`libpnicore` Users Guide

Eugen Wintersberger

February 24, 2017

Contents

1. Introduction

1.1. The PNI library stack

`libpnicore` is one of the PNI libraries developed within the framework of the HDRI project. As shown in Fig. 1.1 `libpnicore` it is the foundation for all libraries in the stack. None of them can be used without it. `libpnicore` provides the basic data structures used by all other PNI libraries. This includes

- well defined data types
- multidimensional arrays
- configuration facilities
- type erasures.

1.2. How to read this manual

For a new user the best way to read this manual is from beginning to the end. One may can omit the next chapter about installation of the library if this is done by your local system administrator. In any case, a new user should should definitely start with Chapter 3.

The library makes heavy use of C++11 features. There are plenty of websites on the internet explaining those new features. For an experienced C++ programmer the Wiki site[3] describing the new features for C++11 might be enough. However, new users with less experience in modern C++ may should purchase one of the excellent books available on this language.

A reader already familiar with `libpnicore` may uses this guide as a short reference to the most important features. In any case, more detailed information about each class can be found in the API documentation.

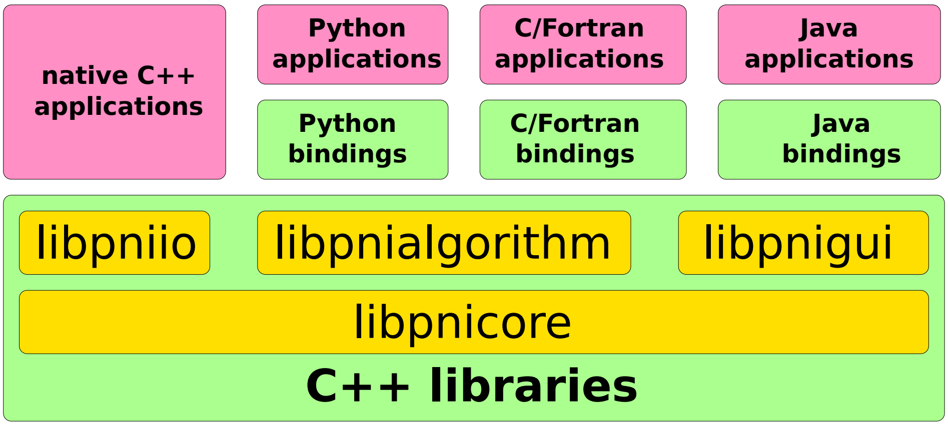

Figure 1.1.: The PNI library stack is a collection of C++ libraries developed with the intention to simplify the process of writing application in the PNI field. `libpnicore`, the library described in this manual is the foundation of this stack. It provides all the basic data structures and facilities used by all other libraries.

2. Installation

Before you building `libpnicore` from sources one should first check if pre-built binary packages are available. Building the library from the sources requires a certain level of expertise which not all users posess. As `libpnicore` is mostly a template library only a few non performance critical components have to be compiled. Therefore, custom builds of the libraries binaries are not necessary in order to get optimum performance.

2.1. Installing pre-built binary packages

Binary packages are currently available for the following platforms

- Debian/GNU Linux
- Ubuntu Linux

2.1.1. Debian and Ubuntu users

As Debian and Ubuntu are closely related the installation is quite similar. The packages are provided by a special Debian repository. To work on the package sources you need to login as `root` user. Use `su` or `sudo su` on Debian and Ubuntu respectively. The first task is to add the GPG key of the HDRI repository to your local keyring

```
> wget -q -O - http://repos.pni-hdri.de/debian_repo.pub.gpg | apt-key add -
```

The return value of this command line should be `OK`. In a next step you have to add new package sources to your system. For this purpose go to `/etc/apt/sources.list.d` and download the sources file. For Debian use

```
> wget http://repos.pni-hdri.de/wheezy-pni-hdri.list
```

and for Ubuntu (Precise)

```
> wget http://repos.pni-hdri.de/precise-pni-hdri.list
```

Once you have downloaded the file use

```
> apt-get update
```

to update your package list and

```
> apt-get install libpnicore1 libpnicore1-dev libpnicore1-doc
```

to install the library. Dependencies will be resolved automatically so you can start with working right after the installation has finished.

2.2. Install from sources

If your OS platform or a particular Linux distribution is currently not supported you have to build `libpnicore` from its sources. As this process usually requires some expert knowledge keep be prepared to get your hands dirty.

2.2.1. Requirements

For a successful build some requirements must be satisfied

- `gcc` >= 4.7 – since version 1.0.0 `libpnicore` requires a mostly C++11 compliant compiler. For the gcc familiy this is 4.7 and upwards
- `BOOST` [1] >= 4.1
- `doxygen` [5] – used to build the API documentation
- `cmake` [2] >= 2.4.8 – the build software used by the `libpnicore`
- `pkg-config` [6] – program to manage libraries
- `cppunit` [4] – a unit test library used for the tests

2.2.2. Obtaining the sources

There are basically two sources from where to obtain the code: either directly from the Git repository on Google Code or a release tarball. The former one should be used if you not only want to use the library but plan to contribute code to it. The latter one is the suggested source if you just want to use the library. As Google Code ceased its download service in January 2014 the tarballs are provided via Google Drive.

For the next steps we assume that the code from the tarball is used.

2.2.3. Building the code

Once downloaded unpack the tarball in a temporary location.

```
> tar xzf libpnicore*.tar.gz
```

which will lead to a new directory named `libpnicore`. As we use `cmake` for building the library, out of place builds are recommended. For this purpose create a new directory where the code will be built and change to this directory

```
> mkdir libpnicore-build
> cd libpnicore-build
```

Now call `cmake` with a path to the original source directory

```
> cmake ../libpnicore
```

A subsequent `make` finally build the library

```
> make
```

This may take a while. Actually building the library is quite fast as `libpnicore` is mostly a template, and thus header-only, library. However, building the test suite is rather time consuming.

2.2.4. Testing the build

Once the build has finished you should definitely run the tests. For this purpose change to the `test` subdirectory in the build directory and run the test script

```
> cd test
> ./run_tests.sh
```

The output is currently a bit crude but there should be 0 failures for all tests.

2.2.5. Installation

If the build has passed the test suite `libpnicore` can be installed from within the build directory with

```
> make install
```

By default the installation prefix is `/usr/local`. If another prefix should be used the `CMAKE_INSTALL_PREFIX` variable must be set when running `cmake` with

```
> cmake -DCMAKE_INSTALL_PREFIX=/opt/pni ../libpnicore
```

which causes the installation prefix to be `/opt/pni`.

3. Using the library

3.1. Include files

To use `libpnicore` in code the appropriate header files must be included. In the simplest case just use the `core.hpp` file

```
#include <pni/core.hpp>
```

which pulls in all the other header files required to work with `libpnicore`. Alternatively, the header files for the different components provided by `libpnicore` can be used

```
#include <pni/core/algorithms.hpp>    //basic algorithms
#include <pni/core/arrays.hpp>        //multidimensional arrays
#include <pni/core/benchmark.hpp>     //benchmark classes
#include <pni/core/configuration.hpp> //program configuration facilities
#include <pni/core/error.hpp>         //exception management
#include <pni/core/type_erasures.hpp> //type erasures
#include <pni/core/types.hpp>         //fundamental data types
#include <pni/core/utilities.hpp>     //general purpose utilities
```

All classes provided by `libpnicore` reside within the `pni::core` namespace. If you do not want to give the namespace explicitly for every type and function use

```
using namespace pni::core;
```

after including the required header files.

3.2. Building and linking

`libpnicore` provides a `pkg-config` file. In the case of a system wide installation this file is most probably allready at the right place in the file system. One can easily check this with

```
>> pkg-config --libs --cflags pnicore
```

for a system wide installation you should get something like this

```
-lpniio -lhdf5 -lz -lboost_filesystem -lpnicore -lboost_program_options\
-lboost_regex -lboost_system
```

For installation locations which are not in the default paths of your system you may get some additional `-I` and `-L` output pointing to the directories where the header files and the library binaries are installed. If `pkg-config` complains that it cannot find a package named `pnicore` then you most probably have to set `PKG_CONFIG_PATH` to the location where the `pkg-config` file of your `libpnicore` installation has been installed.

3.2.1. From the command line

If a single simple program should be compiled the following approach is suggested

```
$> g++ -std=c++11 -O3 -oprogram program.cpp\
       $(pkg-config --cflags --libs pnicore)
```

Please recognize the `-std=c++11` option. `libpnicore` requires a state of the art compiler with full support for C++11.

3.2.2. From within a Makefile

If `make` should be used to build the code add the following lines to your `Makefile`

```
CPPFLAGS=-g -O3 -std=c++11 -fno-deduce-init-list -Wall -Wextra -pedantic \
                 $(shell pkg-config --cflags pnicore)
LDFLAGS=$(shell pkg-config --libs pnicore)
```

This will set the appropriate compiler and linker options for the build.

3.2.3. With Scons

If you use `SCons` for building the code add the following to your `SConstruct` file

```
env = Environment()
env.AppendUnique(CXXFLAGS=["-std=c++11","-pedantic","-Wall","-Wextra"])
env.ParseConfig('pkg-config --cflags --libs pnicore')
```

The `ParseConfig` method of a `SCons` environment is able to parse the output of `pkg-config` and add the flags to the environments configuration.

3.2.4. With CMake

Currently no `cmake`-package files are installed with the library. However, due to `cmakes` `pkg-config` support autoconfiguration can still be done. In one of the toplevel `CMakeLists.txt` files use

```
pkg_search_module(PNICORE REQUIRED pnicore)
```

to load the configuration for `libpnicore` from `pkg-config`. Furthermore we have to add the header file and library installation paths to the configuration

```
link_directories(${PNICORE_LIBRARY_DIRS} ${HDF5_LIBRARY_DIRS})
include_directories(${PNICORE_INCLUDE_DIRS} ${HDF5_INCLUDE_DIRS})
```

Finally the library can easily be added to the build target with

```
target_link_libraries(mytarget ${PNICORE_LIBRARIES})
```

4. Data types

`libpnicore` provides a set of data types of well defined size and utility functions related to type management. The basic header file required to use `libpnicore`s type facilities is

```
#include <pni/core/types.hpp>
```

The data types provided by `libpnicore` include

1. numeric types with all their arithmetic operations
2. string types (currently only one member)
3. and utility types like `binary`, `bool`, and `none`.

All this types together are refered to as *primitive types*. The numeric types are ensured to have the same size on each platform and architecture supported by `libpnicore`. They are mostly `typedef`s to the types defined by the C standard library. However, the utility types `binary`, `bool`, and `none` are unique to `libpnicore` and will be explained in more detail in the last sections of this chapter.

Every type in `libpnicore` is associated with an ID represented by the `type_id_t` enumeration type. Additionally every type belongs to a particular type class defined by the `type_class_t` enumeration type. Table 4.1 gives an overview over the primitive types provided by `libpnicore` and their corresponding `type_id_t` and `type_class_t` values.

4.1. Compile time type identification

To obtain the ID or class of a type at compile time use the `type_id_map` or `type_class_map` type maps.

```
#include <pni/core/types.hpp>

using namespace pni::core;

//determine the type ID for a given type
type_id_map<float32>::type_id == type_id_t::FLOAT32;

//obtain the class of a particular type
type_class_map<float32>::type_class == type_class_t::FLOAT;
```

For IDs the other way around is also possible with the `id_type_map`

type_class_t::	data type	type_id_t::	description
INTEGER	uint8	UINT8	8Bit unsinged integer
	int8	INT8	8Bit signed integer
	uint16	UINT16	16Bit unsigned integer
	int16	INT16	16Bit signed integer
	uint32	UINT32	32Bit unsigned integer
	int32	INT32	32Bit signed integer
	uint64	UINT64	64Bit unsigned integer
	int64	INT64	64Bit signed integer
FLOAT	float32	FLOAT32	32Bit IEEE floating point type
	float64	FLOAT64	64Bit IEEE floating point type
	float128	FLOAT128	128Bit IEEE floating point type
COMPLEX	complex32	COMPLEX32	32Bit IEEE complex float type
	complex64	COMPLEX64	64Bit IEEE complex float type
	complex128	COMPLEX128	128Bit IEEE complex float type
STRING	string	STRING	string type
BINARY	binary	BINARY	binary type
NONE	none	NONE	none type

Table 4.1.: An overview of the primitive data types provided by `libpnicore`.

```
#include <pni/core/types.hpp>

using namespace pni::core;

//determine the type for a given ID
id_type_map<type_id_t::FLOAT32>::type data = ...;
```

For numeric types there are also some other templates for a more detailed type classification

`is_integer_type<T>::value`	`true` if `T` is an integer type
`is_float_type<T>::value`	`true` if `T` is a floating point type
`is_complex_type<T>::value`	`true` if `T` is a complex number type
`is_numeric_type<T>::value`	`true` if `T` is any of the above types

4.2. Identifying types at runtime

The recommended way to deal with type information at runtime are the `type_id_t` enumerations. At some point in time a program might has to determine the type ID of a variable type or of the element type of a container. The basic facility to achieve this is the `type_id` function defined in `pni/core/type_utils.hpp`. The usage of this function is rather simple as shown here

```
#incldue<pni/core/types.hpp>

using namespace pni::core;
```

data type	string representation
uint8	"uint8" , "ui8"
int8	"int8" , "i8"
uint16	"uint16", "ui16"
int16	"int16", "i16"
uint32	"uint32", "ui32"
int32	"int32", "i32"
uint64	"uint64", "ui64"
int64	"int64", "i64"
float32	"float32", "f32"
float64	"float64", "f64"
float128	"float128", "f128"
complex32	"complex32", "c32"
complex64	"complex64", "c64"
complex128	"complex128", "c128"
string	"string", "str"
binary	"binary", "binary"
none	"none"

Table 4.2.: Data types and their string representations.

```
//one could use this with
auto data = get_data(...);

std::cout<<type_id(data)<<std::endl;
```

The important thing to notice here is that no matter what type the get_data function returns, type_id will give you the type ID. In cases where the type ID is given and a classification of the type has to be made four functions are provided where each takes a type ID as its single most argument

is_integer(type_id_t)	returns true if the type ID refers to an integer type
is_float(type_id_t)	returns true if the type ID refers to a float type
is_complex(type_id_t)	returns true if the type ID refers to a complex type
is_numeric(type_id_t)	returns true if the type ID refers to a numeric type

Another important scenario is the situation where a user uses the string representation to tell a program with which type it should work. In such a situation you either want to convert the string representation of a type into a value of type_id_t or vica verse. The library provides two functions for this purpose type_id_from_str which converts the string representation of a type to a value of type_id_t and str_from_type_id which performs the opposite operation. The usage of this two guys is again straight forward.

```
#include <pni/core/types.hpp>
#include <pni/core/type_utils.hpp>
```

```

    using namespace pni::core;

    //get a type id from a string
    string rep = "string";
    type_id_t id = type_id_from_str("str");

    //get a string from a type id
    rep = str_from_type_id(type_id_t::FLOAT32);
```

4.3. The binary type

In many cases uninterpreted binary data should be transfered from one location to the other (a typical example would be to copy the content of one file to another). Typically one would use a `typdef` to something like `uint8` to realize such a type. However, this approach has two disadvantages

1. as `uint8` is a numeric type with all arithmetic operators available which we do not want for uninterpreted binary data
2. a mere `typedef` would make `uint8` and `binary` indistinguishable and thus we could not specialize template classes for each of them.

Consequently `binary` was implemented as a thin wrapper around an appropriately sized integer type with all arithmetic operators stripped away. A short example of how to use binary is the `copy_file.cpp` example in the `examples` directory of the source distribution of `libpnicore`.

examples/copy_file.cpp

```

//---------------------------------------------------------------------------
// This example shows how to implement a simple file copy.
//---------------------------------------------------------------------------

#include <vector>
#include <fstream>
#include <pni/core/types.hpp>

using namespace pni::core;

typedef std::vector<binary> binary_vector;

int main(int ,char **)
{
    //open the input file
    std::ifstream i_stream("Makefile",std::fstream::binary);

    //determine the length of the file
```

```
20      size_t length = i_stream.seekg(0,std::ifstream::end).tellg();
21      i_stream.seekg(0,std::ifstream::beg);
22
23      //allocate memory
24      binary_vector data(length);
25
26      //read data
27      i_stream.read(reinterpret_cast<char*>(data.data()),length);
28      //close input file
29      i_stream.close();
30
31      //open output file
32      std::ofstream o_stream("Makefile.copy",std::fstream::binary);
33      o_stream.write(reinterpret_cast<char*>(data.data()),length);
34      //close the output stream
35      o_stream.close();
36
37      return 0;
38  }
```

In lines 8 and 10 we include the `pni/core/types.hpp` header file and instruct the compiler to use the `pni::core` namespace by default. In line 12 a vector type with binary elements is defined and an instance of this type is allocated in line 24. In line 27 data is read from the input file and stored in the vector. Now, it is clear from here that a vector of type `char` would have perfectly served the same purpose. The major difference is that unlike `char` `binary` has absolutely no semantics. In practice there is nothing much you can do without it rather than store it back to another stream as it is done in line 33.

4.4. The `none` type

The `none` type represents the absence of a type. It is a dummy type of very limited functionality and is mainly used internally by `libpnicore`. One major application of the `none` type is to do default construction of type erasures (see chapter 6). For all practical purposes this type can be ignored.

4.5. The `bool_t` type

Unlike the `C` programming language C++ provides a native `bool` type. Unfortunately the `C++` standardization committee made some unfortunate decisions with `bool` and STL containers. `std::vector` for instance is in most cases specialized for the standard `C++` `bool` type. In the most common STL implementation std::vector is considered an array of individual bits. Meaning that every byte in the vector is storing a total of 8 `bool` values. Consequently we cannot obtain an address for a particular bit but only for the byte where it is stored. Hence `std::vector<bool>` does not provide the `data` method which is required for storage containers used with the `mdarray` templates (see chapter 5).

To overcome this problem a new boolean type was included in `libpnicore` which can be converted to `bool` but uses a single byte for each boolean value and thus can use the `std::vector` template. So use the `libpnicore` `bool_t` type whenever working with `libpnicore` templates or whenever the address of a container element is required. For all other purposes the default C++ `bool` type can be used.

4.6. Numeric type conversion

`libpnicore` provides facilities for save numeric type conversion. These functions are not only used internally by the library they are also available to users. The conversion policy enforced by `libpnicore` is more strict than that of standard C++. For instance you cannot convert a negative integer to an unsigned integer type. The goal of the conversion rules are set up in order to avoid truncation errors as they would typically occur when using the standard C++ rules.

The basic rule for conversion between two integer type A and B is as follows

> A value of type `S` can only be converted to type `B` if the value does not exceed the numeric range of type `B`.

A consequence of this rule is that a signed integer can only be converted to an unsigned type if its value is larger than 0. This is different from the standard C++ rule where the unsigned target type will just overflow.

The second basic rule which governs `libpnicore`s conversion policy is

> During a conversion no information must be lost!

Hence, conversion from a floating point type to an integer type is prohibited as it would most likely lead to truncation and thus a loss of information. Conversion from a scalar float value to a complex value is allowed (as long as the first rule applies to the base type of the complex type) but one cannot convert a complex value to a scalar float type.

Several types cannot be converted to anything than themselves

- `bool_t` which can be only the result of a boolean operation.
- `binary` as this type is considered to be a completely opaque type conversion to any other type is prohibited. Furthermore no type can be converted to binary.
- `string` conversion to string is done exclusively carried out by formatters provided by the IO library.

The library distinguishes between two kinds of type conversion

unchecked conversion the conversion can be done without checking the value

checked conversion the value has to be checked if it fits into the target type.

Table **??** gives an overview between which types conversion is possible and whether unchecked or checked conversion will be used.

source / target	ui8	ui16	ui32	ui64	i8	i16	i32	i64	f32	f64	f128	c32	c64	c128
ui8	U	U	U	U	C	U	U	U	U	U	U	U	U	U
ui16	C	U	U	U	C	C	U	U	U	U	U	U	U	U
ui32	C	C	U	U	C	C	C	U	U	U	U	U	U	U
ui64	C	C	C	U	C	C	C	C	U	U	U	U	U	U
i8	C	C	C	C	U	U	U	U	U	U	U	U	U	U
i16	C	C	C	C	C	U	U	U	U	U	U	U	U	U
i32	C	C	C	C	C	C	U	U	U	U	U	U	U	U
i64	C	C	C	C	C	C	C	U	U	U	U	U	U	U
f32	N	N	N	N	N	N	N	N	U	U	U	U	U	U
f64	N	N	N	N	N	N	N	N	C	U	U	C	U	U
f128	N	N	N	N	N	N	N	N	C	C	U	C	C	U
c32	N	N	N	N	N	N	N	N	N	N	N	U	U	U
c64	N	N	N	N	N	N	N	N	N	N	N	C	U	U
c128	N	N	N	N	N	N	N	N	N	N	N	C	C	U

Table 4.3.: Type matrix showing between which types conversion is possible. `U` and `C` denote unchecked and checked type conversion. `N` indicates type pairs where conversion is impossible as it would violate one of the conversion policies mentioned in the text.

4.6.1. The `convert` function template

At the heart of `libpnicore`s type conversion system is the `convert` function template. The declaration of the template looks somehow like this

```
template<typename ST,typename TT> TT convert(const ST &v);
```

A value of a particular source type (denoted by the template parameter `ST`) is passed as an argument to the `convert` template. The value of this argument will then be converted to a value of the target type `TT` and returned from the function template. This function template throws two exceptions

`type_error`	in situations where the type conversion is not possible
`range_error`	where the source value does not fit into the target type

The behavior of this function can best be demonstrated examples.

```
auto f = convert<float32>(int32(5));
```

In this example a value of type `int32` is successfully converted to a value of type `float32`, while

```
auto f = convert<uint16>(float32(-5)); // throws type_error
```

leads to `type_error`. According to the conversion policies mentioned above a float value cannot be converted to an integer due to truncation issues.

```
auto f = convert<uint32>(int32(-3)); //throws range_error
```

`range_error` will be thrown as a negative value cannot be converted to an unsigned type. A similar situation would be

```
auto f = convert<uint8>(int16(10000)); //throws range_error
```

where `range_error` would indicate that it is impossible to store a value of 10000 in an 8-Bit unsigned variable.

5. Arrays

C++ has no multidimensional array type (MDA) in its standard library. However, MDAs are crucial for the developmemt of scientific applications. One of the reasons for the continuing success of languages like Fortran or Python is their excellent support for MDAs[1]. The lack of an MDA type in C++ was indeed the spark that initiated the developement of `libpnicore`. Before discussing `libpnicore`s array facilities some terminology should be defined:

element type (ET)	referes to the data type of the individual elements stored in an MDA. For MDAs this will typically be a numeric type like an integer or a floating point number.
rank r	denotes the number of dimensions of an MDA
shape $\mathbf{s}$	is a vector of dimension r whose elements are the number of elements along each dimension. The elements of $\mathbf{s}$ are denoted as s_i with $i = 0, \ldots, r-1$

The hart of `libpnicore`'s MDA support is the `mdarray` template. `mdarray` is extremely powerfull. Thus, using `mdarray` directly to define array types is not for the faint harted. To simplify the usage of multidimensional arrays the library provides three templates derived form `mdarray` which are easy to use.

`static_array`	a static arrays whose shape, rank, and element type are fixed at compile time.
`fixed_dim_array`	element type and rank are fixed at compile time but the shape can be changed at runtime.
`dynamic_array`	a fully dynamic array type where only the element type must be known at compile time. The rank as well as the shape can be altered at runtime

These types are all defined in `pni/core/arrays.hpp`. In addition to this two basic templates there are several utility classes and templates like

`array_view`	a template providing a particular view on an array (see Sec. **??**)
`array`	a type erasure that can be used with any instance of an array template (see Chapter **??**)

All array types derived from `mdarray` provide the following features

1. unary and binary arithmetics if the element type is a numeric type.
2. slicing to extract only a part of a large array
3. simple access to data elements using variadic operators.
4. all array types are full STL compliant containers and thus can be used along with STL algorithms.

[1]For Python arrays are introduced by the `numpy` package.

5.1. Array construction and inquery

Constructing arrays is rather simple by means of the `::create` function provided by the array templates. The next example shows how to create arrays using this static function

examples/array_create.cpp

```
1
2  //-----------------------------------------------------------------------
3  // basic array construction
4  //-----------------------------------------------------------------------
5  #include <vector>
6  #include <pni/core/types.hpp>
7  #include <pni/core/arrays.hpp>
8
9  using namespace pni::core;
10
11 //some usefull type definitions
12 typedef dynamic_array<float64> darray_type;
13
14 int main(int ,char **)
15 {
16     //construction from shape
17     auto a1 = darray_type::create(shape_t{1024,2048});
18
19     //construction from shape and buffer
20     auto a2 = darray_type::create(shape_t{1024,2048},
21                                   darray_type::storage_type(1024*2048));
22
23     //construction from initializer lists
24     auto a3 = darray_type::create({5},{1,2,3,4,5});
25
26     return 0;
27 }
```

In line 12 a concrete array type is defined from the `dynamic_array` utility template. The `::create` function comes in three flavors as shown in the previous example

line 17 it takes the shape of the array as a container and constructs the array from this. In this case the storage container is allocated internally.

line 20 the storage container is passed along with the shape.

line 24 here the shape and the container are passed as initializer lists - this can make the syntax more readable in some cases.

After an array has been created we may want to retrieve some of its basic properties. In the next example we do exactly this

examples/array_inquery.cpp

```

//-----------------------------------------------------------------------
// basic array inquery
//-----------------------------------------------------------------------
```

```
5  #include <vector>
6  #include <pni/core/types.hpp>
7  #include <pni/core/arrays.hpp>
8
9  using namespace pni::core;
10
11 //some usefull type definitions
12 typedef dynamic_array<float64> darray_type;
13
14 template<typename ATYPE>
15 void show_info(const ATYPE &a)
16 {
17     std::cout<<"Data type: "<<type_id(a)<<std::endl;
18     std::cout<<"Rank      : "<<a.rank()<<std::endl;
19     std::cout<<"Shape     : (";
20     auto s = a.template shape<shape_t>();
21     for(auto n: s) std::cout<<" "<<n<<" ";
22     std::cout<<")"<<std::endl;
23     std::cout<<"Size      : "<<a.size()<<std::endl;
24 }
25
26 int main(int ,char **)
27 {
28     auto a1 = darray_type::create(shape_t{1024,2048});
29
30     show_info(a1);
31
32     return 0;
33 }
```

The important part is the implementation of the `show_info` function template starting at line 14. The function template `type_id` is used in line 17 to retrieve the type ID of the arrays element type. `rank` in line 18 returns the number of dimension and `size` in line 23 the total number of elements stored in the array. The `shape` template function in line 20 returns the number of elements along each dimension stored in a user provided container type.

The content of arrays can be copied to and from containers using the standard `std::copy` template function from the STL. In addition a version of the assignment operator is provided which allows assignment of values from an initializer list. This is particularly useful for static arrays which basically do not require construction.

```
typedef .... static_array_type;

static_array_type a;

a = {1,2,3,4,5};
```

5.2. Linear access to data

As already mentioned in the first section of this chapter, the array types provided by `libpnicore` are fully STL compliant containers. They provided all the iterators required by the STL. Before we have a look on STL lets first investigate how to simply access data elements in an array

examples/array_linear_access.cpp

```
1
2  //-------------------------------------------------------------------------
3  // Linear data access
4  //-------------------------------------------------------------------------
5  #include <random>
6  #include <pni/core/types.hpp>
7  #include <pni/core/arrays.hpp>
8
9  using namespace pni::core;
10
11 typedef uint16                         channel_type;
12 typedef fixed_dim_array<channel_type,1> mca_type;
13
14 int main(int ,char **)
15 {
16     auto mca = mca_type::create(shape_t{128});
17
18     //initialize
19     std::random_device rdev;
20     std::mt19937 generator(rdev());
21     std::uniform_int_distribution<channel_type> dist(0,65000);
22
23     //generate data
24     for(auto &channel: mca)  channel = dist(generator);
25
26     //subtract some number
27     for(size_t i=0;i<mca.size();++i) if(mca[i]>=10) mca[i] -= 10;
28
29     //set the first and last element to 0
30     mca.front() = 0;
31     mca.back()  = 0;
32
33     //output data
34     for(auto channel: mca) std::cout<<channel<<std::endl;
35
36     return 0;
37 }
```

For all array types the new C++ *for-each* construction can be used as shown in lines 24 and 34. Unchecked access (no index bounds are checked) is provided via the `[]` operator as demonstrated in line 27. Finally, in cases where the index should be checked use the `at()` method like in lines 30 and 31. Some of the operations in this example can be done much more efficient with STL algorithms as demonstrated in the next example

examples/array_stl.cpp

```

//-------------------------------------------------------------------------
// using STL algorithms
```

```
4  //-----------------------------------------------------------------------------
5  #include <algorithm>
6  #include <random>
7  #include <pni/core/types.hpp>
8  #include <pni/core/arrays.hpp>
9
10 using namespace pni::core;
11
12 typedef uint16                         pixel_type;
13 typedef fixed_dim_array<pixel_type,2> image_type;
14
15 int main(int ,char **)
16 {
17     auto image = image_type::create(shape_t{1024,512});
18
19     std::fill(image.begin(),image.end(),0); //use STL std::fill to initialize
20
21     std::random_device rdev;
22     std::mt19937 generator(rdev());
23     std::uniform_int_distribution<pixel_type> dist(0,65000);
24
25     std::generate(image.begin(),image.end(),
26                   [&generator,&dist](){ return dist(generator); });
27
28     //get min and max
29     std::cout<<"Min: "<<*std::min_element(image.begin(),image.end())<<std::endl;
30     std::cout<<"Max: "<<*std::max_element(image.begin(),image.end())<<std::endl;
31     std::cout<<"Sum: ";
32     std::cout<<std::accumulate(image.begin(),image.end(),pixel_type(0));
33     std::cout<<std::endl;
34
35     return 0;
36 }
```

In line 19 `std::fill` is used to initialize the array to 0 and `std::generate` in line 25 fills it with random numbers using a lambda expression . The rest of the example should be trivial (if not, please lookup a good C++ STL reference).

5.3. Multidimensional access

Though being an important feature, linear access to multidimensional arrays is not always useful. In particular the last example where we pretended to work on image data implementing algorithms would be rather tedious if we would have had only linear access. It is natural for such objects to think in pixel coordinates (i, j) rather than the linear offset in memory. `libpnicore` provides easy multidimensional access to the data stored in an array. The next example shows how to use this feature to work only on a small region of the image data as defined in the last example

examples/array_multiindex.cpp

```

//-----------------------------------------------------------------------------
```

```
3  // using STL algorithms
4  //----------------------------------------------------------------------------
5  #include <algorithm>
6  #include <random>
7  #include <pni/core/types.hpp>
8  #include <pni/core/arrays.hpp>
9  
10 using namespace pni::core;
11 
12 typedef uint16                       pixel_type;
13 typedef std::array<size_t,2>         index_type;
14 typedef fixed_dim_array<pixel_type,2> image_type;
15 
16 int main(int ,char **)
17 {
18     auto image = image_type::create(shape_t{1024,512});
19 
20     std::fill(image.begin(),image.end(),0); //use STL std::fill to initialize
21 
22     std::random_device rdev;
23     std::mt19937 generator(rdev());
24     std::uniform_int_distribution<pixel_type> dist(0,65000);
25 
26     std::generate(image.begin(),image.end(),
27                   [&generator,&dist](){ return dist(generator); });
28 
29 
30     size_t zero_count = 0;
31     size_t max_count  = 0;
32     for(size_t i=512;i<934;++i)
33     {
34         for(size_t j=128;j<414;++j)
35         {
36             if(image(i,j) == 0) zero_count++;
37             if(image(index_type{{i,j}}) >= 10000) max_count++;
38 
39         }
40     }
41 
42     std::cout<<"Found 0 in "<<zero_count<<" pixels!"<<std::endl;
43     std::cout<<"Found max in "<<max_count<<" pixels!"<<std::endl;
44 
45     return 0;
46 }
```

The interesting part here are lines 36 and 37. You can pass the multidimensional indexes either as a variadic argument list to the () operator of the array type (as in line 36) or you can use a container like in line 37. The former approach might look a bit more familiar, however, in some cases when decisions have to made at runtime the container approach might fits better. However, passing containers reduces access performance approximately by a factor of 2. Thus, as a rule of thumb you should always use the variadic form when you know the number of dimensions the array has and containers only in those cases where this information is only available at runtime.

5.4. Array views and slicing

In the previous example multiindex access was used to do work on only a small part of the image data. `libpnicore` provides view types for arrays which would make these operations easier. Views are created by passing instances of `slice` to the `()` operator of an array type. Slices in `libpnicore` work pretty much the same as in python. Lets have a look on the following example

examples/array_view.cpp

```cpp

//-----------------------------------------------------------------------------
// using views with STL algorithms
//-----------------------------------------------------------------------------
#include <algorithm>
#include <random>
#include <pni/core/types.hpp>
#include <pni/core/arrays.hpp>

using namespace pni::core;

typedef uint16                          pixel_type;
typedef std::array<size_t,2>            index_type;
typedef fixed_dim_array<pixel_type,2> image_type;

int main(int ,char **)
{
    auto image = image_type::create(shape_t{1024,512});

    std::fill(image.begin(),image.end(),0); //use STL std::fill to initialize

    std::random_device rdev;
    std::mt19937 generator(rdev());
    std::uniform_int_distribution<pixel_type> dist(0,65000);

    std::generate(image.begin(),image.end(),
                  [&generator,&dist](){ return dist(generator); });


    auto roi = image(slice(512,934),slice(128,414));
    auto zero_count = std::count_if(roi.begin(),roi.end(),
                                    [](pixel_type &p){return p==0;});
    auto max_count  = std::count_if(roi.begin(),roi.end(),
                                    [](pixel_type &p){return p>= 10000; });

    std::cout<<"Found 0 in "<<zero_count<<" pixels!"<<std::endl;
    std::cout<<"Found max in "<<max_count<<" pixels!"<<std::endl;

    return 0;
}
```

The view is created in line 30 where the slices are passed instead of integer indices to the `()` operator. A slice selects an entire index range along a dimension. The first argument to the `slice` constructor is the starting index and the last the stop index of the range. The stop index is not included (just as it is the case with Python slices). If the `()` operator of an array is called with any of its arguments being a slice a view object is returned instead of a single value or reference to a single value. View objects are pretty much like arrays themselves. However, they do not hold data by themselves but only a reference to the original array. Like arrays they are fully STL compliant containers and thus can be used with STL algorithms as shown in lines 31 and 33.

View types can be copied and moved and thus can be stored in STL containers as shown in the next example

examples/array_view_container.cpp

```cpp

//-----------------------------------------------------------------------------
// using views with STL algorithms and containers
//-----------------------------------------------------------------------------
#include <algorithm>
#include <random>
#include <pni/core/types.hpp>
#include <pni/core/arrays.hpp>

using namespace pni::core;

typedef uint16                          pixel_type;
typedef std::array<size_t,2>            index_type;
typedef fixed_dim_array<pixel_type,2>   image_type;
typedef image_type::view_type           roi_type;
typedef std::vector<roi_type>           roi_vector;

int main(int ,char **)
{
    auto image = image_type::create(shape_t{1024,512});

    std::fill(image.begin(),image.end(),0); //use STL std::fill to initialize

    std::random_device rdev;
    std::mt19937 generator(rdev());
    std::uniform_int_distribution<pixel_type> dist(0,65000);

    std::generate(image.begin(),image.end(),
                  [&generator,&dist](){ return dist(generator); });

    roi_vector rois;
    rois.push_back(image(slice(512,934),slice(128,414)));
    rois.push_back(image(slice(0,128),slice(4,100)));
    rois.push_back(image(200,slice(450,512)));

    for(auto roi: rois)
    {
        auto zero_count = std::count_if(roi.begin(),roi.end(),
                                        [](pixel_type &p){return p==0;});
```

```
40          auto max_count  = std::count_if(roi.begin(),roi.end(),
41                                          [](pixel_type &p){return p>= 10000; });
42
43          std::cout<<std::endl;
44          std::cout<<"Found 0 in "<<zero_count<<" pixels!"<<std::endl;
45          std::cout<<"Found max in "<<max_count<<" pixels!"<<std::endl;
46          std::cout<<std::endl;
47      }
48
49      return 0;
50  }
```

Here we apply the algorithms from the previous example not to a single but to several selections in the image. As shown in lines 32 to 34 we can safely store views in a container and later iterate over it.

In general views make algorithm development much easier as we have to develop algorithms only for entire arrays. If it should be applied to only a part of an array we can use a view and pass it to the algorithm. As views expose the same interface as an array the algorithm should work on views too.

5.5. Arithmetic expressions

Array and view types fully support the common arithmetic operators `+`, `*`, `/`, and `-` in their binary and unary forms. The binary versions are implemented as expression templates avoiding the allocation of unnecessary temporary and giving the compiler more possibilities to optimize the code. Views, arrays and scalars can be mixed within all arithmetic expressions. There is nothing magical with expression templates as they work entirely transparent to the user. Just use the arithmetic expressions as you are used to

examples/array_arithmetic1.cpp

```

//-----------------------------------------------------------------------------
// using views with STL algorithms
//-----------------------------------------------------------------------------
#include <algorithm>
#include <random>
#include <pni/core/types.hpp>
#include <pni/core/arrays.hpp>

using namespace pni::core;

typedef float64                        number_type;
typedef fixed_dim_array<number_type,2> image_type;

int main(int ,char **)
{
    shape_t s{1024,512};
    auto image      = image_type::create(s);
    auto background = image_type::create(s);
    number_type exp_time = 1.234;
    number_type current  = 98.3445;
```

```
22
23      //compute the corrected image
24      image = (image-background)/exp_time/current;
25
26      return 0;
27  }
```

The important line here is 24 where arrays and scalars are mixed in an arithmetic expression. One can also mix arrays, selections, and scalars as the next examples shows

examples/array_arithmetic2.cpp

```
1
2   //---------------------------------------------------------------------------
3   // using views with STL algorithms
4   //---------------------------------------------------------------------------
5   #include <algorithm>
6   #include <random>
7   #include <pni/core/types.hpp>
8   #include <pni/core/arrays.hpp>
9
10  using namespace pni::core;
11
12  typedef float64                        number_type;
13  typedef fixed_dim_array<number_type,3> stack_type;
14  typedef fixed_dim_array<number_type,2> image_type;
15  typedef fixed_dim_array<number_type,1> data_type;
16
17  int main(int ,char **)
18  {
19      shape_t frame_shape{1024,512};
20      shape_t data_shape{100};
21      shape_t stack_shape{100,1024,512};
22      auto image_stack = stack_type::create(stack_shape);
23      auto background  = image_type::create(frame_shape);
24      auto exp_time    = data_type::create(data_shape);
25      auto current     = data_type::create(data_shape);
26
27      for(size_t i = 0;i<data_shape[0];++i)
28      {
29          std::cout<<"Processing frame "<<i<<" ..."<<std::endl;
30          auto curr_frame = image_stack(i,slice(0,1024),slice(0,512));
31          curr_frame = (curr_frame - background)/exp_time[i]/current[i];
32      }
33
34      return 0;
35  }
```

In line 30 a single image frame is selected from a stack of images and used in line 31 in an arithmetic expression. In fact, what we are doing here is, we are writing the corrected data back on the stack since `curr_frame` is just a view on the particular image in the stack.

5.6. Example: matrix-vector and matrix-matrix multiplication

In the last example matrix vector multiplications are treated. The full code can be viewed in `array_arithmetic3.cpp` in the source distribution. But lets first start with the header

examples/array_arithmetic3.cpp

```
24
25 //---------------------------------------------------------------------------
26 // basic linear algebra
27 //---------------------------------------------------------------------------
28 #include <iostream>
29 #include <algorithm>
30 #include <random>
31 #include <pni/core/types.hpp>
32 #include <pni/core/arrays.hpp>
33
34 using namespace pni::core;
35
36 // define the matrix and  vector types
37 template<typename T,size_t N> using matrix_temp = static_array<T,N,N>;
38 template<typename T,size_t N> using vector_temp = static_array<T,N>;
39
```

Besides including all required header files matrix and vector templates are defined in lines 37 and 38 using the new C++11 template aliasing.

5.6.1. Matrix vector multiplication

The implementation of the matrix vector multiplication is shown in the next block. In other words

$$\mathbf{r} = A\mathbf{v} \text{ or } r_j = A_{j,i}v_i \tag{5.1}$$

with A denoting a $N \times N$ matrix and $\mathbf{r}$ and $\mathbf{v}$ are N-dimensional vectors. In all formulas Einsteins sum convention is used.

examples/array_arithmetic3.cpp

```
//---------------------------------------------------------------------------
// matrix-vector multiplication
template<typename T,size_t N>
vector_temp<T,N> mv_mult(const matrix_temp<T,N> &m,const vector_temp<T,N> &v)
{
    vector_temp<T,N> result;

    size_t i = 0;
    for(auto &r: result)
    {
        const auto row = m(i++,slice(0,N));
        r = std::inner_product(v.begin(),v.end(),row.begin(),T(0));
    }
    return result;
}

//---------------------------------------------------------------------------
```

In line 74 we select the i-th row of the matrix and compute the inner product of the row vector and the input vector in line 75.

5.6.2. Vector matrix multiplication

The vector matrix multiplication

$$\mathbf{r} = \mathbf{v}A \text{ or } r_i = v_j A_{j,i} \tag{5.2}$$

is computed analogously

examples/array_arithmetic3.cpp

```
80  //------------------------------------------------------------------------------
81  // vector-matrix multiplication
82  template<typename T,size_t N>
83  vector_temp<T,N> mv_mult(const vector_temp<T,N> &v,const matrix_temp<T,N> &m)
84  {
85      vector_temp<T,N> result;
86
87      size_t i = 0;
88      for(auto &r: result)
89      {
90          const auto col = m(slice(0,N),i++);
91          r = std::inner_product(col.begin(),col.end(),v.begin(),T(0));
92      }
93      return result;
94  }
95
```

despite the fact that we are choosing the appropriate column instead of a row in line 90.

5.6.3. Matrix matrix multiplication

Finally we need an implementation for the matrix - matrix multiplication

$$C = AB \text{ or } C_{i,j} = A_{i,k} B_{k,j} \tag{5.3}$$

examples/array_arithmetics3.cpp

```
//------------------------------------------------------------------------------
// matrix-matrix multiplication
template<typename T,size_t N>
matrix_temp<T,N> mv_mult(const matrix_temp<T,N> &m1,const matrix_temp<T,N> &m2)
{
    matrix_temp<T,N> result;

    for(size_t i=0;i<N;++i)
    {
        for(size_t j=0;j<N;++j)
        {
            const auto row = m1(i,slice(0,N));
            const auto col = m2(slice(0,N),j);
```

```
109                 result(i,j) = std::inner_product(row.begin(),row.end(),
110                                                  col.begin(),T(0));
111             }
112         }
113         return result;
114     }
115
```

The rows and columns are selected in lines 107 and 108 respectively. Line 109 finally computes the inner product of the row and column vector.

5.6.4. Putting it all together: the main function

Finally the main program shows a simple application of these template functions.

examples/array_arithemtic3.cpp

```
//-----------------------------------------------------------------------------
// define some local types
typedef float64                     number_type;
typedef vector_temp<number_type,3> vector_type;
typedef matrix_temp<number_type,3> matrix_type;


int main(int ,char **)
{
    vector_type v;
    matrix_type m1,m2;
    m1 = {1,2,3,4,5,6,7,8,9};
    m2 = {9,8,7,6,5,4,3,2,1};
    v  = {1,2,3};

    std::cout<<"m1 = "<<std::endl<<m1<<std::endl;
    std::cout<<"m2 = "<<std::endl<<m2<<std::endl;
    std::cout<<"v  = "<<std::endl<<v<<std::endl;
    std::cout<<"m1.v = "<<std::endl<<mv_mult(m1,v)<<std::endl;
    std::cout<<"v.m1 = "<<std::endl<<mv_mult(v,m1)<<std::endl;
    std::cout<<"m1.m2 = "<<std::endl<<mv_mult(m1,m2)<<std::endl;

    return 0;
```

It is important to understand that the appropriate function is determined by the types of the arguments (vector or matrix). This is a rather nice example of how to use the typing system of C++ to add meaning to objects. For the exact implementation of the output operators please consult the full source code in `array_arithmetic3.cpp`.

The output of the program is

```
>./array_arithmetic3
m1 =
| 1 2 3 |
| 4 5 6 |
| 7 8 9 |
```

```
m2 =
| 9 8 7 |
| 6 5 4 |
| 3 2 1 |

v  =
| 1 |
| 2 |
| 3 |

m1.v =
| 14 |
| 32 |
| 50 |

v.m1 =
| 30 |
| 36 |
| 42 |

m1.m2 =
| 30 24 18 |
| 84 69 54 |
| 138 114 90 |
```

6. Type erasures

Templates are powerful tools as they allow the compiler to perform all kinds of optimizations.In addition they help to avoid virtual functions in classes and thus increase performance by avoiding call indirection through the virtual functions table. However, there are two major obstacles with templates

1. template expansion virtually always leas to code generation and this could lead to large binaries which might be a problem on small hardware architectures

2. template libraries and the applications which are using them are harder to maintain.

The last point may requires a bit of explanation. The reason why system administrators are not very happy with programs based on template libraries is that the latter ones are distributed as source code. Consequently whenever a bug is fixed in the library all programs depending on the code required recompilation. For programs using binary libraries only the library has to be updated. This is obviously much easier than recompiling all the programs depending on a library.

A reasonable solution for this problem is the use of type erasures. `libpnicore` provides three different type erasures

`value`	stores a single scalar value of a POD type
`value_ref`	stores the reference to an instance of a POD type
`array`	stores a multidimensional array type

To use type erasures include the `/pni/core/type_erasures.hpp` at the top of your source file.

6.1. The `value` type erasures

6.1.1. Construction

The `value` type erasure stores the value of a single primitive type. Whenever an instance of `value` is constructed memory is allocated large enough to store the value of a particular type.

`value` provides a default constructor. The instance produced by the default constructor holds a value of type `none`.

```
value v;
std::cout<<v.type_id()<<std::endl; //output NONE
```

Though there is not too much one can do with such a type it has the nice advantage that one can default construct an instance of type `value`. In addition a copy and a move constructor is provided. All these constructors are implicit.

The more interesting constructors are explicit. An instance of `value` can be constructed either from a variable from a particular type or from a literal as shown in this next example

```
//explicit construction from a variable
int32 n = 1000;
value v1(n);
std::cout<<v1.type_id()<<std::endl; //output INT32

//explicit construction from a literal
value v2(3.4212);
std::cout<<v2.type_id()<<std::endl; //output FLOAT64

//copy construction
value v3 = v1;
```

As mentioned earlier in this section, whenever an instance of `value` is constructed, memory is allocated to store the quantity that should be hidden in the type erasure. The default constructor would allocate memory for a `none` type with which one can do nothing useful. A typical application for type erasures would be to store primitive values of different type in a container and we would like to make the decision which type to use at runtime. For this purpose one could define a vector type like this

```
typedef std::vector<value> value_vector;
```

However, how would one initialize an instance of this vector? It would not make too much sense to use the default constructor (as we cannot pass type information). The solution to this problem is the `make_value` function which comes in two flavors. The first, as shown in the next code snippet, takes a type ID as a single argument and returns an instance of `value` of the requested type.

```
std::vector<type_id_t>  ids = get_ids();
value_vector values;

for(auto id: ids)
    values.push_back(make_value(id));
```

In addition there is a function template which serves the same purpose

```
value v = make_value<uint32>();
```

Here the type is determined by the template parameter of the function template.

6.1.2. Assignment

Copy and move assignment are provided by the `value` between two of its instances. In both situations the type of the `value` instance on the left handside of the operator changes (this is obvious). Move and copy assignment have the expected semantics.

The more interesting situation appears with assigning new values to an instance of `value`. As memory is only allocated during creation (or copy assignment) assigning a new value does not create a new instance of `value` but rather tries to perform a type conversion between the instance of `value` on the LHS of the operator and the value on the LHS.

```
value v = make_value<float32>(); //creates a value for a float32 value

v = uint16(5); //converts uint16 value to a float32 value
```

The type conversion follows the same rules as described in the section about type conversion earlier in this manual (in fact it uses this functionality). Consequently

```
value v = make_value<float64>();

v = complex32(3,4); //throws type_error
```

will throw a `type_error` exception as a complex number cannot be converted to a single float value.

6.1.3. Retrieving data

Retrieving data from an instance of `value` is done via the `as` template method like this

```
value v = ....;

auto data = v.as<uint8>();
```

The template parameter of `as` determines the data type as which the data should be retrieved. Like for value assignment the method performs a type conversion if necessary and throws `type_error` or `range_error` exceptions if the conversion is not possible or the numeric range of the requested type is too small.

Information about the type of the data stored in the `value` instance can be obtained by means of the `type_id` method.

```
value v = ...;
v.type_id();
```

6.2. The `value_ref` type erasure

The `value` type encapsulates data of an arbitrary type and has full ownership of the data. Sometimes it is more feasible to only store a reference to an already existing data item of a primitive type. If the reference should be copyable the default approach towards this problem would be to use `std::reference_wrapper`. Unfortunately, this template includes the full type information – which is what we want to get rid of when using a type erasure. `libpnicore` for this purpose provides the `value_ref` erasure. It stores a reference to an existing data item and hides all the type information. Though `value_ref` behaves quite similar to `value` there are some subtle differences originating from its nature as a reference type. Thus it is highly recommended to read this section carefully if you are planing to use `value_ref`.

6.2.1. Construction

Like `value`, `value_ref` is default constructible

```
value_ref vref;
```

allowing it to be used in STL containers. However, unlike `value` the default constructed reference points to nowhere. Every access to any of `value_ref`s methods will throw `memory_not_allocated_error` for a default constructed instance of `value_ref`. The preferred way of how to initialize `value_ref` is by passing an instance of `std::reference_wrapper` to it

```
float64 data;
value_ref data_ref(std::ref(data));
```

In addition `value_ref` is copy constructible.

6.2.2. Assignment

The most difficult operation with `value_ref` is assignment. It really depends on the right handside of the assignment operator what happens. One can do copy assignment

```
float32 temperature;
uint32  counter;
value_ref v1(std::ref(temperature)); //reference to temperature
value_ref v2(std::ref(counter));     //reference to counter

v1 = v2; //now v1 is a reference to counter too
```

which has the same semantics as the copy assignment for `std::reference_wrapper` where the reference is copied.

Another possibility is to assign the value of a primitive type to an instance of `value_ref`. In this case two things are taking place

1. the value is converted to the type of the data item the instance of `value_ref` references
2. the converted value is assigned to the referenced data item

Consider this example

```
float32 temperature;
value_ref temp_ref(std::ref(temperature));

temp_ref = uint16(12);
```

In this example the value 12 of type `uint16` is first converted to a `float32` value. This new float value is then assigned to the variable `temperature`. As always with type conversions exceptions will be thrown if the conversion fails.

One can also change the variable an instance of `value_ref` references with

```
value_ref ref = ....;     //reference to some data item
complex64 refractive_index = ...;

ref = std::ref(refractive_index);  //now reference points to refractive_index
```

Finally a value from a `value` instance can be assigned with

```
value v = int32(100);
value_ref ref = ....;

ref = v;
```

in which case type conversion from the internal type of `v` to the internal type of `ref` occurs. Exceptions are thrown if the type conversion fails.

6.2.3. Retrieving data

Data retrieval for `value_ref` works exactly the same way as for `value`. The type provides a template method `as` which can be used to get a copy of the data stored in the item referenced as an instance of a type determined by the template parameter.

```
value_ref ref = ....;

auto data = ref.as<uint32>();
```

Again, type conversion takes place from the original type of the referenced data item to the type requested by the user via the template parameter. Finally, as `value`, `value_ref` provides a `type_id` member function which returns the type ID of the referenced data item.

6.3. Type erasures for arrays

As `libpnicore` provides a virtually indefinite number of array types via its `mdarray` template the `array` type erasure is maybe one of the most important ones. Like the `value` type erasure it will take over full ownership of the array stored in it.

A good introduction into the `array` type erasure is this particular version of the array inquiry example from the previous chapter on arrays.

examples/type_erasure3.cpp

```
#include <vector>
#include <pni/core/types.hpp>
#include <pni/core/arrays.hpp>
#include <pni/core/type_erasures.hpp>

using namespace pni::core;

//some usefull type definitions
typedef dynamic_array<float64> darray_type;
typedef static_array<float64,3,3> sarray_type;
```

```
typedef fixed_dim_array<float64,2> farray_type;

void show_info(const array &a)
{
    std::cout<<"Data type: "<<type_id(a)<<std::endl;
    std::cout<<"Rank      : "<<a.rank()<<std::endl;
    std::cout<<"Shape     : (";
    auto s = a.shape<shape_t>();
    for(auto n: s) std::cout<<" "<<n<<" ";
    std::cout<<")"<<std::endl;
    std::cout<<"Size      : "<<a.size()<<std::endl;
}

int main(int ,char **)
{
    auto a1 = darray_type::create(shape_t{1024,2048});
    auto a2 = farray_type::create(shape_t{1024,2048});
    sarray_type a3;

    std::cout<<"--------------------------------"<<std::endl;
    show_info(array(a1));
    std::cout<<std::endl<<"--------------------------------"<<std::endl;
    show_info(array(a2));
    std::cout<<std::endl<<"--------------------------------"<<std::endl;
    show_info(array(a3));

    return 0;
}
```

In the previous version, where `show_info` was a template function a new version of `show_info` would have been created for each of the three array types used in this example. By using the type erasure only a single version of `show_info` is required which reduces the total code size of the binary.

The current implementation of `array` is rather limited in comparison to the `mdarray` template. Multidimensional access is not provided and only forward iteration is implemented. In addition there is now `array_ref` type erasure which only keeps a reference to an instance of `mdarray`.

The iterators themselves have a subtle speciality. They do not provide a `->` operator. This has a rather simple reason. While all other interators return a pointer to a particular data element in a container the `array` iterators cannot do this (they do no hold any type information). Instead they return an instance of `value` for constant or `value_ref` for read/write iterators. In order to keep the semantics of the `->` operator we would have to return `*value` or `*value_ref` from the `->` operator. However, this is not possible as these objects are just temporaries and would be destroyed once the operator function has returned. However, this is only a small inconvenience as it has no influence on the STL compliance of the iterator. One can still use the `foreach` construction

```
array a(...);

for(auto x: a)
    std::cout<<s<<std::endl;
```

and all STL algorithms with a `array` type erasure.

6.4. An example: reading tabular ASCII data

In this final section a typical use-case for a type erasure will be discussed. One problem that regularly pops up is to read tabular ASCII data. For this example a very simple file format has been used. The file `record.dat` has the following content

examples/record.dat

```
11   -123.23   (-1.,0.23)
13   -12.343   (12.23,-0.2)
16   134.12    (1.23,-12.23)
```

While the elements of the first two columns are integer and float respectively, the third column holds complex numbers. The task is simple: read the values from the file without losing information. This means that we do not want to truncate values (for instance float to integer) or do inappropriate type conversions (for instance convert everything to the complex type) which may add rounding errors.

There are several ways how to approach this problem. The most straight forward one would be to create a `struct` with an integer, a float, and a complex element. However, this approach is rather static. If a column will be added or removed or only the order of the columns is changed we have to alter the code.

In this example a different path has been taken. Each individual line is represented by a record type which consists of a vector whose elements are instances of the `value` type erasure.

examples/type_erasure_record.cpp

```cpp
#include <vector>
#include <iostream>
#include <fstream>
#include <pni/core/types.hpp>
#include <pni/core/type_erasures.hpp>
#include <boost/spirit/include/qi.hpp>
#include <boost/spirit/include/phoenix.hpp>

using namespace pni::core;
using namespace boost::spirit;

typedef int32                   int_type;
typedef float64                 float_type;
typedef complex64               complex_type;
typedef std::vector<value>      record_type;
```

The entire table is again a vector with `record_type` as element type. In addition we have defined a special type to store complex numbers (`complex_type`).

6.4.1. Defining the parsers

One of the key elements for this example is to use the `boost::spirit` parser framework. We define three parsers

1. one for the `complex_type`
2. one for a value which can parser integer, double, and complex numbers
3. and one for the entire record.

The `boost::spirit` framwork is indeed rather complex and requires a deep understanding of some of the additional boost libraries like `fusion` and `phoenix`. However, as we will see, it is worth to become familiar with them as will be shown here.

Add a reference for the boost documentation here

In this next snippet the definition of the complex number parser is shown.

examples/type_erasure_record.cpp

```
//--------------------------------------------------------------------------
template<typename ITERT>
struct complex_parser : public qi::grammar<ITERT,complex_type()>
{
    qi::rule<ITERT,complex_type()> complex_rule;

    complex_parser() : complex_parser::base_type(complex_rule)
    {
        using namespace boost::fusion;
        using namespace boost::phoenix;
        using qi::_1;
        using qi::_2;
        using qi::double_;

        complex_rule = ('('>>double_>>','>>double_>>')')
                         [_val = construct<complex_type>(_1,_2)];
    }
```

We assume complex numbers to be stored as tuples of the form `(real part,imaginary part)`. As we can see in the above example the complex type is assembled from the two double values matched in the rule. The next parser required is the value parser. This parser matches either an integer, a double, or a complex value. It is a good example how to reuse already existing parser in `boost::spirit`.

examples/type_erasure_record.cpp

```
//--------------------------------------------------------------------------
template<typename ITERT>
struct value_parser : public qi::grammar<ITERT,pni::core::value()>
{
    qi::rule<ITERT,pni::core::value()> value_rule;

    complex_parser<ITERT> complex_;

    value_parser() : value_parser::base_type(value_rule)
    {
        using namespace boost::fusion;
        using namespace boost::phoenix;
        using qi::_1;
        using qi::char_;
        using qi::int_;
        using qi::double_;
```

```
        using qi::_val;

        value_rule = (
                    (int_ >> !(char_('.')|char_('e')))[_val =
                    construct<pni::core::value>(_1)]
                    ||
                    double_[_val = construct<pni::core::value>(_1)]
                    ||
```

Finally we need a parser for the entire record. This is rather simple as `boost::spirit` provides a special syntax for parsers who store their results in containers.

examples/type_erasure_record.cpp

```

//-----------------------------------------------------------------------------
// parse an entire record
//-----------------------------------------------------------------------------
template<typename ITERT>
struct record_parser : public qi::grammar<ITERT,record_type()>
{
    qi::rule<ITERT,record_type()> record_rule;

    value_parser<ITERT> value_;

    record_parser() : record_parser::base_type(record_rule)
    {
        using qi::blank;
```

6.4.2. The main program

The main program is rather simple

examples/type_erasure_record.cpp

```
162 // write a single record to the output stream
163 //------------------------------------------------------------------------------
164 void write_record(std::ostream &stream,const record_type &r)
165 {
166     for(auto v: r)
167     {
168         write_value(stream,v);
169         stream<<"\t";
170     }
171     stream<<std::endl; //terminate the output with a newline
172 }
173 
174 //------------------------------------------------------------------------------
175 // write the entire table to the output stream
```

Not all the code will be explained as it is only those parts which are of interest for the `value` type erasure. The program can be divided into two parts:

1. reading the data (in line 166)

2. and writing it back to standard output (in line 172)

As the latter one is rather trivial we will only consider the reading part in this document. The output of the main function is

```
INT32
FLOAT64
COMPLEX32
11      -123.23 (-1,0.23)
13      -12.343 (12.23,-0.2)
16      134.12  (1.23,-12.23)
```

6.4.3. The reading sequence

The entry point for the read sequence is the `read_table` function.

examples/type_erasure_record.cpp

```cpp

//-----------------------------------------------------------------------------
// read an entire table from a stream
//-----------------------------------------------------------------------------
table_type read_table(std::istream &stream)
{
    table_type table;
    string line;

    while(!stream.eof())
    {
        std::getline(stream,line);
        if(!line.empty())
            table.push_back(parse_record(line));
```

The logic of this function is rather straight forward. Individual lines are written from the input stream until `EOF` and passed on to the `parse_record` function which returns an instance of `record_type`. Each record is appended to the table.

The `parse_record` function is where all the magic happens

examples/type_erasure_record.cpp

```cpp

//-----------------------------------------------------------------------------
// read a single record from the stream
//-----------------------------------------------------------------------------
record_type parse_record(const string &line)
{
    typedef string::const_iterator iterator_type;
    typedef record_parser<iterator_type> parser_type;

    parser_type parser;
    record_type record;

```

The definition of this function pretty much demonstrates the power of the `boost::spirit` library. All the nasty parsing work is done by the code provided by `boost::spirit`. The only thing left to do is provide iterators to the beginning and end of the line.

7. Program configuration utilities

One of the most tedious tasks when writing applications is how to handle its configuration. There are several possibilities how a user can tell a program about input arguments and parameters

- via command line options and arguments
- via environment variables of the calling shell
- and via configuration files.

It is important to note that this is information the program has only read-only access too. `libpnicore` provides some simple functions to make programs aware of command line options and arguments as well as of configuration files. The facility provided by `libpnicore` is based on the `boost::program_options` library. However, its interface is much easier to use. What makes this facility so interesting for scientific applications is the fact that every option or argument can be given a distinct data type.

7.1. Command line options and arguments

One possibility to provide information to a program is by means of command line arguments and options which are passed to the program when called from a shell by the user. This is particularly true for Unix systems where many programs are called via the command line rather than via a Desktop.

In this section we will deal with all the basics required to use `libpnicore`s configuration facilities. Read this in any case even if you only want to use a configuration file as many of the things written is true also for configuration files.

7.1.1. Unix conventions

In the Unix world we have to distinguish between command line *arguments* and *options*. The former ones are strings which can typically appear anywhere in the command calling the program from the shell. The meaning of a particular option depends only on its position within the command used to run a program. The latter ones, *options*, are introduced by special tokens in the calling command. This token determines the meaning of a particular option. Thus options can appear in any order after the name of the program. To get a better feeling about arguments and options lets have a look on a typical command line call on a Unix or Linux system

```
> program -oresult.dat --wavelength=1.234 in1.dat in2.dat in3.dat
```

The three input files at the end of the call (`in1.dat`, `in2.dat`, and `in3.dat`) are passed as arguments. They can appear in any order and there is no way to distinguish one from the other and will be processed in the order of their appearance. This is in contrast to `result.dat` and `1.234` which are passed as options. An option can be identified by a *short name* (by convention this must be a single character) as it is the case for `result.dat` or by a *long name*, like for `1.234`, which must be a string without whitespaces. Short names are are introduced by a single '`-`' and followed immediately by the value of the option. Long names start with '`--`' followed by a '`=`' and the value of the option. An option can have both, a long and a short name. While short names are typically used when a program is called interactively by a user to minimize the typing effort, long names are mostly used when a program is called from a script. As they are not limited to a single character long names can be chosen much more descriptive than short names. Being restricted to a single character short names are sometimes only used for options which are frequently used in interactive calls while scarcely used options have only a long name.

7.1.2. Creating a simple program configuration

Creating a configuration for a program involves three classes

- `configuration` which, in the end, will hold the configuration data and provide access to it
- `config_option` describing a single command line option
- `config_argument` which we will use to describe command line arguments

Lets start with the above example. The source code of the program would maybe look like this

```
#include <vector>
#include <pni/core/types.hpp>
#include <pni/core/config/configuration.hpp>
#include <pni/core/config/config_parser.hpp>

using namespace pni::core;

typedef std::vector<string> input_files;

int main(int argc,char **argv)
{
    configuration config;
    config.add_option(config_option<string>("output","o","output file"));
    config.add_option(config_option<float64>("wavelength","w"));
    config.add_argument(config_argument<input_files>("input",-1));

    parse(config,cliargs2vector(argc,argv));

    return 0;
}
```

7.2. Configuration files

Configuration files handled by `libpnicore` follow the `INI`-file syntax as used by Windows. An example file would look like this

```
#experiment.cfg
[beam]
wavelength = 1.543
divergence = 0.12
diameter   = 1.23

[sample]
name = sample1
description = first sample in the series
```

To read this file create the following configuration in the program

```
#include <pni/core/types.hpp>
#include <pni/core/configuration.hpp>
#include <pni/core/config_parser.hpp>

using namespace pni::core;

int main(int argc,char **argv)
{
    configuration config;

    config.add_option(config_option<float64>("beam.wavelength",""));
    config.add_option(config_option<float64>("beam.divergence",""));
    config.add_option(config_option<float64>("beam.diameter",""));
    config.add_option(config_option<string>("sample.name",""));
    config.add_option(config_option<string>("sample.description",""));

    parse(config,"experiment.cfg");

    //use the options
}
```

7.3. Using configuration files and command line options together

8. Benchmark utilities

A. Benchmark results

To check the overall performance of the `mdarray` template provided by the library benchmark programs have been written whose results will be presented in this chapter. Three particular aspects are investigated by the benchmarks

- linear data access via iterators
- data access via multidimensional indexes
- performance of the arithmetic operators

To keep the number of benchmark results within reasonable bounds all benchmarks have been performend with the three predefined specializations of the `mdarray` template: `dynamic_array`, `fixed_dim_array`, and `static_array`. In addition to the plain array templates also their view types have been taken into account. The view types are interesting as they add some additional code which may cause some overhead.

Its (presumed) outstanding performance is the reason why so much scientific software is written in C. In order to show that the code provided by `libpnicore` can be used in high performance applications all benchmarks are normalized to the runtime of equivalient C code. In most situations this means that data access is done via simple pointers.

A.1. Iterator benchmarks

The essential loops whose runtime is measured for this benchmark is shown in Listings 1 and 2. It should be mentioned that for the pointer code only the loop is measured without the time required for allocating memory.

The benchmark results are summarized in Tab. A.1. All numbers in this table are normalized to the raw pointer performance and thus reflect directly any performance penalty or advantage over direct pointer access. Table A.1 shows a small performance penalty of 2 to 4 % for the `dynamic_array`. For `fixed_dim_array` and `static_array` iterator access is as fast as accessing the data via a pointer. In all cases iterating over a view shows significant performance penalties. Using iterators on views is about 2 up to 3 times slower than accessing the data via a pointer. This is simply due to additional overhead the view template introduces.

array type	iterator (r/w)	view iterator (r/w)
`dynamic_array`	1.02/1.04	2.50/2.96
`fixed_dim_array`	0.99/1.00	2.52/2.89
`static_array`	1.00/1.00	2.07/2.40

Table A.1.: Results for the iterator benchmark. r and w denote reading and writing results respectively.

```
for(auto &x: data)
    x = buffer;
```

```
T *data = new T[N];

for(size_t i=0;i<N;++i)
    data[i] = buffer;
```

Listing 1: The left snippet shows the core of the iterator writing benchmark and the right one the equivalent code using plain pointers.

```
for(auto x: data)
    buffer = x;
```

```
T *data = new T[N];

for(size_t i=0;i<N;++i)
    buffer = data[i];
```

Listing 2: The left snippet shows the core of the iterator reading benchmark and the right one the equivalent code using plain pointers.

A.2. Multidimensional index access

One of the major goals for `libpnicore` was to provide an array type which is as easy and intuitive to use as the multidimensional array types provided by Fortran or the numpy Python package. This includes easy access to array elements using a multidimensional index which can be passed either as a variadic list of integers or as a container of an integer type. This immediately raises the question how fast data access via multidimensional indices is in comparison with simple pointer access where the linear offset is computed from the multidimensional index and the number of elements along each dimension of the array.

The results for the benchmark are shown in Tab. A.2. It follows immediately from this table that passing the multidimensional index as a variadic argument list is the fastest way of how to access the data. The performance is virtually equal to those of using direct pointer access. Using the container types `std::vector` or `std::array` to pass the index will cause in a performance penalty of 200 to 300 % for virtually all array types. As with linear access via iterators there is a significant performance penalty when accessing data via a view. One surprising aspect of the results shown in Tab. A.2 is the fact that at least for `fixed_dim_array` and `static_array` variadic access outperforms even pointer access.

The reason for the huge performance penalties is yet unclear. However, we hope that they can be reduced in further releases.

array type	variadic (r/w)	vector (r/w)	array (r/w)
`dynamic_array`	1.03/1.02	3.82/4.63	3.33/3.76
`dynamci_array`-view	3.39/6.31	6.29/6.49	4.23/5.41
`fixed_dim_array`	0.94/0.97	3.27/3.79	3.29/3.53
`fixed_dim_array`-view	3.27/6.04	5.02/6.26	4.03/5.15
`static_array`	0.97/0.97	3.08/3.75	3.25/3.67
`static_array`-view	2.78/3.58	4.66/5.92	4.83/5.33

Table A.2.: Results for the muldimensional index access benchmarks. For array views a significant overhead is added. This makes views at the current state of development rather useless for high performance applications.

```
for(size_t i=0;i<nx;++i)
    for(size_t j=0;j<ny;++j)
        data(i,j) = buffer;
```

variadic index

```
for(index[0]=0;index[0]<nx;++index[0])
    for(index[1]=0;index[1]<ny;++index[1])
        data(index) = buffer;
```

vector/array index

```
for(size_t i=0;i<nx;++i)
    for(size_t j=0;j<ny;++j)
        data[i*ny+j] = buffer;
```

pointer access

Listing 3: Basic loop constructions measured for the multiindex write benchmarks. The code for reading is basically the same - just flip the RHS and LHS of the assignment operator.

operation	`dynamic_array`	`fixed_dim_array`
$a* = b$	1.00	1.00
$a* = s$	0.77	0.77
$a/ = b$	1.00	1.00
$a/ = s$	1.00	1.00
$a+ = b$	1.00	1.00
$a+ = s$	0.77	0.77
$a- = b$	1.00	1.00
$a- = s$	0.77	0.77

Table A.3.: Results for the unary arithmetic benchmarks. Both benchmarked types show rather similar performance. It is interesting, however, that in some cases the array types seem to outperform the pointer implementation.

Finally Listing 3 shows the basic code that has been measured for this benchmark (in this particular case for writing data). The code used for reading data is virtually the same just flip the RHS and the LHS arguments of the assignment operator.

A.3. Arithmetics

Last but not least a feasible array type has to provide arithmetic operators of reasonable performance. This last section compares the unary and binary arithmetic operators for the `mdarray` specializations. Unlike for the other benchmarks the arithmetic benchmarks cover only the `dynamic_array` and `fixed_dim_array` specializations of `mdarray`.

A.3.1. Unary arithmetics

`libpnicores` `mdarray` template provides the following unary arithmetic operators

- `+=` unary addition
- `-=` unary subtraction
- `*=` unary multiplication

operation	dynamic_array	fixed_dim_array	Fortran
$a + b$	1.04	1.00	2.18
$a - b$	1.02	1.00	2.18
$a \times b$	1.05	1.00	2.24
a/b	1.02	1.00	1.46
$a \times b + \frac{d-e}{f}$	1.04	1.00	1.49

Table A.4.: Results for the binary arithmetic benchmarks normalized to the raw pointer implementation of the operations.

- `/=` unary division

where the operations are applied element-wise on the LHS of the operator. All operators accept either an array type or a scalar type as their RHS. The results for the benchmark are shown in Tab. A.3. As can be obtained from Tab. A.3 the unary arithmetic operations are as fast as their equivalent implementations using simple pointer access. Indeed in some cases the operators are faster than the pointer approach. The s and b on the RHS of the operator in Tab. A.3 denote scalar and array arguments on the RHS of the operator respectively.

A.3.2. Binary arithmetics

As already mentioned binary arithmetic operations are implemented with expression templates. Though the reference for the binary benchmarks are still their equivalent C expressions Fortran has also been included in the benchmark. This is in so far of importance as Fortran is, until today, considered the ultimate language for numerics. The results for the binary arithmetic benchmarks are shown in Tab. A.4. The first conclusion which can be drawn from this table is the fact that `dynamic_array` shows an up to 5 % performance penalty over C code while `fixed_dim_array` is virtually as fast as the C implementation of the tested operations. The most astonishing result is, however, the rather low performance of Fortran not only in comparison with the C++ types but also with respect to the C implementation of the operations (as can be seen from the last column of Tab. A.4).

The reason for the bad performance is not yet clear. It might be due to the poor quality of the compiler (we only tested with `gfortran` from the GNU compiler collection). Thus the tests should be repeated using for instance Intel's compiler suite. On the other handside: the GNU compiler collection is the most important for our uses which makes the results for this set of compilers the most relevant. Another reason might be that the benchmark code is written in C++ and the Fortran functions are linked into the C++ code statically. It may be possible that this has some negative effect on the performance of the Fortran code. Whatever might be the reason far the bad Fortran results, a single conclusion can be drawn from this benchmark: Expression templates are a very sensible way to implement operators in C++ and may can help to push C++ in the field of scientific computing.

Bibliography

[1] BOOST C++ libraries, . URL http://www.boost.org/. Online; Apr. 2014.

[2] CMAKE build system, . URL http://www.cmake.org/. Online; Apr. 2014.

[3] Wiki page with C++11 information, . URL http://en.wikipedia.org/wiki/C++11. Online; Apr. 2014.

[4] C++ unit testing framework, . URL http://sourceforge.net/apps/mediawiki/cppunit/index.php?title=Main_Page. Online; Apr. 2014.

[5] Doxygen documentation generator, . URL http://www.stack.nl/~dimitri/doxygen/index.html. Online; Apr. 2014.

[6] Utility to manage libraries on linux and windows, . URL http://www.freedesktop.org/wiki/Software/pkg-config/. Online; Apr. 2014.