RISC-V Parallel Programming for Signal and Image Processing

This book and presented **labs** are devoted to exploring how modern RISC-V processors can provide an effective solution for parallel programming and high-performance processing using Python and optimized scientific libraries.

RISC-V represents a significant shift in the field of processor design. Unlike traditional proprietary instruction set architectures, RISC-V is open, modular, and freely available. This openness has encouraged rapid innovation in universities, research laboratories, and industry alike. Designers are free to implement, extend, and optimize the architecture according to the needs of specific applications. The clean and well-structured design of RISC-V also makes it particularly suitable for both high-performance computing platforms and modern embedded systems. As a result, it has become one of the most promising foundations for the next generation of computing technologies.

A central theme of this book is **performance through parallelism**. Modern RISC-V processors combine **multicore** architectures with powerful **vector-processing** capabilities. Multicore systems improve performance by executing multiple tasks simultaneously, while the RISC-V vector extension enables a single instruction to operate on multiple data elements at once. This form of data-level parallelism is especially well suited to applications in signal and image processing, where the same operations are repeatedly applied to large data sets. By combining multicore execution with vector operations, RISC-V offers a scalable and energy-efficient computing model capable of addressing demanding real-world problems.

Another important aspect of this book is the role of **Python** as a high-level development environment. Python has become one of the most widely used languages in scientific and engineering applications thanks to its clear syntax, readability, and extensive ecosystem of libraries.

Optimized libraries such as **NumPy**, **SciPy**, and OpenCV allow developers to implement complex algorithms quickly and with relatively little code. Although Python itself is a high-level language, many of its scientific libraries rely on highly optimized low-level implementations. This makes it possible to combine the productivity of Python with the performance of optimized numerical routines, including those that take advantage of vector instructions and other hardware-acceleration features available on modern RISC-V processors.

This book aims to bridge the gap between architectural concepts and **practical algorithm development**. It explains how computationally intensive operations can be implemented efficiently while remaining accessible to readers who may not have extensive experience in low-level programming. Through a series of practical examples, the book demonstrates how modern RISC-V processors and high-level programming tools can work together to create efficient, portable, and scalable solutions.

Ultimately, this book is intended for students, researchers, and engineers who are interested in high-performance computing, signal processing, image processing, and modern processor architectures.

Whether the reader is approaching RISC-V for the first time or seeking to deepen their understanding of parallel computing techniques, the goal of this work is to provide a clear and practical path toward efficient algorithm design on one of the most promising computing platforms of our time

Table of Contents

Lab 0 : Introduction

Lab 0

Introduction

This book and labs focuse on how modern **RISC-V processors** can be used to efficiently handle computationally intensive tasks in signal and image processing. These applications require high performance because they involve large data sets and repeated numerical operations such as filtering, convolution, and matrix computations. The RISC-V architecture provides an ideal platform for addressing these challenges.

RISC-V is an open and modular instruction set architecture (ISA) designed to provide a flexible and efficient alternative to traditional proprietary processor architectures.Unlike **commercial ISAs** such as x86 or ARM, RISC-V is freely available and can be implemented, extended, and optimized by universities, researchers, and industry. Its clean and modular design also makes it particularly suitable for both high-performance computing systems and modern embedded platforms.

One of the key strengths of RISC-V is its support for **parallel computing**. Modern RISC-V processors can combine multicore architectures with vector processing capabilities.

Multicore processors improve performance by allowing several tasks to be executed simultaneously, while the RISC-V vector extension (RVV) enables a single instruction to operate on multiple data elements at the same time.

This form of data-level parallelism is especially effective for applications in signal processing, image processing, scientific computing, and machine learning.

By combining multicore parallelism with vector processing, RISC-V offers a powerful and scalable computing model. Programs can distribute workloads across multiple cores while each core processes large data sets using vector operations.

This approach significantly improves both performance and energy efficiency, making RISC-V an attractive platform for research, high-performance computing, and next-generation parallel programming.

Python plays an important role in this context. Thanks to its clear syntax and its rich ecosystem of scientific libraries such as NumPy, SciPy, and OpenCV, Python provides a high-level environment for algorithm development and rapid prototyping. Although Python itself is a high-level language, many of its scientific libraries rely on **highly optimized C and C++ implementations**.

When Python is combined with these **optimized libraries**, computationally intensive operations such as **filtering**, **convolution**, **Fourier transforms**, and large-scale matrix processing can be executed efficiently. These libraries can take advantage of hardware acceleration, including vector instructions (RVV) and SIMD capabilities available on modern RISC-V processors.

As a result, developers can design signal and image processing algorithms that are both easy to implement and highly efficient, achieving a balance between productivity, portability, and performance.

0.1 From SciPy/NumPy to Python, C and -RISC-V Assembly Programming Stack

RISC-V is an open-source, modular CPU instruction set architecture (ISA) that has gained widespread attention due to its flexibility, simplicity, and extensibility. Unlike proprietary ISAs, RISC-V provides a clean and standardized base instruction set with optional extensions, allowing hardware designers to tailor processors for specific applications.

This modularity enables RISC-V cores to span a wide range of platforms, from low-power embedded microcontrollers to high-performance CPUs capable of scientific computation, multimedia processing, and machine learning. Its open-source nature encourages innovation, reduces development costs, and facilitates academic research, making it an ideal choice for experimental architectures and novel computing paradigms.

At the software level, **Python** has emerged as a preferred language for RISC-V programming, particularly for rapid prototyping and high-level computation. Python's interpreted nature, dynamic typing, and rich standard library allow developers to implement algorithms, manipulate data, and experiment with complex workflows quickly and efficiently, without the need to manage low-level hardware details. Python code can perform tasks such as numerical computations, linear algebra operations, image processing, and data analysis in a highly readable and maintainable way.

For instance, using **`NumPy`**, a developer can generate **large matrices** and perform element-wise operations in concise code, while **OpenCV** enables sophisticated image processing with minimal Python instructions.

A 1024×1024 matrix transformation can be written simply as **`b=np.sin(a)+np.cos(a)`**, and image color-space conversions can be executed with a single **cv2.cvtColor call.**

Beneath this high-level code, the Python interpreter, typically CPython on RISC-V Linux systems, converts Python scripts into bytecode stored in **`.pyc`** files and executes it using a virtual machine. The interpreter manages memory, variable scope, and control flow while dispatching computationally intensive operations to C or C++ extension modules for efficiency. This layered execution allows Python developers to maintain the simplicity of high-level scripting while relying on **compiled routines for performance**.

The next layer in the stack is the accelerated libraries, which form the bridge between Python and the hardware. Libraries such as **`NumPy`**, **`SciPy`**, **`OpenCV`**, and **`Pandas`** are implemented in optimized C or C++, allowing them to execute heavy operations efficiently. High-level Python calls, such as **matrix multiplication**, **Fourier transforms**, or **image convolution**, are offloaded to these compiled routines, avoiding slow Python loops. On RISC-V platforms, these libraries can leverage **RISC-V Vector Extensions (RVV)** to perform vectorized operations, processing multiple data elements in parallel, and **SIMD instructions** to accelerate

element-wise computation. As a result, Python code benefits from high-level expressiveness while the libraries perform computations at speeds close to native execution.

At the **C libraries layer**, performance-critical routines are implemented with hardware efficiency in mind. For example, `NumPy` relies on C implementations with optional **BLAS** and **LAPACK** backends to optimize linear algebra operations, while `OpenCV` uses highly optimized C++ routines for image processing. When a Python function like `np.dot` is called, it is dispatched to the corresponding C function, which may be compiled to exploit vectorization, loop unrolling, and **cache-aware memory access**. These optimizations ensure that large-scale computations, such as matrix products or convolution operations, execute efficiently on RISC-V hardware without requiring manual assembly programming from the Python developer.

At the **assembly layer**, the compiler translates C and C++ code into **RISC-V assembly instructions**. Compilers such as **GCC** or **LLVM** generate instructions that fully exploit the processor's features, including the RVV vector units. Vectorized instructions allow multiple elements of arrays or matrices to be processed simultaneously, while memory operations and loops are mapped efficiently to hardware registers and caches. This layer transforms high-level computational routines into instructions executed directly by the CPU, maximizing performance while remaining transparent to the programmer.

The combination of Python, accelerated libraries, C/C++ backends, and RISC-V assembly creates a **highly productive and efficient programming stack**. Developers can write concise, readable Python code while relying on underlying libraries to handle hardware-aware optimization, including vectorization, memory alignment, and instruction-level parallelism. This approach provides several concrete **benefits**.

First, it ensures **high productivity**, as Python code is easy to write, understand, and maintain, allowing developers to focus on problem-solving and algorithm design rather than low-level hardware intricacies.
Second, it delivers **high performance**, since critical loops and computations run in optimized C using vector instructions, enabling operations such as large-scale matrix multiplication, image filtering, or scientific simulations to execute at speeds approaching native code.
Third, it offers **portability**, because the same Python code can run across multiple architectures—including x86, ARM, and RISC-V, while relying on underlying libraries to perform architecture-specific optimizations.
Finally, the stack provides **hardware-aware acceleration**, allowing RISC-V **vector extensions** (**RVV**) **to dramatically speed up** linear algebra routines, signal processing tasks, image filters, and other data-parallel operations, ensuring that Python applications fully leverage the capabilities of modern RISC-V processors.
By combining these layers, from high-level Python code to interpreter, accelerated libraries, C/C++ routines, and assembly execution, the Python-to-RISC-V programming stack achieves a unique balance of usability, portability, and computational efficiency. Developers can rapidly prototype and iterate in Python while relying on optimized compiled routines and vectorized instructions to handle the performance-critical aspects of their applications.

This layered architecture makes **Python a practical language** for demanding workloads on RISC-V platforms, including scientific computing, machine learning, image and signal processing, real-time embedded applications, and other high-performance tasks. Ultimately, the stack demonstrates how a high-level language like Python, when paired with carefully designed libraries and modern processor features, can deliver both developer productivity and hardware efficiency on open, flexible architectures such as RISC-V.

0.2 Example Use-Cases

The versatility of Python combined with accelerated libraries on RISC-V enables a wide range of practical applications across different domains. In **scientific computing**, for instance, developers can perform large-scale numerical computations such as matrix multiplications, linear algebra operations, and complex simulations.

These tasks, which often involve millions of data elements, can be efficiently handled using Python for high-level algorithm design while delegating performance-critical operations to C libraries that exploit RISC-V vector instructions, allowing simulations to run faster and with lower energy consumption. In the domain of **image and video processing**, RISC-V platforms paired with Python and libraries such as OpenCV provide the tools necessary to implement filters, edge detection algorithms, geometric transformations, and real-time video analysis. High-level Python code simplifies algorithm development, while the underlying C++ routines execute efficiently using SIMD or vectorized instructions to process large image frames in parallel.

For **machine learning**, Python serves as the prototyping environment where developers can experiment with models, data preprocessing, and feature extraction using libraries like NumPy. Once algorithms are validated, the computationally intensive parts, such as matrix multiplications, convolutions, or activation functions, can be accelerated with compiled C++ code or optimized RISC-V vector instructions, bridging the gap between rapid prototyping and high-performance execution.

Finally, in **embedded systems**, Python on RISC-V can be employed for processing sensor data on microcontrollers, performing tasks such as filtering, statistical analysis, and signal processing directly on the device.

By leveraging **vectorized** and **parallel operations** provided by RISC-V hardware, even resource-constrained embedded systems can execute real-time computations efficiently, enabling applications in IoT devices, robotics, and autonomous systems. Across all these domains, the combination of Python's high-level expressiveness and the performance of accelerated libraries allows developers to implement complex algorithms quickly, while fully exploiting the computational capabilities of RISC-V processors.

0.3 RISC-V platform for SIMD/MIMD programming

Google developed the **Highway library** to be a **portable and future-proof** solution for **SIMD** programming. Its efficiency on **RISC-V** with the **Vector** extension (**RVV**) is not an accident but a direct result of its core design, which was intentionally created to handle the unique challenges posed by modern, variable-length vector architectures like **RVV** and **ARM SVE** .

Here is a breakdown of why **Highway** is so well-suited for **RISC-V**:

0.3.1 The Core Problem: "Sizeless" Vectors

Traditional **SIMD** libraries often wrap vector data in C++ classes. However, this approach hits a fundamental wall with **RISC-V RVV** and ARM SVE.

- **The Issue**: On these architectures, the length of a vector is not fixed at compile time. It is determined by the hardware and can even change with different CPU settings. This creates a "**sizeless**" type that standard C++ classes cannot easily represent as member variables .
- **The Common (but limiting) Solution**: Some libraries get around this by using special compiler flags to force a fixed vector length. This is only practical when the exact hardware is known in advance, like in a supercomputer .

0.3.2 Highway's Elegant Solution: Tag-Based Design

Highway's primary innovation is its **tag-based design**, which completely separates the **data** from the **operations** on that data .

- **Tags, Not Wrappers**: Instead of wrapping vectors in classes, Highway uses empty "tag" types (like `ScalableTag<float>`) to select the correct operation. The actual vector data is stored using the compiler's own built-in types for RVV (e.g., `vuint32m1_t`) .
- **The Result**: This allows Highway to work directly with the hardware's **sizeless** vector types. It can run efficiently on **any RISC-V CPU with RVV**, regardless of the specific vector length, without needing a recompile or multiple versions of the binary .

0.3.3 Purpose-Built Implementation for RVV

Highway wasn't just designed to **tolerate** RVV; it was built to **embrace its complexity**. The library includes a dedicated and sophisticated backend for RISC-V, located in `rvv-inl.h` .

- **Handling LMUL**: RVV's flexibility includes a register **grouping** feature called **LMUL** (**Length Multiplier**), which can make the effective vector length 1/8th, 1/4th, ..., up to 8 times the base length. Highway's RVV implementation uses a complex macro system to handle all these variants, as well as widening and narrowing operations .
- **Support for Masked Operations**: RVV supports predicated execution (masked operations), and Highway's design fully accounts for this .
- **Enabling Projects like NumPy**: The Highway library acts as a crucial bridge. As highlighted in the **RuyiSDK** community announcement, the **Institute of Softwar**e, **Chinese Academy of Sciences** (**ISCAS**) used Highway to implement RISC-V Vector (RVV) support in NumPy. They leveraged Highway to create an **"end-to-end optimization path from high-level operators down to low-level RVV intrinsic"**. This means NumPy's Python-level array operations can now be efficiently translated into RISC-V vector instructions without the NumPy developers needing to become RVV assembly experts.

0.4 The `NumPy` package

The **`NumPy`** package with RISC-V vector acceleration was primarily prepared by the **Institute of Software, Chinese Academy of Sciences (ISCAS)** .

Here are some details regarding the development period and availability date for the NumPy package with RISC-V vector acceleration:

- **Announcement Date**: The achievement was publicly announced on **January 5, 2026**, by the Ruyi community (a RISC-V software ecosystem community) .
- **Availability**: This means that the NumPy version with RISC-V Vector (RVV) optimizations was made available for testing and use as of **early January 2026** .
- **Development Period**: The available search results do not specify the exact date when development began. However, given the announcement in January 2026, the core development work by the team at the Institute of Software, Chinese Academy of Sciences (ISCAS) likely took place during **2025**.

0.5 Functionalities of the `NumPy` and `SciPy` libraries

0.5.1 Introduction

`NumPy` (Numerical Python) is the fundamental library for numerical computing in Python. It provides high-performance multidimensional arrays and mathematical tools to operate efficiently on large datasets. **`NumPy`** forms the foundation of the scientific Python ecosystem and **is used by libraries such as `SciPy`**, pandas, scikit-learn, and many others.

Its **core objective** is to enable **fast numerical computation** by:

- Storing data in compact, contiguous memory blocks
- Executing operations in compiled C code
- Exploiting **SIMD/vector instructions** (such as **RVV on RISC-V**, AVX on x86, etc.)
- Minimizing Python interpreter overhead

0.5.2 The `ndarray`: Core Data Structure

The central object in NumPy is the **`ndarray`** (n-dimensional array).

Key characteristics:

- Homogeneous data type (all elements share the same type)
- Fixed size
- Stored in contiguous memory (by default)
- Supports multi-dimensional indexing

Example:

```
import numpy as np
a = np.array([[1, 2], [3, 4]])
```

Important **`ndarray`** attributes:

- **`a.shape`** → dimensions
- **`a.dtype`** → data type (**`int32, float64,`** etc.)
- **`a.ndim`** → number of dimensions
- **`a.size`** → total number of elements

Supported data types include:

- Integer: **`int8, int16, int32, int64`**
- Floating point: **`float32, float64`**
- Complex numbers, Boolean, Custom structured types

Broadcasting

Broadcasting allows operations between arrays of different shapes without explicit replication.

Example:

```
a = np.array([1, 2, 3])
b = 10
c = a + b
```

Result:

```
[11, 12, 13]
```

Broadcasting rules:

- Align dimensions from the right

- Dimensions must be equal or 1
- Size 1 dimensions are automatically expanded

This eliminates the need for manual loops or memory duplication.

Indexing and Slicing

NumPy supports powerful data access mechanisms:

Basic slicing:

```
a[0:10]
a[:, 1]
```

0.5.3 Vectorized Computation

One of **NumPy's** most powerful features is **vectorization**.

Instead of writing Python loops:

```
for i in range(N):
    c[i] = a[i] + b[i]
```

Example:

```
import numpy as np
a = np.array([1,2,3])
b = np.array([4,5,6])
print(a + b)
```

Benefits:

- Eliminates Python loop overhead
- Executes in optimized C
- Uses hardware vector instructions
- Greatly improves performance

Vectorized operations include:

- Arithmetic: `+, -, *, /`
- Comparisons
- Logical operations
- Mathematical functions

0.5.4 Mathematical and Statistical Functions

NumPy provides a wide range of numerical functions.

Elementwise math:

- `np.sin(), np.cos()`
- `np.exp(), np.log()`
- `np.sqrt(), np.abs()`

Reductions:

- `np.sum(), np.mean()`
- `np.min(), np.max()`
- `np.std(), np.var()`
- `np.prod()`

0.5.5 Linear algebra:

- `np.dot() (dot product)`
- `np.matmul() or @`
- `np.linalg.inv() (matrix inverse)`
- `np.linalg.eig() (eigenvalues)`
- `np.linalg.solve()`

0.5.6 Logical Operators:

Used for logical operations on arrays.

Function	Description
`np.logical_and()`	Logical AND
`np.logical_or()`	Logical OR
`np.logical_not()`	Logical NOT
`np.logical_xor()`	Logical XOR

Many of these functions may internally use optimized **`BLAS/LAPACK`** libraries.

0.5.7 Statistical Functions

These compute statistics on arrays.

Function	Description
`np.mean()`	Mean (average)
`np.median()`	Median
`np.std()`	Standard deviation
`np.var()`	Variance
`np.sum()`	Sum of elements
`np.min()`	Minimum value
`np.max()`	Maximum value

0.5.8 Array Manipulation Functions

Function	Description
`np.reshape()`	Change array shape
`np.transpose()`	Transpose matrix
`np.concatenate()`	Join arrays
`np.split()`	Split arrays
`np.append()`	Append elements
`np.insert()`	Insert elements

0.5.9 Aggregation Functions

Function	Description
`np.argmax()`	Index of maximum value
`np.argmin()`	Index of minimum value
`np.cumsum()`	Cumulative sum
`np.cumprod()`	Cumulative product

0.6 Functionalities of the `SciPy` Library

`SciPy` library is builty upon the `NumPy` library. The functions of **`SciPy`** are widely used for image filtering, smoothing, and feature detection.

Below is a list of common filtering and image-processing functions with descriptions.

0.6.1 Gaussian Filtering Functions

Function	Description
`gaussian_filter()`	Smooths data using a Gaussian kernel; reduces noise and detail.
`gaussian_filter1d()`	Applies Gaussian smoothing along a single axis.
`gaussian_gradient_magnitude()`	Computes gradient magnitude using Gaussian derivatives.
`gaussian_laplace()`	Applies Laplacian of Gaussian filter for edge detection.

Example:

```
from scipy.ndimage import gaussian_filter
result = gaussian_filter(image, sigma=2)
```

0.6.2 Generic Filtering Functions

Function	Description
`generic_filter()`	Applies a custom function to each neighborhood of an array.
`generic_filter1d()`	Applies a custom 1D filter along a specified axis.

Example:

```
from scipy.ndimage import generic_filter
filtered = generic_filter(image, np.mean, size=3)
```

0.6.3 Uniform and Mean Filters

Function	Description
`uniform_filter()`	Smooths data using an averaging filter.
`uniform_filter1d()`	Applies a 1D uniform filter along an axis.

0.6.4 Minimum / Maximum Filters

Function	Description
`minimum_filter()`	Replaces each element with the minimum value in a neighborhood.
`maximum_filter()`	Replaces each element with the maximum value in a neighborhood.
`minimum_filter1d()`	1D minimum filter.
`maximum_filter1d()`	1D maximum filter.

0.6.5 Median Filtering

Function	Description
`median_filter()`	Replaces each value with the median of neighboring values; good for removing salt-and-pepper noise.

Example:

```
from scipy.ndimage import median_filter
output = median_filter(image, size=3)
```

0.6.6 Edge Detection Filters

Function	Description
`sobel()`	Detects edges using the Sobel operator.
`prewitt()`	Edge detection using Prewitt operator.
`laplace()`	Laplacian filter for detecting edges.

0.6.7 Other Useful Filters

Function	Description
`convolve()`	Performs convolution between an array and a kernel.
`correlate()`	Computes correlation between arrays.
`zoom()`	Resizes an array using interpolation.
`rotate()`	Rotates an image.

0.6.8 Summary

Important filtering functions commonly used with **`NumPy`** arrays include:

- `gaussian_filter(), median_filter()`
- `uniform_filter(), minimum_filter()`
- `maximum_filter()`
- `sobel(), laplace()`
- `generic_filter(), convolve()`

These functions are widely used in image processing, signal processing, computer vision, and scientific computing.

0.7 Common filtering and image-processing functions with descriptions.

0.7.1 Convolution and Correlation Functions

Used for filtering, pattern detection, and signal analysis.

Function	Description
`convolve()`	Performs convolution between two signals or arrays
`convolve2d()`	2D convolution for images
`fftconvolve()`	Fast convolution using FFT
`correlate()`	Computes cross-correlation between signals
`correlate2d()`	2D correlation for image analysis

0.7.2 Fourier Transform Functions

Used to analyze signals in the **frequency domain**.

Function	Description
`fft()`	Fast Fourier Transform
`ifft()`	Inverse FFT
`fft2()`	2-D FFT for images
`ifft2()`	Inverse 2-D FFT
`fftfreq()`	Returns frequency bins
`fftshift()`	Shifts zero frequency to center

These are heavily used in **spectral analysis and filtering**.

0.7.3 Digital Filter Design Functions

Used to create filters such as **low-pass, high-pass, band-pass**.

Function	Description
`butter()`	Butterworth filter design
`cheby1()`	Chebyshev Type I filter
`cheby2()`	Chebyshev Type II filter
`ellip()`	Elliptic filter
`bessel()`	Bessel filter design

Example:

```
from scipy.signal import butter
b, a = butter(4, 0.2)
```

Signal Filtering Functions

Function	Description
`lfilter()`	Applies a digital filter to a signal
`filtfilt()`	Zero-phase filtering
`savgol_filter()`	Savitzky–Golay smoothing filter
`medfilt()`	Median filter for noise removal
`wiener()`	Wiener filter for noise reduction

Signal Analysis Functions

Function	Description
`find_peaks()`	Detects peaks in signals
`peak_widths()`	Measures peak width

Function	Description
`spectrogram()`	Time-frequency representation
`periodogram()`	Power spectral density estimation
`welch()`	Improved power spectral density method

Image Filtering Functions

From `scipy.ndimage`.

Function	Description
`gaussian_filter()`	Gaussian smoothing
`median_filter()`	Removes salt-and-pepper noise
`uniform_filter()`	Mean filtering
`minimum_filter()`	Minimum neighborhood value
`maximum_filter()`	Maximum neighborhood value

Edge Detection Functions

Function	Description
`sobel()`	Sobel edge detector
`prewitt()`	Prewitt edge detector
`laplace()`	Laplacian operator
`gaussian_laplace()`	Laplacian of Gaussian

Image Transformation Functions

Function	Description
`rotate()`	Rotates an image
`zoom()`	Resizes image
`shift()`	Shifts image position
`affine_transform()`	Performs affine transformations

Morphological Image Operations

Function	Description
`binary_dilation()`	Expands binary regions
`binary_erosion()`	Shrinks binary regions
`binary_opening()`	Removes small objects
`binary_closing()`	Fills small holes

Summary

Important SciPy functions used in **signal and image processing** include:

Signal processing

- `convolve(), correlate()`
- `fft(), butter()`
- `lfilter(), filtfilt()`
- `find_peaks(), spectrogram()`

Image processing

- `gaussian_filter(), median_filter()`
- `sobel(), laplace()`
- `rotate(), zoom()`
- `binary_dilation(), binary_erosion()`
-

These tools are **essential** in **digital signal processing (DSP), computer vision, image analysis, and scientific computing**.

0.8 How SciPy uses NumPy

NumPy provides the array structure

SciPy functions operate on NumPy arrays (ndarray).

Example:

```
import numpy as np
from scipy import signal

x = np.array([1,2,3,4])
y = signal.convolve(x, x)
```

Here:

- np.array() creates the data structure.
- SciPy processes the data using that structure.

In short

NumPy provides the fundamental numerical operations used in scientific computing with Python.

It supplies essential capabilities such as array arithmetic, broadcasting, vectorized operations for efficient computation, basic linear algebra routines, and tools for generating random numbers.

These operations are designed to work efficiently on large numerical arrays and can benefit from modern hardware acceleration.

In particular, performance can be significantly improved when the underlying compiled libraries exploit **hardware vectorization** features such as the **vector instruction** extensions of RISC-V (RVV), which allow many data elements to be processed simultaneously in a single instruction.

This vectorized execution model is especially effective for numerical array computations typical of NumPy workloads. On top of this numerical foundation, SciPy builds advanced scientific algorithms and specialized computational methods used in areas such as signal processing, optimization, statistics, and image processing, leveraging both NumPy's array model and the performance benefits of modern vectorized processor architectures such as RISC-V (**RVV**).