Adelsbach/VSIPL
Core Lite Profile
Programming Reference Guide
DD-00014-015

Jan Adelsbach

November 3, 2025

Contents

1 About this Guide

1.1 Legal Information

1.2 Feedback and Contact

For feedback on this document, please use the following email address:
techpubs@adelsbach-research.eu

Please include the page number or a link to the page.

For general contact details, please visit https://adelsbach-research.eu/contact.

2 Overview

2.1 Introduction

The *Adelsbach / VSIPL* is an implmenetation of the digital signal processing API standard of the Object Management Group version VSIPL 1.5. It currently implments the *Core Lite* profile.

This reference manual provides a brief reference of all functionality provided in the *Adelsbach / VSIPL* library. For a more throughfully and complete reference, please refer to the Object Management Group VSIPL 1.5 standard.

2.2 Link Libraries

The following libraries are provided with the distribution. For development it is recommended to link against the more extensive error checking library, whereas for deployment use one of the performance tuned variants.

- `libavsipl_cl.a` Performance tuned library without additional error checking. May make use of processor SIMD features, please see platform details.
- `libavsipl_cl_mp.a` Performance tuned library with shared memory multithreading (OpenMP)
- `libavsipl_cl_dbg.a` Non-performance tuned library with extensive error checking.

3 General Functions

3.1 vsip_cstorage_p - Complex storage type

```
typedef enum _vsip_cmplx_mem {
  VSIP_CMPLX_INTERLEAVED,
  VSIP_CMPLX_SPLIT,
  VSIP_CMPLX_NONE
} vsip_cmplx_mem;

vsip_cmplx_mem vsip_cstorage_f(void);

/* deprecated */
vsip_cmplx_mem vsip_cstorage(void);
```

Description

These functions query the manner in which complex values are stored. This can be in interleaved (real followed by imaginary part in one vector) or split format (two separate vectors).

Return Value

- Returns one of the enumerator values.

Example

```
vsip_cmplx_mem complex_storage;

// Allocate complex storage using the preferred method
complex_storage = vsip_cstorage_f();
```

3.2 vsip_real_p - Complex Real part

```
vsip_scalar_f vsip_real_f(vsip_cscalar_f x);
```

Description

This function extracts the real part of the complex scalar x.

Parameters

- vsip_cscalar_f x: The complex scalar from which to extract the real part.

Return Value

- The real part of the complex scalar.

Example

```
vsip_cscalar_f complex_value = {1.0, 2.0};
vsip_scalar_f real_part;

real_part = vsip_real_f(complex_value);
```

3.3 vsip_imag_p - Complex Imaginary part

```
vsip_scalar_f vsip_imag_f(vsip_cscalar_f x);
```

Description

This function extracts the imaginary part of the complex scalar x.

Parameters

- vsip_cscalar_f x: The complex scalar from which to extract the imaginary part.

Return Value

- The imaginary part of the complex scalar.

Example

```
vsip_cscalar_f complex_value = {1.0, 2.0};
vsip_scalar_f imag_part;

imag_part = vsip_imag_f(complex_value);
```

3.4 vsip_cmplx_p - Create complex number

```
vsip_cscalar_f vsip_cmplx_f(vsip_scalar_f r, vsip_scalar_f i);
```

Description

This function creates a complex scalar from the real part `r` and the imaginary part `i`.

Parameters

- `vsip_scalar_f r`: The real part of the complex scalar.
- `vsip_scalar_f i`: The imaginary part of the complex scalar.

Return Value

- The created complex scalar.

Example

```
vsip_scalar_f real_part = 1.0;
vsip_scalar_f imag_part = 2.0;
vsip_cscalar_f complex_value;

complex_value = vsip_cmplx_f(real_part, imag_part);
```

3.5 vsip_CMPLX_p - Create a Complex Scalar and Store in a Pointer

```
void vsip_CMPLX_f(vsip_scalar_f a, vsip_scalar_f b, vsip_cscalar_f *r);
```

Description

This function creates a complex scalar from the real part `a` and the imaginary part `b` and stores the result in the complex scalar pointed to by `r`.

Parameters

- `vsip_scalar_f a`: The real part of the complex scalar.
- `vsip_scalar_f b`: The imaginary part of the complex scalar.
- `vsip_cscalar_f* r`: Pointer to the complex scalar where the result will be stored.

Example

```
vsip_scalar_f real_part = 1.0;
vsip_scalar_f imag_part = 2.0;
vsip_cscalar_f complex_value;

vsip_CMPLX_f(real_part, imag_part, &complex_value);
```

4 Initialization

4.1 vsip_init - Initialize

```
int vsip_init(void*);
```

Description

This function initializes the VSIPL library and must be called before any other VSIPL functions are used. It can be called multiple times without side effects.

Parameters

- `void*`: The argument is unused and should be set to `0` or `NULL`.

Return Value

- Returns `0` on success.
- Returns a non-zero value on error.

Example

```
int result;

// Initialize the VSIPL library
result = vsip_init(NULL);

if (result != 0) {
    // Handle error
}
```

4.2 vsip_finalize - Finalize

```
int vsip_finalize(void*);
```

Description

This function finalizes the VSIPL library, releasing all internal resources and memory. After calling this function, no other VSIPL functions may be called. The function can be called in a nested manner, but only the outermost call will actually free up the internal initialization memory.

Parameters

- `void*`: The argument is unused and should be set to `0` or `NULL`.

Return Value

- Returns `0` on success.
- Returns a non-zero value on error.

Example

```
int result;

// Finalize the VSIPL library
result = vsip_finalize(NULL);

if (result != 0) {
    // Handle error
}
```

5 Block Support Functions

5.1 vsip_dblockcreate_p - Create a block

```
typedef enum _vsip_memory_hint {
  VSIP_MEM_NONE          = 0,
  VSIP_MEM_RDONLY        = 1,
  VSIP_MEM_CONST         = 2,
  VSIP_MEM_SHARED        = 3,
  VSIP_MEM_SHARED_RDONLY = 4,
  VSIP_MEM_SHARED_CONST  = 5
} vsip_memory_hint;

vsip_block_f*  vsip_blockcreate_f(vsip_length n, vsip_memory_hint h);
vsip_block_i*  vsip_blockcreate_i(vsip_length n, vsip_memory_hint h);
vsip_cblock_f* vsip_cblockcreate_f(vsip_length n, vsip_memory_hint h);
```

Description

These functions create a block of data of the specified type with $n > 0$ elements. The memory hint `h` describes how this data is intended to be used, such as read-only, constant, or shared memory.

Parameters

- `vsip_length n`: The number of elements in the block. Must be greater than 0.
- `vsip_memory_hint h`: Memory hint for the block, indicating properties such as read-only, constant, or shared memory.
  - `VSIP_MEM_NONE` - No memory hint
  - `VSIP_MEM_RDONLY` - The memory is to be used read-only
  - `VSIP_MEM_CONST` - The memory will hold constants
  - `VSIP_MEM_SHARED` - The memory will be shared
  - `VSIP_MEM_SHARED_RDONLY` - The memory will be shared and is read-only
  - `VSIP_MEM_SHARED_CONST` - The memory will be shared and will hold constants

Return Value

- On success, a pointer to the newly created block object is returned.
- On error, `NULL` is returned.

Error Handling

If an error occurs, the function returns `NULL`.

Example

```
vsip_length length = 10;
vsip_memory_hint hint = VSIP_MEM_NONE;
vsip_block_f *float_block;

// Create a float block
float_block = vsip_blockcreate_f(length, hint);

if (float_block == NULL) {
    // Handle error
}

vsip_block_i *int_block;

// Create an integer block
```

```
int_block = vsip_blockcreate_i(length, hint);

if (int_block == NULL) {
    // Handle error
}

vsip_cblock_f *complex_block;

// Create a complex float block
complex_block = vsip_cblockcreate_f(length, hint);

if (complex_block == NULL) {
    // Handle error
}
```

5.2 vsip_blockbind_p - Create a block using existing data

```
typedef enum _vsip_memory_hint {
  VSIP_MEM_NONE          = 0,
  VSIP_MEM_RDONLY        = 1,
  VSIP_MEM_CONST         = 2,
  VSIP_MEM_SHARED        = 3,
  VSIP_MEM_SHARED_RDONLY = 4,
  VSIP_MEM_SHARED_CONST  = 5
} vsip_memory_hint;

vsip_block_f*  vsip_blockbind_f(vsip_scalar_f *p, vsip_length n, vsip_memory_hint h);
vsip_block_i*  vsip_blockbind_i(vsip_scalar_i *p, vsip_length n, vsip_memory_hint h);
```

Description

These functions create a new data block using an existing data array p with $n > 0$ elements and a given memory hint h. The block must be admitted before it can be used.

Parameters

- vsip_scalar_p *p: Pointer to the existing data array.
- vsip_length n: The number of elements in the data array. Must be greater than 0.
- vsip_memory_hint h: Memory hint for the block, indicating properties such as read-only, constant, or shared memory.
  - VSIP_MEM_NONE - No memory hint
  - VSIP_MEM_RDONLY - The memory is to be used read-only
  - VSIP_MEM_CONST - The memory will hold constants
  - VSIP_MEM_SHARED - The memory will be shared
  - VSIP_MEM_SHARED_RDONLY - The memory will be shared and is read-only
  - VSIP_MEM_SHARED_CONST - The memory will be shared and will hold constants

Return Value

- On success, a pointer to the newly created block object is returned.
- On error, NULL is returned.

Error Handling

If an error occurs, the function returns NULL.

Example

```
vsip_scalar_f float_data[10]; // Example float data array
vsip_length length = 10;
vsip_memory_hint hint = VSIP_MEM_NONE;
vsip_block_f *float_block;

// Create a float block
float_block = vsip_blockbind_f(float_data, length, hint);

if (float_block == NULL) {
    // Handle error
}

// Admit the block before using it
int result = vsip_blockadmit_f(float_block, VSIP_TRUE);
```

```
if (result != 0) {
    // Handle error
}
```

5.3 vsip_cblockbind_p - Create a block using existing data (complex)

```
typedef enum _vsip_memory_hint {
  VSIP_MEM_NONE          = 0,
  VSIP_MEM_RDONLY        = 1,
  VSIP_MEM_CONST         = 2,
  VSIP_MEM_SHARED        = 3,
  VSIP_MEM_SHARED_RDONLY = 4,
  VSIP_MEM_SHARED_CONST  = 5
} vsip_memory_hint;

vsip_cblock_f* vsip_cblockbind_f(vsip_scalar_f *r, vsip_scalar_f *i, vsip_length n, vsip_memory_hint h);
```

Description

This function creates a new complex data block using existing data, which can be either interleaved complex numbers or split real and imaginary data arrays. If the imaginary data array `i` is `NULL`, it is assumed that `r` is interleaved and contains $2n > 0$ elements. If the imaginary data array is provided, it is assumed that each of the `r` and `i` arrays contains $n > 0$ elements.

The block must be admitted before it can be used.

Parameters

- `vsip_scalar_p *r`: Pointer to the real part array or the interleaved array.
- `vsip_scalar_p *i`: Pointer to the imaginary part array. If `NULL`, `r` is assumed to be interleaved.
- `vsip_length n`: The number of complex elements. Must be greater than 0.
- `vsip_memory_hint h`: Memory hint for the block, indicating properties such as read-only, constant, or shared memory.
  - `VSIP_MEM_NONE` - No memory hint
  - `VSIP_MEM_RDONLY` - The memory is to be used read-only
  - `VSIP_MEM_CONST` - The memory will hold constants
  - `VSIP_MEM_SHARED` - The memory will be shared
  - `VSIP_MEM_SHARED_RDONLY` - The memory will be shared and is read-only
  - `VSIP_MEM_SHARED_CONST` - The memory will be shared and will hold constants

Return Value

- On success, a pointer to the newly created complex block object is returned.
- On error, `NULL` is returned.

Error Handling

If an error occurs, the function returns `NULL`.

Example

```
vsip_scalar_f real_data[10]; // Example data array
vsip_scalar_f imag_data[10]; // Example imaginary data array
vsip_length length = 10;
vsip_memory_hint hint = VSIP_MEM_NONE;
vsip_cblock_f *block;

// Create a block with split real and imaginary data
block = vsip_cblockbind_f(real_data, imag_data, length, hint);

if (block == NULL) {
```

```
    // Handle error
}

// Admit the block before using it
int result = vsip_cblockadmit_f(block, VSIP_TRUE);
if (result != 0) {
    // Handle error
}

// Create a block with interleaved data
vsip_scalar_f interleaved_data[20]; // Example interleaved data array
block = vsip_cblockbind_f(interleaved_data, NULL, length, hint);

if (block == NULL) {
    // Handle error
}

// Admit the block before using it
result = vsip_cblockadmit_f(block, VSIP_TRUE);
if (result != 0) {
    // Handle error
}
```

5.4 vsip_blockrebind_p - Rebind existing block

```
vsip_scalar_f* vsip_blockrebind_f(vsip_block_f *p, vsip_scalar_f *d);
vsip_scalar_i* vsip_blockrebind_i(vsip_block_i *p, vsip_scalar_i *d);
```

Description

These functions rebind an existing block p to a new user data array d. The new data array must have the same size as the originally bound data array.

The block must be in the released state to be rebound. After rebinding, the block must be admitted before it can be used.

A pointer to the previously bound user data array is returned. If an error occurs, NULL is returned.

Parameters

- vsip_block_p *p: Pointer to the block to be rebound.
- vsip_scalar_p *d: Pointer to the new user data array.

Return Value

- On success, a pointer to the previously bound user data array is returned.
- On error, NULL is returned.

Error Handling

If an error occurs, the function returns NULL.

Example

```
vsip_block_f *float_block;
vsip_scalar_f *new_data;
vsip_scalar_f *old_data;

// Assuming float_block has been properly initialized and is in the released state
old_data = vsip_blockrebind_f(float_block, new_data);

if (old_data == NULL) {
    // Handle error
}

// Admit the block before using it
int result = vsip_blockadmit_f(float_block, VSIP_TRUE);
if (result != 0) {
    // Handle error
}

vsip_block_i *int_block;
vsip_scalar_i *new_int_data;
vsip_scalar_i *old_int_data;

// Assuming int_block has been properly initialized and is in the released state
old_int_data = vsip_blockrebind_i(int_block, new_int_data);

if (old_int_data == NULL) {
    // Handle error
}

// Admit the block before using it
result = vsip_blockadmit_i(int_block, VSIP_TRUE);
if (result != 0) {
```

```
    // Handle error
}
```

5.5 vsip_cblockrebind_p - Rebind existing block (complex)

```
void vsip_cblockrebind_f(vsip_cblock_f *p, vsip_scalar_f *r, vsip_scalar_f *i, vsip_scalar_f **rr, vsip_sc
```

Description

This function rebinds an existing complex block p to new real and imaginary part arrays r and i. If i is NULL, it is implied that r points to an interleaved array. The new array(s) must have the same size as the originally bound data array(s).

The block must be in the released state to be rebound. After rebinding, the block must be admitted before it can be used.

The previously bound data array(s) are stored in rr and ir. If the originally bound data is interleaved, ir will be set to NULL. On error, both rr and ir will be set to NULL.

Parameters

- vsip_cblock_p *p: Pointer to the complex block to be rebound.
- vsip_scalar_p *r: Pointer to the new real part array.
- vsip_scalar_p *i: Pointer to the new imaginary part array. If NULL, r is assumed to point to an interleaved array.
- vsip_scalar_p **rr: Pointer to store the previously bound real part array.
- vsip_scalar_p **ir: Pointer to store the previously bound imaginary part array. Will be NULL if the original data is interleaved.

Error Handling

On error, both rr and ir are set to NULL.

Example

```
vsip_cblock_f *block;
vsip_scalar_f *new_real_part;
vsip_scalar_f *new_imag_part = NULL; // For interleaved data
vsip_scalar_f *old_real_part;
vsip_scalar_f *old_imag_part;

// Assuming block has been properly initialized and is in the released state
vsip_cblockrebind_f(block, new_real_part, new_imag_part, &old_real_part, &old_imag_part);

if (old_real_part == NULL) {
    // Handle error
}
```

5.6 vsip_dblockadmit_p - Admit block data

```
int vsip_blockadmit_f(vsip_block_f *b, vsip_scalar_bl s);
int vsip_blockadmit_i(vsip_block_i *b, vsip_scalar_bl s);
int vsip_cblockadmit_f(vsip_cblock_f*, vsip_scalar_bl s);
```

Description

These functions admit user data to the given block b. After calling this function, the user array may no longer be manually manipulated outside of VSIPL routines. The boolean flag s indicates whether the user data should be consistent with the block data. In most cases, s should be set to VSIP_TRUE.

Parameters

- vsip_dblock_p *b: Pointer to the block to which user data is to be admitted.
- vsip_scalar_bl s: Boolean flag indicating whether the user data should be consistent with the block data.

Return Value

- Returns 0 on success.
- Returns a non-zero value on error.

Error Handling

If an error occurs, the function returns a non-zero value.

Example

```
vsip_block_f *float_block;
vsip_scalar_bl consistent = VSIP_TRUE;
int result;

// Assuming float_block has been properly initialized
result = vsip_blockadmit_f(float_block, consistent);

if (result != 0) {
    // Handle error
}
```

5.7 vsip_blockfind_p - Get user data

```
vsip_scalar_f* vsip_blockfind_f(const vsip_block_f *p);
vsip_scalar_i* vsip_blockfind_i(const vsip_block_i *p);
```

Description

These functions return a pointer to the user data array bound to the given block p. The block must have been bound previously and must be in the released state before calling these functions.

Parameters

- const vsip_block_p *p: Pointer to the block whose user data array is to be queried.

Return Value

- On success, a pointer to the user data array is returned.
- On error, NULL is returned.

Error Handling

If an error occurs, the function returns NULL.

Example

```
vsip_block_f *float_block;
vsip_scalar_f *float_data;

// Assuming float_block has been properly initialized, bound, and released
float_data = vsip_blockfind_f(float_block);

if (float_data == NULL) {
    // Handle error
}
```

5.8 vsip_cblockfind_p - Get user data (complex)

```
void vsip_cblockfind_f(const vsip_cblock_f *p, vsip_scalar_f **rr, vsip_scalar_f **ri);
```

Description

This function queries the user data array(s) of the given complex block p. Depending on the data format, the function sets the pointers rr and ri accordingly:

- If the data is in interleaved format, only rr will be set, and ri will be set to NULL.
- If the data is in split format, both rr and ri will be set to point to the real and imaginary parts, respectively.

The block must have been bound previously and must be in the released state before calling this function.

Parameters

- const vsip_cblock_p *p: Pointer to the complex block whose user data arrays are to be queried.
- vsip_scalar_p **rr: Pointer to the real part of the user array.
- vsip_scalar_p **ri: Pointer to the imaginary part of the user array.

Error Handling

On error, both rr and ri are set to NULL.

Example

```
vsip_cblock_f *block;
vsip_scalar_f *real_part;
vsip_scalar_f *imag_part;

// Assuming block has been properly initialized, bound, and released
vsip_cblockfind_f(block, &real_part, &imag_part);

if (real_part == NULL) {
    // Handle error
}
```

5.9 vsip_blockrelease_p - Release a block

```
vsip_scalar_f* vsip_blockrelease_f(vsip_block_f *b, vsip_scalar_bl u);
vsip_scalar_i* vsip_blockrelease_i(vsip_block_i *b,vsip_scalar_bl u);
```

Description

These functions release the user arrays in a block b and return a pointer to the user data array. The flag u determines whether the data must be maintained during the state change. The block must have been bound previously.

Parameters

- vsip_block_p *: Pointer to the block to be released.
- vsip_scalar_bl u: Flag indicating whether the data should be maintained.

Return Value

- On success, a pointer to the user data array is returned.
- On error, NULL is returned.

Error Handling

If an error occurs, the function returns NULL.

Example

```
vsip_block_f *float_block;
vsip_scalar_bl maintain_data = VSIP_TRUE;
vsip_scalar_f *float_data;

// Assuming float_block has been properly initialized and bound
float_data = vsip_blockrelease_f(float_block, maintain_data);

if (float_data == NULL) {
    // Handle error
}
```

5.10 vsip_cblockrelease_p - Release a block (complex)

```
void vsip_cblockrelease_f(vsip_cblock_f *b, vsip_scalar_bl u, vsip_scalar_f **rr, vsip_scalar_f **ri);
```

Description

This function releases the complex block `b` and queries the user array(s). Depending on the data format, the function sets the pointers `rr` and `ri` accordingly:

- If the data is in interleaved format, only `rr` will be set, and `ri` will be set to `NULL`.
- If the data is in split format, both `rr` and `ri` will be set to point to the real and imaginary parts, respectively.

The flag `u` determines whether the data must be maintained during the state change. The block must have been bound previously and must be in the released state before calling this function.

Parameters

- `vsip_cblock_p *b`: Pointer to the complex block to be released.
- `vsip_scalar_bl u`: Flag indicating whether the data should be maintained.
- `vsip_scalar_p **rr`: Pointer to the real part of the user array.
- `vsip_scalar_p **ri`: Pointer to the imaginary part of the user array.

Error Handling

On error, both `rr` and `ri` are set to `NULL`.

Example

```
vsip_cblock_f *block;
vsip_scalar_bl maintain_data = VSIP_TRUE;
vsip_scalar_f *real_part;
vsip_scalar_f *imag_part;

// Assuming block has been properly initialized and bound
vsip_cblockrelease_f(block, maintain_data, &real_part, &imag_part);

if (real_part == NULL) {
    // Handle error
}
```

5.11 vsip_dblockdestroy_p - Destroy a block

```
void vsip_blockdestroy_f(vsip_block_f *b);
void vsip_blockdestroy_i(vsip_block_i *b);
void vsip_cblockdestroy_f(vsip_cblock_f *b);
```

Description

These functions destroy the block specified by the pointer `b`. Destroying a block involves deallocating the memory associated with it and performing any necessary cleanup operations. After calling one of these functions, the block pointer `b` becomes invalid and should not be used further.

Parameters

- `vsip_dblock_p *b` Pointer to a floating-point or integer block to be destroyed.

Return Value

These functions do not return a value.

Example

```
vsip_block_f *block = vsip_blockcreate_f(10, VSIP_MEM_NONE);
// Use the block...
vsip_blockdestroy_f(block); // Destroy the block when done
```

Notes

Ensure that the block pointer is valid and has been properly initialized before calling these functions. Attempting to destroy an already destroyed block or an invalid pointer may result in undefined behavior.

6 View Support Functions

6.1 vsip_dvcreate_p - Create a Vector View

```
typedef enum _vsip_memory_hint {
  VSIP_MEM_NONE          = 0,
  VSIP_MEM_RDONLY        = 1,
  VSIP_MEM_CONST         = 2,
  VSIP_MEM_SHARED        = 3,
  VSIP_MEM_SHARED_RDONLY = 4,
  VSIP_MEM_SHARED_CONST  = 5
} vsip_memory_hint;

vsip_vview_f*  vsip_vcreate_f(vsip_length n, vsip_memory_hint h);
vsip_cvview_f* vsip_cvcreate_f(vsip_length n, vsip_memory_hint h);
```

Description

This function creates a vector view of the specified length `n` with a given memory hint `h`. The memory hint describes how the data is intended to be used, such as read-only, constant, or shared memory.

Parameters

- `vsip_length n`: The number of elements in the vector view. Must be greater than 0.
- `vsip_memory_hint h`: Memory hint for the vector view, indicating properties such as read-only, constant, or shared memory.

Return Value

- On success, a pointer to the newly created vector view object is returned.
- On error, `NULL` is returned.

Error Handling

If an error occurs, the function returns `NULL`.

Example

```
vsip_length length = 10;
vsip_memory_hint hint = VSIP_MEM_NONE;
vsip_vview_f *vector_view;

// Create a vector view
vector_view = vsip_vcreate_f(length, hint);

if (vector_view == NULL) {
    // Handle error
}
```

6.2 vsip_dvbind_p - Bind a Vector View to a Data Block

```
vsip_vview_f* vsip_vbind_f(const vsip_block_f* b, vsip_offset o, vsip_stride s, vsip_length n);
vsip_vview_i* vsip_vbind_i(const vsip_block_i* b, vsip_offset o, vsip_stride s,  vsip_length n);
vsip_vview_i* vsip_cvbind_f(const vsip_cblock_f* b, vsip_offset o, vsip_stride s, vsip_length n);
```

Description

This function binds a vector view to an existing data block `b` with a specified offset `o`, stride `s`, and length `n`. The vector view provides a view into the data block starting from the offset and stepping by the stride for the specified length.

Parameters

- `const vsip_dblock_p* b`: Pointer to the data block to which the vector view will be bound.
- `vsip_offset o`: Offset within the data block where the vector view starts.
- `vsip_stride s`: Stride between elements in the vector view.
- `vsip_length n`: The number of elements in the vector view.

Return Value

- On success, a pointer to the newly created vector view object is returned.
- On error, `NULL` is returned.

Error Handling

If an error occurs, the function returns `NULL`.

Example

```
vsip_block_f *data_block;
vsip_offset offset = 0;
vsip_stride stride = 1;
vsip_length length = 10;
vsip_vview_f *vector_view;

// Assuming data_block has been properly initialized
vector_view = vsip_vbind_f(data_block, offset, stride, length);

if (vector_view == NULL) {
    // Handle error
}

// The vector view is now bound to the data block
```

6.3 vsip_dvsubview_p - Create a Subview of a Vector View

```
vsip_vview_f* vsip_vsubview_f(const vsip_vview_f* v, vsip_index j, vsip_length n);
vsip_cvview_f* vsip_cvsubview_f(const vsip_cvview_f* v, vsip_index j, vsip_length n);
```

Description

This function creates a subview of an existing vector view `v`, starting from the index `j` and extending for `n` elements. The subview provides a view into a subset of the original vector view.

Parameters

- `const vsip_dvview_p* v`: Pointer to the original vector view from which the subview will be created.
- `vsip_index j`: Starting index within the original vector view for the subview.
- `vsip_length n`: The number of elements in the subview.

Return Value

- On success, a pointer to the newly created subview object is returned.
- On error, `NULL` is returned.

Error Handling

If an error occurs, the function returns `NULL`.

Example

```
vsip_vview_f *original_view;
vsip_index start_index = 5;
vsip_length subview_length = 10;
vsip_vview_f *subview;

// Assuming original_view has been properly initialized
subview = vsip_vsubview_f(original_view, start_index, subview_length);

if (subview == NULL) {
    // Handle error
}

// The subview is now a view into a subset of the original vector view
```

6.4 vsip_dvgetblock_p - Get the Data Block of a Vector View

```
vsip_block_f* vsip_vgetblock_f(const vsip_vview_f* v);
vsip_cblock_f* vsip_cvgetblock_f(const vsip_cvview_f* v);
```

Description

This function returns the data block associated with the vector view v.

Parameters

- const vsip_dvview_p* v: Pointer to the vector view.

Return Value

- On success, a pointer to the data block is returned.
- On error, NULL is returned.

Example

```
vsip_vview_f *vector_view;
vsip_block_f *data_block;

// Assuming vector_view has been properly initialized
data_block = vsip_vgetblock_f(vector_view);

if (data_block == NULL) {
    // Handle error
}
```

6.5 vsip_dvgetoffset_p - Get the Offset of a Vector View

```
vsip_offset vsip_vgetoffset_f(const vsip_vview_f* v);
vsip_offset vsip_cvgetoffset_f(const vsip_cvview_f* v);
```

Description

This function returns the offset within the data block where the vector view v starts.

Parameters

- const vsip_dvview_p* v: Pointer to the vector view.

Return Value

- The offset within the data block.

Example

```
vsip_vview_f *vector_view;
vsip_offset offset;

// Assuming vector_view has been properly initialized
offset = vsip_vgetoffset_f(vector_view);
```

6.6 vsip_dvputoffset_p - Set the Offset of a Vector View

```
vsip_vview_f* vsip_vputoffset_f(vsip_vview_f* v, vsip_offset o);
vsip_cvview_f* vsip_cvputoffset_f(vsip_cvview_f* v, vsip_offset o);
```

Description

This function sets the offset within the data block for the vector view `v` to the specified value `o`.

Parameters

- `vsip_dvview_p* v`: Pointer to the vector view.
- `vsip_offset o`: The new offset within the data block.

Return Value

- On success, a pointer to the modified vector view is returned.
- On error, `NULL` is returned.

Example

```
vsip_vview_f *vector_view;
vsip_offset new_offset = 5;

// Assuming vector_view has been properly initialized
vector_view = vsip_vputoffset_f(vector_view, new_offset);

if (vector_view == NULL) {
    // Handle error
}
```

6.7 vsip_dvgetstride_p - Get the Stride of a Vector View

```
vsip_stride vsip_vgetstride_f(const vsip_vview_f* v);
vsip_stride vsip_cvgetstride_f(const vsip_cvview_f* v);
```

Description

This function returns the stride between elements in the vector view v.

Parameters

- const vsip_dvview_p* v: Pointer to the vector view.

Return Value

- The stride between elements in the vector view.

Example

```
vsip_vview_f *vector_view;
vsip_stride stride;

// Assuming vector_view has been properly initialized
stride = vsip_vgetstride_f(vector_view);
```

6.8 vsip_dvputstride_p - Set the Stride of a Vector View

```
vsip_vview_f* vsip_vputstride_f(vsip_vview_f* v, vsip_stride s);
vsip_cvview_f* vsip_cvputstride_f(vsip_cvview_f* v, vsip_stride s);
```

Description

This function sets the stride between elements in the vector view v to the specified value s.

Parameters

- vsip_dvview_p* v: Pointer to the vector view.
- vsip_stride s: The new stride between elements.

Return Value

- On success, a pointer to the modified vector view is returned.
- On error, NULL is returned.

Example

```
vsip_vview_f *vector_view;
vsip_stride new_stride = 2;

// Assuming vector_view has been properly initialized
vector_view = vsip_vputstride_f(vector_view, new_stride);

if (vector_view == NULL) {
    // Handle error
}
```

6.9 vsip_dvgetlength_p - Get the Length of a Vector View

```
vsip_length vsip_vgetlength_f(const vsip_vview_f* v);
vsip_length vsip_cvgetlength_f(const vsip_cvview_f* v);
```

Description

This function returns the length of the vector view v.

Parameters

- const vsip_dvview_p* v: Pointer to the vector view.

Return Value

- The length of the vector view.

Example

```
vsip_vview_f *vector_view;
vsip_length length;

// Assuming vector_view has been properly initialized
length = vsip_vgetlength_f(vector_view);
```

6.10 vsip_dvputlength_p - Set the Length of a Vector View

```
vsip_vview_f* vsip_vputlength_f(vsip_vview_f* v, vsip_length n);
vsip_cvview_f* vsip_cvputlength_f(vsip_cvview_f* v, vsip_length n);
```

Description

This function sets the length of the vector view `v` to the specified value `n`.

Parameters

- `vsip_dvview_p* v`: Pointer to the vector view.
- `vsip_length n`: The new length of the vector view.

Return Value

- On success, a pointer to the modified vector view is returned.
- On error, `NULL` is returned.

Example

```
vsip_vview_f *vector_view;
vsip_length new_length = 15;

// Assuming vector_view has been properly initialized
vector_view = vsip_vputlength_f(vector_view, new_length);

if (vector_view == NULL) {
    // Handle error
}
```

6.11 vsip_dvgetattrib_p - Get the Attributes of a Vector View

```
void vsip_vgetattrib_f(const vsip_vview_f* v, vsip_vattr_f *a);
void vsip_vgetattrib_i(const vsip_vview_i* v, vsip_vattr_i *a);
void vsip_cvgetattrib_f(const vsip_cvview_f* v, vsip_cvattr_f *a);
```

Description

This function retrieves the attributes of the vector view v and stores them in the structure pointed to by a.

Parameters

- const vsip_dvview_p* v: Pointer to the vector view.
- vsip_dvattr_p *a: Pointer to a structure where the attributes will be stored.

Example

```
vsip_vview_f *vector_view;
vsip_vattr_f attributes;

// Assuming vector_view has been properly initialized
vsip_vgetattrib_f(vector_view, &attributes);

// The attributes of the vector view are now stored in 'attributes'
```

6.12 vsip_dvputattrib_p - Set the Attributes of a Vector View

```
vsip_vview_f* vsip_vputattrib_f(vsip_vview_f* v, const vsip_vattr_f *a);
vsip_vview_i* vsip_vputattrib_i(vsip_vview_i* v, const vsip_vattr_i *a);
vsip_cvview_f* vsip_cvputattrib_f(vsip_cvview_f* v, const vsip_cvattr_f *a);
```

Description

This function sets the attributes of the vector view v to the values specified in the structure pointed to by a.

Parameters

- vsip_dvview_p* v: Pointer to the vector view.
- const vsip_dvattr_p *a: Pointer to a structure containing the new attributes.

Return Value

- On success, a pointer to the modified vector view is returned.
- On error, NULL is returned.

Example

```
vsip_vview_f *vector_view;
vsip_vattr_f new_attributes;

// Assuming vector_view has been properly initialized and new_attributes is set
vector_view = vsip_vputattrib_f(vector_view, &new_attributes);

if (vector_view == NULL) {
    // Handle error
}
```

6.13 vsip_dvget_p - Get an Element from a Vector View

```
vsip_scalar_f  vsip_vget_f(const vsip_vview_f* v, vsip_index j);
vsip_cscalar_f vsip_cvget_f(const vsip_cvview_f* v, vsip_index j);
```

Description

This function retrieves the element at the specified index `j` from the vector view `v`.

Parameters

- `const vsip_dvview_p* v`: Pointer to the vector view.
- `vsip_index j`: Index of the element to retrieve.

Return Value

- The value of the element at the specified index.

Example

```
vsip_vview_f *vector_view;
vsip_index index = 3;
vsip_scalar_f value;

// Assuming vector_view has been properly initialized
value = vsip_vget_f(vector_view, index);
```

6.14 vsip_dvput_p - Set an Element in a Vector View

```
void vsip_vput_f(const vsip_vview_f* v, vsip_index j, vsip_scalar_f x);
void vsip_cvput_f(const vsip_cvview_f* v, vsip_index j, vsip_cscalar_f x);
```

Description

This function sets the element at the specified index `j` in the vector view `v` to the value `x`.

Parameters

- `const vsip_dvview_p* v`: Pointer to the vector view.
- `vsip_index j`: Index of the element to set.
- `vsip_dscalar_p x`: The new value for the element.

Example

```
vsip_vview_f *vector_view;
vsip_index index = 3;
vsip_scalar_f new_value = 10.0;

// Assuming vector_view has been properly initialized
vsip_vput_f(vector_view, index, new_value);
```

6.15 vsip_vrealview_p - Get the Real Part View of a Complex Vector View

```
vsip_vview_f* vsip_vrealview_f(const vsip_cvview_f* v);
```

Description

This function returns a view of the real part of the complex vector view `v`.

Parameters

- `const vsip_cvview_p* v`: Pointer to the complex vector view.

Return Value

- On success, a pointer to the real part view of the complex vector view is returned.
- On error, `NULL` is returned.

Example

```
vsip_cvview_f *complex_vector_view;
vsip_vview_f *real_part_view;

// Assuming complex_vector_view has been properly initialized
real_part_view = vsip_vrealview_f(complex_vector_view);

if (real_part_view == NULL) {
    // Handle error
}
```

6.16 vsip_vimagview_p - Get the Imaginary Part View of a Complex Vector View

```
vsip_vview_f* vsip_vimagview_f(const vsip_cvview_f* v);
```

Description

This function returns a view of the imaginary part of the complex vector view v.

Parameters

- const vsip_cvview_p* v: Pointer to the complex vector view.

Return Value

- On success, a pointer to the imaginary part view of the complex vector view is returned.
- On error, NULL is returned.

Example

```
vsip_cvview_f *complex_vector_view;
vsip_vview_f *imaginary_part_view;

// Assuming complex_vector_view has been properly initialized
imaginary_part_view = vsip_vimagview_f(complex_vector_view);

if (imaginary_part_view == NULL) {
    // Handle error
}
```

6.17 vsip_dvdestroy_p - Destroy a Vector View

```
vsip_block_f*  vsip_vdestroy_f(vsip_vview_f* v);
vsip_block_i*  vsip_vdestroy_i(vsip_vview_i* v);
vsip_cblock_f* vsip_cvdestroy_f(vsip_cvview_f* v);
```

Description

This function destroys a vector view v and returns a pointer to the underlying data block. After calling this function, the vector view is no longer valid, but the data block can still be used.

Parameters

- vsip_dvview_p* v: Pointer to the vector view to be destroyed.

Return Value

- On success, a pointer to the underlying data block is returned.
- On error, NULL is returned.

Error Handling

If an error occurs, the function returns NULL.

Example

```
vsip_vview_f *vector_view;
vsip_block_f *data_block;

// Assuming vector_view has been properly initialized
data_block = vsip_vdestroy_f(vector_view);

if (data_block == NULL) {
    // Handle error
}

// The data block can still be used after the vector view is destroyed
```

6.18 vsip_dvalldestroy_p - Destroy a Vector View and Its Data Block

```
void vsip_valldestroy_f(vsip_vview_f *v);
void vsip_cvalldestroy_f(vsip_cvview_f* v);
```

Description

This function destroys a vector view v and its underlying data block. After calling this function, both the vector view and the data block are no longer valid.

Parameters

- vsip_dvview_p *v: Pointer to the vector view to be destroyed along with its data block.

Example

```
vsip_vview_f *vector_view;

// Assuming vector_view has been properly initialized
vsip_valldestroy_f(vector_view);

// Both the vector view and its data block are now invalid
```

7 Copy Functions

7.1 vsip_dvcopy_p_p - Copy Vector Views

```
void vsip_vcopy_f_f(const vsip_vview_f* a, const vsip_vview_f* r);
void vsip_vcopy_i_i(const vsip_vview_i* a, const vsip_vview_i* r);
void vsip_vcopy_i_f(const vsip_vview_i* a, const vsip_vview_f* r);
void vsip_vcopy_f_i(const vsip_vview_f* a, const vsip_vview_i* r);
void vsip_cvcopy_f_f(const vsip_cvview_f* a, const vsip_cvview_f* r);
```

Description

These functions copy the contents of one vector view to another. The source and destination vector views can be of different types (float or integer), and the functions handle the necessary type conversions.

Parameters

- `const vsip_dvview_p* a`: Pointer to the source vector view.
- `const vsip_dvview_p* r`: Pointer to the destination vector view.

Functions

- `vsip_vcopy_f_f`: Copies from a float vector view to another float vector view.
- `vsip_vcopy_i_i`: Copies from an integer vector view to another integer vector view.
- `vsip_vcopy_i_f`: Copies from an integer vector view to a float vector view.
- `vsip_vcopy_f_i`: Copies from a float vector view to an integer vector view.
- `vsip_cvcopy_f_f`: Copies from a complex float vector view to another complex float vector view.

Example

```
vsip_vview_f *src_float_view;
vsip_vview_f *dst_float_view;

// Assuming all views have been properly initialized

// Copy from float vector view to float vector view
vsip_vcopy_f_f(src_float_view, dst_float_view);
```

8 Vector General

8.1 vsip_dvdot_p - Compute the Dot Product of Two Vector Views

```
vsip_scalar_f  vsip_vdot_f(const vsip_vview_f* a, const vsip_vview_f* b);
vsip_cscalar_f vsip_cvdot_f(const vsip_cvview_f* a, const vsip_cvview_f* b);
```

Description

This function computes the dot product of the vector views `a` and `b` and returns it. The dot product is computed as the sum of the element-wise products of the corresponding elements in the two vectors.

$$\sum_{i}^{n} a_i b_i$$

Parameters

- `const vsip_dvview_p* a`: Pointer to the first vector view.
- `const vsip_dvview_p* b`: Pointer to the second vector view.

Return Value

- The dot product of the two vector views.

Example

```
vsip_cvview_f *complex_vector_a;
vsip_cvview_f *complex_vector_b;
vsip_cscalar_f dot_product;

// Assuming complex_vector_a and complex_vector_b have been properly initialized
dot_product = vsip_cvdot_f(complex_vector_a, complex_vector_b);
```

8.2 vsip_dvmul_p - Element-wise Multiplication of Two Vector Views

```
void vsip_vmul_f(const vsip_vview_f* a, const vsip_vview_f* b, const vsip_vview_f* r);
void vsip_cvmul_f(const vsip_cvview_f* a, const vsip_cvview_f* b, const vsip_cvview_f* r);
```

Description

This function performs element-wise multiplication of the vector views a and b and stores the result in the vector view r.

Parameters

- const vsip_dvview_p* a: Pointer to the first source vector view.
- const vsip_dvview_p* b: Pointer to the second source vector view.
- const vsip_dvview_p* r: Pointer to the destination vector view.

Example

```
vsip_vview_f *vector_view_a;
vsip_vview_f *vector_view_b;
vsip_vview_f *result_vector_view;

// Assuming vector_view_a, vector_view_b, and result_vector_view have been properly initialized
vsip_vmul_f(vector_view_a, vector_view_b, result_vector_view);
```

8.3 vsip_vdiv_p - Element-wise Division of Two Vector Views

```
void vsip_vdiv_f(const vsip_vview_f* a, const vsip_vview_f* b, const vsip_vview_f* r);
```

Description

This function performs element-wise division of the vector view `a` by the vector view `b` and stores the result in the vector view `r`.

Parameters

- `const vsip_vview_p* a`: Pointer to the numerator vector view.
- `const vsip_vview_p* b`: Pointer to the denominator vector view.
- `const vsip_vview_p* r`: Pointer to the destination vector view.

Example

```
vsip_vview_f *vector_view_a;
vsip_vview_f *vector_view_b;
vsip_vview_f *result_vector_view;

// Assuming vector_view_a, vector_view_b, and result_vector_view have been properly initialized
vsip_vdiv_f(vector_view_a, vector_view_b, result_vector_view);
```

8.4 vsip_dvadd_p - Element-wise Addition of Two Vector Views

```
void vsip_vadd_f(const vsip_vview_f* a, const vsip_vview_f* b, const vsip_vview_f* r);
void vsip_cvadd_f(const vsip_cvview_f* a, const vsip_cvview_f* b, const vsip_cvview_f* r);
```

Description

This function performs element-wise addition of the vector views a and b and stores the result in the vector view r.

Parameters

- const vsip_dvview_p* a: Pointer to the first source vector view.
- const vsip_dvview_p* b: Pointer to the second source vector view.
- const vsip_dvview_p* r: Pointer to the destination vector view.

Example

```
vsip_vview_f *vector_view_a;
vsip_vview_f *vector_view_b;
vsip_vview_f *result_vector_view;

// Assuming vector_view_a, vector_view_b, and result_vector_view have been properly initialized
vsip_vadd_f(vector_view_a, vector_view_b, result_vector_view);
```

8.5 vsip_dvsub_p - Element-wise Subtraction of Two Vector Views

```
void vsip_vsub_f(const vsip_vview_f* a, const vsip_vview_f* b, constvsip_vview_f* r);
void vsip_cvsub_f(const vsip_cvview_f* a, const vsip_cvview_f* b, const vsip_cvview_f* r);
```

Description

This function performs element-wise subtraction of the vector view `b` from the vector view `a` and stores the result in the vector view `r`.

Parameters

- `const vsip_dvview_p* a`: Pointer to the minuend vector view.
- `const vsip_dvview_p* b`: Pointer to the subtrahend vector view.
- `const vsip_dvview_p* r`: Pointer to the destination vector view.

Example

```
vsip_vview_f *vector_view_a;
vsip_vview_f *vector_view_b;
vsip_vview_f *result_vector_view;

// Assuming vector_view_a, vector_view_b, and result_vector_view have been properly initialized
vsip_vsub_f(vector_view_a, vector_view_b, result_vector_view);
```

8.6 vsip_dsvmul_p - Multiply a Scalar by a Vector View

```
void vsip_svmul_f(vsip_scalar_f alpha, const vsip_vview_f* b, const vsip_vview_f* r);
void vsip_csvmul_f(vsip_cscalar_f alpha, const vsip_cvview_f* b, const vsip_cvview_f* r);
```

Description

This function multiplies each element of the vector view `b` by the scalar `alpha` and stores the result in the vector view `r`.

Parameters

- `vsip_dscalar_p alpha`: The scalar value to multiply by.
- `const vsip_dvview_p* b`: Pointer to the source vector view.
- `const vsip_dvview_p* r`: Pointer to the destination vector view.

Example

```
vsip_vview_f *src_vector_view;
vsip_vview_f *dst_vector_view;
vsip_scalar_f scalar = 2.0;

// Assuming src_vector_view and dst_vector_view have been properly initialized
vsip_svmul_f(scalar, src_vector_view, dst_vector_view);
```

8.7 vsip_svdiv_p - Divide a Scalar by a Vector View

```
void vsip_svdiv_f(vsip_scalar_f alpha, const vsip_vview_f* b, const vsip_vview_f* r);
```

Description

This function divides the scalar `alpha` by each element of the vector view `b` and stores the result in the vector view `r`.

Parameters

- `vsip_scalar_p alpha`: The scalar value to divide.
- `const vsip_vview_p* b`: Pointer to the source vector view.
- `const vsip_vview_p* r`: Pointer to the destination vector view.

Example

```
vsip_vview_f *src_vector_view;
vsip_vview_f *dst_vector_view;
vsip_scalar_f scalar = 2.0;

// Assuming src_vector_view and dst_vector_view have been properly initialized
vsip_svdiv_f(scalar, src_vector_view, dst_vector_view);
```

8.8 vsip_svadd_p - Add a Scalar to a Vector View

```
void vsip_svadd_f(vsip_scalar_f alpha, const vsip_vview_f* b, const vsip_vview_f* r);
```

Description

This function adds the scalar `alpha` to each element of the vector view `b` and stores the result in the vector view `r`.

Parameters

- `vsip_scalar_p alpha`: The scalar value to add.
- `const vsip_vview_p* b`: Pointer to the source vector view.
- `const vsip_vview_p* r`: Pointer to the destination vector view.

Example

```
vsip_vview_f *src_vector_view;
vsip_vview_f *dst_vector_view;
vsip_scalar_f scalar = 2.0;

// Assuming src_vector_view and dst_vector_view have been properly initialized
vsip_svadd_f(scalar, src_vector_view, dst_vector_view);
```

8.9 vsip_dvneg_p - Negate Elements of a Vector View

```
void vsip_vneg_f(const vsip_vview_f* a, const vsip_vview_f* r);
void vsip_cvneg_f(const vsip_cvview_f* a, const vsip_cvview_f* r);
```

Description

This function negates each element of the vector view a and stores the result in the vector view r.

$$r_i = -a_i$$

Parameters

- const vsip_dvview_p* a: Pointer to the source vector view.
- const vsip_dvview_p* r: Pointer to the destination vector view.

Example

```
vsip_vview_f *src_vector_view;
vsip_vview_f *dst_vector_view;

// Assuming src_vector_view and dst_vector_view have been properly initialized
vsip_vneg_f(src_vector_view, dst_vector_view);
```

8.10 vsip_dvmag_p - Compute Magnitude of Elements of a Vector View

```
void vsip_vmag_f(const vsip_vview_f* a, const vsip_vview_f* r);
void vsip_cvmag_f(const vsip_cvview_f* a, const vsip_vview_f* r);
```

Description

This function computes the magnitude (absolute value) of each element of the vector view a and stores the result in the vector view r.

$$r_i = |a_i|$$

Parameters

- const vsip_dvview_p* a: Pointer to the source vector view.
- const vsip_dvview_p* r: Pointer to the destination vector view.

Example

```
vsip_vview_f *src_vector_view;
vsip_vview_f *dst_vector_view;

// Assuming src_vector_view and dst_vector_view have been properly initialized
vsip_vmag_f(src_vector_view, dst_vector_view);
```

9 Vector Real

9.1 vsip_vminval_p - Find the Minimum Value in a Vector View

```
vsip_scalar_f vsip_vminval_f(const vsip_vview_f* a, vsip_index* j);
```

Description

This function finds the minimum value in the vector view `a` and returns it. The index of the minimum value is stored in the variable pointed to by `j`.

Parameters

- `const vsip_vview_p* a`: Pointer to the vector view.
- `vsip_index* j`: Pointer to a variable where the index of the minimum value will be stored.

Return Value

- The minimum value in the vector view.

Example

```
vsip_vview_f *vector_view;
vsip_index index;
vsip_scalar_f min_value;

// Assuming vector_view has been properly initialized
min_value = vsip_vminval_f(vector_view, &index);
```

9.2 vsip_vmaxval_p - Find the Maximum Value in a Vector View

```
vsip_scalar_f vsip_vmaxval_f(const vsip_vview_f* a, vsip_index* j);
```

Description

This function finds the maximum value in the vector view `a` and returns it. The index of the maximum value is stored in the variable pointed to by `j`.

Parameters

- `const vsip_vview_p* a`: Pointer to the vector view.
- `vsip_index* j`: Pointer to a variable where the index of the maximum value will be stored.

Return Value

- The maximum value in the vector view.

Example

```
vsip_vview_f *vector_view;
vsip_index index;
vsip_scalar_f max_value;

// Assuming vector_view has been properly initialized
max_value = vsip_vmaxval_f(vector_view, &index);
```

9.3 vsip_vsumval_p - Compute the Sum of Elements in a Vector View

```
vsip_scalar_f vsip_vsumval_f(const vsip_vview_f* a);
```

Description

This function computes the sum of all elements in the vector view a and returns it.

$$\sum_{i}^{n} a_i$$

Parameters

- const vsip_vview_p* a: Pointer to the vector view.

Return Value

- The sum of all elements in the vector view.

Example

```
vsip_vview_f *vector_view;
vsip_scalar_f sum;

// Assuming vector_view has been properly initialized
sum = vsip_vsumval_f(vector_view);
```

9.4 vsip_vsumsqval_p - Compute the Sum of Squares of Elements in a Vector View

```
vsip_scalar_f vsip_vsumsqval_f(const vsip_vview_f* a);
```

Description

This function computes the sum of the squares of all elements in the vector view a and returns it.

$$\sum_{i}^{n} a_i^2$$

Parameters

- const vsip_vview_p* a: Pointer to the vector view.

Return Value

- The sum of the squares of all elements in the vector view.

Example

```
vsip_vview_f *vector_view;
vsip_scalar_f sum_of_squares;

// Assuming vector_view has been properly initialized
sum_of_squares = vsip_vsumsqval_f(vector_view);
```

9.5 vsip_vsq_p - Square Elements of a Vector View

```
void vsip_vsq_f(const vsip_vview_f* a, const vsip_vview_f* r);
```

Description

This function squares each element of the vector view a and stores the result in the vector view r.

$$r_i = a_i^2$$

Parameters

- const vsip_vview_p* a: Pointer to the source vector view.
- const vsip_vview_p* r: Pointer to the destination vector view.

Example

```
vsip_vview_f *src_vector_view;
vsip_vview_f *dst_vector_view;

// Assuming src_vector_view and dst_vector_view have been properly initialized
vsip_vsq_f(src_vector_view, dst_vector_view);
```

9.6 vsip_vrecip_p - Compute Reciprocal of Elements of a Vector View

```
void vsip_vrecip_f(const vsip_vview_f* a, const vsip_vview_f* r);
```

Description

This function computes the reciprocal of each element of the vector view a and stores the result in the vector view r.

$$r_i = \frac{1}{a_i}$$

Parameters

- const vsip_vview_p* a: Pointer to the source vector view.
- const vsip_vview_p* r: Pointer to the destination vector view.

Example

```
vsip_vview_f *src_vector_view;
vsip_vview_f *dst_vector_view;

// Assuming src_vector_view and dst_vector_view have been properly initialized
vsip_vrecip_f(src_vector_view, dst_vector_view);
```

9.7 vsip_vmin_p - Element-wise Minimum of Two Vector Views

```
void vsip_vmin_f(const vsip_vview_f* a, const vsip_vview_f* b, const vsip_vview_f* w);
```

Description

This function performs element-wise minimum comparison of the vector views `a` and `b` and stores the result in the vector view `w`. Each element in `w` is the minimum of the corresponding elements in `a` and `b`.

$$w_i = \min(a_i, b_i)$$

Parameters

- `const vsip_vview_p* a`: Pointer to the first source vector view.
- `const vsip_vview_p* b`: Pointer to the second source vector view.
- `const vsip_vview_p* w`: Pointer to the destination vector view.

Example

```
vsip_vview_f *vector_view_a;
vsip_vview_f *vector_view_b;
vsip_vview_f *result_vector_view;

// Assuming vector_view_a, vector_view_b, and result_vector_view have been properly initialized
vsip_vmin_f(vector_view_a, vector_view_b, result_vector_view);
```

9.8 vsip_vmax_p - Element-wise Maximum of Two Vector Views

```
void vsip_vmax_f(const vsip_vview_f* a, const vsip_vview_f* b, const vsip_vview_f* w);
```

Description

This function performs element-wise maximum comparison of the vector views a and b and stores the result in the vector view w. Each element in w is the maximum of the corresponding elements in a and b.

$$w_i = \max(a_i, b_i)$$

Parameters

- const vsip_vview_p* a: Pointer to the first source vector view.
- const vsip_vview_p* b: Pointer to the second source vector view.
- const vsip_vview_p* w: Pointer to the destination vector view.

Example

```
vsip_vview_f *vector_view_a;
vsip_vview_f *vector_view_b;
vsip_vview_f *result_vector_view;

// Assuming vector_view_a, vector_view_b, and result_vector_view have been properly initialized
vsip_vmax_f(vector_view_a, vector_view_b, result_vector_view);
```

9.9 vsip_vsin_p - Element-wise Sine of a Vector View

```
void vsip_vsin_f(const vsip_vview_f* a, const vsip_vview_f* r);
```

Description

This function computes the element-wise sine of the vector view a and stores the result in the vector view r.

$$r_i = \sin(a_i)$$

Parameters

- const vsip_vview_p* a: Pointer to the source vector view.
- const vsip_vview_p* r: Pointer to the destination vector view.

Example

```
vsip_vview_f *src_vector_view;
vsip_vview_f *dst_vector_view;

// Assuming src_vector_view and dst_vector_view have been properly initialized
vsip_vsin_f(src_vector_view, dst_vector_view);
```

9.10 vsip_vcos_p - Element-wise Cosine of a Vector View

```
void vsip_vcos_f(const vsip_vview_f* a, const vsip_vview_f* r);
```

Description

This function computes the element-wise cosine of the vector view a and stores the result in the vector view r.

$$r_i = \cos(a_i)$$

Parameters

- const vsip_vview_p* a: Pointer to the source vector view.
- const vsip_vview_p* r: Pointer to the destination vector view.

Example

```
vsip_vview_f *src_vector_view;
vsip_vview_f *dst_vector_view;

// Assuming src_vector_view and dst_vector_view have been properly initialized
vsip_vcos_f(src_vector_view, dst_vector_view);
```

9.11 vsip_vtan_p - Element-wise Tangent of a Vector View

```
void vsip_vtan_f(const vsip_vview_f* a, const vsip_vview_f* r);
```

Description

This function computes the element-wise tangent of the vector view a and stores the result in the vector view r.

$$r_i = \tan(a_i)$$

Parameters

- const vsip_vview_p* a: Pointer to the source vector view.
- const vsip_vview_p* r: Pointer to the destination vector view.

Example

```
vsip_vview_f *src_vector_view;
vsip_vview_f *dst_vector_view;

// Assuming src_vector_view and dst_vector_view have been properly initialized
vsip_vtan_f(src_vector_view, dst_vector_view);
```

9.12 vsip_vatan_p - Element-wise Arctangent of a Vector View

```
void vsip_vatan_f(const vsip_vview_f* a, const vsip_vview_f* r);
```

Description

This function computes the element-wise arctangent (inverse tangent) of the vector view a and stores the result in the vector view r.

$$r_i = \tan^{-1}(a_i)$$

Parameters

- const vsip_vview_p* a: Pointer to the source vector view.
- const vsip_vview_p* r: Pointer to the destination vector view.

Example

```
vsip_vview_f *src_vector_view;
vsip_vview_f *dst_vector_view;

// Assuming src_vector_view and dst_vector_view have been properly initialized
vsip_vatan_f(src_vector_view, dst_vector_view);
```

9.13 vsip_vexp_p - Element-wise Exponential of a Vector View

```
void vsip_vexp_f(const vsip_vview_f* a, const vsip_vview_f* r);
```

Description

This function computes the element-wise exponential of the vector view a and stores the result in the vector view r.

$$r_i = e^{a_i}$$

Parameters

- const vsip_vview_p* a: Pointer to the source vector view.
- const vsip_vview_p* r: Pointer to the destination vector view.

Example

```
vsip_vview_f *src_vector_view;
vsip_vview_f *dst_vector_view;

// Assuming src_vector_view and dst_vector_view have been properly initialized
vsip_vexp_f(src_vector_view, dst_vector_view);
```

9.14 vsip_vlog_p - Element-wise Natural Logarithm of a Vector View

```
void vsip_vlog_f(const vsip_vview_f* a, const vsip_vview_f* r);
```

Description

This function computes the element-wise natural logarithm of the vector view a and stores the result in the vector view r.

$$r_i = \log(a_i)$$

Parameters

- const vsip_vview_p* a: Pointer to the source vector view.
- const vsip_vview_p* r: Pointer to the destination vector view.

Example

```
vsip_vview_f *src_vector_view;
vsip_vview_f *dst_vector_view;

// Assuming src_vector_view and dst_vector_view have been properly initialized
vsip_vlog_f(src_vector_view, dst_vector_view);
```

9.15 vsip_vlog10_p - Element-wise Base-10 Logarithm of a Vector View

```
void vsip_vlog10_f(const vsip_vview_f* a, const vsip_vview_f* r);
```

Description

This function computes the element-wise base-10 logarithm of the vector view a and stores the result in the vector view r.

$$r_i = \log_{10}(a_i)$$

Parameters

- const vsip_vview_p* a: Pointer to the source vector view.
- const vsip_vview_p* r: Pointer to the destination vector view.

Example

```
vsip_vview_f *src_vector_view;
vsip_vview_f *dst_vector_view;

// Assuming src_vector_view and dst_vector_view have been properly initialized
vsip_vlog10_f(src_vector_view, dst_vector_view);
```

9.16 vsip_vsqrt_p - Element-wise Square Root of a Vector View

```
void vsip_vsqrt_f(const vsip_vview_f* a, const vsip_vview_f* r);
```

Description

This function computes the element-wise square root of the vector view a and stores the result in the vector view r.

$$r_i = \sqrt{a_i}$$

Parameters

- const vsip_vview_p* a: Pointer to the source vector view.
- const vsip_vview_p* r: Pointer to the destination vector view.

Example

```
vsip_vview_f *src_vector_view;
vsip_vview_f *dst_vector_view;

// Assuming src_vector_view and dst_vector_view have been properly initialized
vsip_vsqrt_f(src_vector_view, dst_vector_view);
```

9.17 vsip_vatan2_p - Element-wise Arctangent of Two Vector Views

```
void vsip_vatan2_f(const vsip_vview_f* a, const vsip_vview_f* b, const vsip_vview_f* r);
```

Description

This function computes the element-wise arctangent of the quotient of the corresponding elements in the vector views a and b and stores the result in the vector view r.

$$r_i = \tan^{-1}(a/b)$$

Parameters

- const vsip_vview_p* a: Pointer to the first source vector view.
- const vsip_vview_p* b: Pointer to the second source vector view.
- const vsip_vview_p* r: Pointer to the destination vector view.

Example

```
vsip_vview_f *vector_view_a;
vsip_vview_f *vector_view_b;
vsip_vview_f *result_vector_view;

// Assuming vector_view_a, vector_view_b, and result_vector_view have been properly initialized
vsip_vatan2_f(vector_view_a, vector_view_b, result_vector_view);
```

9.18 vsip_vfill_p - Fill a Vector View with a Scalar Value

```
void vsip_vfill_f(vsip_scalar_f alpha, const vsip_vview_f* r);
```

Description

This function fills the vector view `r` with the scalar value `alpha`.

$$\mathbf{r} = \alpha$$

Parameters

- `vsip_scalar_p alpha`: The scalar value to fill the vector view with.
- `const vsip_vview_p* r`: Pointer to the destination vector view.

Example

```
vsip_vview_f *vector_view;
vsip_scalar_f scalar_value = 2.0;

// Assuming vector_view has been properly initialized
vsip_vfill_f(scalar_value, vector_view);
```

9.19 vsip_vramp_p - Fill a Vector View with a Ramp

```
void vsip_vramp_f(vsip_scalar_f z, vsip_scalar_f d, const vsip_vview_f* r);
```

Description

This function fills the vector view r with a ramp starting at z and incrementing by d.

$$r_i = z + di$$

Parameters

- vsip_scalar_p z: The starting value of the ramp.
- vsip_scalar_p d: The increment value of the ramp.
- const vsip_vview_p* r: Pointer to the destination vector view.

Example

```
vsip_vview_f *vector_view;
vsip_scalar_f start_value = 0.0;
vsip_scalar_f increment = 1.0;

// Assuming vector_view has been properly initialized
vsip_vramp_f(start_value, increment, vector_view);
```

10 Vector Complex

10.1 vsip_cvjdot_p - Compute the Conjugate Dot Product of Two Complex Vector Views

```
vsip_cscalar_f vsip_cvjdot_f(const vsip_cvview_f* a, const vsip_cvview_f* b);
```

Description

This function computes the conjugate dot product of the complex vector views `a` and `b` and returns it. The conjugate dot product is computed as the sum of the element-wise products of the corresponding elements in the first vector and the conjugate of the elements in the second vector.

$$\sum_{i}^{n} a_i \overline{b_i}$$

Parameters

- `const vsip_cvview_p* a`: Pointer to the first complex vector view.
- `const vsip_cvview_p* b`: Pointer to the second complex vector view.

Return Value

- The conjugate dot product of the two complex vector views.

Example

```
vsip_cvview_f *complex_vector_a;
vsip_cvview_f *complex_vector_b;
vsip_cscalar_f conjugate_dot_product;

// Assuming complex_vector_a and complex_vector_b have been properly initialized
conjugate_dot_product = vsip_cvjdot_f(complex_vector_a, complex_vector_b);
```

10.2 vsip_cvjmul_p - Element-wise Complex Conjugate Multiplication of Two Complex Vector Views

```
void vsip_cvjmul_f(const vsip_cvview_f* a, const vsip_cvview_f* b, const vsip_cvview_f* w);
```

Description

This function performs element-wise complex conjugate multiplication of the complex vector views `a` and `b` and stores the result in the complex vector view `w`.

Parameters

- `const vsip_cvview_p* a`: Pointer to the first source complex vector view.
- `const vsip_cvview_p* b`: Pointer to the second source complex vector view.
- `const vsip_cvview_p* w`: Pointer to the destination complex vector view.

Example

```
vsip_cvview_f *complex_vector_a;
vsip_cvview_f *complex_vector_b;
vsip_cvview_f *result_vector;

// Assuming complex_vector_a, complex_vector_b, and result_vector have been properly initialized
vsip_cvjmul_f(complex_vector_a, complex_vector_b, result_vector);
```

10.3 vsip_rcvmul_p - Element-wise Real-Complex Multiplication

```
void vsip_rcvmul_f(const vsip_vview_f* a, const vsip_cvview_f* b, const vsip_cvview_f* r);
```

Description

This function performs element-wise multiplication of the real vector view `a` and the complex vector view `b` and stores the result in the complex vector view `r`.

Parameters

- `const vsip_vview_p* a`: Pointer to the source real vector view.
- `const vsip_cvview_p* b`: Pointer to the source complex vector view.
- `const vsip_cvview_p* r`: Pointer to the destination complex vector view.

Example

```
vsip_vview_f *real_vector;
vsip_cvview_f *complex_vector;
vsip_cvview_f *result_vector;

// Assuming real_vector, complex_vector, and result_vector have been properly initialized
vsip_rcvmul_f(real_vector, complex_vector, result_vector);
```

10.4 vsip_rscvmul_p - Element-wise Scalar-Complex Multiplication

```
void vsip_rscvmul_f(vsip_scalar_f alpha, const vsip_cvview_f* b, const vsip_cvview_f* r);
```

Description

This function performs element-wise multiplication of the scalar `alpha` and the complex vector view `b` and stores the result in the complex vector view `r`.

Parameters

- `vsip_scalar_p alpha`: The scalar value to multiply by.
- `const vsip_cvview_p* b`: Pointer to the source complex vector view.
- `const vsip_cvview_p* r`: Pointer to the destination complex vector view.

Example

```
vsip_scalar_f scalar = 2.0;
vsip_cvview_f *complex_vector;
vsip_cvview_f *result_vector;

// Assuming complex_vector and result_vector have been properly initialized
vsip_rscvmul_f(scalar, complex_vector, result_vector);
```

10.5 vsip_cvconj_p - Element-wise Complex Conjugate of a Complex Vector View

```
void vsip_cvconj_f(const vsip_cvview_f* a, const vsip_cvview_f* r);
```

Description

This function computes the element-wise complex conjugate of the complex vector view a and stores the result in the complex vector view r.

Parameters

- const vsip_cvview_p* a: Pointer to the source complex vector view.
- const vsip_cvview_p* r: Pointer to the destination complex vector view.

Example

```
vsip_cvview_f *complex_vector;
vsip_cvview_f *result_vector;

// Assuming complex_vector and result_vector have been properly initialized
vsip_cvconj_f(complex_vector, result_vector);
```

10.6 vsip_cvmag_p - Compute Magnitude of Complex Vector View

```
void vsip_cvmag_f(const vsip_cvview_f *a, const vsip_vview_f *r);
```

Description

This function computes the element-wise magnitude (absolute value) of each complex element in the vector view `a` and stores the result in the real vector view `r`. The magnitude of a complex number $a + bi$ is calculated as $\sqrt{a^2 + b^2}$.

Parameters

- `const vsip_cvview_p* a`: Pointer to the source complex vector view.
- `const vsip_vview_p* r`: Pointer to the destination real vector view where the magnitudes will be stored.

Example

```
vsip_cvview_f *complex_vector_view;
vsip_vview_f *magnitude_vector_view;

// Assuming complex_vector_view and magnitude_vector_view have been properly initialized
vsip_cvmag_f(complex_vector_view, magnitude_vector_view);
```

10.7 vsip_vcmagsq_p - Element-wise Magnitude Squared of a Complex Vector View

```
void vsip_vcmagsq_f(const vsip_cvview_f* a, const vsip_vview_f* r);
```

Description

This function computes the element-wise magnitude squared of the complex vector view a and stores the result in the real vector view r.

Parameters

- const vsip_cvview_p* a: Pointer to the source complex vector view.
- const vsip_vview_p* r: Pointer to the destination real vector view.

Example

```
vsip_cvview_f *complex_vector;
vsip_vview_f *real_vector;

// Assuming complex_vector and real_vector have been properly initialized
vsip_vcmagsq_f(complex_vector, real_vector);
```

10.8 vsip_vreal_p - Extract Real Part of a Complex Vector View

```
void vsip_vreal_f(const vsip_cvview_f* a, const vsip_vview_f* r);
```

Description

This function extracts the real part of the complex vector view a and stores the result in the real vector view r.

Parameters

- const vsip_cvview_p* a: Pointer to the source complex vector view.
- const vsip_vview_p* r: Pointer to the destination real vector view.

Example

```
vsip_cvview_f *complex_vector;
vsip_vview_f *real_vector;

// Assuming complex_vector and real_vector have been properly initialized
vsip_vreal_f(complex_vector, real_vector);
```

10.9 vsip_vimag_p - Extract Imaginary Part of a Complex Vector View

```
void vsip_vimag_f(const vsip_cvview_f* a, const vsip_vview_f* r);
```

Description

This function extracts the imaginary part of the complex vector view a and stores the result in the real vector view r.

Parameters

- const vsip_cvview_p* a: Pointer to the source complex vector view.
- const vsip_vview_p* r: Pointer to the destination real vector view.

Example

```
vsip_cvview_f *complex_vector;
vsip_vview_f *imag_vector;

// Assuming complex_vector and imag_vector have been properly initialized
vsip_vimag_f(complex_vector, imag_vector);
```

10.10 vsip_vcmplx_p - Create a Complex Vector View from Real and Imaginary Parts

```
void vsip_vcmplx_f(const vsip_vview_f* a, const vsip_vview_f* b, const vsip_cvview_f* r);
```

Description

This function creates a complex vector view `r` from the real vector view `a` and the imaginary vector view `b`.

Parameters

- `const vsip_vview_p* a`: Pointer to the source real vector view.
- `const vsip_vview_p* b`: Pointer to the source imaginary vector view.
- `const vsip_cvview_p* r`: Pointer to the destination complex vector view.

Example

```
vsip_vview_f *real_vector;
vsip_vview_f *imag_vector;
vsip_cvview_f *complex_vector;

// Assuming real_vector, imag_vector, and complex_vector have been properly initialized
vsip_vcmplx_f(real_vector, imag_vector, complex_vector);
```

11 FIR

11.1 vsip_dfir_create_p - Create a FIR Filter

```
typedef enum _vsip_symmetry {
  VSIP_NONSYM            = 0,
  VSIP_SYM_EVEN_LEN_ODD  = 1,
  VSIP_SYM_EVEN_LEN_EVEN = 2
} vsip_symmetry;

typedef enum _vsip_obj_state {
  VSIP_STATE_NO_SAVE = 1,
  VSIP_STATE_SAVE    = 2
} vsip_obj_state;

typedef enum _vsip_alg_hint {
  VSIP_ALG_TIME  = 0,
  VSIP_ALG_SPACE = 1,
  VSIP_ALG_NOISE = 2
} vsip_alg_hint;

vsip_fir_f *vsip_fir_create_f(const vsip_vview_f *kernel, vsip_symmetry symm,
                              vsip_length n, vsip_length d,
                              vsip_obj_state state,
                              vsip_length ntimes, vsip_alg_hint hint);
vsip_cfir_f *vsip_cfir_create_f(const vsip_cvview_f *kernel, vsip_symmetry symm,
                                vsip_length n, vsip_length d,
                                vsip_obj_state state,
                                vsip_length ntimes, vsip_alg_hint hint);
```

Description

This function creates a FIR (Finite Impulse Response) filter with the specified kernel, symmetry, length, decimation factor, state, number of times to apply the filter, and algorithm hint.

Parameters

- `const vsip_dvview_p* kernel`: Pointer to the kernel vector view.
- `vsip_symmetry symm`: Symmetry of the filter kernel.
  - `VSIP_NOSYM` - No symmetry
  - `VSIP_SYM_EVEN_LEN_ODD` - Odd symmetry
  - `VSIP_SYM_EVEN_LEN_EVEN` - Even symmetry
- `vsip_length n`: Length of the filter.
- `vsip_length d`: Decimation factor.
- `vsip_obj_state state`: State of the filter object.
  - `VSIP_STATE_NO_SAVE` - Do not save state
  - `VSIP_STATE_SAVE` - Save state
- `vsip_length ntimes`: Number of times to apply the filter.
- `vsip_alg_hint hint`: Algorithm hint for the filter.
  - `VSIP_ALG_TIME` - Optimize for time
  - `VSIP_ALG_SPACE` - Optimize for memory usage
  - `VSIP_ALG_NOISE` - Optimize for noise

Return Value

- On success, a pointer to the newly created FIR filter object is returned.
- On error, NULL is returned.

Example

```
vsip_vview_f *kernel_view;
vsip_symmetry symm = VSIP_NONSYM;
vsip_length length = 10;
vsip_length decimation = 1;
vsip_obj_state state = VSIP_STATE_SAVE;
vsip_length ntimes = 1;
vsip_alg_hint hint = VSIP_ALG_TIME;
vsip_fir_f *fir_filter;

// Assuming kernel_view has been properly initialized
fir_filter = vsip_fir_create_f(kernel_view, symm, length, decimation, state, ntimes, hint);

if (fir_filter == NULL) {
    // Handle error
}
```

11.2 vsip_dfir_reset_p - Reset a FIR Filter

```
void vsip_fir_reset_f(vsip_fir_f *filt);
void vsip_cfir_reset_f(vsip_cfir_f *filt);
```

Description

This function resets the specified FIR filter to its initial state.

Parameters

- `vsip_dfir_p* filt`: Pointer to the FIR filter to be reset.

Example

```
vsip_fir_f *fir_filter;

// Assuming fir_filter has been properly initialized
vsip_fir_reset_f(fir_filter);
```

11.3 vsip_dfir_getattr_p - Get Attributes of a FIR Filter

```
typedef struct _vsip_fir_attr_f{
  vsip_scalar_vi kernel_len;
  vsip_symmetry symm;
  vsip_scalar_vi in_len;
  vsip_scalar_vi out_len;
  vsip_length decimation;
  vsip_obj_state state;
} vsip_fir_attr_f;

void vsip_fir_getattr_f(const vsip_fir_f *fir, vsip_fir_attr_f *attr);
void vsip_cfir_getattr_f(const vsip_cfir_f *fir, vsip_cfir_attr_f *attr);
```

Description

This function retrieves the attributes of the specified FIR filter and stores them in the structure pointed to by `attr`.

Parameters

- `const vsip_dfir_p* fir`: Pointer to the FIR filter.
- `vsip_dfir_attr_p* attr`: Pointer to a structure where the attributes will be stored.

Example

```
vsip_fir_f *fir_filter;
vsip_fir_attr_f attributes;

// Assuming fir_filter has been properly initialized
vsip_fir_getattr_f(fir_filter, &attributes);

// The attributes of the FIR filter are now stored in 'attributes'
```

11.4 vsip_dfirflt_p - Apply a FIR Filter to a Vector View

```
int vsip_firflt_f(vsip_fir_f *fir, const vsip_vview_f *x, const vsip_vview_f *y);
int vsip_cfirflt_f(vsip_cfir_f *fir, const vsip_cvview_f *x, const vsip_cvview_f *y);
```

Description

This function applies the specified FIR filter to the input vector view x and stores the result in the output vector view y.

Parameters

- vsip_dfir_p* fir: Pointer to the FIR filter.
- const vsip_dvview_p* x: Pointer to the input vector view.
- const vsip_dvview_p* y: Pointer to the output vector view.

Return Value

- Returns 0 on success.
- Returns a non-zero value on error.

Example

```
vsip_fir_f *fir_filter;
vsip_vview_f *input_vector;
vsip_vview_f *output_vector;
int result;

// Assuming fir_filter, input_vector, and output_vector have been properly initialized
result = vsip_firflt_f(fir_filter, input_vector, output_vector);

if (result != 0) {
    // Handle error
}
```

11.5 vsip_dfir_destroy_p - Destroy a FIR Filter

```
int vsip_fir_destroy_f(vsip_fir_f *filt);
int vsip_cfir_destroy_f(vsip_cfir_f *filt);
```

Description

This function destroys the specified FIR filter and frees associated resources.

Parameters

- `vsip_dfir_p* filt`: Pointer to the FIR filter to be destroyed.

Return Value

- Returns `0` on success.
- Returns a non-zero value on error.

Example

```
vsip_fir_f *fir_filter;
int result;

// Assuming fir_filter has been properly initialized
result = vsip_fir_destroy_f(fir_filter);

if (result != 0) {
    // Handle error
}
```

12 FFT

12.1 vsip_*dd*fftop_create_*p* - Create FFT Objects

```
typedef enum _vsip_fft_dir {
    VSIP_FFT_FWD = -1,
    VSIP_FFT_INV = +1
} vsip_fft_dir;

typedef enum _vsip_alg_hint {
  VSIP_ALG_TIME  = 0,
  VSIP_ALG_SPACE = 1,
  VSIP_ALG_NOISE = 2
} vsip_alg_hint;

vsip_fft_f* vsip_ccfftop_create_f(vsip_length length, vsip_scalar_f scale,
                                  vsip_fft_dir sign, vsip_length ntimes,
                                  vsip_alg_hint hint);
vsip_fft_f* vsip_rcfftop_create_f(vsip_length length, vsip_scalar_f scale,
                                  vsip_fft_dir sign, vsip_length ntimes,
                                  vsip_alg_hint hint);
vsip_fft_f* vsip_crfftop_create_f(vsip_length length, vsip_scalar_f scale,
                                  vsip_fft_dir sign, vsip_length ntimes,
                                  vsip_alg_hint hint);
```

Description

These functions create FFT (Fast Fourier Transform) objects for different types of FFT operations:

- `vsip_ccfftop_create_p`: Creates an FFT object for complex-to-complex out-of-place FFT.
- `vsip_rcfftop_create_p`: Creates an FFT object for real-to-complex out-of-place FFT.
- `vsip_crfftop_create_p`: Creates an FFT object for complex-to-real out-of-place FFT.

Each function initializes the FFT object with the specified length, scale factor, direction, number of times to apply the FFT, and algorithm hint.

The performance for supported FFT sizes is standardized as $O(n \log n)$. For sizes not directly supported by the FFT kernels a DFT fallback with a performance of $O(n^2)$ is standardized.

Parameters

- `vsip_length length`: The length of the FFT.
- `vsip_scalar_f scale`: The scale factor to apply to the FFT result.
- `vsip_fft_dir sign`: The direction of the FFT.
  - `VSIP_FFT_FWD` - Forward
  - `VSIP_FFT_INV` - Inverse
- `vsip_length ntimes`: The number of times to apply the FFT.
- `vsip_alg_hint hint`: Algorithm hint for the FFT.
  - `VSIP_ALG_TIME` - Optimize for time
  - `VSIP_ALG_SPACE` - Optimize for memory usage
  - `VSIP_ALG_NOISE` - Optimize for noise

Return Value

- On success, a pointer to the newly created FFT object is returned.
- On error, `NULL` is returned.

Example

```
vsip_length length = 1024;
vsip_scalar_f scale = 1.0;
vsip_fft_dir direction = VSIP_FFT_FWD; // Forward FFT
vsip_length ntimes = 1;
vsip_alg_hint hint = VSIP_ALG_TIME;

vsip_fft_f *fft_cc;
vsip_fft_f *fft_rc;
vsip_fft_f *fft_cr;

// Create complex-to-complex FFT object
fft_cc = vsip_ccfftop_create_f(length, scale, direction, ntimes, hint);
if (fft_cc == NULL) {
    // Handle error
}
```

12.2 vsip_*dd*fftop_*p* - Perform FFT Operations

```
void vsip_ccfftop_f(const vsip_fft_f *fft, const vsip_cvview_f *x, const vsip_cvview_f *y);
void vsip_rcfftop_f(const vsip_fft_f *fft, const vsip_cvview_f *x, const vsip_cvview_f *y);
void vsip_crfftop_f(const vsip_fft_f *fft, const vsip_cvview_f *x, const vsip_vview_f *y);
```

Description

These functions perform FFT (Fast Fourier Transform) operations using the specified FFT object. Each function handles a different type of FFT:

- vsip_ccfftop_*p*: Performs a out-of-place complex-to-complex FFT.
- vsip_rcfftop_*p*: Performs a out-of-place real-to-complex FFT.
- vsip_crfftop_*p*: Performs a out-of-place complex-to-real FFT.

The performance for supported FFT sizes is standardized as $O(n \log n)$. For sizes not directly supported by the FFT kernels a DFT fallback with a performance of $O(n^2)$ is standardized.

Parameters

- const vsip_fft_*p* * fft: Pointer to the FFT object.
- const vsip_*d*vview_*p* * x: Pointer to the input complex vector view
- const vsip_*d*vview_*p* * y: Pointer to the output complex vector view

Example

```
vsip_fft_f *fft_cc;
vsip_fft_f *fft_rc;
vsip_fft_f *fft_cr;
vsip_cvview_f *complex_input;
vsip_cvview_f *complex_output;
vsip_vview_f *real_input;
vsip_vview_f *real_output;

// Assuming fft_cc, fft_rc, fft_cr, complex_input, complex_output, real_input, and real_output have been p

// Perform complex-to-complex FFT
vsip_ccfftop_f(fft_cc, complex_input, complex_output);
```

12.3 vsip_fft_destroy_p - Destroy an FFT Object

```
int vsip_fft_destroy_f(vsip_fft_f *fft);
```

Description

This function destroys the specified FFT object and frees associated resources.

Parameters

- vsip_fft_p* fft: Pointer to the FFT object to be destroyed.

Return Value

- Returns 0 on success.
- Returns a non-zero value on error.

Example

```
vsip_fft_f *fft;
int result;

// Assuming fft has been properly initialized
result = vsip_fft_destroy_f(fft);

if (result != 0) {
    // Handle error
}
```

13 Random Numbers

13.1 vsip_randcreate - Create a Random Number Generator State

```
vsip_randstate *vsip_randcreate(vsip_index seed, vsip_index numprocs,
                                vsip_index id, vsip_rng portable);
```

Description

This function creates and initializes a random number generator state. The function allows for parallel random number generation by specifying the number of processes (`numprocs`) and the process ID (`id`). The `portable` parameter specifies the type of random number generator to use.

Parameters

- `vsip_index seed`: The seed value for the random number generator.
- `vsip_index numprocs`: The number of parallel processes.
- `vsip_index id`: The ID of the current process (must be in the range [0, numprocs-1]).
- `vsip_rng portable`: The type of random number generator to use (e.g., `VSIP_PRNG` for portable random number generation).

Return Value

- On success, a pointer to the newly created random number generator state is returned.
- On error, `NULL` is returned.

Example

```
vsip_randstate *rand_state;
vsip_index seed = 42;
vsip_index numprocs = 1;
vsip_index id = 0;

// Create a random number generator state
rand_state = vsip_randcreate(seed, numprocs, id, VSIP_PRNG);

if (rand_state == NULL) {
    // Handle error
}
```

13.2 vsip_randdestroy - Destroy a Random Number Generator State

```
int vsip_randdestroy(vsip_randstate *state);
```

Description

This function destroys the random number generator state `state` and frees all associated resources. After calling this function, the random number generator state should no longer be used.

Parameters

- `vsip_randstate* state`: Pointer to the random number generator state to be destroyed.

Return Value

- Returns `0` on success.
- Returns a non-zero value on error.

Example

```
vsip_randstate *rand_state;
int result;

// Assuming rand_state has been properly initialized
result = vsip_randdestroy(rand_state);

if (result != 0) {
    // Handle error
}
```

13.3 vsip_vrandu_p - Generate Uniformly Distributed Random Numbers in a Vector View

```
void vsip_vrandu_f(vsip_randstate *state, const vsip_vview_f *r);
```

Description

This function fills the vector view `r` with uniformly distributed random numbers in the range [0, 1) using the random number generator state `state`.

Parameters

- `vsip_randstate* state`: Pointer to the random number generator state.
- `const vsip_vview_p* r`: Pointer to the destination vector view where the random numbers will be stored.

Example

```
vsip_randstate *rand_state;
vsip_vview_f *random_vector;

// Initialize random number generator state
rand_state = vsip_randcreate(42, 0, 1, VSIP_PRNG);

// Assuming random_vector has been properly initialized
vsip_vrandu_f(rand_state, random_vector);

// Clean up
vsip_randdestroy(rand_state);
```

14 Miscellaneous

14.1 vsip_vhisto_p - Compute Histogram of a Vector View

```
typedef enum _vsip_hist_opt {
  VSIP_HIST_RESET = 1,
  VSIP_HIST_ACCUM = 2
} vsip_hist_opt;

void vsip_vhisto_f(const vsip_vview_f *src, vsip_scalar_f min_bin,
                   vsip_scalar_f max_bin, vsip_hist_opt opt,
                   const vsip_vview_f *hist);
```

Description

This function computes the histogram of the elements in the vector view `src` and stores the result in the vector view `hist`. The histogram is computed over the range `[min_bin, max_bin]` with the specified binning options `opt`.

Parameters

- `const vsip_vview_p* src`: Pointer to the source vector view.
- `vsip_scalar_p min_bin`: The minimum value of the histogram bins.
- `vsip_scalar_p max_bin`: The maximum value of the histogram bins.
- `vsip_hist_opt opt`: Options for histogram computation.
  - `VSIP_HIST_RESET` - Reset histogram and compute new
  - `VSIP_HIST_ACCUM` - Accumulate with previous
- `const vsip_vview_f* hist`: Pointer to the destination vector view where the histogram will be stored.

Example

```
vsip_vview_f *src_vector_view;
vsip_scalar_f min_bin = 0.0;
vsip_scalar_f max_bin = 10.0;
vsip_hist_opt hist_options = VSIP_HIST_ACCUM;
vsip_vview_f *hist_vector_view;

// Assuming src_vector_view and hist_vector_view have been properly initialized
vsip_vhisto_f(src_vector_view, min_bin, max_bin, hist_options, hist_vector_view);
```