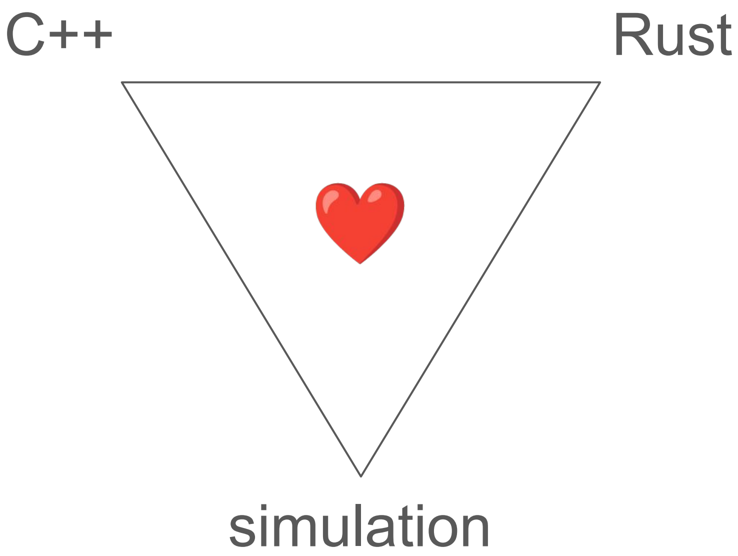
Wrapping C++ in Rust

Máté Kovács, 2024 May 14, Tokyo

A Few Facts about Me

- I love simulation.
- I love C++.
- I love Rust.

Today's Topic is Very Special to Me



"Every triangle is a love triangle if you love triangles."

– Pythagoras

Okay, Seriously…

Key Takeaways

- When the thing you need already exists, but it's written in C++.
- Rewriting it in Rust would be a waste of your time and effort.
- You're better off making the thing usable from Rust: wrapping it.
- Easier than you might think, thanks to great tools and docs!
- Don't panic! The heat problem can be solved in 6 simple steps.

Overview

- Rewrite vs Wrap: When & Why
- The Lab Rat: MFEM
- MFEM: A User's Perspective
- The Wrapper
  - Top-Level Structure
  - Rust FFI using CXX
  - Idiomatic Rust
  - Future Work

Rewrite vs Wrap

When to "Rewrite It in Rust"

- Do you understand the internals from an expert's point of view?
- Do the internals have bugs, e.g. memory safety issues?
- Are you sure that only rewriting can fix those issues?
- Are you willing to put in potentially a lot of effort and time?

When to "Wrap It in Rust"

- Did you answer "No" to any question on the previous slide?
- Do you understand the API from a user's point of view?

The Lab Rat: MFEM

What is MFEM?

- "MFEM is a free, lightweight, scalable C++ library for finite element methods." (Not sure what they mean by "lightweight"; it's absolutely monstrous.😅)
- By the Lawrence Livermore National Laboratory, USA.
- Too many features to even list here…
- Learn more at https://mfem.org.
- The official pronunciation is *em-fem*.
  (Apologies for my consisten mispronunciation…)

My Golden Week of Code

- Try to wrap MFEM in Rust.
- Target a concrete use-case: (a minimized) MFEM Example 1.
- Rely on automation, minimize handwritten C++ glue.
- Encode ownership rules into the Rust API.
- Provide wrappers to support idiomatic Rust.

MFEM: A User's Point of View

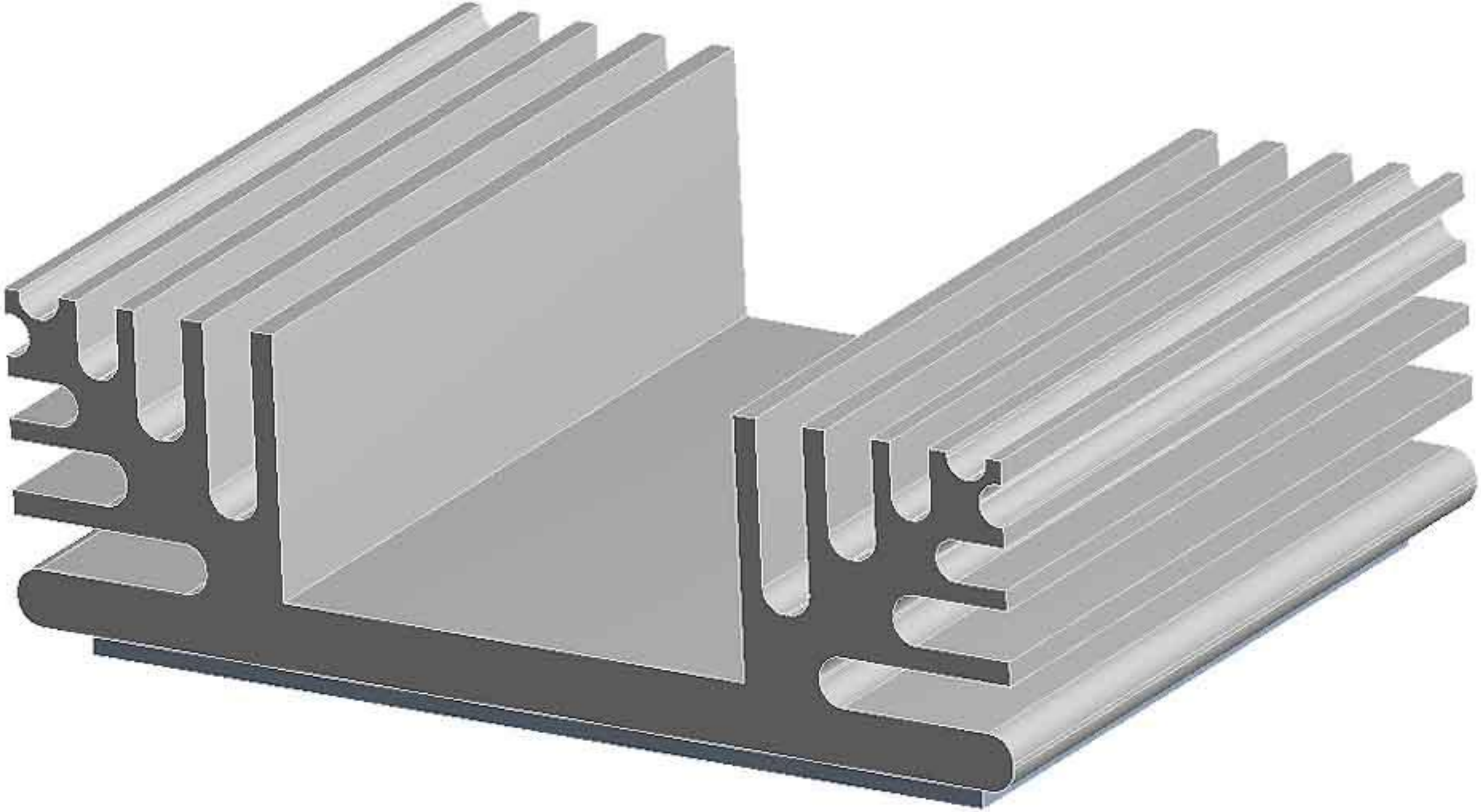
A User's Point of View

- Finite Element Method (FEM) Basics
- Key MFEM Entities and Relationships

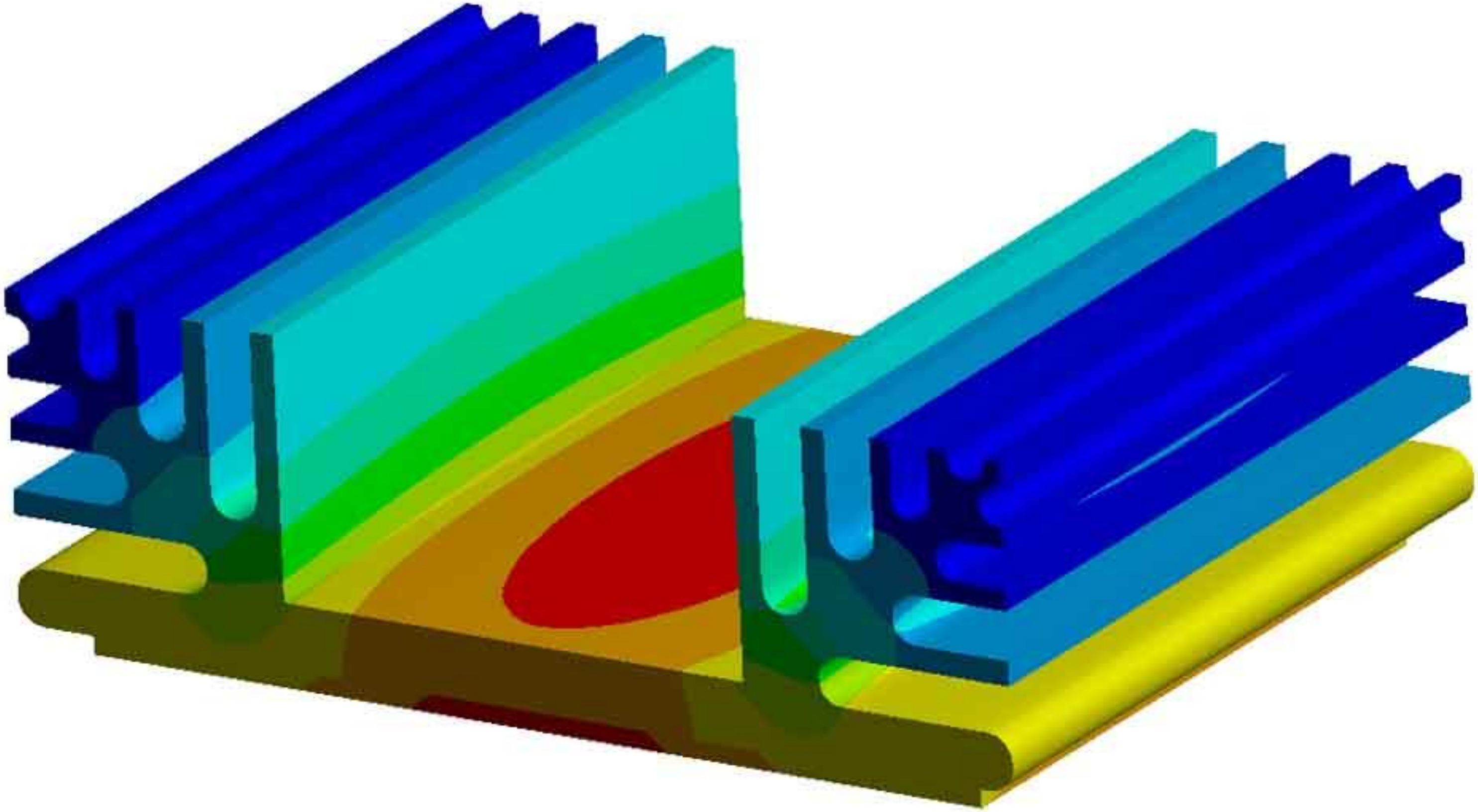
Finite Element Method: Basics

- goal: create computational models of physics
- continuous function → a finite set of variables
- functional analysis → algebra

Example: The Heat Problem



Example: The Heat Problem



Example: The Heat Problem

- $u(x, y, z)$:= temperature at point (x, y, z)
- steady-state (equilibrium) heat equation: $-k \cdot \nabla^2 u = 1$
- [UPDATED] expanded in terms of coordinates:

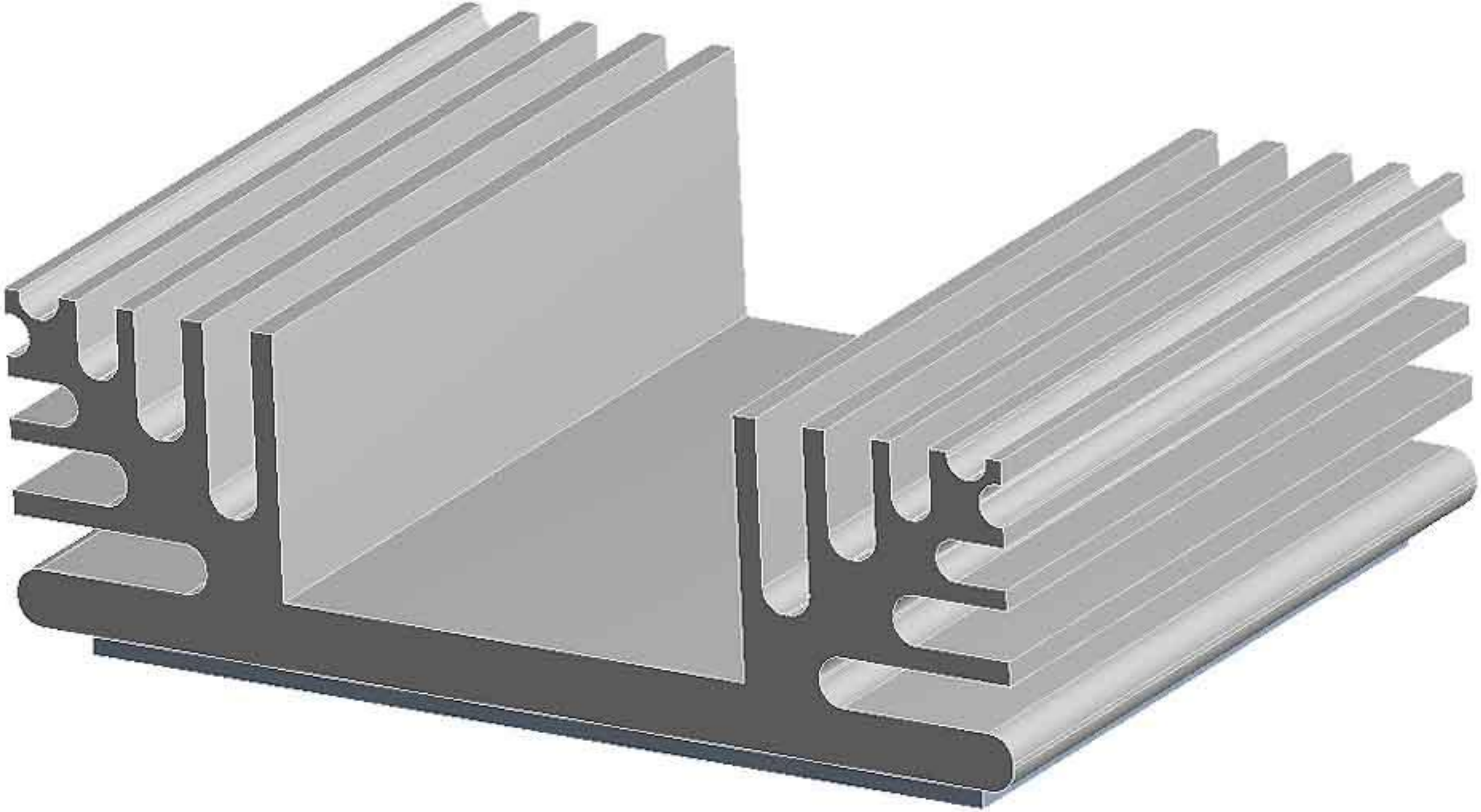
$$-k \cdot \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} \right) = 1$$

- boundary conditions: u must be zero on the surface
- goal: find u

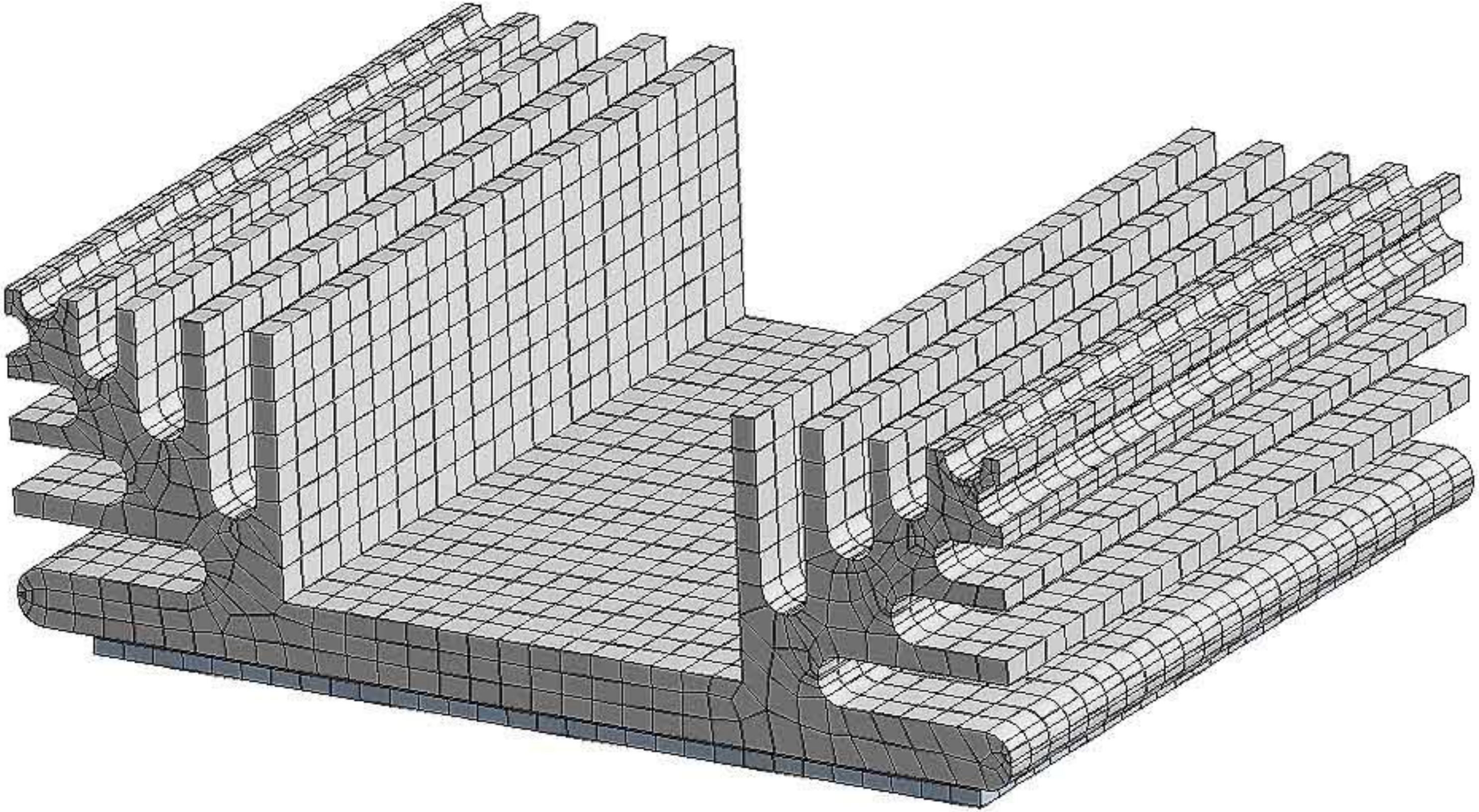
Margin Note: Continuous vs Discrete

- Physical models are analog, but our computers are digital.
- This is the gap that FEMs bridge!
- E.g. temperature (u) is a field; a function over continuous space.
- mathematics: $|\mathbb{R}| > |\mathbb{N}|$

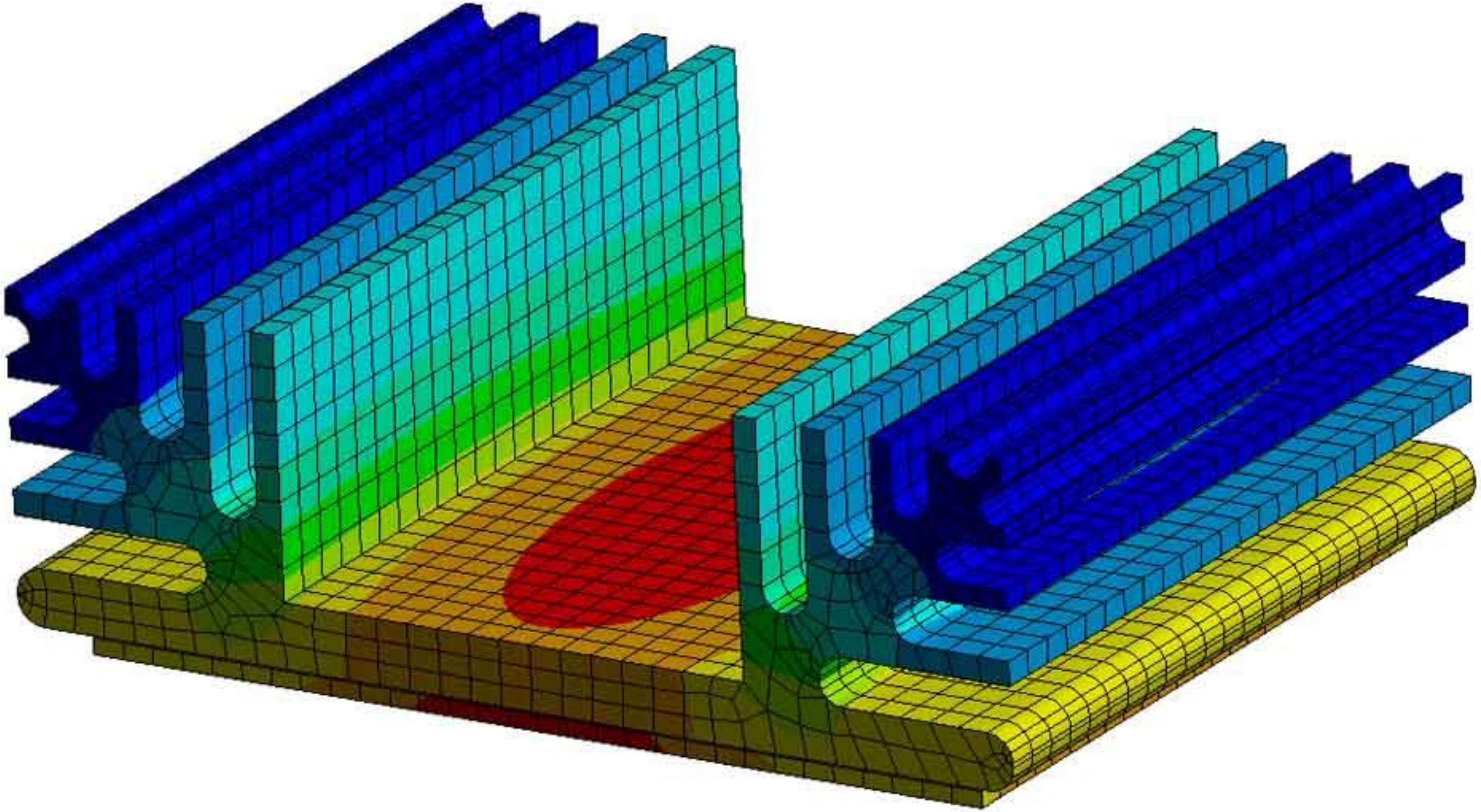
Margin Note: Continuous Domain



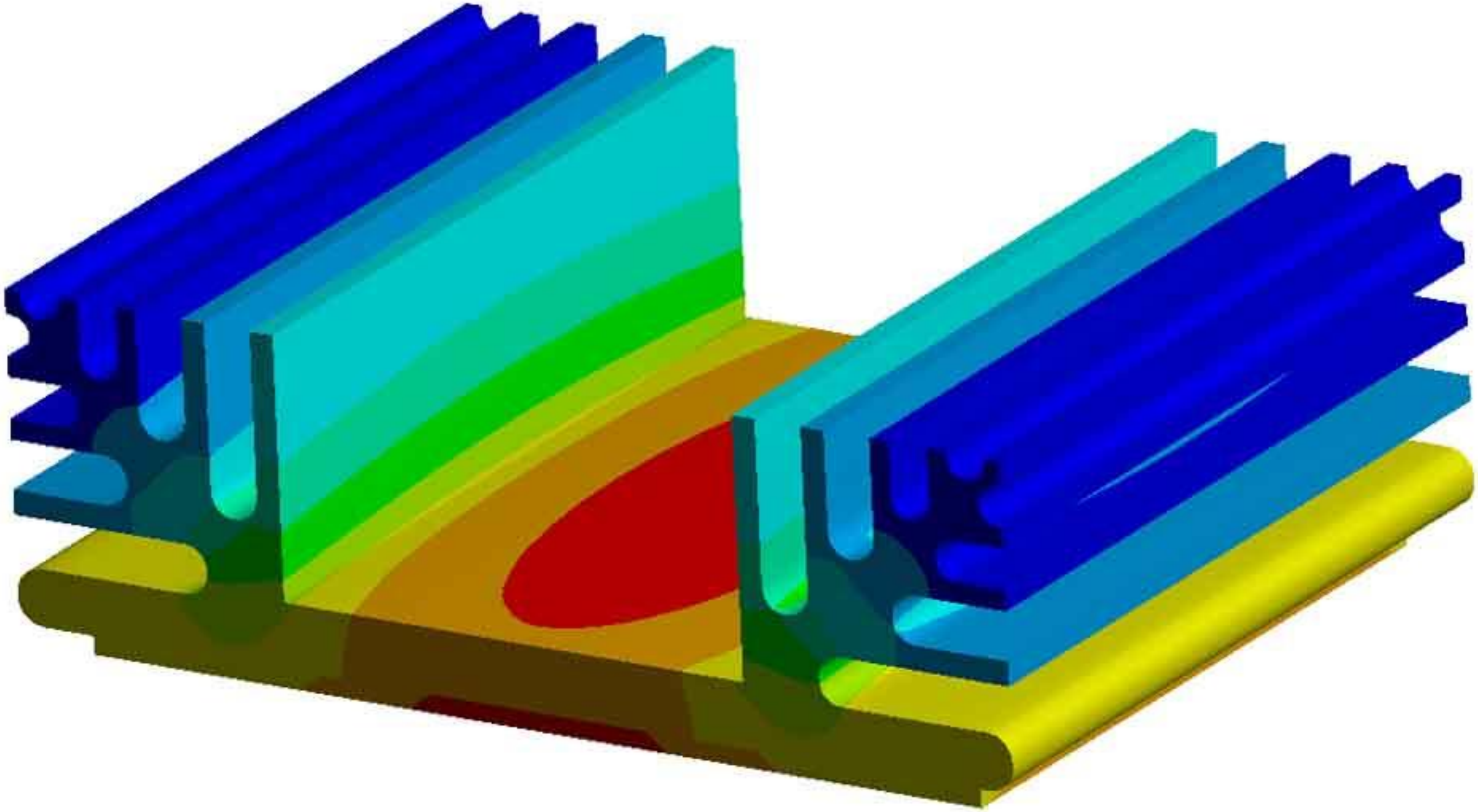
Margin Note: Discretized Domain (Mesh)



Margin Note: Discretized Function (GridFunction)



Margin Note: Continuous Function



MFEM: Key Entities

- `Coefficient`: an abstract field, i.e. a function f(x, y, z)
  - expression evaluation is lazy (i.e. on demand)
- `GridFunction`: a field discretized into an array of numbers
  - expression evaluation is eager (i.e. immediate)
- `Mesh`: a discretization of space, needed to define a `GridFunction`
- `FiniteElementCollection`: a collection of interpolating functions, needed to define a `GridFunction`
- `LinearForm`: a linear operator that maps a `GridFunction` to a number
- `BilinearForm`: a bilinear operator that takes two `GridFunctions` to a number

Quick Demo!

Back to the Heat Problem

- strong formulation: $-\nabla^2 u = f$
- weak formulation: $\int_\Omega \nabla u \cdot \nabla v \, dx = \int_\Omega f v \, dx$
- find u such that **for all v: a(u, v) = b(v)**
- bilinear form (left): a(u, v) := (∇u, ∇v)
- linear form (right): b(v) := (f, v)

MFEM: Example 1

1. Load and refine a `Mesh`.
2. Create a `FiniteElementSpace`.
3. Create a `GridFunction` and set it to an initial guess.
4. Create the `LinearForm` and `BilinearForm`.
5. Translate to and solve the set of algebraic equations.
6. Translate the solution back to a `GridFunction`, and save it.

The Wrapper

(No more integrals, I promise!)

Top-Level Structure

Top-Level Structure

- Monorepo with a multi-package `cargo` workspace.
- 3 layers, each as a separate package:
  - `mfem-cpp`

    Provides the C++ MFEM library as a `cargo` package.
  - `mfem-sys`

    Binds (via `cxx`) to `mfem-cpp` and encodes ownership rules.
  - `mfem`

    Wraps `mfem-sys` to support writing idiomatic Rust.

mfem-cpp

`mfem-cpp`: Summary

- Provides a specific version of MFEM (currently 4.6.0).
- The version of MFEM is part of the package version, e.g.

  `mfem-cpp = "0.1.0+mfem-4.6.0"`
- Has a feature called `bundled`.
  - on: Build MFEM from bundled source code.
  - off: Find (with CMake) MFEM installed on the system.
- The `lib.rs` provides `mfem_path()` to be used by `mfem-sys`.
- Setup copy-pasted from Brian Schwind's `opencascade-rs`.

`mfem-cpp`: Details

- Simple `build.rs` to run CMake (via the `cmake` crate).
- Not going into more detail here to save time…
- Please read the code if you're curious! :)

mfem-sys

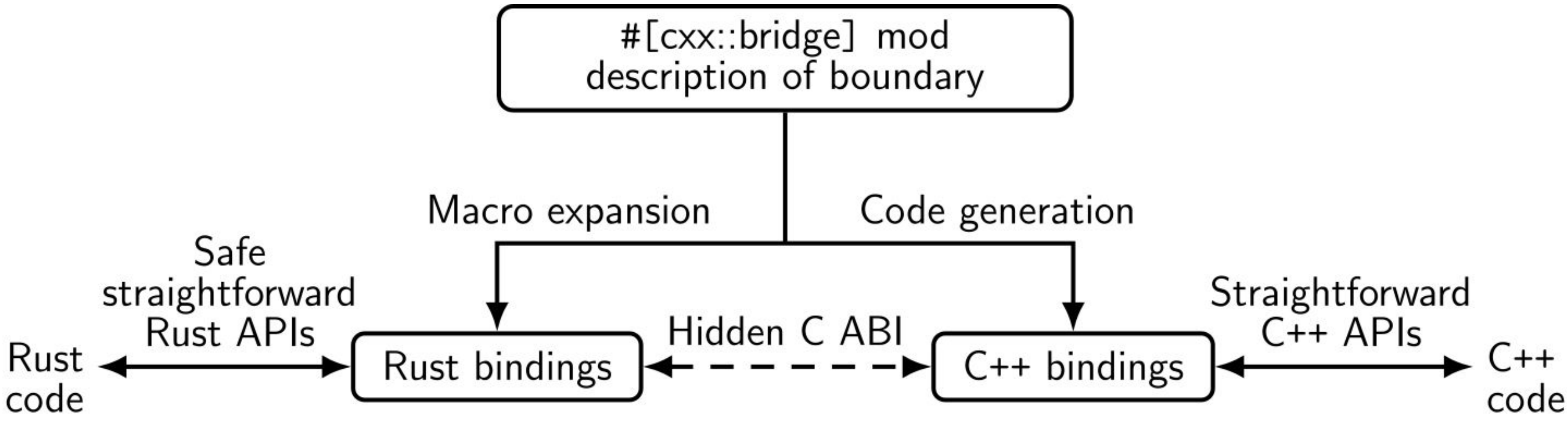
`mfem-sys`: Summary

- Depends on `mfem-cpp`.
- A safe FFI (foreign-function interface) to use MFEM from Rust.
- Uses the `cxx` crate to generate safe and correct bindings.
- Encodes MFEM's ownership rules into Rust's type system.
- Turns various MFEM `int` constants into type-safe Rust `enum`s.

`mfem-sys`: What is CXX?

- "a safe mechanism for calling C++ code from Rust [..]"
- "guides the programmer to express their language boundary [..] where Rust and C++ are semantically very similar"
- "CXX fills in the low level stuff so that you get a safe binding"

mfem-sys: What is CXX? (cont'd)



"The resulting FFI bridge operates at zero or negligible overhead, i.e. no copying, no serialization, no memory allocation, no runtime checks needed." – https://cxx.rs

mfem-sys: CXX in Code

```rust
#[cxx::bridge]
mod ffi {
    // shared types go here

    unsafe extern "C++" {
        // types and functions that bind to C++ go here
        // written in a Rust-like DSL that CXX understands
    }

    extern "Rust" {
        // Rust stuff exposed to C++ goes here
    }
}
```

`mfem-sys`: Translating Example 1

Let's follow the steps of MFEM Example 1 (`ex1.cpp`):

1. Load and refine a `Mesh`.
2. Create a `FiniteElementSpace`.
3. Create a `GridFunction` and set it to an initial guess.
4. Create the `LinearForm` and `BilinearForm`.
5. Translate to and solve the set of algebraic equations.
6. Translate the solution back to a `GridFunction`, and save it.

mfem-sys: Load and Refine a Mesh: C++

```cpp
// ex1.cpp in mfem-cpp


Mesh mesh(mesh_file, 1, 1);
int dim = mesh.Dimension();
int ref_levels =
      (int)floor(log(50000./mesh.GetNE())/log(2.)/dim);
for (int i = 0; i < ref_levels; i++) {
   mesh.UniformRefinement();
}
```

mfem-sys: Load and Refine a Mesh: Rust

```
// ex1.rs in mfem-sys
let_cxx_string!(mesh_file = args.mesh_file);
let mut mesh = Mesh_ctor_file(&mesh_file, 1, 1, true);
let dim = mesh.Dimension();
let ref_levels = f64::floor(
    f64::log2(50000.0 / mesh.GetNE() as f64) / dim as f64
) as u32;
for _ in 0..ref_levels {
    mesh.pin_mut().UniformRefinement(0);
}
```

`mfem-sys`: The "Construct-Unique" Pattern

TL;DR: Constructor-like functions must return a `UniquePtr<T>`, so they require hand-written C++ glue code.

Why: "To the extent that constructors "return" a C++ type by value, they're out of scope for this library because Rust moves (memcpy) are incompatible with C++ moves (which require a constructor to be called). Translating a constructor to `fn new() -> Self` would not be correct." – https://github.com/dtolnay/cxx/issues/221

`mfem-sys`: Mesh Type & Constructors: Rust FFI

```
// lib.rs in mfem-sys, inside unsafe extern "C++" {
type Mesh;
// Learned this pattern from Brian's opencascade-rs crate. :)
#[cxx_name = "construct_unique"]
fn Mesh_ctor() -> UniquePtr<Mesh>;


#[cxx_name = "construct_unique"]
fn Mesh_ctor_file(
    filename: &CxxString, generate_edges: i32,
    refine: i32, fix_orientation: bool) -> UniquePtr<Mesh>;
```

`mfem-sys`: Mesh Constructors: Hand-Written C++ Glue

```
// wrapper.hpp in mfem-sys


// Templated "constructor-like" function
template <typename T, typename... Args>
auto construct_unique(Args... args) -> std::unique_ptr<T> {
    return std::make_unique<T>(args...);
}
```

`mfem-sys`: Mesh Methods: Rust FFI

```rust
// lib.rs in mfem-sys, inside unsafe extern "C++" {
fn Dimension(self: &Mesh) -> i32;
fn GetNE(self: &Mesh) -> i32;
// I hope y'all still remember Pin from Eylon's talk! ;)
fn UniformRefinement(self: Pin<&mut Mesh>, ref_algo: i32);
fn Save(self: &Mesh, fname: &CxxString, precision: i32);

// these are "dirty", they have hand-written C++ glue
fn Mesh_GetNodes(mesh: &Mesh) -> Result<&GridFunction>;
fn Mesh_bdr_attributes(mesh: &Mesh) -> &ArrayInt;
```

mfem-sys: Mesh Methods: Hand-Written C++ Glue

```cpp
// wrapper.hpp in mfem-sys

// Sjors taught me this nifty pattern of
//   writing functions in modern C++. :)
auto Mesh_bdr_attributes(Mesh const& mesh) -> ArrayInt const& {
   return mesh.bdr_attributes;
}

// Now I can do this and get even closer to writing Rust :P
#define fn auto
// Okay, don't actually do this. It's bad. ;)
```

`mfem-sys`: Mesh Methods: Hand-Written C++ Glue

```cpp
// wrapper.hpp in mfem-sys


// You might be wondering how this works, given the FFI signature:
//   fn Mesh_GetNodes(mesh: &Mesh) -> Result<&GridFunction>;
auto Mesh_GetNodes(Mesh const& mesh) -> GridFunction const& {
    auto ptr = mesh.GetNodes();
    if (!ptr) {
        throw mfem_exception("Mesh::GetNodes() == nullptr");
    }
    return *ptr;
}
```

`mfem-sys`: The "TryCatch to Result" Pattern

```cpp
// wrapper.hpp in mfem-sys

class mfem_exception : public std::exception { /* ... */ };

// This is described in the CXX Guide:
// https://cxx.rs/binding/result.html
template <typename Try, typename Fail>
static void trycatch(Try &&func, Fail &&fail) noexcept try {
    func();
} catch (const std::exception &e) {
    fail(e.what());
}
```

`mfem-sys`: Create a FiniteElementSpace: C++

```cpp
// ex1.cpp in mfem-cpp
FiniteElementCollection* fec;
bool delete_fec;
if (order > 0) {
  fec = new H1_FECollection(order, dim); delete_fec = true;
} else if (mesh.GetNodes()) {
  fec = mesh.GetNodes()->OwnFEC(); delete_fec = false;
  cout << "Using isoparametric FEs: " << fec->Name() << endl;
} else {
  fec = new H1_FECollection(order = 1, dim); delete_fec = true;
}
FiniteElementSpace fespace(&mesh, fec);
```

`mfem-sys`: Create a FiniteElementSpace: Rust

```rust
// ex1.rs in mfem-sys
let owned_fec: Option<UniquePtr<H1_FECollection>> = if args.order > 0 {
    Some(H1_FECollection_ctor(
        args.order,
        dim,
        BasisType::GaussLobatto.repr,
    ))
} else if Mesh_GetNodes(&mesh).is_err() {
    Some(H1_FECollection_ctor(1, dim, BasisType::GaussLobatto.repr))
} else {
    None
};
```

mfem-sys: Create a FiniteElementSpace: Rust (cont'd)

```rust
// ex1.rs in mfem-sys
let fec = match &owned_fec {
    Some(ptr) => H1_FECollection_as_FEC(&ptr),
    None => {
        println!("Using isoparametric FEs");
        let nodes = Mesh_GetNodes(&mesh).expect("Mesh has its own nodes");
        let iso_fec = GridFunction_OwnFEC(nodes).expect("OwnFEC exists");
        iso_fec
    }
};

let fespace = FiniteElementSpace_ctor(&mesh, fec, 1, OrderingType::byNODES);
```

mfem-sys: Type-Unsafe Enums in C++

```
// H1_FECollection constructor takes 3 ints
H1_FECollection(const int p, const int dim=3,
  const int btype=BasisType::GaussLobatto);


/// Possible basis types. Note that not all elements can use all BasisType(s).
class BasisType {
public:
  enum {
     Invalid         = -1,
     GaussLegendre   = 0,  ///< Open type
     GaussLobatto    = 1,  ///< Closed type
     Positive        = 2,  ///< Bernstein polynomials
```

`mfem-sys`: Type-Safe Enums

```rust
// lib.rs in mfem-sys, inside #[cxx::bridge] pub mod ffi {
#[repr(i32)]
enum BasisType {
    Invalid = -1,
    /// Open type
    GaussLegendre = 0,
    /// Closed type
    GaussLobatto = 1,
    /// Bernstein polynomials
    Positive = 2,
    // ... and a bunch more ...
}
```

`mfem-sys`: Create FiniteElementSpace: C++ & Rust

```
// ex1.cpp in mfem-cpp
FiniteElementCollection* fec;
// ...
FiniteElementSpace fespace(&mesh, fec);

// C++ above, Rust below

// ex1.rs in mfem-sys
let fec: &FiniteElementCollection = match &owned_fec {
    // ...
};
let fespace = FiniteElementSpace_ctor(&mesh, fec, 1, OrderingType::byNODES);
```

`mfem-sys`: Implicit Ownership Rules in C++

```
// the constructor signature and class def from fespace.hpp in
mfem-cpp
FiniteElementSpace(Mesh* mesh, const FiniteElementCollection* fec,
  int vdim=1, int ordering=Ordering::byNODES);

class FiniteElementSpace {
protected:
  /// The mesh that FE space lives on (not owned).
  Mesh *mesh;
  /// Associated FE collection (not owned).
  const FiniteElementCollection *fec;
```

`mfem-sys`: Explicit Ownership Rules in Rust FFI

```rust
// lib.rs in mfem-sys, inside unsafe extern "C++" {
type FiniteElementSpace<'mesh, 'fec>;

fn FiniteElementSpace_ctor<'mesh, 'fec>(
    mesh: &'mesh Mesh,
    fec: &'fec FiniteElementCollection,
    vdim: i32,
    ordering: OrderingType,
) -> UniquePtr<FiniteElementSpace<'mesh, 'fec>>;

fn GetTrueVSize(self: &FiniteElementSpace) -> i32;
```

`mfem-sys`: Explicit Ownership Rules in Rust FFI

- Definitely my favorite feature of CXX!
- Helped me find a bug that was causing a segmentation fault.
- Smart pointers help, but borrow checking is even more robust.
- Borrow checking uses info that no longer exists at runtime.

  (This is true of static type checking in general.)

mfem-sys: Create and Init a GridFunction: C++ & Rust

```
// ex1.cpp in mfem-cpp
GridFunction x(&fespace);
x = 0.0;


// C++ above, Rust below


// ex1.rs in mfem-sys
let mut x = GridFunction_ctor_fes(&fespace);
GridFunction_SetAll(x.pin_mut(), 0.0);
```

`mfem-sys`: Remember, We're Translating Example 1

Again, the steps of MFEM Example 1 (`ex1.cpp`):

1. Load and refine a `Mesh`.
2. Create a `FiniteElementSpace`.
3. Create a `GridFunction` and set it to an initial guess.

   — We are here; halfway done! —

4. Create the `LinearForm` and `BilinearForm`.
5. Translate to and solve the set of algebraic equations.
6. Translate the solution back to a `GridFunction`, and save it.

`mfem-sys`: Quick Reminder, the Heat Problem

- bilinear form (left): $a(u, v) := (\nabla u, \nabla v)$
- linear form (right): $b(v) := (f, v)$

```
// bilininteg.hpp in mfem-cpp

/** Class for integrating the bilinear form a(u,v) := (Q ∇u, ∇v)
  where Q can be a scalar or a matrix coefficient. */
class DiffusionIntegrator: public BilinearFormIntegrator { /* ... */ };

// lininteg.hpp in mfem-cpp

/// Class for domain integration b(v) := (f, v)
class DomainLFIntegrator : public DeltaLFIntegrator { /* ... */ };
```

mfem-sys: Create the LinearForm: C++ & Rust

```
// ex1.cpp in mfem-cpp
LinearForm b(&fespace);
ConstantCoefficient one(1.0);
b.AddDomainIntegrator(new DomainLFIntegrator(one));
b.Assemble();


// C++ above, Rust below


// ex1.rs in mfem-sys
let mut b = LinearForm_ctor_fes(&fespace);
let one = ConstantCoefficient_ctor(1.0);
let one_coeff = ConstantCoefficient_as_Coeff(&one);
let integrator = DomainLFIntegrator_ctor_ab(one_coeff, 2, 0);
let lfi = DomainLFIntegrator_into_LFI(integrator);
LinearForm_AddDomainIntegrator(b.pin_mut(), lfi);
b.pin_mut().Assemble();
```

`mfem-sys`: Create the BilinearForm: C++ & Rust

```
// ex1.cpp in mfem-cpp
BilinearForm a(&fespace);
a.AddDomainIntegrator(new DiffusionIntegrator(one));
a.Assemble();

// C++ above, Rust below

// ex1.rs in mfem-sys
let mut a = BilinearForm_ctor_fes(&fespace);
let bf_integrator = DiffusionIntegrator_ctor(one_coeff);
let bfi = DiffusionIntegrator_into_BFI(bf_integrator);
BilinearForm_AddDomainIntegrator(a.pin_mut(), bfi);
a.pin_mut().Assemble(0);
```

mfem-sys: Translate to Algebra: C++ & Rust

```cpp
// ex1.cpp in mfem-cpp
OperatorPtr A;
Vector B, X;
a.FormLinearSystem(
  ess_tdof_list,
  x, b,
  A, X, B);
```

```rust
// ex1.rs in mfem-sys
let mut a_mat = OperatorHandle_ctor();
let mut b_vec = Vector_ctor();
let mut x_vec = Vector_ctor();
BilinearForm_FormLinearSystem(
    &a,
    &ess_tdof_list,
    GridFunction_as_Vector(&x),
    LinearForm_as_Vector(&b),
    a_mat.pin_mut(),
    x_vec.pin_mut(),
    b_vec.pin_mut(),
);
```

mfem-sys: Solve the Algebraic Equations: C++ & Rust

```
// ex1.cpp in mfem-cpp
GSSmoother M((SparseMatrix&)(*A));
PCG(*A, M, B, X, 1, 200, 1e-12, 0.0);

// C++ above, Rust below

// ex1.rs in mfem-sys
let a_sparse = OperatorHandle_try_as_SparseMatrix(&a_mat)
    .expect("Operator is a SparseMatrix");
let mut m_mat = GSSmoother_ctor(a_sparse, 0, 1);
let solver = GSSmoother_as_mut_Solver(m_mat.pin_mut());
PCG(OperatorHandle_as_ref(&a_mat), solver, &b_vec, x_vec.pin_mut(),
    1, 200, 1e-12, 0.0);
```

mfem-sys: Translate Back and Save: C++ & Rust

```cpp
// ex1.cpp in mfem-cpp
a.RecoverFEMSolution(X, b, x);

ofstream mesh_ofs("refined.mesh");
mesh_ofs.precision(8);
mesh.Print(mesh_ofs);

ofstream sol_ofs("sol.gf");
sol_ofs.precision(8);
x.Save(sol_ofs);
```

```rust
// ex1.rs in mfem-sys
a.pin_mut().RecoverFEMSolution(
    &x_vec,
    LinearForm_as_Vector(&b),
    GridFunction_as_mut_Vector(x.pin_mut()),
);

let_cxx_string!(
    mesh_filename = "refined.mesh");
mesh.Save(&mesh_filename, 8);

let_cxx_string!(sol_filename = "sol.gf");
GridFunction_Save(&x, &sol_filename, 8);
```

🎉 `mfem-sys`: Example 1 Done! 🎉

Key Points of Rust FFI Bindings With CXX

- Use the construct-unique pattern for constructor-like functions.
- Use the trycatch-to-result pattern to handle C++ exceptions.
- Make the C++ API's implicit ownership rules explicit in Rust.
- Don't be afraid to get your hands dirty and write some C++ glue!
    - Write `Class_as_Base()`, `Class_into_Base()`, etc.
- Try to expose type-safe `enum`s to Rust wherever possible.

`mfem` (the Rust crate)

`mfem`: Idiomatic Rust Wrapper Types and Traits

- Depends on `mfem-sys`.
- Hides sharp bits such as UniquePtr, C/C++ strings, etc.
- Turns constructor-like FFI functions into `Self::new()`, etc.
- Turns method-like FFI functions into real `.method()`s.
- Provides field setters and getters.
- Turns C++ base classes into traits.
- Has identifiers that follow Rust best practices.

`mfem`: Translating Example 1

Again, let's follow the steps of MFEM Example 1 (`ex1.cpp`):

1. Load and refine a `Mesh`.
2. Create a `FiniteElementSpace`.
3. Create a `GridFunction` and set it to an initial guess.
4. Create the `LinearForm` and `BilinearForm`.
5. Translate to and solve the set of algebraic equations.
6. Translate the solution back to a `GridFunction`, and save it.

mfem: Load and Refine a Mesh: C++ vs Rust

```
// ex1.cpp in mfem-cpp
Mesh mesh(mesh_file, 1, 1);
int dim = mesh.Dimension();
int ref_levels = (int)floor(log(50000./mesh.GetNE())/log(2.)/dim);
for (int i = 0; i < ref_levels; i++) {
   mesh.UniformRefinement();
}
```

```
// ex1.rs in mfem
let mut mesh = Mesh::from_file(&args.mesh_file)?;
let dim = mesh.dimension();
let ref_levels = f64::floor(f64::log2(5.0e4 / mesh.get_num_elems() as f64) / dim as f64) as u32;
for _ in 0..ref_levels {
   mesh.uniform_refinement(RefAlgo::A);
}
```

`mfem`: The "Wrapper-Inner" Pattern

```rust
// lib.rs in mfem
pub struct Mesh {
    inner: UniquePtr<mfem_sys::ffi::Mesh>,
}

impl Mesh {
    pub fn new() -> Self { /* ... */ }
    pub fn from_file(path: &str) -> Result<Self, MfemError> { /* ... */ }

    pub fn dimension(&self) -> i32 { /* ... */ }
    pub fn get_num_elems(&self) -> i32 { /* ... */ }
    pub fn get_nodes<'fes, 'a: 'fes>(&'a self) -> Option<GridFunctionRef<'fes, 'a>> { /* ... */ }
    pub fn get_bdr_attributes<'a>(&'a self) -> ArrayIntRef<'a> { /* ... */ }
    pub fn uniform_refinement(&mut self, ref_algo: RefAlgo) { /* ... */ }
    pub fn save_to_file(&self, path: &str, precision: i32) { /* ... */ }
}
```

mfem: Create a FiniteElementSpace: C++ & Rust

```cpp
// ex1.cpp in mfem-cpp
FiniteElementCollection *fec;
bool delete_fec;
if (order > 0) {
  fec = new H1_FECollection(order, dim);
  delete_fec = true;
} else if (mesh.GetNodes()) {
  fec = mesh.GetNodes()->OwnFEC();
  delete_fec = false;
} else {
  fec = new H1_FECollection(
    order = 1, dim);
  delete_fec = true;
}



FiniteElementSpace fespace(&mesh, fec);
```

```rust
// ex1.rs in mfem
let owned_fec: Option<H1FeCollection> =if args.order > 0 {
    Some(H1FeCollection::new(args.order, dim, BasisType::GaussLobatto))
} else if mesh.get_nodes().is_none() {
    Some(H1FeCollection::new(1, dim, BasisType::GaussLobatto))
} else { None };
let owned_nodes = mesh.get_nodes();
let fec: &dyn FiniteElementCollection =match &owned_fec {
   Some(h1_fec) => h1_fec,
   None => {
       println!("Using isoparametric FEs");
       let nodes = owned_nodes.as_ref()
           .expect("Mesh has its own nodes");
       nodes.get_own_fec().expect("OwnFEC exists");
   }
};

let fespace =
    FiniteElementSpace::new(&mesh, fec, 1,OrderingType::byNODES);
```

mfem: The "Base-Trait" Pattern

```
// lib.rs in mfem
trait AsBase<T> { // sibling traits omitted here: AsBaseMut and IntoBase
    fn as_base(&self) -> &T;
}


// In C++, FiniteElementCollection is a base class
// In Rust, there's no inheritance, so it's a trait instead
pub trait FiniteElementCollection:
     AsBase<mfem_sys::ffi::FiniteElementCollection> {
    // ...
}
```

mfem: "H1FeCollection: public FEC" in Rust

```rust
// lib.rs in mfem


// In C++, H1_FECollection is a subclass of FiniteElementCollection
// In Rust, H1FeCollection implements the FiniteElementCollection trait
impl FiniteElementCollection for H1FeCollection {}


impl AsBase<mfem_sys::ffi::FiniteElementCollection> for H1FeCollection {
    fn as_base(&self) -> &mfem_sys::ffi::FiniteElementCollection {
        mfem_sys::ffi::H1_FECollection_as_FEC(&self.inner)
    }
}
```

mfem: Create and Init a GridFunction: C++ & Rust

```
// ex1.cpp in mfem-cpp
GridFunction x(&fespace);
x = 0.0;


// C++ above, Rust below


// ex1.rs in mfem
let mut x = GridFunction::new(&fespace);
x.set_all(0.0);
```

`mfem`: Remember, We're Translating Example 1

Again, the steps of MFEM Example 1 (`ex1.cpp`):

1. Load and refine a `Mesh`.
2. Create a `FiniteElementSpace`.
3. Create a `GridFunction` and set it to an initial guess.

   — We are here; halfway done! —

4. Create the `LinearForm` and `BilinearForm`.
5. Translate to and solve the set of algebraic equations.
6. Translate the solution back to a `GridFunction`, and save it.

mfem: Create the LinearForm: C++ & Rust

```
// ex1.cpp in mfem-cpp
LinearForm b(&fespace);
ConstantCoefficient one(1.0);
b.AddDomainIntegrator(new DomainLFIntegrator(one));
b.Assemble();

// C++ above, Rust below

// ex1.rs in mfem
let mut b = LinearForm::new(&fespace);
let one = ConstantCoefficient::new(1.0);
let integrator = DomainLFIntegrator::new(&one, 2, 0);
b.add_domain_integrator(integrator);
b.assemble();
```

mfem: Create the BilinearForm: C++ & Rust

```
// ex1.cpp in mfem-cpp
BilinearForm a(&fespace);
a.AddDomainIntegrator(new DiffusionIntegrator(one));
a.Assemble();

// C++ above, Rust below

// ex1.rs in mfem
let mut a = BilinearForm::new(&fespace);
let bf_integrator = DiffusionIntegrator::new(&one);
a.add_domain_integrator(bf_integrator);
a.assemble(true);
```

mfem: Translate to Algebra: C++ & Rust

```cpp
// ex1.cpp in mfem-cpp
OperatorPtr A;
Vector B, X;
a.FormLinearSystem(
  ess_tdof_list,
  x, b,
  A, X, B);
```

```rust
// ex1.rs in mfem
let mut a_mat = OperatorHandle::new();
let mut b_vec = Vector::new();
let mut x_vec = Vector::new();
a.form_linear_system(
    &ess_tdof_list,
    &x, &b,
    &mut a_mat, &mut x_vec, &mut b_vec,
);
```

mfem: Solve the Algebraic Equations: C++ & Rust

```
// ex1.cpp in mfem-cpp
GSSmoother M((SparseMatrix&)(*A));
PCG(*A, M, B, X, 1, 200, 1e-12, 0.0);


// C++ above, Rust below


// ex1.rs in mfem
let a_sparse = SparseMatrixRef::try_from(&a_mat)
    .expect("Operator is a SparseMatrix");
let mut m_mat = GsSmoother::new(&a_sparse, 0, 1);
solve_with_pcg(&a_mat, &mut m_mat, &b_vec, &mut x_vec,
    1, 200, 1e-12, 0.0);
```

`mfem`: Translate Back and Save: C++ & Rust

```cpp
// ex1.cpp in mfem-cpp
a.RecoverFEMSolution(X, b, x);

ofstream mesh_ofs("refined.mesh");
mesh_ofs.precision(8);
mesh.Print(mesh_ofs);

ofstream sol_ofs("sol.gf");
sol_ofs.precision(8);
x.Save(sol_ofs);
```

```rust
// ex1.rs in mfem
a.recover_fem_solution(&x_vec, &b, &mut x);

mesh.save_to_file("refined.mesh", 8);
x.save_to_file("sol.gf", 8);
```

🎉 `mfem`: Example 1 Done! 🎉

Key Points of Making Idiomatic Rust Wrappers

- Try to hide (within reason) FFI-specific machinery from the user.
- Use the wrapper-inner pattern to give easy access to
  - constructor-like functions, e.g. `new()`, `from_thing()`
  - methods, getters, and setters.
- Use the base-trait pattern to encode inheritance.
- Choose identifiers that adhere to Rust community standards.

Future Work

- Incrementally extend coverage and translate all Examples.
- Contribute improvements back to upstream MFEM.

  For example, around const correctness.
- Make it work with MPI (Message Passing Interface).

  https://crates.io/crates/mpi
- Cover other components (Hypre, etc.).

Thanks for Listening!

The Code is Open Source!

https://github.com/mkovaxx/mfem-rs

Consider Applying to Braid!

https://braid.tech

Special Thanks To

- Brian Schwind, who taught me about the cxx bridge and whose opencascade-rs crate served as both inspiration and copyable project setup.
- Sjors Donkers, who is a source of wisdom about writing modern C++.
- Guido Cossu & Ivo Timoteo, who continue to answer my endless questions about Finite Element Methods.

Appendix

More Thoughts Re: C++ vs Rust

- The borrow checker is your friend. Give it as much info as possible.

  https://github.com/mkovaxx/mfem-rs/commit/d5ea4db7f92b06cd9b446c162cb2c5b9ee93c2f2
- Result<&T> instead of *const T, etc.

  https://github.com/mkovaxx/mfem-rs/commit/4e3a63fb1f7d95b5e9dd67c4ef05429aeda4e1c4
- Sometimes you must get your hands dirty…

  https://github.com/mkovaxx/mfem-rs/commit/06e65bb72b7ff625ea40dfe326c47d1adb7bfafe
- API defaults: const vs mut

  changing const → mut breaks argument compatibility

  changing mut → const breaks return value compatibility