NAMESPACE` file contains:

```
useDynLib(KernSmooth, .registration = TRUE, .fixes = "F_")
```

This registers all Fortran entry points at package load time and applies the `F_` prefix to their R-level names, preventing symbol conflicts. All eight Fortran-delegating R wrappers use this naming scheme. LINPACK-derived linear-algebra routines (attributed to Cleve Moler in `DESCRIPTION`) reside in `KernSmooth/src/d*` and underpin the local polynomial computations in `locpoly`, `sdiag`, and `sstdiag`.

2.4.5 Architectural Design: Binned Approximation

Every computationally non-trivial function in `KernSmooth` adopts the same two-phase strategy: (1) **bin** the raw data onto an equally-spaced grid using *linear binning*, which preserves the first moment of each data point within its bin interval (unlike simple histogram binning), and (2) **convolve** the bin counts with a kernel weight vector, either via FFT (`bkde`, `bkde2D`, `bkfe`) or via the Fortran local-polynomial routines. This trades a small, controllable bias for a reduction in computational complexity from $O(n \cdot g)$ to $O(n + g \log g)$, where n is the sample size and g is the grid size.

2.4.6 Note on Fortran Source Analysis

A complete cross-language dependency map would also require analyzing the compiled Fortran sources in `KernSmooth/src/`. The `/analyze-c-folder-dependencies` skill (Appendix B) was identified as a candidate tool for this task (it handles both C and Fortran), but this analysis has not yet been executed. Due to the nature of `KernSmooth` package, the Fortran sources are relatively self-contained and do not call back into R, so the lack of a Fortran-level dependency map has not posed a significant obstacle to subsequent phases. The Fortran code is treated as a black box that is called by the R wrappers, so the focus has been on understanding the R-level structure and dependencies.

3 Phase 2: Establishing the Python Build Infrastructure

3.1 Motivation

The conversion strategy retains the original Fortran 77 sources verbatim and relies on `f2py` (part of NumPy) to parse them and generate a Python-callable C-extension. This approach

provides a strong correctness guarantee: numerical differences between the Python and R packages can originate only in the R-to-Python wrapper translation, not in the underlying computational routines. Phase 2 addresses all build-system and packaging concerns before any wrapper code is written.

Two related objectives determine the scope of this phase. The first is local: the extension must compile and install correctly in the HPC development environment so that Python wrapper functions can be developed and tested. The second is distribution: once complete, the package must be available as pre-compiled binary wheels on PyPI for all major platforms, so that end users can install it with a plain `pip install r2py_kernsmooth` without requiring a local Fortran compiler, BLAS library, or build toolchain. Because a Fortran extension produces a platform-specific binary, a separate wheel must be built for each combination of operating system, processor architecture, and CPython version.

3.2 Build System Design

The build backend is `meson-python`, declared in `r2py_kernsmooth/pyproject.toml`:

```
[build-system]
build-backend = "mesonpy"
requires = ["meson-python", "meson", "ninja", "numpy"]

[project]
name = "r2py_kernsmooth"
version = "0.1.0"
dependencies = ["numpy", "scipy"]
```

The `meson` and `ninja` entries in `requires` ensure that PEP 517 isolated builds can locate the Meson build system and the Ninja build tool without requiring them to be pre-installed in the user's environment. The complete build definition resides in `r2py_kernsmooth/meson.build`.

3.2.1 Fortran Sources

Thirteen Fortran 77 fixed-form source files in `r2py_kernsmooth/src/` are compiled into the extension:

File	Origin
`src/blkest.f`	Original `KernSmooth` source
`src/cp.f`	Original `KernSmooth` source
`src/dgedi.f`	Original `KernSmooth` source (LINPACK LU)
`src/dgefa.f`	Original `KernSmooth` source (LINPACK LU)
`src/dgesl.f`	Original `KernSmooth` source (LINPACK LU)
`src/linbin.f`	Original `KernSmooth` source
`src/linbin2D.f`	Original `KernSmooth` source
`src/locpoly.f`	Original `KernSmooth` source
`src/rlbin.f`	Original `KernSmooth` source
`src/sdiag.f`	Original `KernSmooth` source
`src/sstdiag.f`	Original `KernSmooth` source
`src/dqrdc.f`	Added from Netlib (LINPACK QR decomposition, G. W. Stewart 1978)
`src/dqrsl.f`	Added from Netlib (LINPACK QR solve, G. W. Stewart 1978)

The two Netlib additions are necessary because `blkest.f` and `cp.f` call `dqrdc_` and `dqrsl_`, respectively. The original R package satisfies these symbols from R's own internal LINPACK copy (built into the R binary). OpenBLAS provides only BLAS levels 1–3 and does not include LINPACK QR routines. Sourcing them from Netlib is the standard remedy for this gap when porting R packages that depend on LINPACK.

The presence of LINPACK routines (both the original `dgedi/dgefa/dgesl` trio and the newly added `dqrdc/dqrsl`) was discovered iteratively during Phase 2: each link failure at `pip install` time revealed one missing symbol, which was then resolved by adding the corresponding Netlib file.

3.2.2 f2py Wrapper Generation

`f2py` (NumPy 2.4.3) is invoked as a Meson `custom_target`:

```
python -m numpy.f2py @INPUT@ -m _KernSmooth --lower
```

Because all sources are Fortran 77 fixed-form (`.f`), `f2py` generates `_KernSmoothmodule.c` and `_KernSmooth-f2pywrappers.f` (the fixed-form wrapper variant). An initial build template incorrectly named the wrapper file `_KernSmooth-f2pywrappers2.f90` (the free-form Fortran 90 variant), which caused a build failure; the distinction was identified by inspecting the `f2py` output and corrected in the `meson.build outputs` list.

The `-lower` flag lowercases all Fortran symbol names in the generated C wrapper, which is required for standard GFortran symbol conventions.

3.2.3 NumPy Include Directories

The Meson configure step must locate NumPy's C header files at configure time (not at install time), since they are required to compile `fortranobject.c`, the f2py runtime support file. Two `run_command` calls retrieve these directories:

```
incdir_numpy = run_command(py,
  ['-c', 'import numpy; print(numpy.get_include())'],
  check: true).stdout().strip()
incdir_f2py = run_command(py,
  ['-c', 'import numpy.f2py; print(numpy.f2py.get_include())'],
  check: true).stdout().strip()
```

`fortranobject.c` (located in `incdir_f2py`) is included directly in the extension module's source list.

3.2.4 BLAS Discovery

The HPC node (Red Hat Enterprise Linux 9, GFortran 15.2.0, `ld.bfd 2.35.2-67`) provides OpenBLAS at `/usr/lib64/libopenblaso.so.0`, but the unversioned symlink (`libopenblaso.so`) normally installed by the `openblas-openmp-devel` package is absent. Meson's standard `find_library` and `dependency('blas')` mechanisms cannot resolve a bare `.so.0` file. The following discovery chain in `meson.build` addresses this while remaining portable across all build platforms:

```
blas_dep = dependency('blas', required: false)
if not blas_dep.found()
  blas_dep = fc.find_library('openblaso', required: false)
endif
if not blas_dep.found()
  blas_dep = fc.find_library('openblas', required: false)
endif
if not blas_dep.found()
  blas_dep = fc.find_library('blas', required: false)
endif
if not blas_dep.found()
  fs = import('fs')
  if fs.is_file('/usr/lib64/libopenblaso.so.0')
    blas_dep = declare_dependency(link_args:
      ['/usr/lib64/libopenblaso.so.0'])
  else
    error('BLAS not found. Install openblas-devel (Linux) '
        + 'or run: brew install openblas (macOS).')
  endif
endif
```

The first four stages cover systems where BLAS is installed conventionally: `dependency('blas')` queries `pkg-config`; the three `find_library` calls search standard library directories for `openblaso`, `openblas`, and `blas` by name. The fifth and final stage is specific to the HPC cluster, where the `openblas-openmp-devel` package that would provide the unversioned symlink is absent but `/usr/lib64/libopenblaso.so.0` is present. Meson's `fs.is_file()` guard verifies the file's existence before constructing the link argument, ensuring that on any platform where the hardcoded path is absent—such as macOS or a standard Ubuntu installation—the build fails at configure time with a clear, actionable error rather than proceeding silently to a linker failure.

3.2.5 Package Structure

The Python package exposes two components:

`r2py_kernsmooth/_KernSmooth` The compiled Fortran C-extension (private backend). All 11 original subroutines are accessible via `r2py_kernsmooth._KernSmooth.`*`name`*.

`r2py_kernsmooth/__init__.py` The Python wrapper layer that defines the public API.

An early design exposed the raw Fortran subroutines directly in the `r2py_kernsmooth` namespace. This was identified as architecturally incorrect: the Fortran subroutines have a low-level calling convention (all arguments passed by reference, no input validation, no Python-idiomatic defaults) and should not be part of the public API. `__init__.py` was revised to contain only `from . import _KernSmooth` and `__all__ = []`, deferring all public-API decisions to the wrapper functions to be written in Phase 4.

3.3 Installation

The package is installed into the `r-to-python` conda environment via:

```
pip install --no-build-isolation .
```

The `-no-build-isolation` flag is required because the conda environment provides `numpy` and `meson-python` directly, and the Meson configure step calls `numpy.get_include()` and `numpy.f2py.get_include()` at configure time. With build isolation, pip would install `numpy` into an isolated build environment, producing a different path that may conflict with the runtime environment.

The full invocation, including the cluster's `module load gcc` and `conda activate r-to-python` steps, is captured in `r2py_kernsmooth/pip_install.sh`. This script is also suitable for submission as a cluster batch job (`#$ -pe smp 1`, `#$ -q long` directives are included).

After successful installation, `import r2py_kernsmooth` and `import r2py_kernsmooth._KernSmooth` both succeed without error.

3.4 PyPI Distribution Infrastructure

A package containing compiled Fortran code produces a platform-specific binary wheel that cannot be installed on a different operating system or processor architecture. Distributing pre-compiled wheels for all major platforms requires building them in CI on a representative of each target environment; the tool used for this purpose is `cibuildwheel`, the standard wheel-building tool in the scientific Python ecosystem, which manages the appropriate Docker containers for Linux builds and native runners for macOS and Windows automatically.

3.4.1 GitHub Actions Workflow

The workflow `r2py_kernsmooth/.github/workflows/python-publish.yml` is triggered on every GitHub Release publication. It consists of three jobs.

The **`build_wheels`** job runs on the runner matrix `[ubuntu-latest, ubuntu-24.04-arm, macos-13, macos-latest, windows-latest]`. `ubuntu-latest` (Linux x86_64) and `ubuntu-24.04-arm` (Linux aarch64, a native ARM64 runner that requires no QEMU emulation) each launch both `manylinux` (glibc) and `musllinux` (Alpine/musl-libc) Docker containers managed by cibuildwheel, producing PEP 600-compliant wheels for all configured Python versions on both Linux libc variants. `macos-13` and `macos-latest` build macOS x86_64 (Intel) and macOS arm64 (Apple Silicon) wheels natively, respectively. `windows-latest` builds Windows x86_64 wheels natively; a preceding step using the `msys2/setup-msys2` action installs `gfortran`, OpenBLAS, and `pkg-config` into the MSYS2 MinGW64 environment before cibuildwheel is invoked. All configuration options are read from the `[tool.cibuildwheel]` table in `pyproject.toml` (§3.4.2).

The **`build_sdist`** job runs on `ubuntu-latest`, executes `python -m build -sdist`, and uploads the resulting `.tar.gz` archive. The sdist allows users on platforms with no pre-built wheel—and users with non-standard build environments—to compile from source.

The **`pypi-publish`** job, which depends on both preceding jobs, downloads all wheel and sdist artifacts into a single `dist/` directory and publishes them via `pypa/gh-action-pypi-publish@release/v1`. Authentication uses OIDC-based *trusted publishing*: the job requests a short-lived identity token (`id-token: write` permission) that PyPI validates against the registered GitHub repository and Actions environment (`environment: name: pypi`), eliminating the need to store any long-lived credential in GitHub Secrets.

3.4.2 Platform Coverage

The `[tool.cibuildwheel]` configuration in `pyproject.toml` is as follows:

```
[build-system]
build-backend = "mesonpy"
requires = ["meson-python", "meson", "ninja", "numpy"]
```

```
[project]
name = "r2py_kernsmooth"
version = "0.1.0"
requires-python = ">=3.10"
dependencies = ["numpy", "scipy"]

[tool.cibuildwheel]
build = "cp310-* cp311-* cp312-* cp313-* cp314-*"
skip = ["*-manylinux_i686"]
build-frontend = "build"

[tool.cibuildwheel.linux]
before-all = "command -v dnf && dnf install -y openblas-devel || command -v
    yum && yum install -y openblas-devel || apk add --no-cache gfortran
    openblas-dev"

[tool.cibuildwheel.macos]
before-all = "brew install openblas"

[tool.cibuildwheel.macos.environment]
PKG_CONFIG_PATH = "$(brew --prefix openblas)/lib/pkgconfig"

[tool.cibuildwheel.windows]
before-build = "pip install delvewheel"
repair-wheel-command = "delvewheel repair -w {dest_dir} {wheel}"

[tool.cibuildwheel.windows.environment]
PATH = "C:\\msys64\\mingw64\\bin;{PATH}"
PKG_CONFIG_PATH = "C:\\msys64\\mingw64\\lib\\pkgconfig"
```

The principal design decisions are as follows.

Python versions. `cp310-*` through `cp314-*` cover CPython 3.10–3.14; 32-bit Linux (`*-manylinux_i686`) is the only excluded target.

Build frontend. `build-frontend = "build"` invokes the PEP 517 `build` tool rather than `pip wheel`, which is more compatible with `meson-python`.

Linux dependencies. `before-all` runs once per container environment (not once per Python version) and installs OpenBLAS via a package-manager detection chain: `dnf` for manylinux_2_28 / AlmaLinux 8 base images, `yum` for manylinux2014 / CentOS 7 base images, and `apk` for musllinux / Alpine base images. The `apk` branch additionally installs `gfortran`, since Alpine-based musllinux containers do not include it by default, unlike the manylinux containers.

macOS dependencies. `brew install openblas` installs OpenBLAS via Homebrew. The `PKG_CONFIG_PATH` override directs Meson's `dependency('blas')` call to the

Homebrew installation via `pkg-config`; without it, the first four stages of the BLAS discovery chain exhaust their search and the configure-time error in the fifth stage is raised.

Windows toolchain. The `PATH` and `PKG_CONFIG_PATH` overrides expose the MSYS2 MinGW64 installation (set up by the preceding workflow step) to Meson inside each cibuildwheel build subprocess. `delvewheel`—the Windows counterpart of Linux `auditwheel` and macOS `delocate`—bundles the OpenBLAS DLL into the wheel, making it self-contained for users without MSYS2.

Across the five runners, up to 35 binary wheels are produced per release (five Python versions × seven platform/libc combinations: Linux x86_64 manylinux, Linux x86_64 musllinux, Linux aarch64 manylinux, Linux aarch64 musllinux, macOS x86_64, macOS arm64, Windows x86_64).

3.4.3 Coverage Limitations

Three platform categories present in established scientific Python distributions such as scikit-learn are not covered.

Windows arm64. Native wheels for Qualcomm Snapdragon-based Windows PCs require a Fortran compiler for the Windows arm64 ABI. The only readily available option is LLVM `flang` via the MSYS2 CLANGARM64 environment, but its integration with `f2py` and Meson is not yet production-stable. As an interim measure, Windows arm64 users can install the `win_amd64` wheel, which runs transparently under the Windows Prism x86-64 emulation layer.

Linux ppc64le and s390x. IBM POWER and IBM Z mainframe architectures would require QEMU emulation in CI, at approximately 40 minutes per Python version per architecture. Given the extreme rarity of these platforms in scientific Python workflows, they are not included.

Free-threaded Python (cp313t, cp314t). The experimental no-GIL CPython builds are supported by scikit-learn on selected platforms. Compatibility of the `f2py`-generated C extension with the free-threaded CPython ABI has not been verified and is deferred until the free-threaded build reaches stable status.

Before the first release, `r2py_kernsmooth` must be registered on PyPI and the GitHub Actions `pypi` environment configured as a trusted publisher in the project's PyPI settings.

4 Phase 3: Language Dependency Catalog and Conversion Guides

4.1 Motivation

The automated function conversion in Phase 4 maps each R construct to its Python equivalent. For most R standard-library functions, a naïve translation to the apparent NumPy or SciPy analogue introduces silent numerical errors. Documented examples include: R's `fft(inverse=TRUE)` is unnormalized while NumPy's `ifft` normalizes by $1/N$; R's `var()` uses Bessel's correction while `np.var` does not; R's `as.integer()` targets 32-bit integers while NumPy defaults to 64-bit.

One may argue that these discrepancies are well-known and can be handled on a case-by-case basis during the conversion process in Phase 4. However, due to the random nature of the LLM, we cannot guarantee that all language dependencies will be correctly identified and translated whenever a function is converted. If there is not a unified, machine-readable reference of all language dependencies and their translation nuances, then the risk of silent numerical errors in the final Python package is unacceptably high. (Even if the translation is mostly correct, we cannot ensure the consistency of choices or styles, which can make debugging significantly difficult.) Also, there are many language dependencies in R that have so many variations in their usage that it is hard to guarantee a correct translation without a comprehensive reference. For example, the `as.double()` function is used in many different contexts (such as on scalars, vectors, matrices, or even data frames), and its translation to Python can vary depending on the context. If we rely directly on the LLM to identify and translate each usage correctly during the conversion process, we may miss some cases or introduce errors. By contrast, if we systematically extract all language dependencies and their call sites in Phase 3, and then generate detailed conversion guides for each scenario for each dependency, we can ensure that the translation in Phase 4 is consistent and accurate across all functions.

Phase 3 systematically catalogs every R standard-library function called in `all.R`, records its precise call sites, and generates a dedicated translation guide for each. By encoding these nuances in machine-readable Markdown files that are later fed to the conversion sub-agent, the translation quality is improved without depending on the agent's training data knowledge of R semantics solely. Moreover, although we do not modify the machine-generated conversion guides since they are already of high quality, the presence of a structured reference allows for easy manual review and correction of any inaccuracies or omissions in the guides, which can then be fed back into the conversion process to further reduce the risk of silent errors or bugs, giving developers more control over the highly automated translation process.

4.2 Language Dependency Extraction

The `/extract-r-folder-language-dependencies` skill (Appendix B, §B.2) was invoked:

```
/extract-r-folder-language-dependencies \
    KernSmooth/R/ \
    structural_analysis/R/
```

The skill dispatches the `@extract-r-file-language-dependencies` sub-agent (Appendix A, §A.2) on each R file together with its pre-computed structural-analysis JSON. The sub-agent reads the structural JSON to identify the language dependencies of each function, then rescans the R source line by line to record, for every individual call site: the dependency name, the enclosing function name, the line number, and the complete call expression (including multi-line calls).

The result was written to `language_dependency_analysis/R/all.csv`: a 509-row, 4-column CSV (`language_dependency`, `function_name`, `line_number`, `call_body`) covering all 16 functions in `all.R`.

A utility script `language_dependency_analysis/combine_csvs.py` was also written to:

- Recursively glob all `.csv` files under `language_dependency_analysis/R/`.
- Insert a `file_path` column (the relative path of the source R file within the `R/` folder, `.csv` extension replaced with `.R`).
- Concatenate, sort by `[language_dependency, file_path, function_name, line_number]`, and save the combined result to `language_dependency_analysis/combined_table.csv`.

This script is designed to sort the entries in the original `all.csv` to make the entries for the same language dependency contiguous and convenient to be extracted for further tasks. It also makes it easier to support future analysis of additional R files beyond `all.R`.

4.3 Conversion Guide Generation

The `/generate-language-dependency-conversion-guides` skill (Appendix B, §B.2) was invoked:

```
/generate-language-dependency-conversion-guides \
    KernSmooth/R/ \
    language_dependency_analysis/combined_table.csv
```

The skill identified 46 unique `language_dependency` values in the combined CSV and launched 46 instances of the `@generate-language-dependency-conversion-guide` sub-agent (Appendix A, §A.2) in a single parallel batch. Each agent:

1. Received the CSV subset for its assigned dependency.
2. Read the corresponding source lines in `all.R` for contextual understanding of argument types and surrounding logic.

3. Consulted R online documentation (https://www.rdocumentation.org) where the function's behavior was complex or non-obvious.
4. Produced a structured Markdown guide specifying the correct Python translation strategy, with concrete code examples.

The 46 resulting guides were saved to `language_dependency_analysis/conversion_guides/` with filenames matching the dependency name (e.g. `fft.md`, `var.md`, `match.arg.md`).

4.4 Selected Translation Nuances

The following are the most consequential findings from the conversion guides, as they directly informed specific decisions in Phase 4. Together, they constitute the principal risk of silent numerical error in any R-to-Python translation of this package.

4.4.1 FFT Normalization (`fft.md`)

R's `fft(..., inverse=TRUE)` returns the *raw, unnormalized* inverse DFT: for an N-point transform it returns $\sum_{k=0}^{N-1} X_k e^{2\pi i k n/N}$ without dividing by N. NumPy's `np.fft.ifft` normalizes by $1/N$ by default, as per the conventional definition. The R source explicitly divides the inverse-FFT result by the grid size P (or $P_1 \times P_2$ for two-dimensional transforms) after computing `Re()`. When porting to NumPy, those division operations must be *removed*.

4.4.2 Fortran FFI Type Coercions (`as.double.md`, `as.integer.md`)

R's pre-allocation idiom `as.double(rep(0, n))` maps to `np.zeros(n, dtype=np.float64)`. `as.integer(Lvec)` on a vector maps to `Lvec.astype(np.int32)`. The 32-bit integer constraint is non-obvious but critical: standard Fortran `INTEGER` is 32 bits; passing `np.int64` arrays would silently corrupt the values seen by the Fortran subroutine on a 64-bit system.

4.4.3 Sample Variance Denominator (`var.md`)

R's `var(x)` uses Bessel's correction ($n-1$ denominator), i.e. it computes the *sample* variance. NumPy's `np.var` defaults to the *population* variance (n denominator, `ddof=0`). All occurrences of `sqrt(var(x))` in the R source must be translated as `np.std(x, ddof=1)`.

4.4.4 Optional Argument Detection (`missing.md`)

R's `missing(param)` returns `TRUE` if the caller did not supply the argument. Python has no built-in equivalent. The idiomatic replacement is the `None` sentinel default: `def f(x, param=None)` with `if param is None:` guards inside the function body.

4.4.5 Negative-Base Fractional Exponentiation (`sqrt.md`)

Several bandwidth formulas involve odd roots of quantities that may be negative, e.g. $(-3\sqrt{2/\pi}\,/\,\hat{\psi}_6)^{1/7}$. R silently computes real-valued odd roots of negative numbers via its internal rule for fractional exponents. Python's `**` operator raises `ValueError` for negative bases with non-integer exponents. The correct idiom is

```
math.copysign(abs(x) ** (1.0 / n), x)
```

4.4.6 Array Memory Order (`matrix.md`)

R's `matrix()` and Fortran both use *column-major* (Fortran-order) storage. NumPy defaults to *row-major* (C-order). When reshaping arrays produced by Fortran subroutines, `np.reshape(..., order='F')` is required.

4.4.7 Package Lifecycle Hooks (`library.dynam.unload.md`)

R provides package attach/detach lifecycle hooks via `.onAttach` and `.onUnload`. `library.dynam.unload` has no Python equivalent: CPython does not safely support unloading native extension modules. The guide documents that the `.onUnload` body should either be omitted or replaced with an `atexit` callback for non-library cleanup tasks.

4.4.8 Partial String Matching (`match.arg.md`)

R's `match.arg(arg, choices)` validates `arg` against a vector of allowed strings and supports *unambiguous prefix matching* (e.g. `"ep"` matches `"epanechnikov"`). No standard Python equivalent exists. The guide specifies a reusable helper function that performs exact match, then unique-prefix match, then raises a descriptive error.

4.4.9 R `switch` on Strings (`switch.md`)

R's `switch(key, case1=expr1, case2=expr2, ...)` on string keys translates cleanly to a Python dictionary lookup. For cheap scalar computations (as in `KernSmooth`), a plain `dict` of values is sufficient; a `dict` of `lambda` functions is available for cases requiring lazy evaluation.

5 Phase 4: Automated Function Conversion and Package Assembly

5.1 Motivation

With the dependency map (Phase 1), build infrastructure (Phase 2), and translation guides (Phase 3) in place, Phase 4 executes the actual conversion. Two challenges make this

non-trivial even with the preparation work done:

1. Functions must be converted in dependency order: when the conversion agent for a function reads the JSON files of its callees to understand their Python calling conventions, those files must already exist.
2. The Fortran FFI calling convention in Python (via `f2py`) differs fundamentally from R's `.Fortran()` in how it handles output arguments. This difference is invisible from the R source and therefore not naturally handled by pattern matching; it requires targeted post-conversion patching.

5.2 Function Conversion

The `/convert-r-file-to-python` skill (Appendix B, §B.3) was invoked:

```
/convert-r-file-to-python \
    KernSmooth/R/all.R \
    structural_analysis/R/all.json \
    structural_analysis/dependency_levels.csv \
    language_dependency_analysis/conversion_guides/ \
    conversion_results/R/
```

The skill parses `dependency_levels.csv`, filters for `r_file == all.R`, and derives a topological ordering from leaves to roots. It then invokes the `@convert-r-function-to-python` sub-agent (Appendix A, §A.3) once per function, *sequentially* in topological order, providing each agent with:

- The R source fragment for the function.
- The function's JSON dependency map (from `all.json`).
- The folder of 46 language-dependency conversion guides.
- The output folder `conversion_results/R/all.R/` for the resulting JSON artefact.

The conversion order was:

Level	Functions (in conversion order)
2 (deepest leaves)	`linbin`, `rlbin`
1 (intermediate)	`bkfe`, `blkest`, `cpblock`, `linbin2D`, `locpoly`, `sdiag`, `sstdiag`
0 (roots)	`.onAttach`, `.onUnload`, `bkde`, `bkde2D`, `dpih`, `dpik`, `dpill`

The 16 resulting JSON artefacts were written to `conversion_results/R/all.R/`. (Two artefacts — `.onAttach.json` and `.onUnload.json` — use leading-dot filenames and are not visible to standard `ls`; they require `ls -a` to discover.)

5.2.1 Notable Per-Function Translation Decisions

`linbin/rlbin/linbin2D/blkest/cpblock` `.Fortran(F_xxx, ...)[[k]]` mapped to `_KernSmooth.xxx(...)` with pre-allocated NumPy output buffers.

`bkfe` Hermite polynomial recurrence loop translated directly; the manual `/P` division present in the R source was removed per the `fft.md` guide, since NumPy's `np.fft.ifft` already normalizes.

`locpoly` Dual density/regression branch (controlled by whether the `y` argument is `None`); a variable-length bandwidth discretization loop; a 19-argument Fortran subroutine call; and a `math.gamma(drv+1)` post-multiplier for the derivative-order correction.

`sdiag/sstdiag` The `ss`, `Smat`, `uu`, `Umat` matrices are flattened column-major (`order='F'`) before being passed to the Fortran subroutine, consistent with the `matrix.md` guide.

`dpih/dpik` Negative-base fractional exponents (e.g. $(-3\sqrt{2/\pi}/\hat{\psi}_6)^{1/7}$) replaced with `math.copysign(abs(val)**(1/k), val)` per the `sqrt.md` guide.

`bkde2D` Two-dimensional FFT convolution via `np.fft.fft2`/`np.fft.ifft2`; wrap-around kernel construction using Python slice notation to replicate R's decreasing-index sequences.

`.onAttach` `packageStartupMessage` translated to `print(..., file=sys.stderr)`.

`.onUnload` `library.dynam.unload` body omitted per `library.dynam.unload.md`; function registered as an `atexit` callback at module level.

5.3 Package Assembly

The `/combine-python-functions-into-file` skill (Appendix B, §B.3) was invoked:

```
/combine-python-functions-into-file \
    conversion_results/R/all.R/ \
    r2py_kernsmooth/r2py_kernsmooth/__init__.py
```

Before writing, the skill performed a structural scan of `meson.build` and the existing `__init__.py` to verify the correct import path for the Fortran extension (`from . import _KernSmooth`), then corrected discrepancies found in the per-function JSON artefacts:

Original (in JSON artefacts)	Corrected
`from r2py_kernsmooth import _KernSmooth`	`from . import _KernSmooth`
`from r2py_kernsmooth.linbin import linbin`	removed (functions are co-located)
`from scipy.stats import norm` (duplicate)	merged: `from scipy.stats import beta as beta_dist, norm`

The final import block, sorted standard library → third-party → local, was:

```
import atexit
import math
```

```
import sys
import warnings

import numpy as np
from scipy.stats import beta as beta_dist, norm

from . import _KernSmooth
```

The `__all__` list was populated with all 14 public function names (excluding `_on_attach` and `_on_unload`). Functions were written to the file in leaf-to-root order, ensuring all callees are defined before callers. Syntax was confirmed immediately by `ast.parse()`.

5.4 The f2py Return-Value Bug

5.4.1 Observed Symptom

After assembly, running the test script raised:

```
TypeError: 'NoneType' object is not subscriptable
  File "__init__.py", line 36, in linbin
    gcnts = _KernSmooth.linbin(...)[6]
```

5.4.2 Root Cause: R `.Fortran()` vs. f2py Semantics

R's `.Fortran()` always returns a *named list of all arguments*, including modified output arrays. The R idiom `.Fortran(F_linbin, ...)$gcnts` (or equivalently `[[6]]`) extracts the modified output array from that list.

f2py-wrapped Fortran *subroutines* return `None` unconditionally. Output array arguments are modified in-place via C pointer; the Python caller reads results by examining the pre-allocated arrays that were passed in. Scalar output arguments require special treatment: passing a Python `np.float64(0.0)` (immutable) does not allow Fortran to write back. The fix is to use a length-1 NumPy array (`np.zeros(1, dtype=np.float64)`), which is mutable and passes a writable pointer.

The `@convert-r-function-to-python` agent for `linbin` had generated `[6]` subscripting on the `None` return. The agent for `rlbin` (converted first) had correctly identified the in-place semantics and avoided this. The inconsistency arose because each agent operated independently.

5.4.3 Fortran Output Argument Audit

All Fortran source files were read to identify output arguments:

Subroutine	Output argument(s)	Type
`linbin`	`gcnts(*)`	`double precision` array
`lbtwod`	`gcnts(*)`	`double precision` array
`blkest`	`sigsqe`, `th22e`, `th24e`	`double precision` *scalars*
`cp`	`Cpvals(Nmax)`	`double precision` array
`locpol`	`cvest(*)`	`double precision` array
`sdiag`	`Sdg(*)`	`double precision` array
`sstdg`	`SSTd(*)`	`double precision` array

5.4.4 Patches Applied

Seven targeted edits were made to `r2py_kernsmooth/r2py_kernsmooth/__init__.py`:

1. **`linbin`** — Removed `[6]` subscript; read `gcnts` (pre-allocated `np.zeros(M)`) directly after the call.
2. **`linbin2D`** — Removed `out = ...` and `out[8]`; read `gcnts` then reshaped with `order='F'`.
3. **`blkest`** — Changed `sigsqe/th22e/th24e` from `np.float64(0.0)` to `np.zeros(1, dtype=np.float64)`; return values read as `sigsqe[0]`, `th22e[0]`, `th24e[0]`.
4. **`cpblock`** — Removed `out = ...` and `out[12]`; used `Cpvals` (modified in-place) directly in `np.argmin`.
5. **`locpoly`** — Removed `out = ...` and `out[18]`; applied `math.gamma(drv+1)` multiplier to `curvest` directly.
6. **`sdiag`** — Removed `out = ...` and `out[16]`; returned `Sdg` directly.
7. **`sstdiag`** — Removed `out = ...` and `out[18]`; returned `SSTd` directly.

A post-patch `grep` confirmed zero remaining stale `out[integer]` subscripts on Fortran call returns. `ast.parse()` re-confirmed syntax validity.

5.5 Output Validation

The test script `r2py_kernsmooth/tests/test.py` invoked `r2py_kernsmooth.bkde(np.array([1,2,3,4,5]))` with default parameters and printed the result. The output was verified analytically:

- **Grid range:** $[-4.244, 10.244]$. Expected $[\min(x) - \tau h, \max(x) + \tau h]$ where $\tau = 4$ and $h \approx \delta_0 \cdot (243/175)^{1/5} \cdot \sigma \approx 0.776 \times 1.069 \times 1.581 \approx 1.312$, giving $[-4.248, 10.248]$. ✓
- **Grid size:** 401 points (default `gridsize=401`). ✓
- **Peak location:** Maximum density ≈ 0.18989 at $x = 3.0$, matching the mean and median of the symmetric input $\{1, 2, 3, 4, 5\}$. ✓
- **Symmetry:** The density array is a palindrome. ✓
- **Non-negativity:** All values > 0. ✓

- **Normalization:** Trapezoidal integral ≈ 1.0. ✓

After Phase 4, the package was in a runnable state with all 16 functions converted.

6 Phase 5: Code Quality Audit and Namespace Refinement

6.1 Motivation

The automated conversion pipeline prioritises functional correctness. Phase 5 is a manual audit and refactoring pass that addresses three categories of follow-on concerns:

1. **Type annotations:** The conversion agent was instructed to produce complete annotations, but the initial assembly contained bare container types (`np.ndarray`, `dict`) without subtype parameters.
2. **Namespace hygiene:** Module-level imports (`import numpy as np`, etc.) are visible as package attributes; best-practice conventions for managing this were investigated and applied.
3. **Code quality:** A systematic audit identified dead code left over from the R porting process, a latent crash bug, structural duplication, and PEP 8 violations, all of which were resolved.

6.2 Type Annotation Completion

All 14 public function signatures and both private function signatures were reviewed. The following annotation deficiencies were found and corrected:

- Bare `np.ndarray` parameters and return types, which lack dtype information.
- Bare `dict`, `tuple`, and `list` without element type parameters.
- Optional parameters with default `None` missing the `| None` union form (e.g. `bandwidth`, `range_x`, `y`, `degree`).

`from typing import Any` was added to the import block. Every array annotation was expanded to `np.ndarray[Any, np.dtype[np.float64]]`, reflecting the exclusive use of 64-bit floating-point arrays throughout the module. Return types of functions returning dictionaries were updated:

- `rlbin`, `locpoly`, `sdiag`, `sstdiag`, `bkde`, `bkde2D` → `dict[str, np.ndarray[Any, np.dtype[np.float64]]]`
- `blkest` → `dict[str, np.float64]` (scalar values extracted from length-1 arrays)

The `NDArray[np.float64]` alias from `numpy.typing` was evaluated and rejected in favour of the explicit `np.ndarray` base type.

6.3 Namespace Privacy Investigation

Importing `numpy as np` at module level exposes `r2py_kernsmooth.np` as a public package attribute, which is undesirable but difficult to prevent. Two strategies were evaluated:

Underscore aliasing All imports renamed to `_np`, `_math`, etc., with global replacement of all references throughout the file. Implemented and tested (confirmed `hasattr(r2py_kernsmooth, 'np')` returned `False`), then **reverted**. A survey of major scientific Python packages (NumPy, SciPy, pandas) confirmed that underscore-aliasing of dependency imports is not standard practice; all use conventional import names at module level.

`del` cleanup Applicable only to imports used *exclusively at module initialization time*. Only `atexit` qualified: it appeared solely in the `atexit.register(_on_unload)` call. `del atexit` was added immediately after that registration. No other import could be safely deleted because all others (`math`, `np`, etc.) are referenced inside function bodies, where deletion would cause `NameError` at call time.

PEP 562 (`module.__getattr__`) was also evaluated but found inapplicable: it is invoked only for *missing* attributes and cannot hide names already present in `__dict__`. The `__all__` list was confirmed as the authoritative and conventional public-API contract.

6.4 Full Code Audit and Systematic Refactoring

A line-by-line review of `__init__.py` identified seven defect categories. All were resolved in a single comprehensive rewrite.

6.4.1 Dead Code from R Porting

`_on_attach` and `_on_unload` were direct transliterations of R's `.onAttach` and `.onUnload` lifecycle hooks. Python's import machinery provides no equivalent of R's package attach/detach events; neither function was reachable from any Python code path. Their presence also rendered `import atexit`, `import sys`, and the subsequently added `del atexit` statement all useless. All were removed.

An R-era source-control comment inside `cpblock` (`# remove unused 'q' 2007-07-10`) was also removed; it had no meaning in a Python file and referenced an event in the R package's version history.

6.4.2 Silent Crash Bug in Three Public Functions

`locpoly`, `sdiag`, and `sstdiag` all declared `bandwidth: np.ndarray[...] | None = None` (a `None` default with a `| None` annotation added during type annotation completion) yet called `np.asarray(bandwidth, dtype=np.float64)` unconditionally. Passing `None` produced a 0-d array with value `nan`; a subsequent `len(bandwidth)` call on that 0-d array raised `TypeError: len() of unsized object`.

The annotation advertised `None` as a semantically valid input while the implementation crashed silently. In the internal call graph, the bug was unreachable because all three functions are always called from `dpill` with a concrete bandwidth value — but any external caller relying on the type annotation would have encountered it immediately. The resolution mirrors R's `!missing(bandwidth)` semantics via the `None`-sentinel idiom from the `missing.md` conversion guide: the bandwidth positivity check in `locpoly` was corrected from an unconditional form to:

```
if bandwidth is not None and np.any(np.asarray(bandwidth) <= 0):
    raise ValueError("'bandwidth' must be strictly positive")
```

This matches R's `if (!missing(bandwidth) && any(bandwidth <= 0))` guard. `sdiag` and `sstdiag` carry no bandwidth positivity check in the R source and required no guard.

6.4.3 Structural Code Duplication

Two logic blocks appeared identically across three functions each:

Bandwidth discretization (27 lines) In `locpoly`, `sdiag`, and `sstdiag`: handles the scalar-bandwidth path, the vector-of-length-M path, and an error path. Extracted into:

```
def _discretize_bandwidth(bandwidth, M, delta, Q, tau):
    ...  # returns (hdisc, Lvec, indic, Q)
```

The extraction also corrected a robustness defect in the scalar-detection logic. The original code used `elif len(bandwidth) == 1`, which raises `TypeError` for 0-d NumPy arrays produced by `np.asarray` on a plain Python `float` or `np.float64` scalar. The corrected guard, consistent with the pattern already used in `bkde2D`, is:

```
if bw.ndim == 0 or len(bw) == 1:
    ...
```

Prefix-match validation (8 lines) In `bkde` (for `kernel`), `dpih` (for `scalest`), and `dpik` (for both `kernel` and `scalest`). Extracted into:

```
def _resolve_choice(val, choices):
    ...  # exact match -> unique prefix match -> error
```

The error message format was subsequently corrected to match R's `match.arg` output; see Section 6.4.5.

Both helpers are private (underscore-prefixed) and absent from `__all__`, preserving the public API surface.

6.4.4 PEP 8 Signature Reformatting

All function signatures exceeding 88 characters were reformatted to multi-line style with one parameter per line. The most extreme cases (`locpoly`, `sdiag`, `sstdiag`) had reached 210+ characters on a single line prior to reformatting.

6.4.5 Error Handling Alignment with the R Source

A subsequent audit compared every `stop()`, `warning()`, and `match.arg()` call in `KernSmooth/R/all.R` against the corresponding `raise`/`warn` in `__init__.py`. Three discrepancies were identified and corrected.

`_resolve_choice` error message. R's `match.arg` produces a single message for both ambiguous partial matches and non-matching values, irrespective of which failure mode triggered:

```
'arg' should be one of "c1", "c2", ...
```

The original `_resolve_choice` generated two distinct custom-formatted messages, neither matching this format. Both failure paths were unified to produce the R-equivalent message, and the now-unused `param_name` parameter was removed from the function signature and all four call sites.

Invented `None` guard in `locpoly`, `sdiag`, `sstdiag`. An intermediate draft had added an explicit `ValueError("'bandwidth' must be specified")` guard at the top of each function body. No corresponding `stop()` call exists in the R source: when a required argument is absent, R's runtime raises "argument is missing, with no default" at the language level rather than via a user-defined message. The guard was removed from all three functions to eliminate error behavior with no R counterpart.

`locpoly` positivity check pattern. R guards its bandwidth positivity check with `!missing(bandwidth)`: `if (!missing(bandwidth) && any(bandwidth <= 0))`. The Python positivity check in `locpoly` had been written in unconditional form, diverging from this pattern. It was corrected to apply the `None`-sentinel idiom (`bandwidth is not None and ...`), as documented in the `missing.md` conversion guide.

All remaining `stop()` and `warning()` messages in the R source were confirmed as exactly reproduced in the Python module.

6.5 State of the Package After Phase 5

The rewritten `r2py_kernsmooth/r2py_kernsmooth/__init__.py` satisfies the following properties:

- No dead code or unused imports.

- No bare `np.ndarray`, `dict`, `tuple`, or `list` annotations in any public or private function prototype.
- All error and warning messages exactly match the explicit `stop()` and `warning()` calls in `KernSmooth/R/all.R`; no invented error messages are present.
- The `None`-sentinel idiom (`bandwidth is not None and ...`) consistently mirrors R's `!missing(bandwidth)` predicate across all functions that test bandwidth before use.
- No duplicated bandwidth-discretization or argument-validation logic.
- Correct 0-d NumPy array handling in the scalar-bandwidth code path.
- PEP 8 line-length compliance (≤ 88 characters) across all function signatures.
- No R-era changelog comments or R-specific lifecycle hook documentation.

The runtime target is CPython 3.14 (confirmed by the presence of `__pycache__/__init__.cpython-314.pyc`). The package uses Python 3.10+ syntax (`dict[str, ...]`, `float | None` union types, lowercase generics), which is consistent with the runtime version.

7 Phase 6: R Test Conversion and Regression Infrastructure

7.1 Motivation

After Phase 5 established a correct, well-structured Python package, the immediate priority was to build a formal test suite grounded in the R reference implementation. Two related objectives shaped this phase. The first was to port the two regression tests shipped with the R `KernSmooth` package — `KernSmooth/tests/bkfe.R` and `KernSmooth/tests/locpoly.R` — into Python equivalents that exercise the same function calls against the new Python module. The second was to equip those tests with assertion-based numerical comparisons using `rpy2`, so that any future divergence between the Python and R implementations is caught automatically at a quantified tolerance level rather than requiring manual inspection of printed output.

In general, testing the converted Python package against the original R implementation is much easier than testing a brand-new software project, because the R source provides a complete executable specification of intended behavior. Especially for Claude Code, that is a big advantage, since it can focus mainly on the input parameters of each function, with little consideration needed for the output (return values or exceptions), as we have the original R output as the reference.

The R test scripts in the original package source, while not comprehensive, provide a starting point for regression testing and a template for future test additions. The choice to use `rpy2` (which calls the R runtime at test time from Python) rather than hardcoded expected values reflects a deliberate design decision. Hardcoded values must be regenerated whenever the R reference changes, and differences in precision across machines may cause

spurious failures. A `rpy2`-based test calls the R function with the same inputs at each run and asserts element-wise agreement to a specified tolerance, making the tolerance bounds explicit and the reference values always current.

7.2 R Test Conversion

The `/convert-r-tests-to-python` skill (Appendix B, §B.4) was invoked to process the two R test files:

```
/convert-r-tests-to-python \
    KernSmooth/tests/ \
    KernSmooth/ \
    r2py_kernsmooth/ \
    r2py_kernsmooth/tests/
```

The skill scans the R test folder for `.R` scripts, then dispatches the `@convert-r-test-to-python` sub-agent (Appendix A, §A.4) once per file. Both R tests were processed sequentially, producing initial Python scripts that replicated the R function calls and printed results to standard output.

7.2.1 Conversion of `bkfe.R`

`KernSmooth/tests/bkfe.R` is a regression guard against a historical crash bug (fixed in KernSmooth 2.23-5) that was triggered when `gridsize` is an exact power of two. It calls `dpik(x, gridsize=256)` and `bkde(x, gridsize=256, range.x=range(x))` on `x <- 1:100` and verifies that both complete without error. The Python translation maps `x <- 1:100` to `np.arange(1, 101, dtype=np.float64)`, R's two-element `range(x)` result to the `range_x` keyword argument as a Python tuple, and wraps both calls in a `main()` / `if __name__ == "__main__"` guard following the pattern established in `r2py_kernsmooth/tests/test.py`.

7.2.2 Conversion of `locpoly.R`

`KernSmooth/tests/locpoly.R` conditionally loads the `carData` package, runs two `locpoly` calls on the `Prestige` dataset (`truncate=TRUE` and `truncate=FALSE`), and plots the results to demonstrate a penultimate-point artefact. No numeric assertions are present in the original R file. The Python translation faced the absence of `carData` as a Python package; the `Prestige` dataset was sourced from the Rdatasets public CSV mirror at `https://vincentarelbundock.github.io/Rdatasets/csv/carData/Prestige.csv` using only Python standard-library modules (`urllib.request`, `csv.DictReader`), preserving exact row-order correspondence with R. Plotting was replaced by dictionary output via `print()`.

An R reference script was additionally written at `r2py_kernsmooth/tests/R/locpoly.R` to enable future side-by-side numerical comparison with R runs.

7.3 The Two-Layer f2py Compatibility Bug

Running `test_locpoly.py` after package installation produced the following error:

```
_KernSmooth.error: (shape(ss, 0) == m) failed for 1st keyword m: locpol:m=0
```

Diagnosis revealed a two-layer incompatibility between the Phase 4 conversion output and the f2py interface generated by NumPy 2.4.3.

7.3.1 Layer 1: Strict 2D Shape Enforcement

The Phase 4 conversion had generated code that flattened 2D Fortran arrays (e.g., `ss(M,ippp)`, `Smat(ipp,ipp)`) to 1D arrays via `.flatten(order="F")` before passing them to Fortran routines. NumPy 2.4.3's f2py enforces strict 2D shape checks for parameters declared as 2D in the Fortran source, a behavioral change from earlier f2py versions, which accepted 1D contiguous memory as a valid 2D array by pointer reinterpretation. Applying this fix — converting all affected allocations to `order="F"` and removing the `.flatten()` calls — was necessary but not sufficient: the error persisted unchanged.

7.3.2 Layer 2: Absorbed Dimension Arguments

Inspecting the f2py-generated Python interface via `help(_KernSmooth.locpol)` exposed the root cause:

```
locpol(xcnts, ycnts, idrv, delta, hdisc, lvec, indic, midpts, iq, fkap,
       ss, tt, smat, tvec, ipvt, cvest, [m, ipp, ippp])
```

NumPy 2.4.3's f2py had *absorbed* the three Fortran dimension scalars `M`, `ipp`, and `ippp` out of the required positional argument list, converting them into optional parameters with defaults inferred from array shapes (`m : input int, optional, Default: shape(ss, 0)`). The same absorption occurred for `n` and `qq` in `blkest`, and `n`, `qq`, and `Nmax` in `cp`.

Because the Python code still injected these integers at their original Fortran-order positions, every subsequent argument was shifted. For `locpol`: positional argument 9 received `np.int32(M)` instead of the `iq` kernel-indicator it expected; positional argument 11 received the `fkap` weight array instead of the 2D `ss` matrix. Since f2py could not extract a valid first dimension from a 1D float array, `m` defaulted to zero, triggering the shape-check failure. The identical misalignment caused silently wrong results in `sdiag`, `sstdiag`, `blkest`, and `cpblock`.

7.3.3 Fixes Applied

Seven allocation sites across `blkest`, `cpblock`, `locpoly`, `sdiag`, and `sstdiag` were updated to add `order="F"` to their `np.zeros()` calls, with the corresponding `.flatten(order="F")` calls at the Fortran call sites removed. Five Fortran call sites had their absorbed dimension integers removed from the positional argument lists:

- `_KernSmooth.locpol`: removed `np.int32(M)`, `np.int32(pp)`, `np.int32(ppp)`; call reduced from 19 to 16 positional arguments.
- `_KernSmooth.sdiag`: same pattern; call reduced from 17 to 14 positional arguments.
- `_KernSmooth.sstdg`: same pattern; call reduced from 19 to 16 positional arguments.
- `_KernSmooth.blkest`: removed `np.int32(n)` and `np.int32(qq)`.
- `_KernSmooth.cp`: removed `np.int32(n)`, `np.int32(qq)`, and `np.int32(Nmax)`.

The package was reinstalled after each layer of fixes:

```
pip install . --no-build-isolation --force-reinstall
```

After both layers, all five affected Fortran call sites transitioned from crashing or silently incorrect to correct.

7.4 Upgrade to rpy2-Based Assertion Testing

With the f2py compatibility issues resolved, the `/convert-r-tests-to-python` skill was re-invoked with the same arguments. This time the sub-agents produced fully assertion-based `pytest` files rather than print-and-inspect scripts.

The resulting test files use `rpy2` throughout: R packages are loaded once at module level with `importr("KernSmooth")` (and `importr("carData")` for the locpoly test); input data is passed as `ro.FloatVector` or `ro.IntVector`; results are extracted with `.rx2()` and converted to NumPy arrays. Grid coordinates are asserted at `rtol=1e-10`; density and regression estimates at `rtol=1e-6` via `numpy.testing.assert_allclose`. For `test_locpoly.py`, the `Prestige` dataset is extracted directly from R via `ro.r("Prestige$income")` to guarantee exact row correspondence, eliminating the CSV download used in the initial print-based version.

Both sub-agents produced output files without the `test_` prefix required by `pytest` for automatic test-discovery; the files were renamed accordingly (`bkfe.py` → `test_bkfe.py`, `locpoly.py` → `test_locpoly.py`).

The test suite was executed:

```
python -m pytest \
    r2py_kernsmooth/tests/test_bkfe.py \
    r2py_kernsmooth/tests/test_locpoly.py -v
```

All four tests passed under Python 3.14.4 / pytest 9.0.3 in 2.94 seconds:

```
r2py_kernsmooth/tests/test_bkfe.py::test_dpik_gridsize_power_of_2   PASSED
r2py_kernsmooth/tests/test_bkfe.py::test_bkde_gridsize_power_of_2   PASSED
r2py_kernsmooth/tests/test_locpoly.py::test_locpoly_truncate_true    PASSED
r2py_kernsmooth/tests/test_locpoly.py::test_locpoly_truncate_false   PASSED
```

Numerical agreement with R was confirmed: `dpik(1:100, gridsize=256)` returned `11.18973073581589` in Python (R: `11.18973`, truncated to 7 significant figures by R's default printing), and all 401 `locpoly` grid values matched R's output to 7 significant figures. The `rpy2`-based testing pattern established here — loading R packages at module level, using `np.testing.assert_allclose` with explicit tolerances, extracting dataset columns directly from R to guarantee row correspondence — became the project's standard approach for R-vs-Python numerical regression tests.

8 Phase 7: Comprehensive Test Suite Construction

8.1 Motivation

At the conclusion of Phase 6, the test suite comprised four assertion-based tests covering only `dpik`, `bkde`, and `locpoly`. The remaining four public functions — `bkde2D`, `bkfe`, `dpih`, and `dpill` — had received no automated numerical testing since the analytical output validation in Phase 4. No negative tests (exercising error-raising code paths) or edge-case tests (exercising boundary inputs such as NaN, Inf, $n = 1$, degenerate grid ranges, and very small or very large bandwidths) existed for any function.

A systematic, comprehensive test suite was needed for two reasons. The practical reason is that functions like `dpill`, which orchestrate five internal helpers and three Fortran routines through a multi-stage bandwidth selection procedure, are difficult to validate analytically; only a battery of numerical tests against the R reference can surface the class of silent errors that the f2py compatibility bugs of Phase 6 demonstrated are possible. The methodological reason is that the tests serve as a living specification: each test documents a precise input/output contract between the Python package and the R reference, against which future modifications to either the wrapper code or the Fortran interface can be validated.

8.2 Comprehensive Test Generation

The `/generate-python-file-tests` skill (Appendix B, §B.5) was invoked:

```
/generate-python-file-tests r2py_kernsmooth/r2py_kernsmooth/__init__.py
```

8.2.1 Public Interface Filtering

The skill parsed `__init__.py` to enumerate all function definitions, then fetched the KernSmooth CRAN reference manual to identify the seven user-facing exports: `bkde`, `bkde2D`, `bkfe`, `dpih`, `dpik`, `dpill`, `locpoly`. The remaining seven `__all__` members (`blkest`, `cpblock`, `linbin`, `linbin2D`, `rlbin`, `sdiag`, `sstdiag`) are internal utilities with no R documentation and were excluded from the generation queue. The existing `test_`

`bkfe.py` and `test_locpoly.py` from Phase 6 were reviewed and supplemented with new files rather than replaced, preserving the established `rpy2` patterns.

The `@generate-python-function-tests` sub-agent (Appendix A, §A.5) was invoked sequentially once per public function. In each sug-agent, the agent parsed the function signature and docstring to identify all parameters and their types, then fetched the corresponding R documentation page (KernSmooth) to extract the parameter descriptions, usage examples, and any explicit error conditions. The agent then generated a battery of positive, negative, and edge tests covering all documented behaviors and error paths for that function. The generated tests were written to new files named `test_`*`fn`*`_positive.py`, `test_`*`fn`*`_negative.py`, and `test_`*`fn`*`_edge.py` in the `r2py_kernsmooth/tests/` folder, following the `pytest` convention of `test_` prefix for automatic discovery. Each test uses `rpy2` to call the R reference implementation with the same inputs and assert numerical agreement within specified tolerances.

In order to cover the possible use cases, exceptions, and edge cases for each function as much as possible, the prompt for agent `@generate-python-function-tests` was designed to elicit a comprehensive set of tests. For positive tests, the agent was prompted to first extract all parameters and then find all possible combinations of them based on the R documentation and function body. For negative tests, the agent was prompted to identify all explicit error conditions processed in the function body and related functions (e.g., invalid parameter values) and generate tests that trigger those errors. For boundary and edge tests, the agent was prompted to consider boundary inputs such as empty arrays, single-element arrays, `NaN`/`Inf` values, degenerate grid ranges, and very small or very large bandwidths. The agent was also prompted to ensure that all generated tests are self-contained and can be run independently, with clear assertions and descriptive test names. The resulting test suite for each function is intended to provide comprehensive coverage of the function's behavior against the R reference implementation, serving both as a regression guard and as documentation of the expected input/output contracts.

Due to different argument evaluation strategies between R and Python, some negative and edge tests that would trigger a specific error in R (e.g., missing required arguments) do not trigger the same (or even a similar) error in Python because of the presence of default values or the way Python handles missing arguments, and vice versa. In such cases, the agent was prompted to test them via a loose principle. For example, if both R and Python throw an error, the test should be considered passed. If R throws an error but Python does not, the test should be considered failed, and the agent will automatically investigate the discrepancy and determine whether the Python implementation needs to be updated to match R's behavior or if the test case is not applicable due to language differences. There are some cases where we accept the tests where one language throws an error, while the other does not, but returns arrays with `NaN` or `Inf` values. Under those circumstances, the tests are written in a way determined by the agent to be informative about the language differences and the expected behavior in each language.

8.2.2 Per-Function Test Content

Each invocation produced three test files: positive cases (`test_fn_positive.py`), negative cases (`test_fn_negative.py`), and edge cases (`test_fn_edge.py`). All tests use `rpy2` throughout to obtain live R reference values at test time.

`bkde` Positive tests cover all 5 kernel types, both `canonical` modes, auto-computed and explicit bandwidth, varied `gridsize` and `range_x`, `truncate` mode, integer input coercion, partial kernel-name abbreviations (e.g. `"n"` matching `"normal"`), and density integration to ≈ 1.0. Negative tests cover negative and zero bandwidth, unrecognized kernel name, and the ambiguous prefix `"b"` (matches both `"box"` and `"biweight"`). Edge tests cover $n = 1$, $n = 2$, constant data, very small and very large bandwidth, degenerate `range_x`, output non-negativity, and τ-boundary tests for all 5 kernels.

`bkde2D` Positive tests cover bivariate normal input, various gridsize combinations, asymmetric and scalar bandwidths, explicit `range_x`, `truncate` mode (confirmed to have no effect at large bandwidths, matching R), integer coercion, and density integration. Negative tests cover negative and zero bandwidth in each component, 1-column input, NaN in input, and inverted `range_x`. Edge tests cover $n = 1$ through $n = 3$, constant data, very large bandwidth, and minimum gridsize.

`bkfe` Positive tests cover `drv` $\in \{0, 2, 4, 6, 8, 10\}$, varied sample sizes and distributions, `gridsize` variants, and `binned` mode. Edge tests include the power-of-two `gridsize` regression guard (`gridsize = 256` and `gridsize = 512`) carried forward from Phase 6.

`dpih` Positive tests cover all three `scalest` options (`minim`, `stdev`, `iqr`) and all six `level` values, including a cross-product of `(level, scalest)` combinations, and an analytical formula check for `level = 0`. Edge tests verify scale-invariance ($\texttt{dpih}(cx) \approx c \cdot \texttt{dpih}(x)$) and `minim` monotonicity against R.

`dpik` Positive tests cover all 5 kernels $\times$ all 6 levels (30 combinations), all 3 `scalest` $\times$ 6 levels (18 combinations), `canonical` mode, partial-name abbreviations, and a kernel bandwidth ordering invariant for the non-canonical case. Edge tests verify scale invariance and location shift invariance against live R reference values, including small $c = 0.001$.

`dpill` Positive and edge tests cover standard regression scenarios, various `blockmax`, `divisor`, `trim`, `proptrun`, and `gridsize` settings, scale and shift invariance, and multiple regression function types. Negative tests all passed immediately. However, 60 of the 83 positive and edge tests failed on first run due to a pre-existing bug discovered during this phase (see Section 8.3.1).

`locpoly` Positive tests cover regression mode with `drv` $\in \{0, 1, 2, 3\}$, scalar and vector bandwidth, `truncate` and `binned` modes, density mode (`y=None`), and the Prestige dataset canonical example from the KernSmooth reference manual. Edge tests verify density non-negativity, density integration to ≈ 1.0 for $n = 1000$, and degenerate grid parameters.

8.2.3 Test Count Summary

Function	Positive	Negative	Edge	Total	Initial pass
`bkde`	28	9	20	57	57
`bkde2D`	28	12	23	63	63
`bkfe`	24	9	16	49	49
`dpih`	32	15	24	71	71
`dpik`	52	23	44	119	119
`dpill`	38	17	28	83	21
`locpoly`	38	11	23	72	72
Total	**240**	**96**	**178**	**514**	**452**

8.3 Systematic Bug Resolution

The 62 initial failures were all in the `dpill` tests. The `/fix-bugs-found-in-tests` skill (Appendix B, §B.5) was invoked to resolve them:

```
/fix-bugs-found-in-tests r2py_kernsmooth/tests/ r2py_kernsmooth/ KernSmooth/
```

The skill runs `pytest` on the test folder, isolates the first failing test, dispatches the `@fix-bug-found-in-test` sub-agent (Appendix A, §A.5) with the full traceback and relevant source context, reinstalls the package if the fix modifies library code, and repeats until all tests pass. Three distinct bugs were identified and fixed across three sub-agent invocations.

8.3.1 Bug 1: `blkest` f2py Scalar Output Binding

The first failing test crashed with:

```
ValueError: cannot convert float NaN to integer
```

at `_discretize_bandwidth` inside `locpoly`, called from `dpill`. Inspecting the f2py wrapper via `python -c "import r2py_kernsmooth._KernSmooth as k; print(k.blkest.__doc__)"` revealed that the three scalar output arguments of `blkest` — `sigsqe`, `th22e`, and `th24e` — were declared as `input float` (passed by value) rather than as writeable output arguments. The Fortran subroutine computes these values by direct assignment (`sigsqe = ...`), but f2py passed them as scalar copies; the Fortran writes never propagated back to the Python caller. Consequently, `blkest` always returned zeros for all three outputs, causing `dpill` to evaluate `gamseh = sigsqQ / (abs(0) * n)`, a zero denominator, propagating NaN as the bandwidth into `locpoly`.

This bug had been silently present since at least Phase 6's argument-order corrections. The prior session fixed the *positions* of arguments passed to `blkest` but did not address

the *direction* of data flow for the scalar outputs. The `dpill` function had never been tested with inputs large enough to expose the NaN cascade.

The fix comprised three coordinated changes:

1. A full f2py signature file `r2py_kernsmooth/src/_KernSmooth.pyf` was generated from all Fortran source files, with `intent(out)` annotations added to `sigsqe`, `th22e`, and `th24e` in the `blkest` block.
2. `r2py_kernsmooth/meson.build` was updated to pass `_KernSmooth.pyf` as the sole f2py input rather than the bare `.f` source files, so that f2py honours the explicit intent declarations rather than inferring them from the Fortran source.
3. `blkest()` in `r2py_kernsmooth/r2py_kernsmooth/__init__.py` was updated to unpack the three scalar outputs as return values: `sigsqe, th22e, th24e = _KernSmooth.blkest(...)`, removing the three pre-allocated `np.zeros(1)` arrays that had formerly been passed as unused input placeholders.

8.3.2 Bug 2: `dpill` Range Computed Before Trimming

After the `blkest` fix, 49 `dpill` tests still failed with Python returning NaN while R returned valid bandwidths. A step-by-step trace of `dpill` for a seed-42, $n = 200$ test case was carried out, logging all intermediate values. The trace revealed that Python and R diverged at the computation of `gamseh`: Python produced `0.19950` while R produced `0.19197`. This difference was traced back to the range $b - a$ used in the formula: Python used 5.046 (the range of the full sorted input before trimming), while R used 3.856 (the range after trimming 2% of points from each tail). With the smaller (post-trim) range, R computed a larger bandwidth `lamseh = 0.13285`, which was wide enough that `sdiag` encountered no near-singular design matrices at empty grid bins. With Python's smaller `lamseh = 0.10453`, `sdiag` produced 29 `Inf` values at zero-count bins, and `np.sum(Sdg * xcounts)` yielded NaN under IEEE 754 ($\infty \times 0 =$ NaN), propagating through the remainder of `dpill`.

The root cause is a fundamental asymmetry between R's lazy default-argument evaluation and Python's eager evaluation. In R, `dpill`'s signature declares `range.x = range(x)`: this expression is not evaluated at call time but at first use inside the function body. Since R's `dpill` reassigns `x` to a sorted and trimmed version before `range.x` is first accessed, the computed range reflects the trimmed data. Python's `range_x = None` sentinel default cannot replicate this behavior automatically; the assignment must appear in the function body *after* the trim step. The Phase 4 translation had placed it before. The fix reordered the code block:

```
sort_idx = np.argsort(x)
x = x[sort_idx]; y = y[sort_idx]
indlow = int(np.floor(trim * len(x)))
indupp = len(x) - int(np.floor(trim * len(x)))
x = x[indlow:indupp]; y = y[indlow:indupp]
```

```python
if range_x is None:
    range_x = (x[0], x[-1])  # matches R's lazy default eval (post-trim)
```

After this fix, `dpill` for the seed-42, $n = 200$ case returned `0.1379083775114942`, matching R exactly. This R lazy-evaluation pattern is potentially relevant to any other function in `__init__.py` that accepts `range_x=None` and may internally reorder or subset `x` before using the range; a post-fix audit confirmed that no other function trims its input data, so no analogous bug exists elsewhere.

8.3.3 Bug 3: `NaN == NaN` Test Assertion

After the `dpill` range fix, two tests remained failing, both verifying the invariant that `dpill(unsorted_input)` and `dpill(sorted_input)` return the same value. For one specific degenerate seed ($n = 100$, seed 13), both Python and R legitimately return NaN due to an ill-conditioned intermediate computation; the test was designed to assert equality between the two calls. Under IEEE 754, `NaN != NaN` by definition, and `pytest.approx` does not handle this case. The sub-agent corrected the assertion in both affected test files:

```python
both_nan = np.isnan(h_unsorted) and np.isnan(h_sorted)
assert both_nan or h_unsorted == pytest.approx(h_sorted, rel=1e-10)
```

8.4 Documented Behavioral Divergences

Several cases were identified in which the Python package's behavior differs from R's in a way that cannot be straightforwardly corrected. Each is documented by a `UserWarning` emitted by the corresponding negative test rather than treated as a test failure:

`bkde` / degenerate `range_x` ($a = b$) R returns a grid of NaN values; Python raises `ZeroDivisionError` from the `(b-a) / (gridsize-1)` step.

`bkfe` / `Inf` or `-Inf` in input R raises an error inside `seq()` when constructing the grid from an infinite range; Python propagates NaN via IEEE 754 arithmetic.

`dpih` / `level=-1` R returns `numeric(0)` silently (a zero-length vector, a valid R type); Python raises `UnboundLocalError` because no branch assigns the result variable for a negative level.

`dpill` / `divisor=0` R returns a valid result via integer division yielding Inf at an intermediate step; Python raises `ZeroDivisionError`.

`bkde2D` / `Inf` in input R raises inside `seq.default()`; Python returns a partially non-finite result.

Additionally, several negative tests where both Python and R raise but with differing message phrasing (Python omits the R function-signature prefix that R prepends to error messages) emit `UserWarning` rather than failing.

8.5 State of the Package After Phase 7

After all three bugs were fixed and the package rebuilt, the complete test suite was run:

```
python -m pytest r2py_kernsmooth/tests/ -q
```

Result: **518 passed, 0 skipped, 0 failed** in 6.40 seconds under Python 3.14.4 / pytest 9.0.3. The suite covers all 7 public functions across 21 test files, with 240 positive tests, 96 negative tests, and 178 edge-case tests, all asserting live numerical agreement or similar exception handling against R `KernSmooth` 2.23 via `rpy2`.

References

[1] M. P. Wand and M. C. Jones (1995). *Kernel Smoothing*. Chapman and Hall, London.

[2] S. J. Sheather and M. C. Jones (1991). A reliable data-based bandwidth selection method for kernel density estimation. *Journal of the Royal Statistical Society, Series B*, **53**(3), 683–690.

[3] D. Ruppert, S. J. Sheather, and M. P. Wand (1995). An effective bandwidth selector for local least squares regression. *Journal of the American Statistical Association*, **90**(432), 1257–1270.

[4] M. P. Wand (1994). Fast computation of multivariate kernel estimators. *Journal of Computational and Graphical Statistics*, **3**(4), 433–445.

Appendices

A Agent Specifications

This appendix reproduces the complete specifications of the sub-agents deployed in this project. Each file is stored under `.claude/agents/` in the project repository. The YAML front matter names the agent and provides a one-line description; the body specifies the execution steps and output schema in structured natural language. These files are the normative definitions of agent behavior: what the agent receives as input, how it processes it, and what it must output.

A.1 Agents Used in Phase 1

`analyze-r-file-dependencies`

```
---
name: analyze-r-file-dependencies
description: Analyzes an R script to extract structural dependency information function by function,
     categorizing calls into language, internal, and external dependencies.
---

# Analyze an R File for Dependencies

## Description

When provided with an R file (‘.R‘), your task is to parse the file, identify every user-defined function
     , and trace all function calls made within their bodies. You must categorize these dependencies into
      strict groups and output the result as a structured JSON object.

## Execution Steps

### Step 1: Extract Function Declarations

Scan the target R file and identify every user-defined function declared within it (e.g., ‘my_func <-
     function(...) { ... }‘). Create a list of these function names.

### Step 2: Analyze Function Bodies and Trace Calls

For every function identified in Step 1, thoroughly analyze its execution body. Extract every function
     call made within that body and categorize it into one of the following three mutually exclusive
     lists:

* **Language Dependencies:**
    * *Definition:* Functions provided by Base R (e.g., ‘c()‘, ‘lapply()‘, ‘print()‘) or functions from
     installed CRAN/Bioconductor packages (e.g., ‘mutate()‘, ‘ggplot()‘).
    * *Heuristic:* Include any function call that includes a package namespace prefix (e.g., ‘dplyr::
     select‘) or any standard function that is not defined locally.
* **Internal Dependencies:**
    * *Definition:* Functions that are defined within the exact same R file, or functions that are
     explicitly defined in other ‘.R‘ files within the same project directory.
* **External Dependencies:**
    * *Definition:* Foreign language interfaces (typically C, C++, or Fortran) invoked from within the R
     code.
    * *Heuristic:* Look specifically for calls to ‘.Call()‘, ‘.C()‘, ‘.Fortran()‘, ‘.External()‘, or ‘
     useDynLib()‘. Extract the name of the compiled routine being invoked (usually the first string
     argument passed to these interfaces).
```

````
### Step 3: Generate JSON Output

Construct a JSON object where each key is the name of a function identified in Step 1. The value for each
     key must be an object containing exactly three arrays of strings: ‘language_dependencies‘, ‘
     internal_dependencies‘, and ‘external_dependencies‘. Deduplicate all function names within these
     arrays.

## Output Format Schema

You must output valid JSON strictly adhering to this structure. Do not include markdown code blocks if
     the system expects raw JSON.

‘‘‘json
{
  "function_name_1": {
    "language_dependencies": ["c", "lapply", "dplyr::filter"],
    "internal_dependencies": ["helper_function_within_file"],
    "external_dependencies": [".Call(\"c_routine_name\")"]
  },
  "function_name_2": {
    "language_dependencies": ["print"],
    "internal_dependencies": ["function_name_1"],
    "external_dependencies": []
  }
}
‘‘‘
````

A.2 Agents Used in Phase 3

extract-r-file-language-dependencies

````
---
name: extract-r-file-language-dependencies
description: Extracts detailed location and usage data for language dependencies in an R file, utilizing
     pre-calculated structural analysis JSON.
---

# Extract Language Dependencies from an R File

## Description

When provided with an R source file (‘.R‘) and its corresponding structural analysis JSON (generated by
     the ‘@analyze-r-file-dependencies‘ agent), your task is to locate every instance where those
     language dependencies are invoked. You must extract the exact line numbers and textual calls for
     each dependency, and output the results as a well-formatted, sorted CSV.

## Execution Steps

### Step 1: Ingest Inputs

Parse the provided structural analysis JSON to identify the ‘language_dependencies‘ associated with each
     user-defined function (the JSON keys).

### Step 2: Locate and Extract Invocations

Scan the raw text of the ‘.R‘ file. For every function listed in the JSON, find its declaration in the R
     file and scan its body for the specific ‘language_dependencies‘ listed in its JSON array.

For every individual call to these dependencies, extract the following data points:
````

```
* **‘language_dependency‘**: The exact string name of the dependency as listed in the JSON (e.g., ‘c‘, ‘
    dplyr::filter‘).
* **‘function_name‘**: The name of the user-defined function (the JSON key) where this dependency is
    currently being executed.
* **‘line_number‘**: The exact line number in the ‘.R‘ file where the invocation of the dependency begins
    .
* **‘call_body‘**: The exact code snippet of the invocation, including its arguments (e.g., ‘dplyr::
    filter(df, column == "value")‘). If the call spans multiple lines, capture the entire complete
    statement.

### Step 3: Format and Sort Output

Compile the extracted data points into a CSV format with a header row.

## Output Format Schema

You must output valid CSV data strictly adhering to this structure. Include the header row. Enclose ‘
    call_body‘ strings in double quotes to prevent formatting errors from commas or line breaks within
    the code.

‘‘‘csv
language_dependency,function_name,line_number,call_body
c,my_func_1,14,"c(1, 2, 3)"
c,my_func_2,42,"c(\"A\", \"B\")"
dplyr::filter,my_func_1,15,"dplyr::filter(data, val > 0)"
lapply,my_func_1,22,"lapply(list, function(x) {
  x + 1
})"
print,my_func_2,45,"print(\"Processing complete.\")"
‘‘‘
```

generate-language-dependency-conversion-guide

```
---
name: generate-language-dependency-conversion-guide
description: Generates a conversion guide for translating a specific language dependency from R to Python
    .
---

# Generate Language Dependency Conversion Guide

## Description

When provided with a target ‘base_folder‘ and a CSV text input matching the following schema:
‘‘‘csv
language_dependency,file_path,function_name,line_number,call_body
exp,all.R,locpoly,663,"exp(seq(log(hlow), log(hupp), length.out = Q))"
exp,all.R,sdiag,768,"exp(seq(log(hlow), log(hupp), length.out = Q))"
exp,all.R,sstdiag,845,"exp(seq(log(hlow), log(hupp), length.out = Q))"
‘‘‘
Your task is to generate a comprehensive conversion guide for the specified ‘language_dependency‘ (e.g.,
    ‘exp‘). The guide must include:
1. A clear explanation of the ‘language_dependency‘’s core functionality in R.
2. A detailed analysis of its usage within the provided CSV data, including the surrounding context
    retrieved directly from the source R files.
3. A conversion guide to equivalent Python functions/methods, providing general Python code snippets for
    all scenarios in the CSV.

## Execution Steps

### Step 1: Ingest Inputs and Gather Context
```

```
- Iterate through each row in the CSV. Using the provided `base_folder`, navigate to the exact `file_path` and `line_number`.
- Read the target line and the surrounding code blocks. You must read enough lines to fully capture multi-line function calls, identify the data types of arguments being passed (e.g., scalars vs. vectors), and understand how the result interacts with adjacent logic.
- If the R dependency's behavior is complex or ambiguous, search `https://www.rdocumentation.org/` to confirm its official functionality, default arguments, and edge cases.
- If there is ambiguity about the valid types of the functions' parameters or return values, you can also look them up in the document `https://cran.r-project.org/web/packages/KernSmooth/refman/KernSmooth.html`.

### Step 2: Determine Python Equivalents

- **Prioritize Vectorization:** Because R inherently vectorizes operations, you must default to evaluating `numpy`, `scipy`, or `pandas` as the primary Python equivalents.
- Choose the library that best matches R's vectorized nature. For example, if translating R's `exp`, strongly prefer `numpy.exp()` over the standard library `math.exp()` unless the context explicitly proves only a scalar is being processed.
- Carefully map R arguments to Python kwargs (e.g., translating R's `seq(..., length.out=Q)` to Python's `np.linspace()`).
- You are free to search online documentation for the chosen Python libraries to ensure you are using the most efficient and idiomatic functions.

## Output Format Schema

You must output the final conversion guide strictly in well-structured Markdown format. Use the following outline to ensure consistency and readability:

### 1. Overview of `{language_dependency}` in R

*Provide a concise explanation of what the dependency does, its typical inputs, and expected outputs.*

### 2. Contextual Usage Analysis

*Summarize how the dependency is applied across the provided dataset. Mention the data types involved and any recurring patterns found in the source files.*

### 3. Python Conversion Strategy

*State the chosen Python library (e.g., `numpy`) and explain why it is the most robust equivalent for these specific usage contexts.*

### 4. Step-by-Step Conversion Examples

*For every functional-distinct usage found in the CSV, provide a subsection containing:*
- **Locations:** Files and Function names.
- **Original R Context:** The types of input parameters and return values, and a generalized code snippet for demonstration.
- **Python Equivalent:** A complete, executable Python code snippet demonstrating the converted logic.
- **Explanation:** A brief breakdown of the syntax translation, addressing any nuances like zero-based indexing, argument mapping, or library imports.
```

A.3 Agents Used in Phase 4

convert-r-function-to-python

```
---
name: convert-r-function-to-python
description: Converts an R function to its Python equivalent using a provided JSON dependency map and conversion guides.
---
```

````
# Convert an R Function to Python

## Description

Rewrite a provided R function into Python. You will receive the R function code, a JSON file detailing
     its dependencies, a folder of language dependency conversion guides, and a target output folder. You
      must maintain the exact logic and functionality of the original code.

## Execution Steps

### Step 1: Infer Types and Define Prototype

Identify all parameters and return types by referencing the R function's CRAN documentation (e.g., the `
     KernSmooth` reference manual `https://cran.r-project.org/web/packages/KernSmooth/refman/KernSmooth.
     html`).
* Create a Python function prototype using complete type annotations.
* **Never omit any types or subtypes inside type annotations.** For example, `dict[str, int]` or `np.
     ndarray[Any, np.dtype[np.float64]]` cannot be simplified as `dict` or `np.ndarray` if the subtypes
     are known. Also, if there are multiple possible types for a parameter, include all of them in the
     annotation (e.g., `str | None`).
* *Example:* `def example_function(param1: int, param2: np.ndarray[Any, np.dtype[np.float64]]) -> np.
     ndarray[Any, np.dtype[np.float64]]:`

### Step 2: Resolve Dependencies

Analyze the provided JSON dependency map. Handle each dependency category as follows:
* **Language Dependencies:** Find the corresponding `{language_dependency}.md` files (e.g., `min.md`, `
     seq.md`) in the conversion guides folder for Python translation strategies and examples. You don't
     have to follow them strictly, but they should guide your translation of R constructs to Python.
* **Internal Dependencies:** Locate and review the previously converted definitions in the output folder
     (`{function_name}.json`). Strictly follow their Python logic to ensure correct integration.
* **External Dependencies:** Map external calls directly to Python's `_KernSmooth` module. *Example: Map
     `.Fortran(F_sdiag)` directly to `_KernSmooth.sdiag()` and can be imported via `from . import
     _KernSmooth`. The `F_` prefix is a convention for Fortran calls in R, and the corresponding Python
     function has already been named without this prefix. No functions have return values, and the
     parameters are passed by reference. If you know little about R's Fortran interface, consult the
     documentation `https://masuday.github.io/fortran_tutorial/r.html`.*

### Step 3: Rewrite Function Body

Translate the R function body into Python syntax.
* Strictly preserve the original logic and functional intent.
* Apply the patterns learned from the language conversion guides.
* Handle R-to-Python structural differences (e.g., 1-based vs. 0-based indexing, data structures, control
      flow) appropriately.

## Output Format

Output strictly valid JSON to `{function_name}.json` in the output folder. **Do not wrap the output in
     markdown code blocks.** The function body may contain multiple lines, and you should include every
     line as a separate string in the array. Follow the following structure strictly (even if there are
     no imports, you must include an empty array for imports):
```json
{
  "imports": [
    "import numpy as np",
    "from _KernSmooth import sdiag"
  ],
  "function_prototype": "def converted_function(param1: int, param2: np.ndarray[Any, np.dtype[np.float64
     ]) -> np.ndarray[Any, np.dtype[np.float64]:",
  "function_body": [
    "    # Function body goes here",
    "    pass"
  ]
````
````

```
}
```
```

## A.4 Agents Used in Phase 6

### convert-r-test-to-python

```
---
name: convert-r-test-to-python
description: Converts a specified R test file to its Python equivalent using provided library contexts.

Convert an R Test File to Python

Description

Translate a provided R test script into an equivalent Python test script. To ensure accurate translation,
      you must rely on the provided paths to the target R test file, the source R library folder, and the
      corresponding Python library folder. You must maintain the exact test coverage, logic, and
     assertions of the original R code while adapting to standard Python testing conventions (e.g., using
      `pytest`) if necessary.

Execution Steps

Step 1: Analyze Dependencies and Context

- Read the provided R library directory to understand the function signatures, data structures, and logic
      being tested in the R script.
- Read the provided Python library directory to identify the equivalent Python functions, classes, and
     modules that the new test file will need to import.

Step 2: Resolve Datasets and Fixtures

- Scan the R test file for any required datasets or external files.
- Look for these files locally first. If they require downloading from an external source, prompt the
     user for the correct URL or source if it is not explicitly defined in the R script.
- Ensure the translated Python code uses appropriate libraries (e.g., `pandas` or built-in `csv`) to load
      these datasets correctly.

Step 3: Translate the Test Logic

- Translate the R test file into Python syntax. Do not change any logic in the file; the workflow should
     remain exactly identical.
- If the R test file is simply a collection of calls to the R library functions and print statements, you
      must use `rpy2` to interface with the R environment, call the functions in the R library, and
     replicate the same logic with the Python library, and then use `assert` statements from `pytest` or
     `unittest` to validate the alignment of results.
- If the R test file contains plots or visualizations that are not part of the R library, you should
     ignore them and check the alignment of numerical results instead.
- Map R testing framework functions (like those from `testthat`) to their exact Python equivalents (e.g.,
      standard `assert` statements for `pytest` or `unittest.TestCase` methods), if applicable.
- Pay strict attention to handling 1-based indexing in R versus 0-based indexing in Python, as well as
     differences in how NA/NaN values are treated.

Step 4: Save the Python Test File

- Check if the user specified an output folder in their prompt.
- If an output folder is specified, save the translated Python test file there.
- If no output folder is specified, create a `tests/` directory inside the provided Python library folder
      (if it doesn't already exist) and save the file there. The file should be named identically to the
     original R test file but with a `.py` extension (e.g., `example.R` becomes `example.py`, but if
```
```

```
    using ‘pytest‘, the file would be named ‘test_example.py‘).
```

A.5 Agents Used in Phase 7

generate-python-function-tests

```
---
name: generate-python-function-tests
description: Generates Python function tests for a given function in a given Python file, benchmarking
    against its original R implementation.
---

# Generate Python Function Tests

## Description

When provided with a target Python function name, a Python file path, and the original R function/package
     reference, your task is to generate comprehensive unit tests using ‘pytest‘. The tests must cover
    positive cases, negative cases, and boundary/edge cases to ensure the Python function’s behavior
    strictly mirrors the original R implementation.

## Expected Inputs
- ‘TARGET_FUNCTION‘: The name of the Python function to test.
- ‘FILE_PATH‘: The path to the Python file containing the function.
- ‘R_FUNCTION‘ / ‘R_PACKAGE‘: The original R function and package to benchmark against.

## Execution Steps

### Step 1: Review the Target Function and Its Context

- Navigate to ‘FILE_PATH‘ and locate ‘TARGET_FUNCTION‘.
- Read the function’s implementation, analyzing its parameters, return values, and any internal logic or
    dependencies on other modules.
- Analyze the docstring to understand the intended behavior, expected inputs, and outputs. If there is no
     docstring, infer the function’s purpose from the code.
- Identify specific edge cases or mathematical constraints based on the logic (e.g., division by zero,
    null checks).

### Step 2: Review the Documentation of the Original Function

- Navigate to the documentation of the original R function (e.g., searching CRAN documentation or using R
    ’s internal ‘help()‘ documentation).
- Extract key information about the function’s expected behavior, input parameters, default arguments,
    return values, and any relevant side effects or constraints.
- *Example:* For a function from the ‘KernSmooth‘ package, review its documentation at ‘https://cran.r-
    project.org/web/packages/KernSmooth/refman/KernSmooth.html‘. Use this to understand the data types
    it accepts and exactly what output structures it returns.

### Step 3: Generate Positive Tests

Generate at least 8 positive test cases covering all valid input scenarios (typical cases and varied data
     types) using ‘pytest‘. You have to cover all the use cases and input types that the original R
    function supports.
- **All parameters and their combinations must be tested.** To achieve this, you have to first extract
    all the parameters of the Python function and their default values (if any) from the docstring or
    code. Then, you have to generate test cases that cover all possible combinations of these parameters.

- **Naming Convention:** Save the tests in the Python project’s ‘tests/‘ directory as ‘test_<
    TARGET_FUNCTION>_positive.py‘. (Create the ‘tests/‘ directory if it does not exist). If the file
    already exists, append the new tests to it without removing any existing tests. It is fine if you
    find existing tests are enough to cover all scenarios, in that case, you can skip the generation of
```

```
    additional tests.
- **Setup:** Define descriptive test functions (e.g., `test_<TARGET_FUNCTION>_with_standard_array`). Set
    up necessary input data mimicking valid R data structures.
- **R Execution:** Use `rpy2` to execute the original R function with the test input data and retrieve
    the expected output.
- **Python Execution & Assertion:** Call the target Python function with the identical input data.
- **Comparison:** Assert that the outputs match. *Crucial:* When comparing floats or numerical arrays
    between R and Python, use `numpy.testing.assert_allclose` or `math.isclose` to account for minor
    floating-point precision differences between the two languages.

### Step 4: Generate Negative Tests

Generate negative test cases covering invalid input scenarios to ensure the Python function handles
    errors identically to the R function. Scan the Python function and all related functions for all
    error-handling logic (e.g., `raise ValueError`); all branches must be tested.
- **Naming Convention:** Save as `test_<TARGET_FUNCTION>_negative.py` in the `tests/` directory in the
    Python project. If the file already exists, append the new tests to it without removing any existing
     tests. It is fine if you find existing tests are enough to cover all scenarios, in that case, you
    can skip the generation of additional tests.
- **Setup:** Define inputs that should trigger failures (e.g., incorrect data types, out-of-bounds
    parameters, missing required arguments).
- **Execution & Comparison:** 1. Use `rpy2` to call the R function and catch the resulting R error
    message as a string.
  2. Use `pytest.raises(Exception) as exc_info` to catch the Python function's error.
  3. **Pass Condition:** If both functions raise an error, the test passes. If one raises an error and
    the other does not, the test fails.
  4. **Warning Condition:** Compare the text of the Python error message (`str(exc_info.value)`) with the
     caught R error message. If they differ in phrasing, do not fail the test, but use Python's `
    warnings.warn()` to print a warning indicating the discrepancy in error messaging.

### Step 5: Generate Boundary and Edge Case Tests

Generate at least 8 tests focusing strictly on the functional limits and extremes. You have to cover all
    edge cases that the original R function handles, such as minimum/maximum values, `NaN`, `Inf`, empty
     inputs, and singleton values. The goal is to ensure that the Python function's behavior in these
    edge cases is identical to the R function's behavior.
- **Naming Convention:** Save as `test_<TARGET_FUNCTION>_edge.py` in the `tests/` directory in the Python
     project. If the file already exists, append the new tests to it without removing any existing tests.
     It is fine if you find existing tests are enough to cover all scenarios, in that case, you can skip
     the generation of additional tests.
- **Setup:** Use inputs such as minimum/maximum possible values, `NaN`, `Inf`, empty strings, empty
    arrays/DataFrames, or singleton values.
- **Execution:** Determine whether the R function resolves these extremes with a valid output or an error
    .
- **Comparison:** Use the exact same validation logic applied in Step 3 (for successful outputs) or Step
    4 (for expected errors) to ensure the Python script's edge-case handling is identical to R's.
```

fix-bug-found-in-test

```
---
name: fix-bug-found-in-test
description: Analyzes and fixes a failing test for a Python function to ensure it correctly reflects the
    corresponding R library's behavior.
---

# Fix Bug Found in a Test

## Description

You will be provided with a failing Python test, a Python library folder, and the corresponding original
    R library folder. The test is currently failing due to an assertion error.
```

```
Your task is to analyze the test, compare the Python implementation to the original R implementation, and
      fix the root cause of the failure. The ultimate goal is to ensure the Python test passes
     successfully and accurately validates that the Python function matches the R function's behavior.

## Execution Steps

### Step 1: Analyze the Failing Test

- **Review the Test:** Understand the test function's objective and the expected behavior of the Python
     function being evaluated.
- **Analyze the Failure:** Identify the specific failing assertion. Review the error message to
     understand the discrepancy between the expected and actual outputs.
- **Validate Test Setup:** Check if the test inputs, mock data, and parameters are correctly structured
     and properly passed to the function.

### Step 2: Compare with the Original R Implementation

- **Review R Logic:** Examine the original R implementation of the target function to understand its
     expected outputs for the given inputs.
- **Compare Implementations:** Cross-reference the logic, data handling, and output formatting of the R
     function against the Python function.
- **Identify Edge Cases:** Note any specific scenarios or edge cases handled by the R function that might
      be missing or misrepresented in the Python implementation or the test itself.

### Step 3: Implement the Fix

- **Prioritize Consistency:** Your primary directive is to ensure logical consistency between the Python
     library and the original R library.
- **Determine the Root Cause:** Identify whether the assertion failure is due to a flawed test (e.g.,
     incorrect expected output, bad inputs) or a bug in the Python library code itself.
- **Apply the Correction:** - If the **test** is flawed: Modify the test inputs, expected outputs, or
     assertion structure so it correctly expects the R implementation's behavior.
  - If the **Python library** is flawed: Update the Python library code to match the R implementation so
     the test passes.
- **Document Intentional Deviations:** If fixing the bug requires introducing a deliberate logical
     inconsistency between the Python and R libraries, you must document this clearly in the code
     comments and provide a rationale for the divergence.
- **Verify:** Ensure the test now runs and passes successfully.
```

A.6 Supplementary Agent

The following agent is defined in the repository but has not yet been invoked in this project. It is included for completeness and as a reference for the Fortran source analysis task identified as a potential next step in Phase 1.

`analyze-c-file-dependencies`

```
---
name: analyze-c-file-dependencies
description: Analyzes a provided C file to extract structural dependency information function by function
      (including function-like macros), categorizing calls strictly into internal and external
     dependencies.
---

# Analyze a C File for Dependencies

## Description
```

```
When provided with a C file (‘.c‘ or ‘.h‘), your task is to parse the file, identify every user-defined
     function or function-like macro, and trace all function-like calls (functions and function-like
     macros) made within their bodies. You must categorize these dependencies into strict groups and
     output the result as a structured JSON object.

## Execution Steps

### Step 1: Extract Function or Macro Declarations

Scan the target C file and identify every user-defined function or function-like macro defined within it
     (e.g., ‘int my_func() { ... }‘). Create a list of these function or macro identifiers.

### Step 2: Analyze Function or Macro Bodies and Trace Calls

For every function or macro identified in Step 1, thoroughly analyze its execution body. Extract the
     identifier of every function or function-like macro invoked within that body.

**Exclusions:**
* Completely ignore functions or macros that are part of the C standard library (e.g., ‘printf‘, ‘malloc
     ‘, ‘strlen‘, ‘size_t‘).
* Extract **only the identifier** (e.g., ‘PyArg_ParseTuple‘, ‘.Call‘), never the arguments or the
     parentheses.

Categorize the extracted identifiers into the following two mutually exclusive lists. You may need to
     traverse and scan other files in the same project folder according to the included header files, and
      you have to use the following heuristics:

* **Internal Dependencies:**
    * *Definition:* Identifiers for functions or macros defined within the same C file, or those that
     appear in other ‘.c‘ or ‘.h‘ files within the same project folder, which are included via ‘#include‘
      directives into the target file.
* **External Dependencies:**
    * *Definition:* Identifiers for functions or macros belonging to known external, third-party
     libraries or language bindings (for example, R API calls like ‘Rf_protect‘, Python C API calls like
     ‘Py_Initialize‘, etc.). Only if you cannot find the definition of the function or macro in the same
     file or any other file in the same project folder, you should classify it as an external dependency.

### Step 3: Generate JSON Output

Construct a JSON object where each key is the name of a function or macro identified in Step 1. The value
      for each key must be an object containing exactly two arrays of strings: ‘internal_dependencies‘
     and ‘external_dependencies‘. Deduplicate all identifiers within these arrays.

## Output Format Schema

You must output valid JSON strictly adhering to this structure. Do not include markdown code blocks if
     the system expects raw JSON.

{
  "my_custom_algorithm": {
    "internal_dependencies": ["helper_function_within_file", "project_utils_init"],
    "external_dependencies": ["Rf_protect", "PyArg_ParseTuple"]
  },
  "helper_function_within_file": {
    "internal_dependencies": [],
    "external_dependencies": []
  }
}
```

B Skill (Command) Specifications

This appendix reproduces the complete specifications of the top-level skills (slash commands) deployed in this project. Each file is stored under `.claude/commands/`. Skills are the entry points invoked interactively; they orchestrate one or more sub-agents (Appendix A) and handle batch dispatch, error recovery, and output persistence.

B.1 Skills Used in Phase 1

`analyze-r-folder-dependencies`

```
---
name: analyze-r-folder-dependencies
description: Analyzes all R scripts in a folder to extract structural dependency information file by file
     , categorizing function calls into language, internal, and external dependencies.
---

# Analyze an R Folder for Dependencies

## Description

When provided with a target folder, your task is to recursively scan for all R scripts, use the `@analyze
     -r-file-dependencies` agent on each file individually, and save the results as structured JSON files.


## Execution Steps

### Step 1: Scan for R Files

Recursively scan the provided target folder and all of its subdirectories for files with `.R` or `.r`
     extensions. Compile a complete list of their absolute file paths.

### Step 2: Process and Analyze

Iterate through the list of identified R files sequentially. For each file:
1. Invoke the `@analyze-r-file-dependencies` agent against the file and record its final JSON output.
2. If the command fails or throws an error for a specific file, log the error to the console and
     immediately continue to the next file in your list. Do not halt the entire operation.

### Step 3: Save Output

For every successfully analyzed R file, save the resulting JSON output according to these strict rules:
* **Naming Convention:** Use the exact name of the original R script, but replace the `.R` or `.r`
     extension with `.json` (e.g., `data_cleaning.R` becomes `data_cleaning.json`).
* **Output Location:** If the user did not specify the output folder, save the new `.json` file in `
     structural_analysis/{folder_name}/`, where the `{folder_name}` is the name of the target folder. The
      relative path must be the same as its corresponding original R file.
```

B.2 Skills Used in Phase 3

`extract-r-folder-language-dependencies`

```
---
name: extract-r-folder-language-dependencies
description: Recursively processes a folder of R files, utilizing pre-calculated structural analysis
     JSONs to generate detailed CSV reports of language dependency locations.
---
```

```
# Extract Language Dependencies from an R Folder

## Description

When provided with a target directory containing R source files and a separate directory containing their
     corresponding structural analysis results (in JSON format), your task is to process every R script
     recursively. You will invoke the `@extract-r-file-language-dependencies` agent on each matched file
     pair (the `.R` file and its corresponding `.json` analysis), and save the extracted dependency data
     strictly as `.csv` files in a mirrored output directory.

## Execution Steps

### Step 1: Scan for R Source Files

Recursively scan the provided target directory and all of its subdirectories. Compile a complete list of
     absolute file paths for every file with an `.R` or `.r` extension.

### Step 2: Map to JSON and Process

Iterate through the list of identified R files sequentially. For each R file:
1. **Locate the JSON Input:** Find the corresponding dependency analysis JSON file. It will be located in
      the provided structural analysis directory, sharing the exact same relative path as the R file, but
      with the `.R`/`.r` extension replaced by `.json`.
2. **Execute File-Level Agent:** Invoke the `@extract-r-file-language-dependencies` agent, providing it
     with the R file and its paired JSON file to generate the dependency data.
3. **Error Handling:** If the agent fails, throws an error, or if the required JSON file is missing, log
     a clear error to the console for that specific file and immediately proceed to the next R file in
     the list. Do not halt the batch execution.

### Step 3: Save Output as CSV

For every successfully analyzed file pair, save the resulting output from the agent according to these
     strict rules:
* **Data Format:** The output must be saved strictly as a CSV file.
* **Naming Convention:** Use the exact base name of the original R script, but replace the `.R` or `.r`
     extension with `.csv` (e.g., `data_cleaning.R` becomes `data_cleaning.csv`).
* **Output Directory:** Maintain the original directory structure. Save the `.csv` file into the user-
     specified output directory, using the exact same relative path as the original R file.
* **Default Output Directory:** If the user did not explicitly specify an output directory, create a
     default directory path named `language_dependency_analysis/{target_folder_name}/` (where `{
     target_folder_name}` is the base name of the target R folder) and mirror the relative paths there.
```

generate-language-dependency-conversion-guides

```
---
name: generate-language-dependency-conversion-guides
description: Orchestrates the generation of conversion guides for translating language dependencies from
     R to Python based on a provided CSV file.
---

# Generate Language Dependency Conversion Guides

## Description

When provided with a target `base_folder` and a CSV file matching the following schema:
```csv
language_dependency,file_path,function_name,line_number,call_body
Re,all.R,bkde,78,"Re(fft(kappa*gcounts, TRUE))"
Re,all.R,bkde2D,156,"Re(fft(rp*sp, inverse = TRUE)/(P1*P2))"
Re,all.R,bkfe,232,"Re(fft(kappam*Gcounts, TRUE))"
abs,all.R,dpill,531,abs(th24Q)
```
```

```
any,all.R,locpoly,610,any(bandwidth <= 0)
as.double,all.R,blkest,265,as.double(x)
as.double,all.R,blkest,265,as.double(y)
as.double,all.R,blkest,267,as.double(xj)
```
Your task is to orchestrate the batch processing of this CSV. You must extract rows corresponding to each unique `language_dependency` into separate CSV-formatted strings (each including the header row), invoke the `@generate-language-dependency-conversion-guide` agent for each subset, and save the resulting guides into a specified output directory.

Execution Steps

Step 1: Identify Unique Dependencies

Parse the input CSV file to identify all unique values in the `language_dependency` column.

Note: The CSV is pre-sorted so identical dependencies are grouped together, allowing for efficient sequential processing.

Step 2: Extract and Process Each Dependency

Iterate through the list of identified unique `language_dependency` values sequentially. For each unique dependency:
1. **Extract the CSV Subset:** Isolate all rows matching the current `language_dependency`. Prepend the standard CSV header row to this subset to create a valid CSV text string.
2. **Execute Conversion Agent:** Invoke the `@generate-language-dependency-conversion-guide` agent. Pass the `base_folder` and the extracted CSV string as inputs to generate the specific conversion guide.
3. **Error Handling:** If the agent fails or throws an error during generation, log a clear error message to the console specifying the failed `language_dependency`, and immediately proceed to the next dependency in your list. Do not halt the overall batch execution.

Step 3: Save Output as Markdown Files

For every successfully generated conversion guide, save the agent's markdown output to the file system according to these strict rules:
* **Naming Convention:** Name the output file exactly `{language_dependency}.md` (e.g., `Re.md`, `abs.md`).
* **Output Directory:** Save the files to the user-specified output directory. If the user did not explicitly provide one, create and use a default directory path named `language_dependency_analysis/conversion_guides/`.

B.3 Skills Used in Phase 4

convert-r-file-to-python

```
---
name: convert-r-file-to-python
description: Converts all functions in an R file to Python by parsing dependency graphs and sequentially invoking the function conversion agent.
---

# Convert an R File to Python

## Description

Translate all functions in an entire R file into Python. You will use the R source file, a comprehensive JSON dependency map, a CSV dependency graph, and a language dependency conversion guide folder. You must parse the dependencies, establish the correct conversion order, and invoke the `@convert-r-function-to-python` agent for each function.

## Execution Steps
```

```
### Step 1: Extract Function Definitions and Dependencies

* Parse the provided JSON file. The top-level keys represent all the function names within the target R
    file.
* For each identified function, extract its corresponding R code definition from the R source file.
* Isolate the JSON value associated with each function key to serve as its specific dependency map.

### Step 2: Sort Functions by Dependency Order (Leaves to Roots)

* Filter the provided dependency CSV file by the `r_file` column to isolate entries relevant only to the
    current target file.
* Sort the functions identified in Step 1 topologically, from leaves to roots. Ensure that child
    functions always precede their parent functions in the sorted execution list.
* *Note:* If a child function listed in the CSV is not found in the current R file's JSON/source, assume
    it has been converted elsewhere. Ignore it for the purposes of sorting the current file.

### Step 3: Execute Sequential Conversions

Iterate through the sorted function list from Step 2 sequentially. For each function:
1. **Invoke Conversion Agent:** Call `@convert-r-function-to-python`. Provide the following context:
   * The extracted R code definition.
   * The extracted JSON dependency map.
   * The folder containing language dependency conversion guides.
   * The target output folder for the specific R file. It should be `{target_output_folder}/{R_file_name
     }/`. For example, `{target_output_folder}/all.R/`. If the `{target_output_folder}` is not specified,
      the default should be `conversion_results/R/`.
2. **Error Handling:** If the agent fails or throws an error for a specific function, log a clear error
    message to the console. **Do not halt** the overall batch execution; proceed to the next function.
```

combine-python-functions-into-file

```
---
name: combine-python-functions-into-file
description: Combines all converted Python functions (stored as JSON) from a specified directory into a
    unified target Python file, handling import deduplication and project integration.
---

# Combine Converted Python Functions into a File

## Description

Given a source folder containing JSON-formatted function data and a target Python file, your task is to
    extract, format, and combine these functions into a single Python script. You must resolve duplicate
     imports, format the code syntactically, and integrate it safely into the target project structure.

## Execution Steps

### Step 1: Discover and Map JSON Files

* Scan the specified source folder for all files ending with `.json`.
* The filename (excluding the `.json` extension) represents the original R function name.

### Step 2: Extract and Format Function Definitions

For each JSON file, extract the `imports`, `function_prototype`, and `function_body`.
* **Concatenation:** Combine the `function_prototype` and the `function_body` array into a single multi-
    line string.
* **Indentation:** Ensure standard Python indentation. The `function_prototype` should have zero
    indentation, and every line within the `function_body` must be indented by exactly 4 spaces relative
     to the prototype.
```

```
*Example JSON Structure:*
‘‘‘json
{
  "imports": [
    "import numpy as np",
    "from . import _KernSmooth"
  ],
  "function_prototype": "def rlbin(X: np.ndarray[np.float64], Y: np.ndarray[np.float64], gpoints: np.
    ndarray[np.float64], truncate: bool = True) -> dict:",
  "function_body": [
    "    n = len(X)",
    "    M = len(gpoints)",
    "    return {\"xcounts\": xcnts, \"ycounts\": ycnts}"
  ]
}
‘‘‘

### Step 3: Deduplicate and Sort Imports

* Aggregate all ‘imports‘ arrays from every JSON file.
* Remove any duplicate import statements.
* Sort the deduplicated imports logically:
  1. Standard library imports (e.g., ‘import math‘).
  2. Third-party imports (e.g., ‘import numpy as np‘).
  3. Local project imports (e.g., ‘from .linbin import linbin‘).
* Combine the sorted imports and the formatted function definitions into one final multi-line string.

### Step 4: Validate and Output the Combined Program

Do not blindly overwrite the target Python file. You must ensure the generated code aligns with the
    existing project architecture:
1. **Structural Scan:** Analyze the target project’s build and configuration files (e.g., ‘pyproject.toml
    ‘, ‘meson.build‘, ‘__init__.py‘) to verify correct module naming conventions and dependency paths.
2. **Binding Corrections:** If the structural scan reveals discrepancies in how external C/Fortran
    bindings are exposed (e.g., ‘_KernSmooth‘ is built under a different path than what the JSON imports
     suggest), automatically fix the import paths in your combined string to match the reality of the
    build system.
3. **Safe Write:** Output the finalized string into the target Python file. If the file already contains
    code, integrate the new functions safely without breaking existing syntax or overwriting required
    project-level configurations.
```

B.4 Skills Used in Phase 6

convert-r-tests-to-python

```
---
name: convert-r-tests-to-python
description: Batch converts all R test files in a specified folder to Python by invoking the convert-r-
    test-to-python agent.
---

# Convert all R Test Files in a Folder to Python

## Description

Translate an entire directory of R test files into Python. You will require the paths to the R test
    source folder, the source R library folder, and the corresponding Python library folder. For every R
     test file discovered, you must invoke the ‘@convert-r-test-to-python‘ agent to handle the
    individual translation.

## Execution Steps
```

```
### Step 1: Discover All R Test Files

- Scan the provided R test source folder recursively.
- Identify and compile a list of all R scripts (files ending with `.R` or `.r`). Pay special attention to
     standard test naming conventions (e.g., files starting with `test-` or `test_`).

### Step 2: Execute Conversions Iteratively

- Iterate through the compiled list of R files from Step 1 sequentially.
- For each file, invoke the conversion agent by calling the `@convert-r-test-to-python` agent.
- Pass the following explicit context to the agent for each invocation:
  1. The exact path to the current R test file.
  2. The path to the source R library folder.
  3. The path to the corresponding Python library folder (e.g., the root of the `r2py_kernsmooth` package
     , ensuring the agent can correctly locate or create the `tests/` directory as per its instructions).
  4. The output folder if specified.

### Step 3: Error Handling and Logging

- Monitor the output of the `@convert-r-test-to-python` agent.
- If the agent fails, hallucinates, or throws an error for a specific file, output a clear error message
    to the console noting the specific file that failed.
- **Do not halt** the batch execution. Catch the exception and proceed immediately to the next file in
    the list until all files have been attempted.
```

B.5 Skills Used in Phase 7

generate-python-file-tests

```
---
name: generate-python-file-tests
description: Generates Python unit tests for all public interfaces in a given Python file by parsing the
    file, referencing R documentation, and invoking the function test generator agent sequentially.
---

# Generate Python File Tests

## Description

When provided with a target Python file path and its corresponding R package documentation reference,
    your task is to parse the file for all function definitions, filter out internal helpers to isolate
    public interfaces, and sequentially invoke the `@generate-python-function-tests` agent to build
    comprehensive `pytest` suites for each public function.

## Execution Steps

### Step 1: Extract All Python Function Definitions

* Parse the target Python source file to identify and extract all function definitions.
* Compile a comprehensive list of all function names present in the file, regardless of their naming
    conventions (e.g., ignoring leading underscores for now).

### Step 2: Filter Public Interfaces via R Documentation

* Navigate to the documentation of the original R package (e.g., via CRAN, local `.Rd` files, or the
    package's reference manual). *Example:* For the `KernSmooth` package, review its documentation at `
    https://cran.r-project.org/web/packages/KernSmooth/refman/KernSmooth.html`. Use this to identify all
     the public interfaces of the R package.
* Cross-reference the comprehensive list of Python functions from Step 1 against the exported (public)
    functions detailed in the R documentation.
```

```
* Filter the list to retain *only* the functions that are documented as public interfaces in the original
      R package. Discard internal or private helper functions from the execution queue.

### Step 3: Execute Sequential Test Generation

Iterate through the filtered list of public functions from Step 2 sequentially. For each function:
1. **Invoke Test Generation Agent:** Call the `@generate-python-function-tests` agent. Provide the
     following context:
   * The target Python function name.
   * The target Python file path.
   * The original R package/function reference for functional equivalence benchmarking.
2. **Error Handling:** If the agent fails, times out, or throws an error while generating tests for a
     specific function, log a clear error message to the console. **Do not halt** the overall batch
     execution; proceed immediately to the next function in the queue.
```

fix-bugs-found-in-tests

```
---
name: fix-bugs-found-in-tests
description: Automatically runs a test suite, identifies failing tests, invokes the fix-bug-found-in-test
      agent to resolve them, and repeats until all tests pass.
---

# Fix Bugs Found in Tests

## Description

You will be provided with a folder containing Python test files, a target Python library folder, and the
     corresponding original R library folder. Your task is to execute the entire test suite, identify any
      test failures, and systematically invoke the `@fix-bug-found-in-test` agent to resolve the root
     cause of each failure. You will repeat this test-and-invoke cycle autonomously until the entire test
      suite passes.

## Execution Steps

### Step 1: Execute the Test Suite

- Navigate to the designated folder containing the Python test files.
- Run the full test suite using `pytest` for all files matching the pattern `test_*.py`.
- Capture the full terminal output from `pytest`. If all tests pass, exit successfully.
- If there are failures, isolate the very first failing test to begin the debugging process. Address
     failures sequentially.

### Step 2: Invoke the Fix Agent

- For the isolated failing test, invoke the `@fix-bug-found-in-test` agent.
- Ensure you pass the agent the necessary context it requires: the specific failing Python test (the
     output error message, the function, and the test file), the Python library folder, and the
     corresponding original R library folder.
- Allow the `@fix-bug-found-in-test` agent to complete its execution.

### Step 3: Validate and Loop

- Once the `@fix-bug-found-in-test` agent has finished its task, reinstall the fixed package, and then
     rerun the entire test suite using `pytest` across all `test_*.py` files to ensure the fix worked and
      no regressions were introduced.
- If new or remaining tests fail, repeat Steps 1 through 3 for the next failing test.
- Continue this cycle until `pytest` returns a 100% pass rate.
- *Safety Constraint:* If the `@fix-bug-found-in-test` agent attempts to fix the exact same failing test
     3 times consecutively without the test passing, halt the loop and request user intervention to
     prevent an infinite loop.
```

B.6 Workflow Utility

The following skill was used at the end of each phase to produce the structured phase summaries stored in `docs/daily_summaries/`, which are the primary source material for this report.

`summarize-research-report`

```
---
name: summarize-research-report
description: Summarizes the current session's activities, modifications, and findings into a precise,
     formal, and concise research report.
---

# Summarize Session into Research Report

This command analyzes the current working session and synthesizes the activities into a formal, concise
     research report.

## Prompt

Review the history of the current session, including all user requests, tool executions, codebase queries
     , and file modifications. Based on this context, generate a formal research report summarizing the
     work accomplished.

The report must adhere strictly to the following parameters:
- **Tone**: Precise, formal, and objective. Avoid conversational filler.
- **Length**: Highly detailed in facts but strictly concise in wording. Do not exceed 500-700 words
     unless absolutely necessary to capture critical technical details.
- **Specificity**: Explicitly mention specific file names, library versions, metrics, or core
     architectural decisions discussed or modified during the session. Avoid generic summaries.

Please structure the report using the following markdown format:

### 1. Abstract

Provide a 2-3 sentence high-level overview of the session's primary objective and the final outcome or
     state of the project.

### 2. Methodology & Actions Taken

Detail the concrete steps taken during the session. Include:
- Which files were created, modified, or analyzed.
- What tools or commands were executed.
- Any bugs investigated and the approach taken to resolve them.

### 3. Key Findings & Results

Highlight the core results of the session:
- Output metrics, successful compilations, or structural changes.
- Key technical insights or discoveries made during the work.
- Resolutions to any problems encountered.

### 4. Conclusion & Next Steps

Briefly conclude the report by stating the final status of the session's goal.
```