The Julia language and C++

The perfect marriage?

C++Now 2018

[Keno Fischer (https://github.com/Keno)](https://github.com/Keno), [Bart Janssens (https://github.com/barche)](https://github.com/barche)

Short bio: Bart

Work: Royal Military Academy (http://www.rma.ac.be) of Belgium

C++:

- K-3D (http://www.k-3d.org): BoostCon 2007 presentation on Boost.Python (http://www.k-3d.org/k3d_wiki/images/8/8b/Extreme_Object_Models_Using_Boost.Python.pdf) by previous maintanter Tim Shead
- Coolfluid3 (https://coolfluid.github.io): C++Now 2013 presentation on using Boost.Proto with Eigen (https://github.com/boostcon/cppnow_presentations_2013/blob/master/fri/proto-eigen-fem.pdf?raw=true)

C++ and Julia:

- CxxWrap.jl (https://github.com/JuliaInterop/CxxWrap.jl)
- QML.jl (https://github.com/barche/QML.jl)
- Trilinos.jl (https://github.com/barche/Trilinos.jl)

Outline

- Why Julia?
- Julia intro: a C++ programmer's perspective
- CxxWrap.jl
- Cxx.jl (https://github.com/Keno/Cxx.jl)
- Code repository: https://github.com/barche/cppnow2018-julia (https://github.com/barche/cppnow2018-julia)

About Julia

Why another language?

- Solve the "two language problem":
  - Prototype in a simple language
  - Write production code in a fast language

So what is it?

- High-level programming language for scientific computing
- Dynamic language
- Strongly typed, with user-defined types and generics
- JIT-compiled using LLVM
- Native interface to C
- Central concept: **(dynamic) multiple dispatch**

A simple function

In [1]:
```
function add(a,b)
    return a + b
end
```

Out[1]:
```
add (generic function with 1 method)
```

Shorter version:

In [2]:
```
add(a,b) = a+b
```

Out[2]:
```
add (generic function with 1 method)
```

In [3]:
```
add(1,2)
```

Out[3]:
```
3
```

In [4]:
```
add(1.0, 2.0)
```

Out[4]:
```
3.0
```

Where are the types?

In [5]:
```
typeof(1)
```
Out[5]:
```
Int64
```
In [6]:
```
typeof(add(1,2))
```
Out[6]:
```
Int64
```
In [7]:
```
typeof(add(1.0,2))
```
Out[7]:
```
Float64
```

A look at the assembly

Each combination of types for the arguments compiles a different version of the code.

In [8]:
```
@code_native add(1,2)
```

```
	.section	__TEXT,__text,regular,pure_instructions
Filename: In[2]
	pushq	%rbp
	movq	%rsp, %rbp
Source line: 1
	leaq	(%rdi,%rsi), %rax
	popq	%rbp
	retq
	nopw	(%rax,%rax)
```

In [9]:
```
@code_native add(1.0,2.0)
```

```
	.section	__TEXT,__text,regular,pure_instructions
Filename: In[2]
	pushq	%rbp
	movq	%rsp, %rbp
Source line: 1
	addsd	%xmm1, %xmm0
	popq	%rbp
	retq
	nopw	(%rax,%rax)
```

Equivalence with C++

The Julia function `add(a,b) = a+b` is equivalent to the following C++ function:

```
template<typename A, typename B>
auto add(A a, B b) -> decltype(a + b)
{
    return a + b;
}
```

- Template function valid for any type `A` and `B`
- C++ compiles a new version for every combination of types
- Automatic (static) computation of the return type (`decltype` annotation)

Types

In [10]:
```
# Define a type
struct MyNumber
    n::Int
end
```

In [11]:
```
# Create an instance
const mynum = MyNumber(2)
```

Out[11]:
```
MyNumber(2)
```

What about our `add` function?

In [12]:

```
add(mynum, 2)
```

```
MethodError: no method matching +(::MyNumber, ::Int64)
Closest candidates are:
  +(::Any, ::Any, ::Any, ::Any...) at operators.jl:424
  +(::Complex{Bool}, ::Real) at complex.jl:247
  +(::Char, ::Integer) at char.jl:40
  ...

Stacktrace:
 [1] add(::MyNumber, ::Int64) at ./In[2]:1
 [2] include_string(::String, ::String) at ./loading.jl:522
```

In [13]:

```
add(a::MyNumber, b) = MyNumber(a.n + b)
```

Out[13]:

```
add (generic function with 2 methods)
```

In [14]:
```
methods(add)
```

Out[14]:

2 methods for generic function **add**:

- add(a::**MyNumber**, b) at In[13]:1
- add(a, b) at In[2]:1

In [15]:
```
add(mynum, 2)
```

Out[15]:
```
MyNumber(4)
```

Inheritance

Julia supports single inheritance from abstract base types:

In [16]:

```
abstract type MyBase end
struct MyConcrete <: MyBase
    a::Int
end
# This kind of introspection is built-in:
supertype(MyConcrete)
```

Out[16]:

```
MyBase
```

In [17]:

```
geta(x::MyBase) = x.a
some_a = MyConcrete(3)
geta(some_a)
```

Out[17]:

```
3
```

Constness

In [18]:
```
some_a.a = 2
```

```
type MyConcrete is immutable

Stacktrace:
 [1] include_string(::String, ::String) at ./loading.jl:522
```

In [19]:
```
# Make a mutable subtype:
mutable struct MutableConcrete <: MyBase
    a::Int
end

mutable_a = MutableConcrete(4)
# Still works:
@show geta(mutable_a)
# We can change it now:
mutable_a.a = 42
geta(mutable_a)
```

```
geta(mutable_a) = 4
```

Out[19]:
```
42
```

Generic types

In [20]:

```
struct Point{T,N}
    coords::NTuple{N,T}
end
p1 = Point((1,2,3))
```

Out[20]:

```
Point{Int64,3}((1, 2, 3))
```

In [21]:

```
p2 = Point((2.0, 3.0))
```

Out[21]:

```
Point{Float64,2}((2.0, 3.0))
```

In [22]:

```julia
# While this is allowed in C++, it's not in Julia:
struct Combined{T,N1,N2}
    coords::NTuple{N1+N2,T}
end
```

```
MethodError: no method matching +(::TypeVar, ::TypeVar)
Closest candidates are:
  +(::Any, ::Any, ::Any, ::Any...) at operators.jl:424

Stacktrace:
 [1] include_string(::String, ::String) at ./loading.jl:522
```

Multiple dispatch

- In Julia, not writing types means any type can be used
- It still tracks the types and computes the correct return type
- Each combination of argument types yields a newly compiled version of the method
- Type computation can happen dynamically (at runtime) or statically (during JIT compilation)
- The static type computation is equivalent to the C++ version
- Deciding which function to call is **multiple dispatch** (static or dynamic)

Dynamic dispatch

Consider a heterogeneous array:

In [23]:
```
A1 = [1, 2.0, [3,4]]
```

Out[23]:
```
3-element Array{Any,1}:
 1
 2.0
  [3, 4]
```

In [24]:
```
# Take the norm of every element
norm.(A1)
```

Out[24]:
```
3-element Array{Real,1}:
 1
 2.0
 5.0
```

Dynamic dispatch

- Because every element has a different type, the `norm` function is chosen at *runtime*, depending on the actual content of the array
- In C++, this can realized using a common base class and a virtual `norm` function.

Let's add a second array:

In [25]:
```
A2 = [[5, 6.0], 7im, 8]
```

Out[25]:
```
3-element Array{Any,1}:
   [5.0, 6.0]
 0+7im
  8
```

Now adding this to the previous array:

In [26]:
```
A1 .+ A2
```

Out[26]:
```
3-element Array{Any,1}:
     [6.0, 7.0]
 2.0+7.0im
     [11, 12]
```

This performs a dynamic dispatch based on *both* arguments to +

Julia for C++ programmers: summary

	Julia	**C++**
Types	✔	✔
Inheritance	Single, abstract	Multiple, concrete
Constness	Type level	Variable level
Methods part of class	✘	✔
Generic types	✔	✔
Computed field types	not yet	✔
Introspection	✔	✘
Multiple dispatch	dynamic	static
Single dispatch	N/A	dynamic
Dot operator overload	✔	✘

CxxWrap

- Github page: https://github.com/JuliaInterop/CxxWrap.jl (https://github.com/JuliaInterop/CxxWrap.jl)
- Makes use of the native C calling ability of Julia
- Most code for wrapping C++ is written in C++, compiled with your favourite compiler
- Inspired by Boost.Python and pybind11

Calling C functions from Julia

Using `ccall`:

In [27]:
```julia
ccall(:fabs, Float64, (Float64,),-1)
```

Out[27]:
```
1.0
```

The overhead is low:

In [28]:
```
using BenchmarkTools
```

In [29]:
```
@btime ccall(:fabs, Float64, (Float64,),-1.0)
```

```
  3.969 ns (0 allocations: 0 bytes)
```

Out[29]:
```
1.0
```

In [30]:
```
@btime abs(-1.0)
```

```
  2.538 ns (0 allocations: 0 bytes)
```

Out[30]:
```
1.0
```

CxxWrap approach for functions

- First argument to `ccall` can be a function pointer
- For C-like functions: pass pointer directly
- Otherwise: pass a function pointer that has the actual arguments *and* a pointer to an `std::function` closure

Example

Code from `tutorials/cxxwrap/hello_world` (tutorials/cxxwrap/hello_world):

```
#include <iostream>
#include <jlcxx/jlcxx.hpp>

void hello() { std::cout << "hello world!" << std::endl; }

JULIA_CPP_MODULE_BEGIN(registry)
  jlcxx::Module& mod = registry.create_module("Hello");

  mod.method("hello", hello);
  mod.method("hello_lambda", [] () { std::cout << "hello lambda!" << std::endl; }
);

JULIA_CPP_MODULE_END
```

CMake:

```
project(Hello)
cmake_minimum_required(VERSION 2.8.12)

find_package(JlCxx REQUIRED)

set(CMAKE_INSTALL_RPATH "${JlCxx_DIR}/../")
set(CMAKE_INSTALL_RPATH_USE_LINK_PATH TRUE)
set(CMAKE_MACOSX_RPATH 1)
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -std=c++11")

add_library(hello SHARED hello.cpp)
target_link_libraries(hello JlCxx::cxxwrap_julia)
```

Running in Julia

In [31]:
```
# Customize this to your actual build directory
buildroot = joinpath(ENV["HOME"], "src/build/cppnow2018");
```

In [32]:
```
using CxxWrap
```

In [33]:
```
wrap_modules(joinpath(buildroot, "hello/libhello"))
```

In [34]:
```
Hello.hello()
```

```
hello world!
```

In [35]:
```
Hello.hello_lambda()
```

```
hello lambda!
```

Behind the scenes

Without the macro:

```
extern "C" void register_julia_modules(void* void_reg)
{
  jlcxx::ModuleRegistry& registry = *reinterpret_cast<jlcxx::ModuleRegistry*>(voi
d_reg);
  try
  {
    jlcxx::Module& mod = registry.create_module("Hello");
    mod.method("hello", hello);
  }
  catch (const std::runtime_error& e) { jl_error(e.what()); }
}
```

- This can be called from Julia using `ccall`
- Other C entry points in `libcxxwrap_julia`

Building a method

The method function is declared as:

```
template<typename R, typename... Args>
FunctionWrapperBase& method(const std::string& name,  std::function<R(Args...)> f
);
```

Here, the `FunctionWrapperBase` object stores all the information needed to build a method:

- The name
- Return type
- Argument types
- Pointers to regular function and `std::function`

In Julia, it looks like this:

In [36]:
```
struct CppFunctionInfo
  name::Any
  argument_types::Array{Type,1}
  reference_argument_types::Array{Type,1}
  return_type::Type
  function_pointer::Ptr{Void}
  thunk_pointer::Ptr{Void}
end
```

The function pointer

The function pointer must be directly callable by Julia. It is a pointer to `apply`:

```
template<typename R, typename... Args>
struct CallFunctor
{
  using return_type = decltype(ReturnTypeAdapter<R, Args...>()(...));

  static return_type apply(const void* functor, mapped_julia_type<Args>... args)
  {
    try
    {
      return ReturnTypeAdapter<R, Args...>()(functor, args...);
    }
    catch(const std::exception& err)
    {
      jl_error(err.what());
    }
    return return_type();
  }
};
```

Function definition wrap-up

Steps to define the Julia functions:

1. Execute the `register_julia_modules` entry point
2. Julia requests all `CppFunctionInfo` by calling the appropriate C function
3. Methods themselves are defined in Julia, wrapping `ccall` to the appropriate pointers

Difference with Python: No C interface to define methods

So how efficient is this?

Test code:

```
function half_loop(n::Array{Float64,1}, out_arr::Array{Float64,1})
  test_length = length(n)
  for i in 1:test_length
        out_arr[i] = half_x(n[i])
  end
end
```

```
mod.method("half_loop_cpp",
  [](jlcxx::ArrayRef<double> in, jlcxx::ArrayRef<double> out)
  {
    std::transform(in.begin(), in.end(), out.begin(), [](const double d) { return
0.5*d; });
  });
```

Results

Loop over 50 M elements:

	Time (s)
Julia	0.079545
C++	0.081665
ccall	0.145599
cxxwrap, c function	0.143931
cxxwrap, lambda	0.277089
cfunction from C++	0.521564

Exposing types

```
class A
{
public:
  A(int value) : m_value(value) {}

  virtual int get_a() { return m_value; }
private:
  int m_value;
};

JULIA_CPP_MODULE_BEGIN(registry)
  jlcxx::Module& mod = registry.create_module("Types");

  mod.add_type<A>("A")
    .constructor<int>()
    .method("get_a",&A::get_a);

JULIA_CPP_MODULE_END
```

In [37]:
```
# Load the module
wrap_modules(joinpath(buildroot, "types/libtypes"))
```

In [38]:
```
a = Types.A(42)
```

Out[38]:
```
AAllocated(Ptr{Void} @0x00007f90a1795900)
```

In [39]:
```
Types.get_a(a)
```

Out[39]:
```
42
```

Memory management

3 types are created for `A`:

```
abstract type A <: CxxWrap.CppAny end

struct ARef <: A
  cpp_object::Ptr{Void}
end

mutable struct AAllocated <: A
  cpp_object::Ptr{Void}
end
```

- The `Allocated` type runs the destructor upon garbage collect (typically returned by constructors).
- `A` itself is abstract...

Inheritance

```
class TwiceA : public A
{
public:
  TwiceA(int value) : A(value) {}
  virtual int get_a() { return 2*A::get_a(); }
};

mod.add_type<TwiceA>("TwiceA", jlcxx::julia_type<A>())
    .constructor<int>();
```

In [40]:
```
a2 = Types.TwiceA(42)
```
Out[40]:
```
TwiceAAllocated(Ptr{Void} @0x00007f90a176a450)
```
In [41]:
```
Types.get_a(a2)
```
Out[41]:
```
84
```
In [42]:
```
# Only one Julia method exists:
methods(Types.get_a)
```
Out[42]:

1 method for generic function **get_a**:

- get_a(arg1::**Union{CxxWrap.SmartPointer{T2}, T2} where T2<:A**)

In [43]:
```
supertype(Types.TwiceA)
```
Out[43]:
```
A
```

Smart pointers

Some standard library smart pointers are automatically supported:

```
mod.method("smartA", [] (int x) { return std::make_shared<A>(x); } );
```

In [44]:
```
smart_a = Types.smartA(1)
```

Out[44]:
```
CxxWrap.SmartPointerWithDeref{A,0x0000000121cc0870,Ptr{Void} @0x0000000121caf7
00,Ptr{Void} @0x0000000121caf730,Ptr{Void} @0x0000000121caf750}(Ptr{Void} @0x0
0007f90a17b45b0)
```

In [45]:
```
Types.get_a(smart_a)
```

Out[45]:
```
1
```

Generic types

The class to wrap:

```
template<typename T1, typename T2>
class SimplePair
{
public:
  typedef T1 x_type;
  typedef T2 y_type;

  SimplePair(const T1 x, const T2 y) : m_x(x), m_y(y) {}
  T1 get_x() const { return m_x; }
  T2 get_y() const { return m_y; }
private:
  T1 m_x;
  T2 m_y;
};
```

Wrapping code for `bool`:

```
auto pair_type = mod.add_type<jlcxx::Parametric<jlcxx::TypeVar<1>, jlcxx::TypeVar
<2>>>("SimplePair");
  // Apply just for bool
  pair_type.apply<SimplePair<bool,bool>>([](auto wrapped)
  {
    typedef typename decltype(wrapped)::type WrappedT;
    typedef typename WrappedT::x_type x_type;
    typedef typename WrappedT::y_type y_type;
    wrapped.template constructor<x_type,y_type>();
    wrapped.method("get_x", &WrappedT::get_x);
    wrapped.method("get_y", &WrappedT::get_y);
  });
```

Automatic combination of types:

```
pair_type.apply_combination<SimplePair,
                            jlcxx::ParameterList<int, float, double>,
                            jlcxx::ParameterList<int, float, double>>
  ([](auto wrapped)
  {
    typedef typename decltype(wrapped)::type WrappedT;
    typedef typename WrappedT::x_type x_type;
    typedef typename WrappedT::y_type y_type;
    wrapped.template constructor<x_type,y_type>();
    wrapped.method("get_x", &WrappedT::get_x);
    wrapped.method("get_y", &WrappedT::get_y);
  });
```

Testing in Julia

In [46]:
```
Types.SimplePair(x::T1,y::T2) where {T1,T2} = Types.SimplePair{T1,T2}(x,y)
```

In [47]:
```
p1 = Types.SimplePair(false,false)
```

Out[47]:
```
SimplePairAllocated{Bool,Bool}(Ptr{Void} @0x00007f90a1791d40)
```

In [48]:
```
Types.get_x(p1)
```

Out[48]:
```
false
```

In [49]:

```
methods(Types.get_x)
```

Out[49]: 10 methods for generic function **get_x**:

- get_x(arg1::**Union{CxxWrap.SmartPointer{T2}, T2} where T2<:SimplePair{Bool,Bool}**)
- get_x(arg1::**Union{CxxWrap.SmartPointer{T2}, T2} where T2<:SimplePair{Int32,Int32}**)
- get_x(arg1::**Union{CxxWrap.SmartPointer{T2}, T2} where T2<:SimplePair{Int32,Float32}**)
- get_x(arg1::**Union{CxxWrap.SmartPointer{T2}, T2} where T2<:SimplePair{Int32,Float64}**)
- get_x(arg1::**Union{CxxWrap.SmartPointer{T2}, T2} where T2<:SimplePair{Float32,Int32}**)
- get_x(arg1::**Union{CxxWrap.SmartPointer{T2}, T2} where T2<:SimplePair{Float32,Float32}**)
- get_x(arg1::**Union{CxxWrap.SmartPointer{T2}, T2} where T2<:SimplePair{Float32,Float64}**)
- get_x(arg1::**Union{CxxWrap.SmartPointer{T2}, T2} where T2<:SimplePair{Float64,Int32}**)
- get_x(arg1::**Union{CxxWrap.SmartPointer{T2}, T2} where T2<:SimplePair{Float64,Float32}**)
- get_x(arg1::**Union{CxxWrap.SmartPointer{T2}, T2} where T2<:SimplePair{Float64,Float64}**)

In [50]:
```
p2 = Types.SimplePair(Int32(3),1.0)
```
Out[50]:
```
SimplePairAllocated{Int32,Float64}(Ptr{Void} @0x00007f90a17517c0)
```
In [51]:
```
p3 = Types.SimplePair("2", 3)
```
```
MethodError: no constructors have been defined for SimplePair{String,Int64}

Stacktrace:
 [1] SimplePair(::String, ::Int64) at ./In[46]:1
 [2] include_string(::String, ::String) at ./loading.jl:522
```

Template types with non-type parameters

```
template<typename T, int I>
class NonType
{
public:
  NonType(const T x) : m_x(x) {}
  T compute() const { return I*m_x; }
private:
  T m_x;
};
```

Wrapping code

```
mod.add_type<jlcxx::Parametric<jlcxx::TypeVar<1>, jlcxx::TypeVar<2>>>("NonType")
    .apply<NonType<double, 2>>([](auto wrapped)
  {
    typedef typename decltype(wrapped)::type WrappedT;
    wrapped.method("compute", &WrappedT::compute);
  });
```

Fails...

```
error: static_assert failed "No parameters found when applying type. Specialize j
lcxx::BuildParameterList
      for your combination of type and non-type parameters."
```

Building parameter lists

```
template<typename T>
struct BuildParameterList
{
  typedef ParameterList<> type;
};

// Match any combination of types only
template<template<typename...> class T, typename... ParametersT>
struct BuildParameterList<T<ParametersT...>>
{
  typedef ParameterList<ParametersT...> type;
};
```

Need to add:

```
template<typename T, int I>
struct BuildParameterList<NonType<T, I>>
{
typedef ParameterList<T, std::integral_constant<int, I>> type;
};
```

Finally, in Julia:

In [53]:
```
nt = Types.NonType{Float64,Int32(2)}(2)
```

Out[53]:
```
NonTypeAllocated{Float64,2}(Ptr{Void} @0x00007f90a1c66490)
```

In [54]:
```
Types.compute(nt)
```

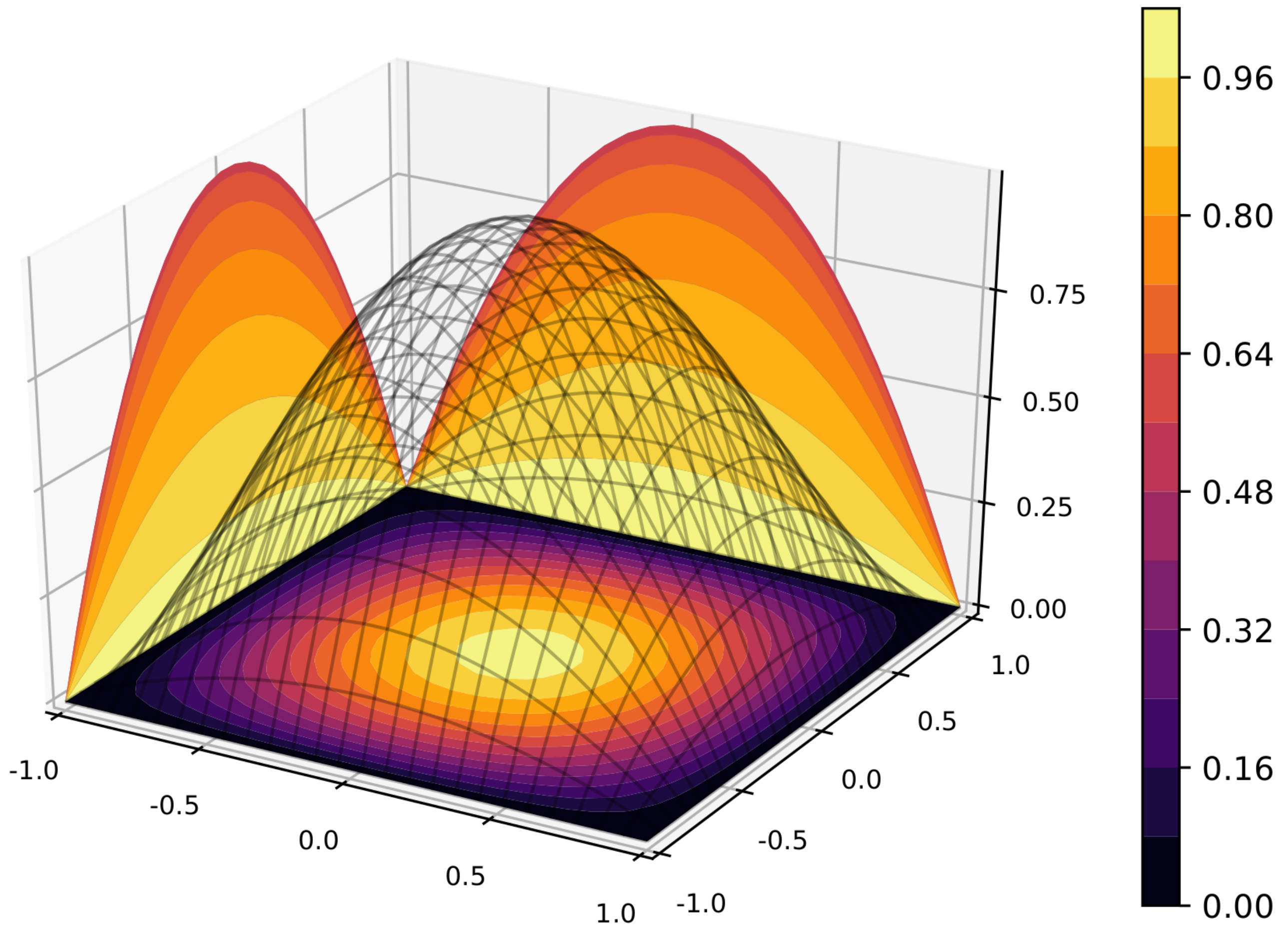
Out[54]:
```
4.0
```

Trilinos.jl (https://github.com/barche/Trilinos.jl)

- Interface to some parts of the Trilinos (https://trilinos.org/) C++ library
- Focus on the Tpetra matrix library and iterative solvers
- Uses CxxWrap.jl
- Example: 2D Laplace in C++ and Julia

Laplace example:

Solve the equation $\nabla^2\varphi + f = 0$ with $f = 2h^2((1-x^2)+(1-y^2))$ and 0 on the boundary:



C++ code

```
template<typename MatrixT>
void fill_laplace2d(MatrixT& A, const CartesianGrid& g)
{
  Teuchos::TimeMonitor local_timer(*fill_time);
  const auto& rowmap = *A.getRowMap();
  const local_t n_my_elms = rowmap.getNodeNumElements();

  // storage for the per-row values
  global_t row_indices[5] = {0,0,0,0,0};
  scalar_t row_values[5] = {4.0,-1.0,-1.0,-1.0,-1.0};

  for(local_t i = 0; i != n_my_elms; ++i)
  {
    const global_t global_row = rowmap.getGlobalElement(i);
    const local_t row_n_elems = laplace2d_indices(row_indices, global_row, g);
    row_values[0] = 4.0 - (5-row_n_elems);
    A.replaceGlobalValues(global_row, Teuchos::ArrayView<global_t>(row_indices,ro
w_n_elems), Teuchos::ArrayView<scalar_t>(row_values,row_n_elems));
  }
}
```

Julia code

```
function fill_laplace2d!(A, g::CartesianGrid)
  rowmap = Tpetra.getRowMap(A)
  n_my_elms = Tpetra.getNodeNumElements(rowmap)

  # storage for the per-row values
  row_indices = [0,0,0,0,0]
  row_values = [4.0,-1.0,-1.0,-1.0,-1.0]

  for i in 0:n_my_elms-1
    global_row = Tpetra.getGlobalElement(rowmap,i)
    row_n_elems = laplace2d_indices!(row_indices, global_row, g)
    row_values[1] = 4.0 - (5-row_n_elems)
    Tpetra.replaceGlobalValues(A, global_row, Teuchos.ArrayView(row_indices,row_n
_elems), Teuchos.ArrayView(row_values,row_n_elems))
  end
end
```

Timing comparison for 1000 x 1000 matrix (in ms)

	C++	Julia
Graph construction	190.7	115.2
Source term	41.17	32.01
Matrix filling	137.1	102.5
Dirichlet setup	0.899	0.761
Check time	41.06	29.42

Conclusions

- C++ integration with minimal overhead
- Expanding on C++ code in pure Julia is possible
- Some CxxWrap todo's:
  - Better Array wrappers
  - std containers
  - exploit Julia dot operator overload for member access